Mobile processes and termination[*]

Romain Demangeon[1], Daniel Hirschkoff[1], and Davide Sangiorgi[2]

[1] ENS Lyon, Université de Lyon, CNRS, INRIA, France
[2] INRIA and Università di Bologna, Italy

Abstract. This paper surveys some recent works on the study of termination in a concurrent setting. Processes are π-calculus processes, on which type systems are imposed that ensure termination of the process computations. Two approaches are exposed. The first one draws on the method of logical relations, which has been extensively used in the analysis of sequential languages. The second approach exploits notions from term rewriting.

1 Introduction

It is a pleasure and a honour to write a paper in this volume dedicated to Peter D. Mosses. Peter has been a strong promoter of the semantics of programming languages (e.g., [Mos01,Mos04,Mos06]), in particular *structured semantics*. A carefully devised semantics makes the meaning of the language constructs clear and unambiguous, and it can be used to prove fundamental behavioural properties. In this paper we make use of structured operational semantics to prove properties of *termination* in the computations originated by systems of processes. We thus review a strand of work that we have pursued during the past few years whose goal is precisely termination in concurrency.

A term terminates if all its reduction sequences are of finite length. As far as programming languages are concerned, termination means that computation in programs will eventually stop. In computer science, termination has been extensively investigated in term rewriting systems [DM79,DH95] and λ-calculi [Gan80,Mit96], where strong normalization is a more commonly used synonym.

Termination is however interesting also in concurrency. For instance, if we interrogate a process, we may want to know that an answer is eventually produced (termination alone does not guarantee this since other security properties, e.g., deadlock-freedom [Kob98], are also involved, but termination would be the main ingredient in a proof). Similarly, when we load an applet we would 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 eventually accept interactions with the environment.

We study termination in the setting of the π-calculus. This is a very expressive formalism. A number of programming language features can be encoded, including functions, objects, and state (in the sense of imperative languages) [SW01a]. As a consequence, however, the notoriously-hard problems of termination for these features hit the π-calculus too. Concurrency, then, adds a further dimension of complexity.

Two main forms of type techniques have been used to handle termination in the π-calculus. The first form makes use of logical relations, and appears in, e.g., [YBK01,San06]. Logical relations are well-known in functional languages and are used, in particular, to prove termination of typed λ-calculi. The second form of techniques appears in, e.g., [DS06b,DHKS07,DHS08], and borrows ideas from *term rewriting systems*. Roughly, termination is ensured by identifying a measure which decreases after finite steps of reduction.

The logical relation techniques allow one to treat processes that have a functional flavour. Intuitively, a channel is *functional* if it appears only once in input, and the input is replicated. In other words, the service offered by the channel is always available and does not change over time.

We show how to apply the logical relation technique to a small sublanguage of the π-calculus, $\mathcal{P}^-$. This is a non-deterministic language, with only asynchronous outputs, and in which all names are functional. One of the reasons for having to restrict 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 $\mathcal{P}^-$ itself is not very expressive. It is however a powerful language for the termination property. We then show that the termination of $\mathcal{P}^-$ implies that of a larger language, $\mathcal{P}$. For this, we use process calculus techniques, most notably techniques for behavioural preorders.

It remains an open problem how to directly apply logical relations to process languages broader than "functional" languages akin to $\mathcal{P}^-$.

On the other hand, with term-rewriting techniques we fail to capture non-trivial functional behaviours. For instance, it appears difficult to type the processes encoding the simply-typed λ-calculus — we discuss this briefly in Section 5.2. However the term-rewriting techniques allow us to handle sophisticated stateful processes.

The paper is thus made of two parts. The first, describing the logical relation techniques, is in Section 4. The proofs are only sketched. For more details, we refer to [San06], where, moreover, the language of processes is richer. For instance, here the only first-order value is unit, whereas in [San06] arbitrary first-order values are allowed.

The second part of the paper, dealing with the term-rewriting techniques, is treated in Section 5. This part is more brief: we only discuss the basic ideas, referring to the literature for the various extensions and enhancements that have been studied.

2 The π-calculus

To describe our contributions, we will work using the syntax given in Table 1, which is based on the process constructs of the standard (monadic) π-calculus [Mil99,SW01a].

$$a, b, c, d, \ldots, x, y, z \ldots \textit{Names}$$

		Values
v, w	$::= a$	name
	$\mid \quad \star$	unit value

		Processes
M, N	$::= \mathbf{0}$	nil process
	$\mid \quad H\lfloor \widetilde{v} \rfloor$	recursion call
	$\mid \quad a(x). M$	input
	$\mid \quad \overline{v}w. M$	output
	$\mid \quad M \mid M$	parallel
	$\mid \quad M + M$	sum
	$\mid \quad \boldsymbol{\nu}\widetilde{a}\, M$	restriction
H	$::= X$	recursion variable
	$\mid \quad \mu X(\widetilde{x}). M$	recursive definition

Table 1. Syntax of processes

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. In particular, 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 to manipulate 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, written $\star$.

