Types, or: Where's the difference between CCS and π?

Davide Sangiorgi*

INRIA
Davide.Sangiorgi@inria.fr

Abstract. The π-calculus is the paradigmatic calculus of mobile processes. With respect to previous formalisms for concurrency, most notably CCS, the most novel aspect of π-calculus is probably its rich theory of *types*. We explain the importance of types in the π-calculus on a concrete example: the *termination* property.
A process M terminates if it cannot produce an infinite sequence of reductions $M \xrightarrow{\tau} M_1 \xrightarrow{\tau} M_2 \ldots$. Termination is a useful property in concurrency. For instance, a terminating applet, when loaded on a machine, will not run for ever, possibly absorbing all computing resources (a 'denial of service' attack). Similarly, termination guarantees that queries to a given service originate only finite computations.
We consider the problem of proving termination of non-trivial subsets of CCS and π-calculus. In CCS the proof is purely combinatorial, and is very simple. In the π-calculus, by contrast, combinatorial proofs appear to be very hard. We show how to solve the problem by taking into account type information.

1 Introduction

The π-calculus is the paradigmatic calculus of mobile processes. With respect to previous formalisms for concurrency, most notably CCS, the most novel aspect of π-calculus is probably its rich theory of *types*. We explain the importance of types in the π-calculus on a concrete example: the *termination* property.

'Terminating' is different from 'convergent'. A convergent term has *at least one* finite (complete) internal run; in a terminating term, by contrast, *all* internal runs are finite (there may even be an infinity of internal runs, and of unbounded length). For instance, the process

$$(a.\,\overline{c} + !a.\,\overline{a}) \mid \overline{a}$$

is convergent but non-terminating. (When the values communicated are not important, we abbreviate inputs $a(x).\,M$ and outputs $\overline{a}v.\,M$ as $a.\,M$ and $\overline{a}.\,M$. The replication operator $!\pi.\,M$ can be thought of as an infinite number of copies of $\pi.\,M$ in parallel; in a language with recursion, it can be taken as an abbreviation for $\mu X.\,(\pi.\,(M \mid X))$.) Also, 'terminating' is different from 'finite'. In a terminating process, only the internal runs are required to be finite. In a finite process, by contrast, all runs are finite, including

* Work supported by the EC project "PROFUNDIS" (IST-2001-33100)

runs consisting of external actions such as inputs and outputs. For instance, $!a.\overline{b}$ is terminating but is not finite.

Termination is a useful property, especially for mobile processes. For instance, if we interrogate a process, we may want to know that an answer is eventually produced. Similarly, when we load an applet we may like to know that the applet will not run for ever on our machine, possibly absorbing all the computing resources (a 'denial of service' attack). In general, if the lifetime of a process can be infinite, we may want to know that the process does not remain alive simply because of non-terminating internal activity, and that, therefore, the process will always accept interactions with the environment. For similar reasons of protection, all programs of languages for active networks such as Plan [HKM+99] must terminate.

We consider the problem of proving termination of a non-trivial subset of the π-calculus. The corresponding problem in CCS has a simple solution, and the proof is purely combinatorial. In the π-calculus, by contrast, a combinatorial proof appears to be very hard. We shall solve the problem by taking into account type information.

We explain why the termination problem is much harder in the π-calculus than in CCS. Consider the process language given by the grammar production

$$M ::= (\boldsymbol{\nu} a_1, \ldots, a_n\,)(!a_1(x).\,M_1 \mid \ldots \mid !a_n(x).\,M_n \mid \overline{b_1}v_1 \mid \ldots \mid \overline{b_r}v_r) \qquad (1)$$

where $n, r \geq 0$. This language includes non-terminating processes, such as

$$\boldsymbol{\nu} a\,(\overline{a} \mid !a.\,\overline{a})\,,$$

and

$$(\boldsymbol{\nu} a, b\,)(\overline{a} \mid !a.\,\overline{b} \mid !b.\,\overline{a}),$$

in which the inputs at a and b are mutually recursive. Further, regardless of what the values v_j are, the size of a process may grow with reduction, because replications are persistent and reductions may liberate new processes (which can themselves be replicated). We can avoid mutual recursions by imposing an order on the inputs: require that in any term $(\boldsymbol{\nu} a_1, \ldots, a_n\,)(!a_1(x).\,M_1 \mid \ldots \mid !a_n(x).\,M_n \mid \overline{b_1}v_1 \mid \ldots \mid \overline{b_r}v_r)$ process M_i only uses names a_j with $j < i$. This condition does indeed guarantee termination if, as in CCS, values are first order and therefore cannot be links.[1] For the proof, it suffices to define a measure that decreases after each reduction. This is easy to do because a process liberated by a replication satisfies the same condition on the usage of names as the replication itself.

The same argument does not work if values can be links, as in the π-calculus. For instance, the process liberated by a replication $!a_i(x).\,M_i$ can be $M_i\{{}^{v_j}\!/\!_x\}$ with $v_j = a_j$ and $j > i$. Indeed if values can be links, then the language (1) does include non-terminating processes even if inputs are ordered; an example is

$$R \stackrel{\text{def}}{=} \boldsymbol{\nu} a\,(!a(x).\,\overline{x}x \mid \overline{a}a)\,.$$

Modulo structural congruence ($\equiv$), a relation that allows us some simple rearrangements of the structure of a process, R reduces to itself.

[1] We prefer the word 'link' to 'channel' because the latter has a more restrictive connotation.

Intuitively, R does not terminate because it has self-applications, that is, outputs in which the value emitted is related to the link along which the output occurs. A discussion on the (very subtle) problems given by self-application is worthwhile. These problems are well-known in the λ-calculus. The process R, as well as the other examples of self-applications below, are not translations of λ-expressions, but are inspired by the λ-calculus.

In R, the self-applications are syntactic: in $\overline{x}x$ and $\overline{a}a$ the link of the output and the value emitted are identical. If all self-applications were syntactic, then termination could be ensured, operationally, by checking that after each reduction no output of the form $\overline{a}a$ exists. Unfortunately, non-termination may also arise because of 'implicit' self-applications: these are outputs $\overline{a}b$ in which the link b emitted is different from the link a of the output, but b may be used by a recipient to access a. An example of implicit self-application is the process $\overline{a}b.\,!b(x).\,\overline{a}x$. Indeed, under certain hypothesis the process is behaviourally equivalent to $\overline{a}a$ [Mer01]. This form of implicit self-application is the reason for the divergence of

$$Q \stackrel{\text{def}}{=} \boldsymbol{\nu} b\,(\overline{a}b \mid !b(z).\,\overline{a}z) \mid !a(x).\,(\boldsymbol{\nu} y\,(\overline{x}y \mid !y(z).\,\overline{x}z))\,,$$

which in two steps can reduce to a process weakly bisimilar to Q itself.

