Termination of processes

Davide Sangiorgi*

December 2001†

Abstract

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 ensure termination of a non-trivial subset of the π-calculus by a combination of conditions on types and on the syntax. The proof of termination is in two parts. The first uses the technique of logical relations – a well-know technique of λ-calculi – on a small set of non-deterministic 'functional' processes. The second part of the proof uses techniques of process calculi, in particular techniques of behavioural preorders.

1 Introduction

The last ten years have seen a lot of work on calculi for process mobility such as the π-calculus [Mil99, SW01]. With respect to previous formalisms for concurrency, perhaps the most novel aspect of these calculi is the rich theory of types. Typically, types are used to guarantee *safety* properties such as absence of communication errors (for instance, agreement between values emitted and values expected along a given link), absence of certain interferences among processes, deadlock freedom. A safety property expresses a condition that is invariant under reduction. In type systems, the invariance is expresses by the fundamental theorem of types, subject reduction. In this paper, by contrast, we consider *liveness* properties.

One of the most important liveness property is *termination*. A term M terminates if all internal runs of M are finite; that is, M has no infinite computation

$$M \xrightarrow{\tau} M_1 \xrightarrow{\tau} M_2 \xrightarrow{\tau} \ldots .$$

If such an infinite computation exists, then M is non-terminating (or *diverges*).

'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

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

*Work supported by the EC project "PROFUNDIS".

†Revised May 2002

is convergent but non-terminating. (When the values communicated are not important, we abbreviate inputs $a(x). M$ and outputs $\overline{a}\langle v\rangle. M$ 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 $\mu X. (\pi. (M \mid X))$.) The difference between 'terminating' and 'convergent' has deep consequences in proofs. For instance, in simply-typed λ-calculi a combinatorial proof of convergence – also called weak normalisation – is easy (see for instance Turing's proof in [Gan80a]), whereas a combinatorial proof of termination – also called strong normalisation – is much harder [Gan80b, Lév77, JM98]. 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 runs consisting of external actions such as inputs and outputs. For instance, $!a. \overline{b}$ is terminating but is not finite.

In the π-calculus, as well as in the λ-calculus, termination is an undecidable property. Termination has been studied extensively in the λ-calculus [GLT88, Mit96]. Termination is also important in concurrency. 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.

The π-calculus is a very expressive formalism. In it, a number of programming language features can be encoded, including functions, objects, and state (in the sense of imperative languages) [SW01]. As a consequence, however, the notoriously-hard problems of termination for these features hit the π-calculus too.

We ensure termination in the π-calculus by a combination of conditions based on types and on the syntax. This allows us to use standard type constructs and to have simple syntactic conditions. We did not succeed in finding powerful conditions that are purely syntactic, or operational. We explain why, informally, below.

In first-order process calculi such as CCS [Mil89], a subclass of processes widely used for analysis and verification is that of regular processes. This class is defined, syntactically, by the constraint that no parallel composition appears inside recursive definitions. Regular processes are precisely the finite-state processes (possibly up to injective substitutions and bisimilarity). In the π-calculus, the class of regular processes is not very interesting, at least as far termination is concerned:

- The class has a limited expressive power. In the π-calculus it is common to use replicated processes of the form $!a(\widetilde{x}). M$, and these processes are non-regular, unless M is trivial. For instance, $!a. b. \mathbf{0}$ is non-regular but terminates. We are after conditions that ensure termination also for processes that are non-regular.

- Regular processes may not terminate, an example being

$$\boldsymbol{\nu} a\, (\overline{a} \mid !a. \overline{a}). \tag{1}$$

The termination problem is much harder in the π-calculus than in CCS. Consider for instance the process language given by the grammar production

$$M \quad ::= \quad (\boldsymbol{\nu} a_1, \ldots, a_n\,)(!a_1(x). M_1 \mid \ldots \mid !a_n(x). M_n \mid \overline{b_1}\langle v_1\rangle \mid \ldots \mid \overline{b_r}\langle v_r\rangle) \tag{2}$$

where $n, r \geq 0$. This language includes non-terminating processes, such as (1), 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 $(\boldsymbol{\nu} a_1, \ldots, a_n)(!a_1(x).\,M_1 \mid \ldots \mid !a_n(x).\,M_n \mid \overline{b_1}\langle v_1\rangle \mid \ldots \mid \overline{b_r}\langle v_r\rangle)$ 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] A combinatorial proof of termination, based on a a measure that decreases after each reduction, is not hard. The measure is simple 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 (2) does include non-terminating processes even if inputs are ordered; an example is

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

Modulo structural congruence ($\equiv$), a relation that allows us some simple rearrangements of the structure of a process, R reduces to itself. To ensure termination in the π-calculus we need more sophisticated conditions. Termination is indeed a good example of the difference between CCS with value passing and the π-calculus.