The monadic π-calculus serves as a basis for the type systems we shall present. We will sometimes adopt variants or restrictions of it, that we now briefly discuss.

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 versions 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.

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

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

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

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

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

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

Table 2. Transition rules

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 $M \in \mathrm{TER}$, if M is not divergent.

The localised π-calculus.

For the study based on logical relations, we shall work in a *localised* calculus [Mer01], in which 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 this part of the paper, where we explore methods based on logical relations, locality is essential in the results involving logical relations: most of our results rely on it.

Other process operators.

Below in the paper, we shall also consider other 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 represents an infinite parallel composition of copies of M (replication will in particular be useful for the definition of the type systems discussed in Section 5). Their transition rules are:

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

In the paper, we will often move interchangeably between recursion and the *replication* construct, as they have the same expressiveness [SW01a].

3   The simply-typed calculus

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

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

where the *connection type* $\sharp T$ is the type of a link that carries tuples of values of type T, and $\mathtt{unit}$ 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 $\mathtt{unit}$, whereas a first-order link has a type $\sharp\,\mathtt{unit}$.

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 (asynchronous output, replication) are similar to those of (standard) output and parallel composition.

$$\text{T-Par}: \quad \frac{\vdash M \qquad \vdash N}{\vdash M \mid N} \qquad\qquad \text{T-Sum}: \quad \frac{\vdash M \qquad \vdash N}{\vdash M + N}$$

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

$$\text{T-Rvar}: \quad \frac{X \in \langle \widetilde{T} \rangle \qquad \vdash \widetilde{v} : \widetilde{T}}{\vdash X \lfloor \widetilde{v} \rfloor}$$

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

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

$$\text{T-Nil}: \quad \vdash \mathbf{0} \qquad\qquad \text{T-Unit}: \quad \vdash \star : \texttt{unit} \qquad\qquad \text{T-Var}: \quad \frac{x \in T}{\vdash x : T}$$

Table 3. Typing rules

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).

The main point in simple types for the π-calculus is to guarantee that in all executions of a typable process, the type of values transmitted along a channel agrees with the type assigned to the channel. Simple types do not tell anything about termination of processes. The type systems we present in this paper will however be built on top of the type system for simple types.

4 Terminating processes, via logical relations

In this section, we discuss how to isolate a subcalculus of the *localised* π-calculus in which all processes terminate by exploiting logical relations.

In sequential higher-order languages, it is well-known that termination may be broken by features such as self-applications, recursion, higher-order state. Our language of terminating processes is defined by four constraints. Three of them, mostly syntactic, are reported in Condition 2. The first condition is for recursive inputs; the second and third conditions control state. The last constraint – condition 1 of Definition 3 – 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 2 (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 2 to mean that M itself and all its process subterms respect the constraints.

Definition 3 (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 2, where $\widetilde{a} = \mathsf{fn}(M)$.*

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

The proof of this theorem is in two parts; the first one is sketched in Sections 4.1-4.3, the second one in Section 4.4. Throughout these sections, all processes and values are well typed, and substitutions map names onto values of the same type. Moreover, processes have no free recursion variables.

4.1 $\mathcal{P}^-$: 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 (Section 4.4).

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 4 (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)*

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

(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 4) in later proofs.

$$\textit{Pre-processes}$$

$$P^{\mathrm{NF}} ::= \boldsymbol{\nu}a\,(I_a^{\mathrm{NF}} \mid P^{\mathrm{NF}}) \quad \text{with } \mathsf{fn}(I_a^{\mathrm{NF}}) \cap \mathtt{in}(P^{\mathrm{NF}}) = \emptyset$$

$$\Big| \quad M^{\mathrm{NF}}$$

$$\Big| \quad I_a^{\mathrm{NF}}$$

$$\textit{Resources}$$

$$I_a^{\mathrm{NF}} ::= \,!a(x).\,M^{\mathrm{NF}} \qquad \text{with } a \notin \mathsf{fn}(M^{\mathrm{NF}})$$

$$\textit{Processes}$$

$$M^{\mathrm{NF}} ::= \boldsymbol{\nu}a\,(I_a^{\mathrm{NF}} \mid M^{\mathrm{NF}})$$

$$\Big| \quad M^{\mathrm{NF}} + M^{\mathrm{NF}}$$

$$\Big| \quad M^{\mathrm{NF}} \mid M^{\mathrm{NF}}$$

$$\Big| \quad \bar{v}w$$

$$\Big| \quad \mathbf{0}$$

$$\textit{Values}$$

$$v ::= a \qquad \qquad \text{name}$$

$$\Big| \quad \star \qquad \qquad \text{unit value}$$

Table 4. The normal forms

Definition 5 (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.*

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 $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}}^{\mathrm{NF}} \mid P)$.