Here is however an example of a process that does not terminate because of self-application but that cannot possibly be considered behaviourally equivalent to a process with syntactic self-applications:

$$\begin{aligned} M \stackrel{\text{def}}{=}\ & !a(x).\,\boldsymbol{\nu} b\,(\overline{x}b \mid b(y).\,\overline{y}x) \\ & \mid !v(z).\,\overline{z}a \\ & \mid \overline{a}v\,. \end{aligned}$$

M has the following divergent computation:

$$\begin{array}{lll} M & = & !a(x).\,\boldsymbol{\nu} b\,(\overline{x}b \mid b(y).\,\overline{y}x) \mid !v(z).\,\overline{z}a \mid \overline{a}v \\ & \xrightarrow{\tau} & \overline{v}b \mid b(y).\,\overline{y}v \mid !a(x).\,\boldsymbol{\nu} b\,(\overline{x}b \mid b(y).\,\overline{y}x) \mid !v(z).\,\overline{z}a \qquad [av] \\ & \xrightarrow{\tau} & b(y).\,\overline{y}v \mid !a(x).\,\boldsymbol{\nu} b\,(\overline{x}b \mid b(y).\,\overline{y}x) \mid \overline{b}a \mid !v(z).\,\overline{z}a \qquad [vb] \\ & \xrightarrow{\tau} & \overline{a}v \mid !a(x).\,\boldsymbol{\nu} b\,(\overline{x}b \mid b(y).\,\overline{y}x) \mid !v(z).\,\overline{z}a \qquad [ba] \\ & \equiv & M \\ & \xrightarrow{\tau} & \ldots \end{array}$$

where the column on the right indicates the action performed in the reduction, that is, the link along which the reduction occurs and the value exchanged. The culprit of the non-termination of M is the subterm $\overline{x}b \mid b(y).\,\overline{y}x$. This process, although behaviourally quite different from a syntactic self-application, does represent a self-application because the name b transmitted at x is used in an input inside which x appears in an output.

But self-applications can be even more nasty. They can arise in processes that, at the beginning, contain no patterns of dependencies between inputs and outputs – not even a 'weak' pattern as the one of M above. An example is the process N below. It is hard to see where an implicit self-application is located in the syntax of N. For instance, there is an output $\overline{a_2}a_1$ but the input at a_1 does not use a_2. However, recursive dependencies among names can arise dynamically in N, at run time.

$$\begin{aligned} N \stackrel{\text{def}}{=} & \,!a(x).\,(\boldsymbol{\nu} a_1, a_2\,)(\overline{a_2}a_1 \mid !a_1(y).\,\overline{x}y \mid !a_2(y).\,\overline{x}y) \\ & \mid !a_0(z).\,\overline{a}z \\ & \mid \overline{a}a_0 \end{aligned}$$

Moreover, whereas the divergent run of M has at least one action – $[av]$ – that repeats itself an infinite number of times, N has a divergent computation in which all actions performed are different. The infinite computation of N consists of cycles of increasing length; a cycle begins with a reduction at a and ends with a reduction at a_0. These are the actions for the first three cycles:

$$\begin{aligned} & [a\, a_0], [a_2\, a_1], [a_0\, a_1], \\ & [a\, a_1], [a_4\, a_3], [a_1\, a_3], [a_0\, a_3]\,, \\ & [a\, a_3], [a_6\, a_5], [a_3\, a_5], [a_1\, a_5], [a_0\, a_5]\,. \end{aligned}$$

We shall avoid the problems given by self-applications by means of types, moving to the *simply-typed π-calculus* (Section 3).

Besides mutual recursion and self-application, the other major issue for the termination of π-calculus processes is *state*. In all examples of processes seen so far, all links are *functional*, in the sense that each link appears only once in input, and the input is replicated. In other words, the service offered by the link is always available and does not change over time. Here are examples of stateful terms:

$$\begin{aligned} & N_1 \stackrel{\text{def}}{=} a(x).\,M && N_3 \stackrel{\text{def}}{=} a(x).\,(N \mid a(y).\,M) \\ & N_2 \stackrel{\text{def}}{=} !a(x).\,M + N && N_4 \stackrel{\text{def}}{=} !b(x).\,(N \mid !a(y).\,M) \\ & && N_5 \stackrel{\text{def}}{=} \mu X(y).\,a(x).\,\overline{y}v.\,X\lfloor x \rfloor \end{aligned}$$

(where $\mu X(y).\,M$ is a recursive definition with formal parameter y and body M, and $X\lfloor v \rfloor$ is a recursive call with actual parameter v). In N_1, the input at a is ephemeral and can therefore vanish. In N_2 the input at a vanishes if the summand N is used first. In N_3 and N_5, different outputs at a may activate different processes (in N_5 this happens because the continuation of the input uses the parameter y of the recursive definition). Finally, in N_4 the input at a is not immediately available.

It is known that in imperative sequential languages higher-order references can break termination [HMST95]. Since higher-order references can be modeled in π-calculus, using recursion, some constraints on state are expected. It is not sufficient, however, to transport the conditions for imperative sequential languages onto the π-calculus, which has more diverse forms of state. We will impose two constraints on state: the parameters of recursive definitions should be first order; certain forms of nesting between inputs are disallowed.

Thus, our language $\mathcal{P}$ of terminating π-calculus processes has the constraints on recursive inputs, self-applications, and state mentioned above. The proof of termination is in two parts. The first part uses the technique of logical relations. Logical relations are well-known in functional languages and are used, in particular, to prove termination of typed λ-calculi. We have not been able to apply the technique to the whole language

$\mathcal{P}$. We have only been able to apply it to a small sublanguage, $\mathcal{P}^-$. This is a non-deterministic language, with only asynchronous outputs, and in which all names are functional. One of the reasons for restricting the logical-relation technique to $\mathcal{P}^-$ is that we need several times the Replication Theorems (laws for the distributivity of replicated processes). These theorems hold only if the names are functional.

The language $\mathcal{P}^-$ is not very expressive. It is however a powerful language for the termination property. The second part of our proof shows that the termination of $\mathcal{P}^-$ implies that of $\mathcal{P}$. For this, we use process calculus techniques, most notably techniques for behavioural preorders.

In this paper, some proofs are only sketched. For more details, we refer to [San01], where, moreover, the language of processes is richer. For instance, here the only first-order value is unit, whereas in [San01] arbitrary first-order values are allowed.

Structure of the paper Sections 2 and 3 contain background material on the π-calculus. In Section 4 we define termination and the language $\mathcal{P}$. The proof of termination of $\mathcal{P}$ is developed in the Section 5 to 8. First, we prove the termination of the language $\mathcal{P}^-$ (Section 5-7). Then, in Section 8, we sketch how the termination of $\mathcal{P}$ is derived from that of $\mathcal{P}^-$.

Related work Termination has been widely studied in sequential languages [GLT88,Mit96], but very little has been done in concurrency. Kobayashi's type system [Kob00] ensures that in every fair reduction sequence a process trying to perform a communication will eventually succeed. Types can express causality, obligation, and time limit on the usage of a link. The main differences with our work are: the types of [Kob00] are rather sophisticated; the properties guaranteed are different (a well-typed process in [Kob00] may still have a divergent computation); the system of [Kob00] cannot handle various forms of replications, for instance it cannot handle the processes encoding the simply-typed λ-calculus.

The closest to our work is Berger, Honda, and Yoshida's [YBK01], which is the first study of termination in the π-calculus. The main differences with our work are the following. First, the conditions that [YBK01] impose for termination are almost entirely expressed by means of *graph types* [Yos96]. By contrast, we separate typing and syntactic conditions. This allows us, for instance, to use more standard notions of types. Second, the language $\mathcal{P}^{\mathrm{BHY}}$ of [YBK01] is a language of functional processes. For instance, every name has only one input occurrence, and reduction is confluent. It is indeed close to the subset of $\mathcal{P}^-$ (Section 5) without sums. Thus, the λ-calculus with resources (Section 9, [BL00]), or deterministic subsets of it such as the λ-calculus with multiplicities, cannot not be encoded in $\mathcal{P}^{\mathrm{BHY}}$. Also, in $\mathcal{P}^{\mathrm{BHY}}$ all outputs are bound; that is, only private names can be transmitted. Third, the encoding of the simply-typed (call-by-name) λ-calculus into the π-calculus is fully abstract if the π-calculus language is taken to be $\mathcal{P}^{\mathrm{BHY}}$. By contrast, the encoding is not fully abstract with respect to $\mathcal{P}$: standard counterexamples to full abstraction of λ-calculus encodings [SW01] live in $\mathcal{P}$. Fourth, the technique of logical relations is used by both us and [YBK01], but the details are quite different. For instance, the proof in [YBK01] exploits the property that the processes are confluent, by allowing also certain outputs underneath prefixes to be consumed. Finally, the types of [YBK01] also guarantee that certain visible actions will eventually be performed. This property does not hold in our case.