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 $\overline{x}\langle x\rangle$ and $\overline{a}\langle a\rangle$ 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 $\overline{a}\langle a\rangle$ exists. Unfortunately, non-termination may also arise because of 'implicit' self-applications: these are outputs $\overline{a}\langle b\rangle$ 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 $\overline{a}\langle b\rangle.\,!b(x).\,\overline{a}\langle x\rangle$. Indeed, under certain hypothesis the process is behaviourally equivalent to $\overline{a}\langle a\rangle$ [Mer01]. (The main hypothesis is *locality*, see below.) This form of implicit self-application is the reason for the divergence of

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

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:

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

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

M has the following divergent computation:

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

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 $\overline{x}\langle b\rangle \mid b(y).\,\overline{y}\langle x\rangle$. 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 $\overline{a_2}\langle a_1\rangle$ but the input at a_1 does not use a_2. However, recursive dependencies among names can arise dynamically in N, at run time.

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

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:

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

Following typed λ-calculi, 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:

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

(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 defer discussions and counterexamples to Sections 4 and 11, as the problems raised by state are quite technical. We only anticipate that we will impose two constraints on state: the parameters of recursive definitions should be first order; certain forms of nesting between inputs are disallowed.

Two remarks should be made on the π-calculus language we use. First, the π-calculus is *localised*, in the sense of [Mer01]; that is, the recipient of a link cannot use it in input. This feature 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.

Second, our π-calculus language includes first-order values, that is, values that do not contain links, and operations for manipulating them. Examples of first-order values are integers, booleans, pairs of booleans, lists of integers. We do not give a concrete syntax for first-order values. To gain in generality, we only give abstract conditions (on the syntax, and on the operational and typing rules) that first-order values should satisfy. For the same reason, we do not commit ourselves to a specific reduction strategy for evaluation of values. (Encodings of values as processes exist, but only for some concrete examples, and assume specific evaluation strategies [Mil99, SW01].)

In summary, our language $\mathcal{P}$ of terminating processes is the localised π-calculus with the addition of arbitrary first-order expressions, subject to the constraints on recursive inputs, self-applications, and state mentioned above.

The proof of termination of (the processes in) $\mathcal{P}$ 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, $\mathcal{P}_0$. 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 $\mathcal{P}_0$ 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 $\mathcal{P}_0$ is not very expressive. It is however a powerful language for the termination property. The second part of our proof shows that how the termination of $\mathcal{P}$ is derived from that of $\mathcal{P}_0$. For this, we use process calculus techniques, most notably techniques for behavioural preorders. An unusual feature of $\mathcal{P}_0$ that we exploit are infinite sums, that is, sums over a possibly-infinite set of summands. In the π-calculus literature, sum operators – let alone infinite sums – are usually omitted.

Structure of the paper Sections 2 and 3 contain background material on the π-calculus. Exceptions are the abstract conditions on syntax, transitions, and types (Conditions 2.1, 2.2, and 3.1) that define first-order values and boolean expressions, which are new. 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 10. First, we prove the termination of a language $\mathcal{P}^-$ of monadic functional non-deterministic processes (Section 5-7). Then, in Section 8, we extend $\mathcal{P}^-$ with polyadicity and arbitrary first-order values, thus obtaining the language $\mathcal{P}_0$. Finally, in Sections 9 and 10, we prove that the termination of $\mathcal{P}_0$ implies that of $\mathcal{P}$. In Section 11 we justify, by means of counterexamples, the termination conditions on state. In Section 12 we discuss our termination conditions on the Join calculus. In Section 13 we report some conclusions and suggestions for future work.