4.2 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 6 (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.

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

Definition 7 (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 $I_b \in \mathcal{L}^{b\_T}$,

$$\boldsymbol{\nu} b \left(I_b \mid \boldsymbol{\nu} a \left(P \mid \overline{a} b \right) \right) \in \mathcal{L}^{\diamond} . \tag{1}$$

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

In Definition 7, the most important clause is the last one. The process in (1) is similar to those used for translating function application into the π-calculus [SW01a]. Therefore a possible reading of (1) is that P is a function and I_b its argument. In (1), P does not know b (because it is fresh), and I_b does not know a (because it is restricted). However, P and I_b may have common free names, in output position.

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

Before presenting the termination proof for $\mathcal{P}^-$, we present some general results on the π-calculus. We formulate them on $A\pi_+$: this is the π-calculus of Table 1, well-typed, without recursion, with only asynchronous outputs, and with the addition of replication. Here is the grammar of $A\pi_+$:

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

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

The Replication Theorems The proofs with the logical relations make extensive use of the Sharpened Replication Theorems [SW01a]. 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 8 (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 1. *Relation $\sim$ is a congruence on $A\pi_+$.*

Lemma 2 (Sharpened Replication Theorems, for $A\pi_+$). *Suppose a does not appear free in input position in $M, N, N_1, N_2, \pi.\, N$. We have:*

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

A *wire* is a process of the form $!a(x).\,\bar{b}x$. The lemma in this section says that, under certain conditions, wires do not affect termination.

Lemma 3 (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 $\nu c' \, (R \mid !c'(x).\,\bar{c}x)$ diverges iff $R\{c/c'\}$ diverges.*

Closure properties We also need a few closure properties; to begin with, closure of the class $\mathcal{P}^-$ of processes under reduction; then a closure of logical relations under $\sim$, closure of terminating processes under deterministic reduction, and a few properties of the logical predicates.

Lemma 4 (Closure under reduction for $\mathcal{P}^-$). $R \in \mathcal{P}^-$ *and* $R \xrightarrow{\tau} R'$ *imply* $R' \in \mathcal{P}^-$.

Lemma 5 (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 $Q \in \mathcal{L}^A$.*

Lemma 6. *If $P \in \mathcal{L}$ then $P \in \mathrm{TER}$.*

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. $\square$

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

Lemma 8. *If $a, b : T$ then $!a(x).\,\bar{b}x \in \mathcal{L}^{a-T}$.*

Proof. Suppose $T = \sharp\,\mathtt{unit}$. Then

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

(it terminates after one step), therefore by Lemma 7 and 5 $\nu a \, (!a(x).\,\bar{b}x \mid \bar{a}v) \in \mathrm{TER}$.

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

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

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

Lemma 9. *Let c be a higher-order name. We have $\nu b \, (I_b \mid \bar{b}c) \in \mathrm{TER}$ iff $(\nu b, c')(I_b \mid \bar{b}c' \mid !c'(x).\,\bar{c}x) \in \mathrm{TER}$, where c' is fresh.*

Proof. Follows from Lemma 3. $\square$

Relatively independent resources The final ingredient that we need for the main theorem (both its assertion and its proof) is that of relatively independent resources, roughly indicating a bunch of replicated processes without references to each other. We also need some lemmas that allow us to transform processes in the language so to have relatively independent resources.

Definition 9. *Resources* $I_{a_1}, \ldots, I_{a_n}$ *are said to be* 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 10. *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. $\square$

Lemma 11. *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. $\square$

Lemma 12. *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 $\equiv \subseteq \sim$. $\square$

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* $\mathtt{in}(P) \cap \mathtt{fn}(I_{\widetilde{a}}) = \emptyset$ *imply* $\boldsymbol{\nu}\widetilde{a} \, (I_{\widetilde{a}} \mid P) \in \mathcal{L}^A$.

Proof. By Lemmas 12 and 5 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 only consider two 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 $Q' \sim \boldsymbol{\nu}b\,(I_b \mid \bar{b}c)$ and the latter process is in TER iff the process $(\boldsymbol{\nu}b\,,c')(I_b \mid !c'(x).\,\overline{c}x \mid \bar{b}c')$ is in TER (Lemma 9), 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 8) and we know that $I_b \in \mathcal{L}$.
- $\widetilde{a'} = \{c\}$. Then $Q' \sim \boldsymbol{\nu}c\,(I_c \mid \bar{b}c)$ and the latter process is in TER because cannot reduce.
- $\widetilde{a'} = \{b, c\}$. Then