2 The localised π-calculus

The syntax of the calculus, in Table 1, has the process constructs of the standard (monadic) π-calculus [Mil99,SW01]. The calculus is *localised* [Mer01], that is, the recipient of a link cannot use it in input. Formally, in an input $a(x).\, M$, name x cannot appear free in M as subject of an input. (The subject of an input is the name at which the input is performed; for instance, the subject of $a(x).\, M$ is a.) Locality has been found useful in practice – it is adopted by a number of experimental languages derived from the π-calculus, most notably Join [FG96] – and has also useful consequences on the theory [Mer01]. In our work, locality is essential: most of our results rely on it.

Bound names, free names, and names of a process M, respectively written $\mathsf{bn}(M)$, $\mathsf{fn}(M)$, and $\mathsf{n}(M)$, are defined in the usual way. We do not distinguish α-convertible terms. Unless otherwise stated, we also assume that, in any term, each bound name is different from the free names of the term and from the other bound names. In a statement, we sometimes say that a name is *fresh* to mean that it does not occur in the objects of the statement, like processes and actions. A name a is *fresh for M* if a does not occur in M. If R' is a subterm of R we say that R' is *guarded in R* if R' is underneath a prefix of R; otherwise R' is *unguarded in R*. We use a tilde to indicate a tuple. All notations are extended to tuples in the usual way.

In a recursive definition $\mu X(\widetilde{x}).\, M$, the recursion variable is X and the formal parameters are $\widetilde{x}$. The actual parameters of a recursion are supplied in a recursion call $H\lfloor\widetilde{v}\rfloor$. We require that, in a recursive definition $\mu X(\widetilde{x}).\, M$, the only free recursion variable of M is X. This constraint simplifies some of our proofs, but can be lifted. Moreover, the recursion variable X should be guarded in the body M of the recursion.

When a recursion has no parameters we abbreviate $\mu X().\, R$ as $\mu X.\, R$, and calls $(\mu X.\, R)\lfloor\,\rfloor$ and $X\lfloor\,\rfloor$ as $\mu X.\, R$ and X, respectively. For technical reasons, we find it convenient a restriction operator that introduces several names at once.

A *first-order value* is a value that does not contain links. Examples are: an integer, a boolean value, a pair of booleans, a list of integers. To facilitate the reading of the proofs, in this paper the only first-order value is the *unit* value, $\star$. (Arbitrary first-order values are studied in [San01].)

The SOS rules for the transition relation of the processes of the calculus are presented in Table 2, where α ranges over actions. (The symmetric of PAR-1, COM-1, CLOSE-1, and SUM-1 have been omitted.) They are the usual transition rules for the π-calculus, in the early style.

2.1 Other process operators

For the proofs in the paper, we consider various π-calculus languages. They are defined from the operators introduced above, with transition rules as by Table 2, plus: *asynchronous output*, $\overline{v}w$, that is, output without continuation; and *replication*, $!M$, which

$a, b, c, d, \ldots, x, y, z \ldots$			*Names*
			Values
v, w	$::=$	a	name
	$\mid$	$\star$	unit value
			Processes
M, N	$::=$	$\mathbf{0}$	nil process
	$\mid$	$H\lfloor \widetilde{v} \rfloor$	recursion call
	$\mid$	$a(x).\, M$	input
	$\mid$	$\overline{v}w.\, M$	output
	$\mid$	$M \mid M$	parallel
	$\mid$	$M + M$	sum
	$\mid$	$\boldsymbol{\nu}\widetilde{a}\, M$	restriction
H	$::=$	X	recursion variable
	$\mid$	$\mu X(\widetilde{x}).\, M$	recursive definition

Table 1. Syntax of processes

$$\text{INP: } a(x).\,M \xrightarrow{a\,v} M\{v/x\} \qquad \text{REC: } \frac{M\{\mu X(\widetilde{x}).\,M/X\}\{\widetilde{v}/\widetilde{x}\} \xrightarrow{\alpha} M'}{(\mu X(\widetilde{x}).\,M)\lfloor\widetilde{v}\rfloor \xrightarrow{\alpha} M'}$$

$$\text{OUT: } \overline{a}v.\,M \xrightarrow{\overline{a}v} M \qquad \text{RES: } \frac{\boldsymbol{\nu}\widetilde{b}\,M \xrightarrow{\alpha} M'}{(\boldsymbol{\nu}a,\widetilde{b}\,)M \xrightarrow{\alpha} \boldsymbol{\nu}a\,M'}\ a \notin \mathsf{n}(\alpha)$$

$$\text{OPEN: } \frac{\boldsymbol{\nu}\widetilde{b}\,M \xrightarrow{\overline{x}a} M'}{(\boldsymbol{\nu}a,\widetilde{b}\,)M \xrightarrow{(\boldsymbol{\nu}a\,)\overline{x}a} M'}\ x \neq a$$

$$\text{SUM-1: } \frac{M \xrightarrow{\alpha} M'}{M + N \xrightarrow{\alpha} M'} \qquad \text{PAR-1: } \frac{M \xrightarrow{\alpha} M'}{M \mid N \xrightarrow{\alpha} M' \mid N}\ \text{if } \mathsf{bn}(\alpha) \cap \mathsf{fn}(N) = \emptyset$$

$$\text{COM-1: } \frac{M \xrightarrow{a\,v} M' \qquad N \xrightarrow{\overline{a}v} N'}{M \mid N \xrightarrow{\tau} M' \mid N'}$$

$$\text{CLOSE-1: } \frac{M \xrightarrow{ab} M' \qquad N \xrightarrow{\boldsymbol{\nu}b\,\overline{a}b} N'}{M \mid N \xrightarrow{\tau} \boldsymbol{\nu}b\,(M' \mid N')}\ \text{if } b \notin \mathsf{fn}(M)$$

Table 2. Transition rules

represents an infinite parallel composition of copies of M. Their transition rules are:

$$\overline{a}v \xrightarrow{\overline{a}v} \mathbf{0} \qquad \frac{M \mid !M \xrightarrow{\alpha} M'}{!M \xrightarrow{\alpha} M'}$$

3 The simply-typed calculus

The grammar of types for the simply-typed π-calculus is:

$$T ::= \sharp\,T \;\Big|\; \mathtt{unit}$$

where the *connection type* $\sharp\,T$ is the type of a link that carries tuples of values of type T, and `unit` is the only *first-order type* (that is, the type of a first-order value).

A *link* is a name of a connection type. A link is *first order* if it carries first-order values. It is *higher order* if it carries higher-order values (i.e., links). Note that 'first-order name' is different from 'first-order link': a first-order name has a type `unit`, whereas a first-order link has a type $\sharp\,$`unit`.

Our type system is à la Church, thus each name has a predefined type. We assume that for each type there is an infinite number of names with that type. We write $x \in T$ to mean that the name x has type T. Similarly, each recursion variable X has a predefined tuple of types, written $X \in \langle\widetilde{T}\rangle$, indicating the types of the arguments of the recursion. A judgment $\vdash M$ says that M is a well-typed process; a judgment $\vdash v : T$ says that v is a well-typed value of type T. For values v, w we write $v : w$ to mean that v and w have the same type.

The typing rules are in Table 3. The typing rules for the operators of Section 2.1 (asynchronous output, replication) are similar to those of (standard) output and parallel composition.