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 13, [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 process calculus

The syntax of the calculus, in Table 1, has all the process constructs of the standard polyadic π-calculus [Mil99, SW01], with the addition of first-order values, which are explained later. The calculus is *localised* [Mer01], that is, the recipient of a link cannot use it in input. Formally, in an input $a(\widetilde{x}).\, M$, names $\widetilde{x}$ 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 $a(\widetilde{x}).\, M$ is a.)

As it is customary in the π-calculus, we make no syntactic difference between links and variables: they are all names. We use the words 'name' and 'link' with a different meaning. A link is a name that can be used to perform communications. Names used as variables for first-order values, however, are not links. (Formally, a link is a name of connection type, see Section 3.)

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. Similarly, the names of a value v and of a boolean expression B, written $\mathsf{n}(v)$ and $\mathsf{n}(B)$, are the names that appear in v and in B. 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.

Intuitively, 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 (note that a first-order value need not be atomic). We also allow operations on first-order values; examples are addition and predecessor on integers, boolean negation. An expression $f()$ is an *atomic value*. An expression $f(\widetilde{v})$, for $\widetilde{v}$ non-empty, can represent a composite value, such as a pair or a list, or an non-reduced value expression such as $3 + 4$. We also allow to test values, in the construct `if` B `then` M `else` N. Here, the condition B is a boolean expression on values; as such, B may contain values but may not contain processes. We do not provide a grammar neither for values nor for boolean expressions. Instead, to gain generality, we state some abstract conditions that values and boolean expressions have to satisfy. Below are some syntactic conditions; other conditions, on evaluation and types, will be given in Sections 2.1 and 3.

Condition 2.1 (Syntactic conditions on values and boolean expressions)

1. The set of names in a value or a boolean expression is finite;
2. first-order values can be tested for equality (in the if-then-else construct);
3. values and boolean expressions are closed under substitution; that is, $v\{^{w}\!/\!_{x}\}$ is a value, for all values v, w and name x, and similarly for a boolean expression $B\{^{w}\!/\!_{x}\}$;
4. if $x \in \mathsf{n}(v)$, then $\mathsf{n}(v\{^{w}\!/\!_{x}\}) = (\mathsf{n}(v) - \{x\}) \cup \mathsf{n}(w)$; otherwise, if $x \notin \mathsf{n}(v)$, then $\mathsf{n}(v\{^{w}\!/\!_{x}\}) = \mathsf{n}(v)$; similarly for $\mathsf{n}(B\{^{w}\!/\!_{x}\})$;
5. the set of closed first-order values (that is, the first-order values that do not contain names) is countable. $\square$

			Values
v, w	$::=$	x	name
	$\mid$	$f(\widetilde{v})$	first-order values
B			*Boolean expressions on values*
			Processes
M, N	$::=$	$\mathbf{0}$	nil process
	$\mid$	$H\lfloor\widetilde{v}\rfloor$	recursion call
	$\mid$	$a(\widetilde{x}).\, M$	input
	$\mid$	$\overline{v}\langle\widetilde{w}\rangle.\, M$	output
	$\mid$	$M \mid M$	parallel
	$\mid$	$M + M$	sum
	$\mid$	$\texttt{if}\ B\ \texttt{then}\ M\ \texttt{else}\ M$	if-then-else
	$\mid$	$\boldsymbol{\nu}\widetilde{a}\, M$	restriction
H	$::=$	X	recursion variable
	$\mid$	$\mu X(\widetilde{x}).\, M$	recursive definition

Table 1: Syntax of processes

2.1 Transition relation

We do not give the rules for the evaluation relation of values and boolean expressions. We simply assume that there is such a relation, $\mapsto$, and that it satisfies the conditions below. A value v *does not reduce* if there is no v' such that $v \mapsto v'$. Value v *has a reduction sequence of length* n if there are $v_1, \dots, v_n$ such that $v \mapsto v_1 \mapsto v_2 \dots \mapsto v_n$. Similar terminology applies to boolean expressions.