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

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 $Q = \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid M_1 \mid M_2) \in \text{TER}$. Using the Replication Theorems we have

  $$
  Q \sim Q_1 \mid Q_2
  $$

  where

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

  Now, $Q \in \text{TER}$ iff each Q_i is so. The latter is true by the induction on the structure. $\qquad\square$

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

4.4 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 $0 \le i < 11$. 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 $[\![M]\!] \in \text{TER}$ implies $M \in \text{TER}$. 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 10. *We say that 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 13. *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 [San06] for the details.

$$M^{\mathrm{NF}} ::= \dots \ \Big| \ \bar{a}v.\, M^{\mathrm{NF}}$$

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

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

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

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

and Lemma 13. □

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

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

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

□

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

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

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 [San06]. Also, we refer to [San06] for more discussions on the importance of the clauses (2) and (3) of Condition 2 (on nesting of inputs and on state) for the termination of the processes of $\mathcal{P}$.

5 Methods based on Rewriting Theory

In this section, we present a series of approaches to termination in the π-calculus based on rewriting techniques. In these works, a compositional type system is defined on top of processes, and the soundness of this analysis (that is, that every typable process terminates) is justified by exhibiting a well-founded order that decreases along reductions. The definition of this order typically exploits constructions like the lexicographic composition of two orders, or the multiset extension of an order.

We present type systems of increasing expressiveness, allowing one to type-check more and more processes. By essence, these type systems make it possible to reason about processes that exhibit non-trivial stateful structures, which is not the case for the analysis presented in the previous sections. On the other hand, it is not clear how to adapt these types to to handle higher-order functions. So the two approaches (the one based on logical relations and the one based on term-rewriting) seem incomparable in terms of expressiveness.

We start by presenting the general approach in its simplest form. We then turn to refinements of the initial type system. We also review results about the complexity of type inference in these systems.

5.1 A basic type system

We begin with a very basic type system, that we hope conveys well the intuition about the term-rewriting approach. We call this system S1 (it corresponds to the first type system of [DS06a]).

The calculus we work with is the one of Table 1, but we find it convenient to have replicated inputs in place of recursion (see Section 2). Also, the constraint on locality (the obligation on the recipient of a name to use it in output only), which was necessary for logical relations, is not needed now, and can therefore be removed.

The type system extends the system of simple types for the π-calculus (cf. Table 3). Channels that are used according to type $\sharp\,\texttt{unit}$ are written as CCS channels (we write $a.\,M$ instead of $a().\,M$ and $\overline{a}.\,M$ instead of $\overline{a}\star.\,M$). Typing judgements for values are of the form $\vdash\ a : \sharp^k T$ where k is the *level* of a (we write in this case $\texttt{lvl}(a) = k$). Typing judgements for processes are of the form $\vdash M : m$, where M is a process and m is a natural number (the *weight* associated to M).

The typing rules for system S1 are presented on Figure 5.1. (The presentation we give is not exactly the same as in [DS06a]. The present reformulation is equivalent, and we hope more clear.)

The type system controls divergences as follows. The weight associated to a process M is the maximum level of a channel which is used to emit a message in M, where outputs occurring under an operator of replication are ignored. For a replication to be typable, the level of the channel on which the replicated input occurs must be strictly greater than the level of all channels that are used in output in the continuation processes (again, outputs occurring under an inner

replication are not taken into account), that is, this level must dominate the weight associated to the process (rule (**Rep1**)).

$$(\textbf{Nil}) \ \frac{}{\vdash M : 0} \qquad\qquad (\textbf{Res}) \ \frac{\vdash M : m \qquad a \in T}{\vdash (\nu a) \ M : m}$$

$$(\textbf{Par}) \ \frac{\vdash M : m \qquad \vdash N : n}{\vdash M \ | \ N : max(m, n)} \qquad (\textbf{In}) \ \frac{\vdash a : \sharp^{k} T \qquad x \in T \qquad \vdash M : m}{\vdash \Gamma : a(x). \, Mm}$$

$$(\textbf{Out}) \ \frac{\vdash M : m \qquad \vdash a : \sharp^{k} T \qquad \vdash w : T}{\vdash \overline{a}w. \, M : max(m, k)}$$

$$(\textbf{Rep1}) \ \frac{\vdash M : m \qquad \vdash a : \sharp^{k} T \qquad x \in T \qquad k > m}{\vdash \, !a(x). \, M : 0}$$

Fig. 1. Weight-based type system for termination

Consider the following examples of diverging processes:

$$M_1 \overset{\text{def}}{=} \, !a. \overline{a} \ | \ \overline{a} \qquad M_2 \overset{\text{def}}{=} \, !a. \overline{b} \ | \ !b. \overline{a} \ | \ \overline{a} \qquad M_3 \overset{\text{def}}{=} c(x). !a. \overline{x} \ | \ \overline{c}a \ | \ \overline{a}$$

They are ruled out by the typing rule for replication. If we write $a \in \sharp^{k_a}\texttt{unit}$, $b \in \sharp^{k_b}\texttt{unit}$ and $c \in \sharp^{k_c}(\sharp^{k_x}\texttt{unit})$, then type-checking the replication in M_1 imposes $k_a > k_a$; type-checking the replication in M_2 imposes $k_a > k_b > k_a$; finally, type-checking the replication in M_3 imposes $k_x > k_x$, as the typing rules for input and output have the effect of unifying k_a and k_x.