A process is *closed* if all its free names have a connection type (i.e., they are links). The closed processes are the 'real' processes, those that, for instance, are supposed to be run, or to be tested. If a process is not closed, it has some names yet to be instantiated. A *closing substitution* for a process R is a substitution σ such that $R\sigma$ is closed (R may already be closed, and σ may just rename some of its links).

$$\text{T-PAR}: \frac{\vdash M \qquad \vdash N}{\vdash M \mid N} \qquad\qquad \text{T-SUM}: \frac{\vdash M \qquad \vdash N}{\vdash M + N}$$

$$\text{T-RVAR}: \frac{X \in \langle \widetilde{T} \rangle \qquad \vdash \widetilde{v} : \widetilde{T}}{\vdash X\lfloor \widetilde{v} \rfloor} \qquad\qquad \text{T-REC}: \frac{X \in \langle \widetilde{T} \rangle \qquad \widetilde{x} \in \widetilde{T} \qquad \vdash \widetilde{v} : \widetilde{T} \qquad \vdash M}{\vdash (\mu X(\widetilde{x}).\, M)\lfloor \widetilde{v} \rfloor}$$

$$\text{T-OUT}: \frac{\vdash v : \sharp T \qquad \vdash w : T \qquad \vdash M}{\vdash \overline{v}w.\, M} \qquad \text{T-INP}: \frac{\vdash v : \sharp T \qquad x \in T \qquad \vdash M}{\vdash v(x).\, M}$$

$$\text{T-RES}: \frac{x_i \in \sharp T_i \ \text{ for some } T_i \ (1 \leq i \leq n) \qquad \vdash M}{\vdash (\boldsymbol{\nu} x_1 \ldots x_n\,)M}$$

$$\text{T-NIL}: \ \vdash \mathbf{0} \qquad\qquad \text{T-UNIT}: \ \vdash \star : \texttt{unit} \qquad\qquad \text{T-VAR}: \frac{x \in T}{\vdash x : T}$$

Table 3. Typing rules

4 Terminating processes

Our goal in this paper is to isolate a subset of processes as large as possible that terminate.

Definition 1. *A process M* diverges *(or* is divergent*) if there is an infinite sequence of processes $M_1, \ldots, M_n, \ldots$ with $M_1 = M$, such that, for all i,*

$$M_i \xrightarrow{\tau} M_{i+1}.$$

M terminates *(or* is terminating*), written $M \in \text{TER}$, if M is not divergent.* □

We have explained in Section 1 what makes termination hard: self-applications, recursive inputs, state. Our language of terminating processes is defined by four constraints. Three of them, mostly syntactic, are reported in Condition 1. The first condition is for recursive inputs; the second and third conditions control state. The last constraint – condition 1 of Definition 2 – controls self-applications.

We explain the new terminology used in the conditions. A name a appears free in *output position in* N if N has a free occurrence of a in an output prefix. An input is *replicated* if the input is inside the body of a recursive definition. Thus, a process M has free replicated first-order inputs if M contains a free first-order input inside the body of a recursive definition. For instance, if a is a first-order link then

$$\mu X\,.\,a(y).\,(\mathbf{0} \mid X)$$

has free replicated first-order inputs, whereas

$$\boldsymbol{\nu} a\,(\mu X\,.\,a(y).\,(\mathbf{0} \mid X)) \mid b(z).\,\mathbf{0}$$

has not.

Condition 1 (Termination constraints on the grammar)

1. *Let* $\widetilde{a} = a_1, \ldots, a_n$*. In a process* $\boldsymbol{\nu}\widetilde{a}\, M$*, if* $a_i(x).\,N$ *is a free input of* M*, then the following holds:*
   (a) $a_i \in \widetilde{a}$*,*
   (b) names a_j *with* $j \geq i$ *do not appear free in output position in* N*.*
2. *In an higher-order input* $a(x).\,M$*, the continuation* M *does not contain free higher-order inputs, and does not contain free replicated first-order inputs.*
3. *In a recursive definition* $\mu X(\widetilde{x}).\,M$*:*
   (a) $\widetilde{x}$ *are first order,*
   (b) M *has no unguarded output and no unguarded if-then-else.* □

Condition (1b) poses no constraints on occurrences of names not in $\widetilde{a}$. The condition can be made simpler, but weaker, by requiring that names $\widetilde{a}$ do not appear free in output position in M.

To illustrate the meaning of condition (3a), consider a 1-place buffer, that receives values at a link a and retransmits them at a link b:

$$\mu X\,.\,a(x).\,\overline{b}x.\,X$$

This process respects the condition regardless of whether the values transmitted are first order or higher order. By contrast, a delayed 1-place buffer

$$\mu X(x).\,a(y).\,\overline{b}x.\,X\lfloor y \rfloor\,,$$

which emits at b the second last value received by a, respects the condition only if the values transmitted are first order.

We say that a process M respects the constraints of Condition 1 to mean that M itself and all its process subterms respect the constraints.

Definition 2 (Language $\mathcal{P}$). *$\mathcal{P}$ is the set of processes such that $M \in \mathcal{P}$ implies:*

1. M *is typable in the simply-typed* π*-calculus;*
2. $\boldsymbol{\nu}\widetilde{a}\, M$ *respects the constraints of Condition 1, where* $\widetilde{a} = \mathsf{fn}(M)$*.* □

Theorem 1. *All processes in $\mathcal{P}$ terminate.*

The proof of this theorem is in two parts; one in Sections 5-7, the other is sketched in Section 8. In the remainder of the paper, all processes and values are well typed, and substitutions map names onto values of the same type. Moreover, processes have no free recursion variables.

5 $\mathcal{P}^-$: Monadic functional non-deterministic processes

We define a very constrained calculus $\mathcal{P}^-$ whose processes will be proved to terminate using the technique of logical relations. We will then use $\mathcal{P}^-$ to derive the termination of the processes of the language $\mathcal{P}$ of Theorem 1.

The processes of $\mathcal{P}^-$ are functional, that is, the input end of each link occurs only once, is replicated, and is immediately available (cf., the uniform-receptiveness discipline, [San99]). To emphasize the 'functional' nature of these processes, we use the (input-guarded) replication operator $!a(x).\, M$ instead of recursion. Processes can however exhibit non-determinism, due to the presence of the sum operator. Outputs are asynchronous, that is, they have no continuations.

For the definition of $\mathcal{P}^-$, and elsewhere in the paper, it is useful to work up to structural congruence, a relation that allows us to abstract from certain details of the syntax of processes.

Definition 3 (Structural congruence). *Let R be a process of a language $\mathcal{L}$ whose operators include parallel composition, restriction, replication, and $\mathbf{0}$. We write $R \equiv_1 R'$ if R' is obtained from R by rewriting, in one step, a subterm of R using one of the rules below (from left to right, or from right to left)*

$$
\begin{aligned}
!R &= R \mid !R \\
\boldsymbol{\nu}\widetilde{a}\; \boldsymbol{\nu}\widetilde{b}\; R &= (\boldsymbol{\nu}\widetilde{a}, \widetilde{b}\,)R \\
(\boldsymbol{\nu}\widetilde{a}, a, b, \widetilde{b}\,)R &= (\boldsymbol{\nu}\widetilde{a}, b, a, \widetilde{b}\,)R \\
R_1 \mid R_2 &= R_2 \mid R_1 \\
R_1 \mid (R_2 \mid R_3) &= (R_1 \mid R_2) \mid R_3 \\
R \mid \mathbf{0} &= R
\end{aligned}
$$

(Note that if $R \equiv_1 R'$ then R' need not be in $\mathcal{L}$.) Structural congruence, $\equiv$, *is the reflexive and transitive closure of $\equiv_1$.* □

The definition of $\mathcal{P}^-$ uses the syntactic categories of *processes*, *pre-processes*, and *resources*. The *normal forms* for processes, pre-processes and resources of $\mathcal{P}^-$ are given in Table 4, where $\mathtt{in}(P)$ are the names that appear free in P in input position. Each new name is introduced with a construct of the form $\boldsymbol{\nu} a\ (!a(x).\, N \mid P)$ where the resource $!a(x).\, N$ is the only process that can ever input at a. In the definition of resources, the constraint $a \notin \mathsf{fn}(M^{\mathrm{NF}})$ prevents mutual recursion (calls of the replication from within its body).