Condition 2.2 (Operational conditions on values and boolean expressions)

1. Names cannot reduce;
2. for every value or boolean expression e there is a bound on the length of reduction sequences that e may have;
3. for every value or boolean expression e, if $e \mapsto e'$ then $\mathsf{n}(e') \subseteq \mathsf{n}(e)$. $\square$

Note that the evaluation relation on values and boolean expressions need not be deterministic. 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, and SUM-1 have been omitted.) They are the usual transition rules for π-calculus, in the early style. In the table, $[\![\,]\!]$ is a partial function on boolean expressions with result in the ordinary two-valued boolean domain $\{\mathsf{true}, \mathsf{false}\}$. Rules OUT-1, OUT-2, IF-1, IF-2, IF-3 make no obligations on when values should reduce. For instance, in

$$\overline{a}\langle 3+4\rangle \mid a(x).\, M$$

the reduction $3 + 4 \mapsto 7$ could take place before or after the communication at a. More specific reduction strategies are obtained by adding side conditions to the rules. Again, the reason for this choice is generality: side conditions decrease the possibilities of reduction, and the termination of processes with unconstrained rules implies that of the processes with constrained rules.

2.2 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*, $\overline{v}\langle\widetilde{w}\rangle$, that is, output without continuation; *replication*, $!M$, which represents an infinite parallel composition of copies of M; and *indexed sum*, $\sum_i M_i$, that is, a sum with countably-many summands. Their transition rules are:

$$\overline{a}\langle\widetilde{v}\rangle \xrightarrow{\overline{a}\langle\widetilde{v}\rangle} \mathbf{0} \qquad\qquad \frac{M \mid !M \xrightarrow{\alpha} M'}{!M \xrightarrow{\alpha} M'} \qquad\qquad \frac{M_i \xrightarrow{\alpha} M_i'}{\sum_i M_i \xrightarrow{\alpha} M_i'}$$

3 The simply-typed π-calculus

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

$$T ::= \sharp\,\langle\widetilde{T}\rangle \;\Big|\; t$$

where the *connection type* $\sharp\,\langle\widetilde{T}\rangle$ is the type of a link that carries tuples of values of type $\widetilde{T}$, and t ranges over *first-order types*, that is, the types of first-order values.

$$\text{INP:}\quad a(\widetilde{x}).\,M \xrightarrow{a\langle\widetilde{v}\rangle} M\{\widetilde{v}/\widetilde{x}\}$$

$$\text{OUT-1:}\quad \overline{a}\langle\widetilde{v}\rangle.\,M \xrightarrow{\overline{a}\langle\widetilde{v}\rangle} M$$

$$\text{OUT-2:}\quad \frac{w \mapsto w'}{\overline{a}\langle\widetilde{v}, w, \widetilde{u}\rangle.\,M \xrightarrow{\tau} \overline{a}\langle\widetilde{v}, w', \widetilde{u}\rangle.\,M}$$

$$\text{SUM-1:}\quad \frac{M \xrightarrow{\alpha} M'}{M + N \xrightarrow{\alpha} M'}$$

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

$$\text{COM-1:}\quad \frac{M \xrightarrow{a\langle\widetilde{v}\rangle} M' \qquad N \xrightarrow{\boldsymbol{\nu}\widetilde{b}\,\overline{a}\langle\widetilde{v}\rangle} N'}{M \mid N \xrightarrow{\tau} \boldsymbol{\nu}\widetilde{b}\,(M' \mid N')}\ \text{if } \widetilde{b} \notin \mathsf{fn}(M)$$

$$\text{RES:}\quad \frac{\boldsymbol{\nu}\widetilde{b}\,M \xrightarrow{\alpha} M'}{(\boldsymbol{\nu}a, \widetilde{b}\,)M \xrightarrow{\alpha} \boldsymbol{\nu}a\,M'}\ a \notin \mathsf{n}(\alpha)$$

$$\text{OPEN:}\quad \frac{\boldsymbol{\nu}\widetilde{b}\,M \xrightarrow{\boldsymbol{\nu}\widetilde{c}\,\overline{x}\langle\widetilde{v}\rangle} M'}{(\boldsymbol{\nu}a, \widetilde{b}\,)M \xrightarrow{(\boldsymbol{\nu}a,\widetilde{c}\,)\overline{x}\langle\widetilde{v}\rangle} M'}\ x \neq a\ ,\ a \in \mathsf{n}(\widetilde{v}) - \widetilde{c}$$