As announced, a typable process does not exhibit a diverging computation. This is expressed by the following result, that in turn relies on a subject reduction property of the type system.

Theorem 3 (Soundness).
If $\vdash M : m$ for some m, then M terminates.

Proof (Sketch). The main idea is to define a measure on processes that decreases along reductions. A communication involving a non replicated input prefix makes the process shrink. The interesting case is when a replicated input is triggered:

$$\overline{a}v. \, M \ | \ !a(x). \, N \ \overset{\tau}{\longrightarrow} \ M \ | \ N\{v/x\} \ | \ !a(x). \, N \ .$$

In this case, by typability (rule (**Rep1**)), all outputs released by the consumption of the $a(x)$ prefix (the outputs in $N\{v/x\}$) are at a level strictly smaller than the level of a.

We hence define, as a measure on processes, the multiset of levels of names used in output (outputs occurring under a replication are ignored). This measure

decreases strictly (for the multiset extension of the order on natural numbers) along every reduction of a typable process. Soundness is then proved by contradiction: an infinite sequence of reductions emanating from a typable process would induce an infinitely strictly decreasing sequence of multisets of natural numbers, which is impossible.

5.2 Enhancements of the basic type system

Recursion A process like $M_4 \stackrel{\mathrm{def}}{=} \, !a.\,b.\,\overline{a}$ cannot be typed using the rules of Figure 5.1, because applying the rule for replication imposes $k_a > k_a$, if k_a is the level associated to channel a. In this process, the output on a is seen as a 'recursive call': triggering the replicated process located at a might lead to an emission on a itself. However, this particular recursive call is innocuous, as an output on a *and* an output on b have to be consumed in order to produce a single output on a.

In order to be able to recognise some processes that exploit recursion as terminating, a further type system, called S2 here, is presented in [DS06a]. The basic idea is to keep the analysis we have presented above, but to treat *sequences of input prefixes* as a whole. For this, we manipulate typing judgements for processes of the form $\vdash M : \mathcal{M}$, where $\mathcal{M}$ is a multiset of natural numbers (we use $>_{mul}$ to denote the multiset extension of the standard ordering on natural numbers). Accordingly, the typing rules for parallel composition and for replicated inputs in system S2 are as follows ($\uplus$ stands for multiset sum):

$$(\mathbf{Par2}) \; \frac{\vdash M : \mathcal{M}_1 \qquad \vdash N : \mathcal{M}_2}{\vdash M \mid N : \mathcal{M}_1 \uplus \mathcal{M}_2}$$

$$(\mathbf{Rep2}) \; \frac{\vdash M : \mathcal{M} \qquad \vdash a_1 : \sharp^{k_1} T_1 \qquad \ldots \qquad \vdash a_q : \sharp^{k_q} T_q \\ x_1 \in T_1 \qquad \ldots \qquad x_q \in T_q \qquad \{k_1, \ldots, k_q\} >_{mul} \mathcal{M}}{\vdash \, !a_1(x_1)\ldots a_q(x_q).\,M : \emptyset}$$

Notice how in rule (**Rep2**), the sequence of input prefixes $a_1(x_1)\ldots a_q(x_q)$ is treated as a whole. In particular, process M_4 can be type-checked: if we write $\mathtt{lvl}(b) = k_b$, rule (**Rep2**) requires $\{k_a, k_b\} >_{mul} \{k_a\}$.

The soundness proof for system S2 is an adaptation of the one for the simpler type system seen above. There is essentially one additional technical difficulty: along the execution of a process, we need to reason about partially consumed sequences of input prefixes.

Recursive Data Structures The limits of system S2 in terms of expressiveness appear when trying to type-check complex processes that mimic the behaviour of list-like or tree-like data structures.

More precisely, a process like

$$M_5 \stackrel{\mathrm{def}}{=} \, !p(x,y).\,x.\,(\overline{y} \mid \overline{p}\langle x,y\rangle)$$

can typically arise in the encoding of the traversal of a list structure (for instance, the list can be used to implement a symbol table, in which data are stored and searched using concurrent accesses). We adopt here a *polyadic* version of the π-calculus, where tuples of names can be transmitted along channels: this represents a mild extension, and the type systems described above can be easily adapted to handle polyadicity.