Normal forms are not closed under reduction. For instance, if $M^{\mathrm{NF}} \xrightarrow{\tau} N$ then N may not be a process of the grammar in the table. However, N is structurally congruent to a normal form. We therefore define processes, resources, pre-processes by closing the normal forms with structural congruence. We need the reduction-closure property (Lemma 8) in later proofs.

Definition 4 (Language $\mathcal{P}^-$). *The sets $\mathcal{PR}$ of* processes, *$\mathcal{RES}$ of* resources, *and $\mathcal{P}^-$ of* pre-processes *are obtained by closing under $\equiv$ the corresponding (well-typed) normal forms in Table 4.* □

$$
\begin{array}{rcll}
 & & & \textit{Pre-processes} \\
P^{\mathrm{NF}} & ::= & \boldsymbol{\nu} a\,(I_a^{\mathrm{NF}} \mid P^{\mathrm{NF}}) & \text{with } \mathsf{fn}(I_a^{\mathrm{NF}}) \cap \mathtt{in}(P^{\mathrm{NF}}) = \emptyset \\
 & \mid & M^{\mathrm{NF}} & \\
 & \mid & I_a^{\mathrm{NF}} & \\
 & & & \textit{Resources} \\
I_a^{\mathrm{NF}} & ::= & !a(x).\, M^{\mathrm{NF}} & \text{with } a \notin \mathsf{fn}(M^{\mathrm{NF}}) \\
 & & & \textit{Processes} \\
M^{\mathrm{NF}} & ::= & \boldsymbol{\nu} a\,(I_a^{\mathrm{NF}} \mid M^{\mathrm{NF}}) & \\
 & \mid & M^{\mathrm{NF}} + M^{\mathrm{NF}} & \\
 & \mid & M^{\mathrm{NF}} \mid M^{\mathrm{NF}} & \\
 & \mid & \overline{v} w & \\
 & \mid & \mathbf{0} & \\
 & & & \textit{Values} \\
v & ::= & a & \text{name} \\
 & \mid & \star & \text{unit value}
\end{array}
$$

Table 4. The normal forms

Thus $M \in \mathcal{PR}$ if there is M^{NF} with $M \equiv M^{\mathrm{NF}}$. Pre-processes include resources and processes (which explains why pre-processes are indicated with the symbol $\mathcal{P}^{-}$). Pre-processes are ranged over by P, Q; resources by I_a, processes by M, N. If $\widetilde{a}$ is $a_1, \ldots, a_n$ then $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid P)$ abbreviates $\boldsymbol{\nu}a_1\,(I_{a_1} \mid \ldots \boldsymbol{\nu}a_n\,(I_{a_n} \mid P)\ldots)$, and similarly for $\boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}} \mid P)$.

6 Logical relations on processes

We recall the main steps of the technique of logical relations in the λ-calculus:

1. assignment of types to terms;
2. definition of a typed logical predicate on terms, by induction on the structure of types; the base case uses the termination property of interest;
3. proof that the logical terms (i.e., those in the logical predicate) terminate;
4. proof, by structural induction, that all well-typed terms are logical.

For applying logical relations to the π-calculus we follow a similar structure. Some of the details however are rather different. For instance, in the π-calculus an important role is played by a closure property of the logical predicate with respect to bisimilarity, and by the (Sharpened) Replication Theorems. Further, in the λ-calculus typing rules assign types to terms; in the π-calculus, by contrast, types are assigned to names. To start off the technique (step 1), we therefore force an assignment of types to the pre-processes. We use A to range over the types for pre-processes:

$$A ::= \diamond \;\Big|\; b\_\sharp T$$

where T is an ordinary type, as by the grammar in Section 3. If $R, R' \in \mathcal{P}^{-}$ then R' *is a normal form of* R if R' is a normal form and $R \equiv R'$.

Definition 5 (Assignment of types to pre-processes). *A normal form of a (well-typed) pre-process P is either of the form*

- $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid M)$*, or*
- $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid I_b)$*, with* $b \notin \widetilde{a}$*.*

In the first case we write $P : \diamond$, in the latter case we write $P : b\_T$, where T is the type of b. □

Lemma 1. *A pre-process has a unique type.* □

We define the logical predicate $\mathcal{L}^A$ by induction on A.

Definition 6 (Logical relations).

- $P \in \mathcal{L}^{\diamond}$ *if* $P : \diamond$ *and* $P \in \mathrm{TER}$*.*
- $P \in \mathcal{L}^{a\_\sharp\,\mathtt{unit}}$ *if* $P : a\_\sharp\,\mathtt{unit}$*, and for all* $v : \mathtt{unit}$*,*

$$\boldsymbol{\nu}a\,(P \mid \overline{a}v) \in \mathcal{L}^{\diamond}\,.$$

– *$P \in \mathcal{L}^{a\_\sharp T}$, where T is a connection type, if $P : a\_\sharp T$, and, for all b fresh for P and for all $I_b \in \mathcal{L}^{b\_T}$,*

$$\boldsymbol{\nu} b\,(I_b \mid \boldsymbol{\nu} a\,(P \mid \overline{a}b)) \in \mathcal{L}^{\diamondsuit}\,. \tag{2}$$

We write $P \in \mathcal{L}$ if $P \in \mathcal{L}^{A}$, for some A. □

In Definition 6, the most important clause is the last one. The process in (2) is similar to those used for translating function application into π-calculus [SW01]. Therefore a possible reading of (2) is that P is a function and I_b its argument. In (2), P does not know b (because fresh), and I_b does not know a (because restricted). However, P and I_b may have common free names, in output position.

7 Termination of $\mathcal{P}^-$

We first present (Sections 7.1 and 7.2) some general results on the π-calculus. We present them on Aπ_+: this is the π-calculus of Section 2, well-typed, without recursion, with only asynchronous outputs, and with the addition of replication. Here is the grammar of Aπ_+:

$$M ::= a(x).\,M \;\Big|\; \overline{v}w \;\Big|\; M+M \;\Big|\; M \mid M \;\Big|\; !M \;\Big|\; \boldsymbol{\nu}\widetilde{a}\,M$$

where values $v, w, \ldots$ are as in Table 4.

In the second part (Sections 7.3-7.5), we derive the termination of $\mathcal{P}^-$.

7.1 The Replication Theorems

In the proofs with the logical relations we make extensive use of the Sharpened Replication Theorems [SW01]. These express distributivity properties of private replications, and are valid for (strong) barbed congruence. We write $M \downarrow_a$ if $M \xrightarrow{\alpha} M'$ where α is an input or an output along link a.