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

$$\text{IF-1:}\quad \frac{B \mapsto B'}{\texttt{if } B \texttt{ then } M \texttt{ else } N \xrightarrow{\tau} \texttt{if } B' \texttt{ then } M \texttt{ else } N}$$

$$\text{IF-2:}\quad \frac{[\![B]\!] = \mathsf{true}, \qquad M \xrightarrow{\alpha} M'}{\texttt{if } B \texttt{ then } M \texttt{ else } N \xrightarrow{\alpha} M'}$$

$$\text{IF-3:}\quad \frac{[\![B]\!] = \mathsf{false}, \qquad N \xrightarrow{\alpha} N'}{\texttt{if } B \texttt{ then } M \texttt{ else } N \xrightarrow{\alpha} N'}$$

Table 2: Transition rules

A *link* is a name of a connection type. A link is *first order* if it only carries first-order values. It is *higher order* if it may also carry higher-order values (i.e., links). Note that 'first-order name' is different from 'first-order link': a first-order name has a type t (for some t), whereas a first-order link has a type $\sharp\,\langle\widetilde{t}\rangle$ (for some $\widetilde{t}$).

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.

An expression is *closed* if all names it contains have a connection type (i.e., they are links). Therefore, since first-order values do not contain links, a *closed first-order value* is simply one that does not contain names (in accordance with what stated in Condition 2.1(5)). Similarly a process is *closed* if all its free names have a connection type. 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 typing rules are in Table 3. Once more, the rules for first-order values and boolean expressions do not interest us. We therefore assume an oracle to infer judgements $\vdash v : T$ and $\vdash B : \texttt{Bool}$, subject only to the conditions below, where we use e for a value or a boolean expression. (We also assume that first-order types include the boolean type `Bool`.)

Condition 3.1 (Type conditions on values and boolean expressions)

1. For any name x, we have $\vdash x : T$ iff $x \in T$;
2. a value $f(\widetilde{v})$, if typed, has a first-order type;
3. if $\vdash v : t$, then v does not contain links;
4. if $\vdash e : T$, and $x : w$ then $\vdash e\{^w\!/\!_x\} : T$;
5. if $e \mapsto e'$ and $\vdash e : T$ then also $e' : T$. □

Lemma 3.2 Let e be a value or a boolean expression, and suppose $e \mapsto e'$. If e is closed then also e' is closed. □

The typing rules for the operators of Section 2.2 (asynchronous output, replication, indexed sum) are similar to those of (standard) output, parallel composition, and sum.

4 Terminating processes

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

Definition 4.1 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,

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

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

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

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

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

$$\text{T-IF}: \frac{\vdash B : \texttt{Bool} \qquad \vdash M \qquad \vdash N}{\vdash \ \texttt{if}\ B\ \texttt{then}\ M\ \texttt{else}\ N} \qquad \text{T-NIL}: \ \vdash \mathbf{0}$$

Table 3: Typing rules

M *terminates* (or *is terminating*), written $M \in$ TER, if M is not divergent. $\square$

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 4.2. The first condition is for recursive inputs; the second and third conditions control state. The last constraint is expressed using types, as condition 1 of Definition 4.3, and 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

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

has free replicated first-order inputs, whereas

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

has not.

Condition 4.2 (Termination constraints on the grammar)

1. Let $\widetilde{a} = a_1, \ldots, a_n$. In a process $\boldsymbol{\nu}\widetilde{a}\, M$, if $a_i(\widetilde{x}).\, N$ is a free input of M, then the following holds:

   (a) $a_i \in \widetilde{a}$,

   (b) names a_j with $j \ge i$ do not appear free in output position in N.

2. In an higher-order input $a(\widetilde{x}).\, M$, 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. $\square$

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:

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

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

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

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 4.2 to mean that M itself and all its process subterms respect the constraints.

Definition 4.3 (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 4.2, where $\widetilde{a} = \mathsf{fn}(M)$. $\square$

Theorem 4.4 All processes in $\mathcal{P}$ terminate.

Most of the paper is devoted to proving this theorem. The proof is in two parts; one in Sections 5-8, the other in Sections 9-10.