In M_5, channel p can be seen as a node constructor, x as a node of the list and y as its successor. To represent a node a in the list having node b as successor, we trigger the replication by inserting process $\overline{p}\langle a, b\rangle$. This has the effect of spawning an instance of the continuation, of the form $a.\,(\overline{b} \mid \overline{p}\langle a, b\rangle)$. This process intuitively trades a request on a for a request on b and reconstructs the node with an output on p, which corresponds to (a simplified representation of) the way we model in the π-calculus an access at a which is propagated to b.

The encoding of lists in the π-calculus moreover imposes that names x and y in M_5 should have the same type, and hence the same level. This is intuitively the case because they represent the address of two nodes that play the same rôle in the (encoding of the) recursive structure. As a consequence, in M_5, the weight consumed is equal to the weight which is released when triggering the two input prefixes (on p and x − here, if $p \in \sharp^{k_p}(\sharp^k\texttt{unit}, \sharp^k\texttt{unit})$, the constraint in rule (**Rep2**) gives $\{k_p, k\} >_{mul} \{k_p, k\}$). Process M_5 hence cannot be typed using the type system presented above.

This motivates the definition of a more refined type system in [DS06a], that exploits a well-founded partial order between names. The main modification is as follows: for a replicated process to be typable, either we have $\{k_1, \ldots, k_q\} > \mathcal{M}$ (as in rule (**Rep2**)), or $\{k_1, \ldots, k_q\} = \mathcal{M}$ and the partial order between names decreases: we are trading inputs for outputs of the same level, but going down in the partial order.

This is the case in process M_5, provided we impose that x dominates y according to the partial order. In this more refined type system, partial order information is attached to the type of channels: for M_5, the type of p is of the form $\sharp^{k_p}_{(1,2)}(\sharp^k\texttt{unit}, \sharp^k\texttt{unit})$, which enforces that the first argument of an output on p to be greater than its second argument. This way, M_5 can be accepted as terminating.

As we said, the typing hypothesis associated to p in the typing derivation for M_5 induces some constraints on the usages of p: for an output of the form $\overline{p}\langle a, b\rangle$ to be typable, partial order information should specify that a dominates b. Accordingly, in

$$M_6 \stackrel{\text{def}}{=} M_5 \mid \overline{p}\langle a, b\rangle \mid \overline{p}\langle b, c\rangle \mid \overline{p}\langle c, a\rangle \ ,$$

in order to type the outputs on p, we are bound to introduce a well-founded partial order on the (free) names a, b and c. Since this relation is necessarily cyclic, we cannot type-check M_6 (note that M_6 is not diverging, but $M_6 \mid \overline{a}$ is).

From lists to trees. The type system we have described above is further enriched in [DHS08]. In that more refined system, one is able to type-check processes

corresponding to tree structures. An example is given by

$$M_7 \stackrel{\text{def}}{=} !p(x,y,z).\,x.\,(\overline{y} \mid \overline{z} \mid \overline{p}\langle x,y,z \rangle) \ .$$

Here, p is used to construct nodes of binary trees: x has two children, y and z. This extension leads to some technicalities, that are related to the fact that we must allow the weight to grow along the firing of a replication (a request on a node is propagated to several child structures).

Typing Termination in the λ-calculus There exist encodings of the λ-calculus into the π-calculus, given a reduction strategy in the λ-calculus (cf. [SW01b]). The type system of Section 3 makes it possible to translate a typable (according to the simply typed λ-calculus) function of the λ-calculus into a typable π-calculus process. Thus this provides a method to show that a λ-term does not exhibit divergences in the call by name or to call by value disciplines.

On the other hand, the same approach cannot be followed (at least not directly) using the term-rewriting-based type systems presented in this section. The argument is non-trivial. We discuss here that for the system S1.

Lemma 14. *There exists a simply typed λ-term whose call-by-value encoding into the π-calculus cannot be typed according to system S1.*

Proof. The call-by-value encoding of a λ-term M is given by a process $[\![M]\!]_p$, which is parametrised upon a *location channel p*. It is defined by the following clauses:

$$[\![\lambda x.\ M]\!]_p \quad \stackrel{\text{def}}{=} \quad (\nu y)\,\overline{p}y.\,!y(x,q).\,[\![M]\!]_q \qquad\qquad [\![x]\!]_p \quad \stackrel{\text{def}}{=} \quad \overline{p}x$$

$$[\![M\ N]\!]_p \quad \stackrel{\text{def}}{=} \quad (\nu q,r)\,\big([\![M]\!]_q \mid [\![N]\!]_r \mid q(y).\,r(z).\,\overline{y}\langle z,p \rangle\big)$$

The call-by-name encoding is defined using similar ideas (see [SW01b]). In order for the encoding of a λ-term to be typable using S1, we must analyse the replicated inputs that are introduced when encoding λ-abstractions. It turns out that different usages of the same λ-calculus variable may induce constraints that make it impossible to apply rule (**Rep1**) of Figure 5.1.