Definition 7 (Barbed congruence).
A relation $\mathcal{R}$ on closed processes is a barbed bisimulation *if whenever $(M, N) \in \mathcal{R}$,*

1. *$M \downarrow_a$ implies $N \downarrow_a$, for all links a;*
2. *$M \xrightarrow{\tau} M'$ implies $N \xrightarrow{\tau} N'$ for some N' with $(M', N') \in \mathcal{R}$;*
3. *the variants of (1) and (2) with the roles of M and N swapped.*

Two closed processes M and N are barbed bisimilar *if $(M, N) \in \mathcal{R}$ for some barbed bisimulation $\mathcal{R}$.*

Two processes M and N are barbed congruent, *$M \sim N$, if $C[M]$ and $C[N]$ are barbed bisimilar, for every context C such that $C[M]$ and $C[N]$ are closed.* □

Lemma 2. *Relation $\sim$ is a congruence on* Aπ_+. □

Lemma 3 (Sharpened Replication Theorems, for Aπ_+). *Suppose a does not appear free in input position in $M, N, N_1, N_2, \pi.\,N$. We have:*

1. *$\boldsymbol{\nu} a\,(!a(x).\,M \mid !N) \sim\, !\boldsymbol{\nu} a\,(!a(x).\,M \mid N)$;*
2. *$\boldsymbol{\nu} a\,(!a(x).\,M \mid N_1 \mid N_2) \sim \boldsymbol{\nu} a\,(!a(x).\,M \mid N_1) \mid \boldsymbol{\nu} a\,(!a(x).\,M \mid N_2)$;*
3. *$\boldsymbol{\nu} a\,(!a(x).\,M \mid \pi.\,N) \sim \pi.\,\boldsymbol{\nu} a\,(!a(x).\,M \mid N)$, where π is any input or output prefix;*
4. *$\boldsymbol{\nu} a\,(!a(x).\,M \mid (N_1 + N_2)) \sim \boldsymbol{\nu} a\,(!a(x).\,M \mid N_1) + \boldsymbol{\nu} a\,(!a(x).\,M \mid N_2)$.* □

7.2 Wires

A *wire* is a process of the form $!a(x).\,\overline{b}x$. The main result in this section says that, under certain conditions, wires do not affect termination. We write $R \longrightarrow_{\mathsf{E}} R'$ if a transition $R \xrightarrow{\tau} R'$ can be inferred using the transition rules of $\mathrm{A}\pi_{+}$ plus the rule

$$\text{SUM-INT-1: } M_1 + M_2 \xrightarrow{\tau} M_1$$

and its symmetric. We say that R *E-diverges* if R has a divergent computation $R \longrightarrow_{\mathsf{E}} R_1 \longrightarrow_{\mathsf{E}} R_2 \ldots$.

Lemma 4 (in $\mathrm{A}\pi_{+}$). *If R E-diverges, then R diverges.* ☐

Lemma 5 (in $\mathrm{A}\pi_{+}$). *Suppose c' is a name that does not occur free in R in input position, and that R only uses input-guarded replications. If $R \mid !c'(x).\,\overline{c}x$ diverges, then $R\{c/c'\}$ E-diverges.*

Proof. The proof is in two parts. First, one shows that a reductions for $R \mid !c'(x).\,\overline{c}x \xrightarrow{\tau} R' \mid !c'(x).\,\overline{c}x$ can be mimicked by sequence of reductions $R\{c/c'\}(\longrightarrow_{\mathsf{E}})^{\star} \equiv R'\{c/c'\}$. This sequence can however be empty if the reduction from $R \mid !c'(x).\,\overline{c}x$ is a communication at c' (and the output consumed is not part of a sum); the sequence is non-empty otherwise.

In the second part of the proof one shows that in a divergent computation from $R \mid !c'(x).\,\overline{c}x$ there are an infinite number of steps that are not communications at c'. This implies, by the arguments above, that also $R\{c/c'\}$ E-diverges.

Lemma 6 (in $\mathrm{A}\pi_{+}$). *Suppose c' is a name that does not occur free in R in input position, and R only uses input-guarded replication. Then $\boldsymbol{\nu} c'\ (R \mid !c'(x).\,\overline{c}x)$ diverges iff $R\{c/c'\}$ diverges.*

Proof. The hard implication is the one from left to right, and follows from Lemmas 5 and 4, and the fact that a top restriction preserves divergences.

7.3 Closure properties

Lemma 7. *Suppose $R \in \mathcal{P}^{-}$. If $R \equiv\xrightarrow{\alpha} R'$ then $R \xrightarrow{\alpha}\equiv R'$.* ☐

Lemma 8 (Closure under reduction for $\mathcal{P}^{-}$). *$R \in \mathcal{P}^{-}$ and $R \xrightarrow{\tau} R'$ imply $R' \in \mathcal{P}^{-}$.*

Proof. Because of Lemma 7, it is sufficient to prove the result when R is a normal form. Moreover, it suffices to prove it for processes: resources have no reductions; only pre-processes of type $\diamond$ can reduce, and these are also processes. The proof is by structural induction.

Lemma 9 (Closure under $\sim$ for the logical relations). *Suppose $P, Q \in \mathcal{P}^{-}$, and $P \sim Q$. If $P \in \mathcal{L}^{A}$, then also $Q \in \mathcal{L}^{A}$.*

Proof. First, note that if P and Q are barbed congruent, then they must have the same type. Then the thesis, for $A = \diamond$, follows from the fact that $\sim$ preserves termination.

Suppose $A = a\_\sharp\, T$, and T is a connection type (the case of `unit` type is simpler): we have to show that if b is a fresh name, for any $I_b \in \mathcal{L}^{b\_T}$,

$$\boldsymbol{\nu} b\,(I_b \mid \boldsymbol{\nu} a\,(Q \mid \overline{a}b)) \in \mathcal{L}^{\diamond}\,.$$

We can assume that b is fresh also for P, and therefore, since $P \in \mathcal{L}^{a\_\sharp\, T}$,

$$\boldsymbol{\nu} b\,(I_b \mid \boldsymbol{\nu} a\,(P \mid \overline{a}b)) \in \mathcal{L}^{\diamond}\,.$$

But since $\sim$ is a congruence, we are done, using the result of the lemma for $A = \diamond$.

Lemma 10. *If $P \in \mathcal{L}$ then $P \in$ TER.*

Proof. If $P \in \mathcal{L}^{\diamond}$ then the result follows by definition of $\mathcal{L}^{\diamond}$. Otherwise $P \in \mathcal{L}^{a\_T}$, for some a, T, and P cannot reduce.

We write $R \xrightarrow{\tau}_{\mathrm{d}} R'$ if $R \xrightarrow{\tau} R'$ and this is the only possible transition for R (i.e., for all α, R'' such that $R \xrightarrow{\alpha} R''$, we have $\alpha = \tau$ and $R' \equiv R''$).

Lemma 11. *If $R \xrightarrow{\tau}_{\mathrm{d}} R'$ and $R' \in$ TER then also $R \in$ TER.* $\square$

Lemma 12. *If $a, b : T$ then $!a(x).\, \overline{b}x \in \mathcal{L}^{a\_T}$.*

Proof. Suppose $T = \sharp\,$`unit`. Then

$$\boldsymbol{\nu} a\,(!a(x).\, \overline{b}x \mid \overline{a}v) \xrightarrow{\tau}_{\mathrm{d}} \sim \overline{b}v$$

(it terminates after one step), therefore by Lemma 11 and 9 $\boldsymbol{\nu} a\,(!a(x).\, \overline{b}x \mid \overline{a}v) \in$ TER.

Otherwise, take a fresh c and any I_c, and consider the process

$$P \stackrel{\mathrm{def}}{=} \boldsymbol{\nu} c\,(I_c \mid \boldsymbol{\nu} a\,(!a(x).\, \overline{b}x \mid \overline{a}c))$$

We have $P \xrightarrow{\tau}_{\mathrm{d}} \sim \boldsymbol{\nu} c\,(I_c \mid \overline{b}c)$, and the latter process cannot reduce further, therefore $P \in$ TER, reasoning as above.