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 $\mathcal{P}^-$ (more precisely its extension $\mathcal{P}_0$ in Section 8) to derive the termination of the processes of the main language we are interested in, namely the language $\mathcal{P}$ of Theorem 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 a sum operator. For technical reasons, we allow sums with countably-many summands (indexed sums): this feature will be essential in proofs of Section 9 involving stateful processes. Outputs are asynchronous, that is, they have no continuations. The behaviour of these operators has been defined in Section 2.2.

To facilitate the reading of the proofs, the calculus is monadic and `unit` is the only first-order type. (The extension of $\mathcal{P}^-$ with polyadicity and arbitrary first-order values is studied in Section 8). Therefore the grammar for the types of values is:

$$T \quad ::= \quad \sharp\,\langle T\rangle \;\Big|\; \mathtt{unit} \tag{3}$$

As the calculus is monadic, we omit angle brackets in output prefixes, and in input and output actions. We assume that for each type there is an uncountable set of names with that type. This form of 'infinity' on names, and that on sums, are the only features of $\mathcal{P}^-$ that go beyond the π-calculus of Section 2.

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 5.1 (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)

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

(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 $\equiv_1$. $\square$

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: L is a countable indexing set; $\mathtt{in}(P)$ 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 7.9) in later proofs.

Definition 5.2 (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. $\square$

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 $\mathcal{P}^-$).

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

Table 4: The normal forms for the language $\mathcal{P}^-$

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:

$$A ::= \diamond \;\Big|\; b\_{\sharp}\,\langle T \rangle$$

where T is an ordinary type, as by grammar (3) in Section 5. 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 6.1 (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. $\square$

Lemma 6.2 A pre-process has a unique type.

Proof The normal forms of a process are structurally congruent. $\square$

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

Definition 6.3 (Logical relations)

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

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

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

$$\boldsymbol{\nu} b\,(I_b \mid \boldsymbol{\nu} a\,(P \mid \overline{a}b)) \in \mathcal{L}^{\diamondsuit}\,. \tag{4}$$

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

In Definition 6.3, the most important clause is the last one. The process in (4) is similar to those used for translating function application into π-calculus [SW01]. Therefore a possible reading of (4) is that P is a function and I_b its argument. In (4), 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.

Lemma 6.4 $P \in \mathcal{L}^{a\_\sharp\langle T\rangle}$, where T is a connection type, if $P : a\_\sharp\langle T\rangle$ and there exists b fresh for P such that for all $I_b \in \mathcal{L}^{b\_T}$,

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

Proof The set TER of terminating processes is closed under injective substitutions. □

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 $\mathrm{A}\pi_\Sigma$: this is the π-calculus of Section 2, well-typed, without recursion and if-then-else, with only asynchronous outputs, and with the addition of indexed sum and replication. Here is the grammar of $\mathrm{A}\pi_\Sigma$:

$$M ::= a(\widetilde{x}).\,M \;\Big|\; \overline{v}\langle\widetilde{w}\rangle \;\Big|\; \sum_{i\in L} M_i \;\Big|\; M \mid M \;\Big|\; !M \;\Big|\; \boldsymbol{\nu}\widetilde{a}\,M$$

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 7.1 (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. $\square$

Lemma 7.2 Relation $\sim$ is a congruence on $\mathrm{A}\pi_\Sigma$. $\square$

Lemma 7.3 (Sharpened Replication Theorems, for $\mathrm{A}\pi_\Sigma$) Suppose a does not appear free in input position in $M, N, N_1, N_2, \pi.\, N$. We have:

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

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 $\mathrm{A}\pi_\Sigma$ plus the rule

$$\textsc{sum-int:} \quad \sum_{i \in L} M_i \xrightarrow{\tau} M_i$$

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 7.4 Suppose $R \longrightarrow_{\mathsf{E}} R_1 \xrightarrow{\tau} R_2$ where $R \longrightarrow_{\mathsf{E}} R_1$ uses rule SUM-INT. Then either $R \xrightarrow{\tau} R_2$, or there is R' such that $R \xrightarrow{\tau} R' \longrightarrow_{\mathsf{E}} R_2$. $\square$

Lemma 7.5 (in $\mathrm{A}\pi_\Sigma$) If R E-diverges, then R diverges.