For instance, if we consider the following typing judgement in the λ-calculus

$$f : (\sigma \rightarrow \tau) \rightarrow \tau \rightarrow \tau, u : \sigma \rightarrow \tau, v : \sigma \vdash \big(f\ \ \lambda x.\ (f\,u\,(u\,v))\big) : \tau \rightarrow \tau \ ,$$

we observe that the encoding of $\big(f\ \ \lambda x.\ (f\,u\,(u\,v))\big) : \tau \rightarrow \tau$ cannot be typed according to S1. The interested reader can write down the details of the encoding. It appears, in doing that, that the abstraction on x is translated into a replicated input on some channel y, and that the type of y must be unified with the type of some y_1 channel which is used to encode the application $(u\,v)$. Since y_1 is used in output within the scope of the replication on y, we get a form of 'recursive call', which prevents us from typing the process. It can be noted that the extensions of S1 presented above do not help either in typing the encoding of this λ-term.

5.3 On the Complexity of Type Inference

[DHKS07] presents an analysis of the problem of type inference for (some of) the systems we have discussed above.

Theorem 4. *The type inference problem for system S1 is polynomial.*

Proof (Sketch). Type inference in S1 boils down to searching for cycles in a directed graph: we construct a graph by associating a vertex to every name used in the process, and we draw an edge from a to b to represent the constraint $\mathtt{lvl}(a) > \mathtt{lvl}(b)$. This algorithm is implemented in [Kob07], and in [Bou08], where a more expressive variant of S1, described in [DHS08], is also implemented.

On the contrary, we prove in [DHKS07] the following result on the type inference problem for system S2 of Section 5.2:

Theorem 5. *The type inference problem for system S2 is NP-complete.*

The intuitive reason for this result is that in rule (**Rep2**), we are using a multiset ordering, which leads to a combinatorial number of possible level assignments. This allows us to reduce the problem **3SAT** to the problem of type inference.

We describe the main ideas behind the reduction, because it provides a good illustration of how system S2 works.

Proof. An instance $\mathcal{I}$ of **3SAT** is made of n clauses $(C_i)_{i \leq n}$ of three literals each, $C_i = l_i^1, l_i^2, l_i^3$. Literals are possibly negated propositional variables taken from a set $V = \{v_1, \ldots, v_m\}$. The problem is to find a mapping from V to $\{True, False\}$ such that, in each clause, at least one literal is set to True.

Given an instance $\mathcal{I}$ of **3SAT**, we describe how we build an instance of the problem of the type inference. We fix a particular name true. To each variable $v_k \in V$, we associate two names x_k and x_k', and define the process

$$M_k \quad \overset{\mathrm{def}}{=} \quad !\mathsf{true}.\,\mathsf{true}.\,\overline{x_k}.\,\overline{x_k'} \ \mid \ !x_k.\,x_k'.\,\overline{\mathsf{true}} \ .$$

We then consider a clause $C_i = \{l_i^1, l_i^2, l_i^3\}$ from $\mathcal{I}$. For $j \in \{1, 2, 3\}$ we let $y_i^j = x_k$ if l_i^j is v_k, and $y_i^j = x_k'$ if l_i^j is $\neg v_k$. We then define the process

$$N_i \quad \overset{\mathrm{def}}{=} \quad !y_i^1.\,y_i^2.\,y_i^3.\,\overline{\mathsf{true}} \ .$$

We call $\mathcal{I}_\mathsf{t}$ the problem of finding a typing derivation in S2 for the process $M \overset{\mathrm{def}}{=} M_1 \mid \ldots \mid M_m \mid N_1 \mid \ldots \mid N_n$.

The constraints corresponding to the existence of such a solution are as follows (the level associated to name true is noted t):

– for each M_k:

$$\big(\mathtt{lvl}(x_k) = t \wedge \mathtt{lvl}(x_k') < t\big) \quad \vee \quad \big(\mathtt{lvl}(x_k') = t \wedge \mathtt{lvl}(x_k) < t\big) \ . \qquad (2)$$

– for each N_i:

$$t \leq \mathtt{lvl}(y_i^1) \ \lor \ t \leq \mathtt{lvl}(y_i^2) \ \lor \ t \leq \mathtt{lvl}(y_i^3) \ . \tag{3}$$

We now prove that '$\mathcal{I}_t$ has a solution' is equivalent to '$\mathcal{I}$ has a solution'.