Lemma 13. *Let c be a higher-order name. We have $\boldsymbol{\nu} b\,(I_b \mid \overline{b}c) \in$ TER iff $(\boldsymbol{\nu} b\,, c')(I_b \mid \overline{b}c' \mid !c'(x).\, \overline{c}x) \in$ TER, where c' is fresh.*

Proof. Follows from Lemma 6.

Lemma 14. *$M_1 \mid M_2 \in$ TER iff, for each i, $M_i \in$ TER.*

Proof. M_1 and M_2 have no free input. This means that M_1 and M_2 cannot interact.

7.4 Relatively independent resources

Definition 8. *Resources $I_{a_1}, \ldots, I_{a_n}$ are* relatively independent *if none of the names $a_1, \ldots, a_n$ appears free in output position in any of the resources $I_{a_1}, \ldots, I_{a_n}$.*

A term $P \in \mathcal{P}^-$ has relatively independent resources *if, for all subterms P' of P, the resources that are unguarded in P' are relatively independent.* □

Lemma 15. *For each $I_a \in \mathcal{RES}$ there is $J_a \in \mathcal{RES}$ with $I_a \sim J_a$ and J_a has relatively independent resources.*

Proof. Induction on the structure of a normal form of I_a, using the Replication Theorems.

Lemma 16. *For each $P \in \mathcal{P}^-$ there is $Q \in \mathcal{P}^-$ with $P \sim Q$ and Q has relatively independent resources.*

Proof. Similar to the previous lemma.

Lemma 17. *For each $P \in \mathcal{P}^-$ there is a normal form $Q \in \mathcal{P}^-$ with $P \sim Q$ and Q has relatively independent resources.*

Proof. If in the previous lemmas the initial process is in normal form, then also the transformed process is. The result then follows from the fact that $\equiv \subseteq \sim$.

7.5 Main theorem

Theorem 2. *Let $\widetilde{a} = a_1, \ldots, a_n$. Suppose that resources $I_{a_1}, \ldots, I_{a_n}$ are relatively independent, and $I_{a_i} \in \mathcal{L}$ for each i. Then $P : A$ and $\mathtt{in}(P) \cap \mathsf{fn}(I_{\widetilde{a}}) = \emptyset$ imply $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid P) \in \mathcal{L}^A$.*

Proof. By Lemmas 17 and 9 we can assume that P is a normal form and has relatively independent resources. We proceed by induction on the structure of P. We call Q the process $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid P)$. We consider the most interesting cases.

- $P = \overline{b}c$.
  In this case, $A = \diamond$. We have to show that $Q \in \mathrm{TER}$. We have:

$$Q = \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid P) \sim \boldsymbol{\nu}\widetilde{a'}\,(I_{\widetilde{a'}} \mid P) \stackrel{\mathrm{def}}{=} Q'$$

  where $\widetilde{a'} = \widetilde{a} \cap \{b, c\}$ (here we exploit the fact that the resources are relatively independent).
  There are 4 subcases:
  - $\widetilde{a'} = \emptyset$. Then $Q' \sim \overline{b}c$, which is in TER.
  - $\widetilde{a'} = \{b\}$. Then $Q' \sim \boldsymbol{\nu}b\,(I_b \mid \overline{b}c)$ and the latter process is in TER iff the process $(\boldsymbol{\nu}b\,, c')(I_b \mid !c'(x).\,\overline{c}x \mid \overline{b}c')$ is in TER (Lemma 13), where c' is fresh. And now we are done, exploiting the definition of $\mathcal{L}$ on the type of b, for $!c'(x).\,\overline{c}x \in \mathcal{L}$ (Lemma 12) and we know that $I_b \in \mathcal{L}$.
  - $\widetilde{a'} = \{c\}$. Then $Q' \sim \boldsymbol{\nu}c(I_c \mid \overline{b}c)$ and the latter process is in TER because cannot reduce.

- $\widetilde{a'} = \{b, c\}$. Then

  $$\begin{aligned} Q' &\sim (\boldsymbol{\nu} b, c\,)(I_b \mid I_c \mid \overline{b}c) \\ &\sim \boldsymbol{\nu} c\,(I_c \mid \boldsymbol{\nu} b\,(I_b \mid \overline{b}c)) \stackrel{\text{def}}{=} Q'' \end{aligned}$$

  since c is fresh for I_b (the relatively-independence hypothesis). Now, Q'' is in TER by definition of $\mathcal{L}$ on higher types (precisely, the type of b).

– $P = M_1 \mid M_2$. (Thus $P : \diamond$)
  We have to show that $Q = \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid M_1 \mid M_2) \in$ TER. Using the Replication Theorems we have

  $$Q \sim Q_1 \mid Q_2$$

  where

  $$Q_i \stackrel{\text{def}}{=} \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid M_i)\,.$$

  By Lemma 14, $Q \in$ TER iff each Q_i is so. The latter is true by the induction on the structure.

– $P = !b(x).\,M$. Then $Q : b\_\sharp\, T$. We assume that T is a connection type. Then $Q \in \mathcal{L}$ if for a fresh c, for any $I_c \in \mathcal{L}^{c\_T}$,

  $$R \stackrel{\text{def}}{=} \boldsymbol{\nu} c\,(I_c \mid \boldsymbol{\nu} b\,(Q \mid \overline{b}c)) \in \text{TER}\,.$$

  Since c is fresh, we can assume $c = x$. We thus have

  $$\begin{aligned} R &= \boldsymbol{\nu} x\,(I_x \mid \boldsymbol{\nu} b\,(\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid !b(x).\,M) \mid \overline{b}x)) \\ &\equiv (\boldsymbol{\nu} x, b, \widetilde{a}\,)(I_x \mid I_{\widetilde{a}} \mid !b(x).\,M \mid \overline{b}x) \end{aligned}$$

  and

  $$R \xrightarrow{\tau}_{\text{d}} \sim (\boldsymbol{\nu} x, \widetilde{a}\,)(I_x \mid I_{\widetilde{a}} \mid M) \stackrel{\text{def}}{=} R'$$

  (here we exploit the fact that replications are not recursive, and that $b \notin \mathsf{fn}(I_{\widetilde{a}})$). Using the induction on the structure we derive $R' \in$ TER (note that $I_x, I_{\widetilde{a}}$ are relatively independent).

– $P = \boldsymbol{\nu} x\,(I_x \mid P')$, where $I_x = !x(y).\,M$.
  We have, using the Replication Theorems:

  $$\begin{aligned} Q &= \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid \boldsymbol{\nu} x\,(I_x \mid P')) \\ &\sim \boldsymbol{\nu} x\,(\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid I_x) \mid \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid P')) \stackrel{\text{def}}{=} Q' \end{aligned}$$

  Call $R \stackrel{\text{def}}{=} \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid I_x)$. Then $R \in \mathcal{L}$ using the inductive hypothesis. Moreover, using the Replication Theorems,

  $$R \sim !x(y).\,\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid M) \stackrel{\text{def}}{=} J_x$$

  and $J_x \in \mathcal{L}$, using Lemma 9. Thus we have

  $$Q' \sim (\boldsymbol{\nu} x, \widetilde{a}\,)(J_x \mid I_{\widetilde{a}} \mid P') \stackrel{\text{def}}{=} Q''$$

  where $J_x, I_{\widetilde{a}}$ are relatively independent. Finally we can apply the induction hypothesis to P' and infer $Q'' \in$ TER.

Corollary 1. *If $P \in \mathcal{P}^-$ then $P \in \mathcal{L}$.* □

8 Proofs based on simulation

The language $\mathcal{P}^-$ is non-trivial, but not very expressive. It is however a powerful language for the termination property, in the sense that the termination of the processes in $\mathcal{P}^-$ implies that of a much broader language, namely the language $\mathcal{P}$ of Theorem 1.