Proof Each process $R \in \mathrm{A}\pi_\Sigma$ has only a finite number of unguarded sums. As a consequence, R can only have finite sequences of reductions in which each step uses rule SUM-INT. Therefore if R E-diverges, then R has a divergent computation in which an infinite number of steps do not use rule SUM-INT. From this and Lemma 7.4 we conclude that R diverges. $\square$

Lemma 7.6 (in $\mathrm{A}\pi_\Sigma$) 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 In this proof, for any process R, we write R^c for the process $R\{c/c'\}$.

Consider a reduction $R \mid !c'(x).\, \overline{c}x \xrightarrow{\tau} R' \mid !c'(x).\, \overline{c}x$ (the case of a reduction $R \mid !c'(x).\, \overline{c}x \xrightarrow{\tau} \boldsymbol{\nu} b\,(R' \mid !c'(x).\, \overline{c}x)$ is similar). This is either given by a reduction $R \xrightarrow{\tau} R'$ (which cannot be a communication along c'), or by a communication between R and $c'(x).\, \overline{c}x$ in which an output $\overline{c'}b$ of R is consumed. In the first case, there is also a reduction $R^c \longrightarrow_{\mathsf{E}} {R'}^c$.

In the second case, we have $R \xrightarrow{\overline{c'}b} R''$, for some R'' and $R' = R'' \mid \overline{c}b$. In the derivation proof of this transition, the output $\overline{c'}b$ consumed can be part of some sums, which disappear as effect of rules SUM-1 and SUM-2; the communication (rule COM-2) then liberates the output $\overline{c}b$. The same effect is obtained in R^c by using the rule SUM-INT, once for each sum that has to be eliminated. The resulting process is $\equiv$ with R'.

This explains why 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 $R^c(\longrightarrow_{\mathsf{E}})^\star \equiv R'^c$. 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.

It therefore remains to show 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 would imply, by the arguments above, that also R^c E-diverges. We now consider this remaining property.

Let R be any process in Aπ_Σ. We say that two outputs in R are *independent* if they are not subterms of different summands of a sum (thus if two outputs are not independent, when one is consumed the other is lost). Process R may only have a finite number of unguarded outputs that are pairwise independent. As a consequence, and since R only uses input-guarded replication, if R does not have inputs at c', then $R \mid !c'(x).\,\overline{c}x$ may only have finite sequences of reductions in which each step is a communication at c'. We conclude that in a divergent computation of $R \mid !c'(x).\,\overline{c}x$ there must be an infinite number of steps that are not communications at c'. $\square$

Lemma 7.7 (in Aπ_Σ) 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 7.6 and 7.5, and the fact that a top restriction preserves divergences. $\square$

7.3 Closure properties

Lemma 7.8 Suppose $R \in \mathcal{P}^-$. If $R \equiv\xrightarrow{\alpha} R'$ then $R \xrightarrow{\alpha}\equiv R'$. $\square$

Lemma 7.9 (Closure under reduction for $\mathcal{P}^-$) $R \in \mathcal{P}^-$ and $R \xrightarrow{\tau} R'$ imply $R' \in \mathcal{P}^-$.

Proof Because of Lemma 7.8, 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. The interesting case is given by the production

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

that is the only production in which there are two parallel processes one of which can have a free input. We need the following easy facts, where $\mathcal{P}^{-\mathrm{NF}}, \mathcal{RES}^{\mathrm{NF}}, \mathcal{PR}^{\mathrm{NF}}$ are the subset of normal forms for $\mathcal{P}^-, \mathcal{RES}, \mathcal{PR}$.

- the set $\mathcal{PR}$ of processes is closed under substitution;

- If $M^{\mathrm{NF}} \xrightarrow{\overline{a}v} R$ then $R \equiv M'$, for some $M' \in \mathcal{PR}^{\mathrm{NF}}$;
- If $M^{\mathrm{NF}} \xrightarrow{\boldsymbol{\nu} b\,\overline{a}b} R$ then $R \equiv \boldsymbol{\nu}\widetilde{c}(I_{\widetilde{c}} \mid I_b \mid M')$, for some $I_{\widetilde{c}}, I_b \in \mathcal{RES}^{\mathrm{NF}}, M' \in \mathcal{PR}^{\mathrm{NF}}$ with $b \notin \mathsf{fn}(I_{\widetilde{c}})$;
- If $I_a^{\mathrm{NF}} \xrightarrow{av} R$ then $R \equiv I_a^{\mathrm{NF}} \mid M$, for some $M \in \mathcal{PR}^{\mathrm{NF}}$.