First, if $\mathcal{I}$ has a solution $S : V \to \{True, False\}$ then fix $t = 2$, and set $\mathtt{lvl}(x_k) = 2, \mathtt{lvl}(x'_k) = 1$ if v_k is set to True, and $\mathtt{lvl}(x_k) = 1, \mathtt{lvl}(x'_k) = 2$ otherwise. We check easily that condition (2) is satisfied; condition (3) also holds because S is a solution of $\mathcal{I}$.

Conversely, if $\mathcal{I}_t$ has a solution, then we deduce a boolean mapping for the literals in the original **3SAT** problem. Since constraint (2) is satisfied, we can set v_k to True if $\mathtt{lvl}(x_k) = t$, and False otherwise. We thus have that v_k is set to True iff $\mathtt{lvl}(x_k) = t$, iff $\mathtt{lvl}(x'_k) < t$. Hence, because constraint (3) is satisfied, we have that in each clause C_i, at least one of the literals is set to True, which shows that we have a solution to $\mathcal{I}$.

In [DHKS07], we also introduce a variant of system S2, which is strictly more expressive, and which relies on algebraic comparisons between multisets of names (instead of comparisons using only the multiset ordering). We show that type inference for this variant is polynomial. However, extending this variant with partial order information (as in Section 5.2), in order to allow processes where the weight does not decrease along the triggering of a replication, is more difficult than in the case of system S2.

6 Conclusions

We have discussed two type-based approaches for guaranteeing termination of π-calculus processes. The first one uses the method of logical relations, transplanted from functional languages. The second approach exploits notions from term-rewriting theory. We now give some additional comments about the results we have presented.

Logical relations techniques In Theorem 1, we have established termination for $\mathcal{P}$, 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.

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,SW01a] – the process analogous of the polymorphic λ-calculus.

We have applied 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.

Term-rewriting techniques The various type systems we have presented in Section 5 all share the same general approach, that somehow is more syntactic than the proof strategy based on logical relations. Indeed, the central idea is to design the termination analysis based on the structure of terms, in order to be able to define a measure that decreases along reductions of processes. As we have shown, this approach turns out to provide a great flexibility, making it possible to combine various tools from rewriting theory to type-check processes.

A further development of this approach has been studied in [RDS09], where the rewriting-based methods are applied to study termination in versions of the π-calculus that feature constructs for higher-order communication and, more generally, for the manipulation of process terms. This should open the way in particular for the adaptation of our results to recent proposals for models of distributed programming, where forms of code mobility are provided. This is left for future work.

Another interesting and natural direction for extensions of these studies is to work on the integration of the two approaches we have described. The processes that we can type in the first approach, although they may exhibit non-determinism, have a definite functional flavour. The second approach based on rewriting techniques has a more limited expressiveness on "functional" processes, but allows us to type more non-trivial "imperative" processes. We are currently working on combining the two approaches, with the goal of being able to handle systems in which both functional and imperative processes appear.

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.

[Bou08]   P. Boutillier. Implementation of a hybrid type system for termination in the π-calculus. Training period report, ENS Lyon, 2008.

[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, 1994.

[DH95]   Nachum Dershowitz and Charles Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, 1995.

[DHKS07]   R. Demangeon, D. Hirschkoff, N. Kobayashi, and D. Sangiorgi. On the Complexity of Termination Inference for Processes. In *Proc. of TGC'07*, volume 4912 of *LNCS*, pages 140–155. Springer, 2007.

[DHS08]   R. Demangeon, D. Hirschkoff, and D. Sangiorgi. Static and Dynamic Typing for the Termination of Mobile Processes. In *Proc. of IFIP TCS'08*. Springer, 2008.

[DM79]   Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DS06a]   Y. Deng and D. Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7):1045–1082, 2006.

[DS06b]   Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Inf. Comput.*, 204(7):1045–1082, 2006.

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

[Gan80]   Robin O. Gandy. Proofs of strong normalization. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[Kob98]   Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.

[Kob07]   N. Kobayashi. TyPiCal: Type-based static analyzer for the Pi-Calculus. available from `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`, 2007.

[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.

[Mos01]   Peter D. Mosses. The varieties of programming language semantics. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*, pages 165–190. Springer, 2001.

[Mos04]   Peter D. Mosses. Exploiting labels in structural operational semantics. *Fundam. Inform.*, 60(1-4):17–31, 2004.

[Mos06]   Peter D. Mosses. Formal semantics of programming languages: - an overview -. *Electr. Notes Theor. Comput. Sci.*, 148(1):41–73, 2006.

[RDS09]   R.Demangeon, D.Hirschkoff, and D. Sangiorgi. Termination in Higher-Order Concurrent Calculi. In *Proc. of FSEN'09*, 2009. to appear.

[San99]   D. Sangiorgi. The name discipline of uniform receptiveness. *Theo. Comp. Sci.*, 221:457–493, 1999.

[San06]   Davide Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 16(1):1–39, 2006.

[SW01a]   D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[SW01b]   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.