We move to $\mathcal{P}$ by progressively extending the language $\mathcal{P}^-$. The technique for proving termination of the extensions is as follows. The extensions define a sequence of languages $\mathcal{P}_0, \ldots, \mathcal{P}_{11}$, with $\mathcal{P}_0 = \mathcal{P}^-$, $\mathcal{P}_{11} = \mathcal{P}$, and $\mathcal{P}_i \subset \mathcal{P}_{i+1}$ for all $0 \leq i < 11$. For each i, we exhibit a transformation $[\![\cdot]\!]_i$, defined on the normal forms of $\mathcal{P}_{i+1}$, with the property that a transformed process $[\![M]\!]_i$ belongs to $\mathcal{P}_i$ and $[\![M]\!] \in \mathrm{TER}$ implies $M \in \mathrm{TER}$. We then infer the termination of the processes in $\mathcal{P}_{i+1}$ from that of the processes in $\mathcal{P}_i$.

For this kind of proofs we use process calculus techniques, especially techniques for *simulation*. If R' simulates R, then R' can do everything R can, but the other way round may not be true. For instance, $a.\,(b.\,c+d)$ simulates $a.\,b$, but the other way round is false. Simulation is not much interesting as a behavioural equivalence. Simulation is however interesting for reasoning about termination, and is handy to use because of its co-inductive definition.

Definition 9. *A relation $\mathcal{R}$ on closed processes is a* strong simulation *if $(M, N) \in \mathcal{R}$ implies:*

- *whenever $M \xrightarrow{\alpha} M'$ there is N' such that $N \xrightarrow{\alpha} N'$ and $(M', N') \in \mathcal{R}$.*

A process N simulates *M, written $M \preceq N$, if for all closing substitutions σ there is a strong simulation $\mathcal{R}$ such that $(M\sigma, N\sigma) \in \mathcal{R}$.* □

Relation $\preceq$ is reflexive and transitive, and is preserved by all operators of the π-calculus. Hence $\preceq$ is a precongruence in all languages $\mathcal{P}_i$. An important property of $\preceq$ for us is:

Lemma 18. *If $M \preceq M'$ then $M' \in \mathrm{TER}$ implies $M \in \mathrm{TER}$.* □

Each language $\mathcal{P}_i$ $(i > 0)$ is defined by exhibiting the additional productions for the normal forms of the processes and the resources of the language. We only show a few examples of extensions and their correctness proofs. We refer to [San01] for the details.

$$M^{\mathrm{NF}} ::= \ldots \;\Big|\; \overline{a}v.\, M^{\mathrm{NF}}$$

Proof. Consider a transformation $[\![\cdot]\!]$ that acts on outputs thus:

$$[\![\overline{b}v.\, M]\!] \stackrel{\mathrm{def}}{=} \overline{b}v \mid [\![M]\!]\,.$$

and is an homomorphism elsewhere. Its correctness is given by the law

$$\overline{b}v.\, M \preceq \overline{b}v \mid M$$

and Lemma 18. □

$$I_a^{\mathrm{NF}} ::= \ldots \;\Big|\; I_a^{\mathrm{NF}} \mid I_a^{\mathrm{NF}}$$

Proof. With the addition of this production, there can be several input-replicated processes at the same link. The correctness of this extension is inferred using the law

$$!a(x).\, R \mid !a(x).\, R' \preceq !a(x).\, (R \mid R')\,.$$

$\square$

$$I_{\tilde{a}}^{\mathrm{NF}} ::= \ldots \;\Big|\; I_{\tilde{a}}^{\mathrm{NF}} + I_{\tilde{a}}^{\mathrm{NF}}$$

Proof. We use the law $R_1 + R_2 \preceq R_1 \mid R_2$. $\square$

The addition of nested inputs and of recursion with first-order state are more difficult, though the basis of the proof is the same. Again, we refer to [San01]. Also, we refer to [San01] for more discussions on the importance of the clauses (2) and (3) of Condition 1 (on nesting of inputs and on state) for the termination of the processes of $\mathcal{P}$.

9 Conclusions and future work

We have proved termination of $\mathcal{P}$, the simply-typed (localised) π-calculus subject to three syntactic conditions that constrain recursive inputs and state. In the proof, we have first applied the logical-relation technique to a subset of processes with only functional names, and then we have extended the termination property to the whole language by means of techniques of behavioural preorders.

The termination of $\mathcal{P}$ implies that of various forms of simply-typed λ-calculus: usual call-by-name, call-by-value, and call-by-need, but also enriched λ-calculi such as concurrent λ-calculi [DCdLP94], λ-calculus with resources and λ-calculus with multiplicities [BL00]. Indeed all encodings of λ-calculi into π-calculus we are aware of, restricted to simply-typed terms, are also encodings into $\mathcal{P}$. The λ-calculus with resources, λ^{res}, is a form of non-deterministic λ-calculus with explicit substitutions and with a parallel composition. Substitutions have a multiplicity, telling us how many copies of a given resource can be made. Due to non-determinism, parallelism, and the multiplicity in substitutions (which implies that a substitution cannot be distributed over a composite term), a direct proof of termination of λ^{res}, with the technique of logical relations, although probably possible, is non-trivial.

The main focus of this paper have been the proof techniques of termination for processes. For future work, an important issue to look at is the expressiveness of the language $\mathcal{P}$ obtained. At present we know little about it, other than $\mathcal{P}$ can encode various λ-calculus dialects. We are particularly interested in applications to parallel and distributed object-oriented languages. This will probably require extending $\mathcal{P}$ with other features, such as primitives for distribution and migration.

We have only considered the simply-typed π-calculus – the process analogous of the simply-typed λ-calculus. It should be possible to adapt our work to more complex types, such as those of the polymorphic π-calculus [Tur96,SW01] – the process analogous of the polymorphic λ-calculus.

We have been able to apply the logical-relation technique only to a small set of 'functional' processes. Then we have had to use ad hoc techniques to prove the termination of a larger language. To obtain stronger results, and to extend the results more easily to other languages (for instance, process languages with communication of terms such as the Higher-Order π-calculus) a deeper understanding of the logical-relation technique in concurrency would seem necessary.

References

[BL00] G. Boudol and C. Laneve. λ-calculus, multiplicities and the π-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[DCdLP94] M. Dezani-Ciancaglini, U. de Liguoro, and U. Piperno. Fully abstract semantics for concurrent λ-calculus. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 16–35. Springer Verlag, 1994.

[FG96] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join calculus. In *Proc. 23th POPL*. ACM Press, 1996.

[GLT88] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1988.

[HKM+99] M. Hicks, P. Kakkar, J.T. Moore, C.A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Conf. on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 86–93. ACM, June 1999.

[HMST95] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 1995.

[Kob00] N. Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *IFIP Conf. TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*, pages 365–389. IFIP, Springer Verlag, August 2000.

[Mer01] M. Merro. Locality in the π-calculus and applications to object-oriented languages. PhD thesis, Ecoles des Mines de Paris, 2001.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

[San99] D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221:457–493, 1999.

[San01] D. Sangiorgi. Termination of processes. ftp://ftp-sop.inria.fr/mimosa/personnel/davides/ter.ps, December 2001.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[Tur96] N.D. Turner. *The polymorphic pi-calculus: Theory and Implementation.* PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[YBK01] N. Yoshida, M. Berger, and Honda. K. Strong normalisation in the π-Calculus. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS-01)*, pages 311–322. IEEE Computer Society, 2001.

[Yos96] N. Yoshida. Graph types for monadic mobile processes. In *Proc. FST & TCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer Verlag, 1996. Full paper appeared as Technical Report, ECS-LFCS-96-350, 1996, Edinburgh.