We can then finish off the proof. Suppose $I_a^{\mathrm{NF}} \xrightarrow{ab} R$ and $M^{\mathrm{NF}} \xrightarrow{\boldsymbol{\nu} b\,\overline{a}b} R'$ (the case of free output is simpler) and $\boldsymbol{\nu} a\,(I_a^{\mathrm{NF}} \mid M^{\mathrm{NF}}) \xrightarrow{\tau} R'' \stackrel{\mathrm{def}}{=} \boldsymbol{\nu} a\,(\boldsymbol{\nu} b\,(R \mid R'))$. We have to prove $R'' \in \mathcal{PR}$. Using the previous facts we have

$$\begin{array}{rcll} R & \equiv & I_a^{\mathrm{NF}} \mid N & \text{for some } N \in \mathcal{PR}^{\mathrm{NF}} \\ R' & \equiv & \boldsymbol{\nu}\widetilde{c}\,(I_{\widetilde{c}} \mid I_b \mid M') & \text{for some } I_{\widetilde{c}}, I_b \in \mathcal{RES}^{\mathrm{NF}}, M' \in \mathcal{PR}^{\mathrm{NF}} \text{ with } b \notin \mathsf{fn}(I_{\widetilde{c}}) \end{array}$$

Therefore

$$\begin{array}{rcl} R'' & = & (\boldsymbol{\nu} a\,, b)(I_a^{\mathrm{NF}} \mid N \mid \boldsymbol{\nu}\widetilde{c}\,(I_{\widetilde{c}} \mid I_b \mid M')) \\ & \equiv & \boldsymbol{\nu} a\,(I_a^{\mathrm{NF}} \mid \boldsymbol{\nu}\widetilde{c}\,(I_{\widetilde{c}} \mid \boldsymbol{\nu} b\,(I_b \mid (M' \mid N)))) \stackrel{\mathrm{def}}{=} R''' \end{array}$$

and R''' is a process in normal form. Hence $R'' \in \mathcal{PR}$. □

Lemma 7.10 (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 $A = a\_\sharp\,\langle T\rangle$, 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 $P \in \mathcal{L}^{a\_\sharp\,\langle T\rangle}$,

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

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

Lemma 7.11 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 7.12 If $R \xrightarrow{\tau}_{\mathrm{d}} R'$ and $R' \in$ TER then also $R \in$ TER. □

Lemma 7.13 If $a, b : T$ then $!a(x).\,\overline{b}x \in \mathcal{L}^{a\_T}$.

Proof Suppose $T = \sharp\langle \mathtt{unit} \rangle$. 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 7.12 and 7.10 $\boldsymbol{\nu} a\,(!a(x).\,\overline{b}x \mid \overline{a}v) \in$ 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. $\square$

Lemma 7.14 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 7.7. $\square$

Lemma 7.15 $\sum_i M_i \in$ TER iff, for each i, $M_i \in$ TER. $\square$

Lemma 7.16 $M_1 \mid M_2 \in$ 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. $\square$

7.4 Relatively independent resources

Definition 7.17 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. $\square$

Lemma 7.18 Suppose $!a(x).\,M \in \mathcal{RES}$, and $I_b \in \mathcal{RES}$, and $a \notin \mathsf{fn}(I_b)$. Then also $!a(x).\,\boldsymbol{\nu} b\,(I_b \mid M) \in \mathcal{RES}$. $\square$

Lemma 7.19 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. We need the Replication Theorems and Lemma 7.18. $\square$

Lemma 7.20 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 7.21 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 $\equiv \subseteq \sim$. $\square$

7.5 Main theorem

Theorem 7.22 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 7.21 and 7.10 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 $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid P)$.

- $P = \overline{b}c$.

  In this case, $A = \diamond$. We have to show that $Q \in$ TER. We have:

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

  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 7.14), 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 7.13) 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

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

    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).

- $P = \overline{b}v$ and $v : \mathtt{unit}$

  Similar to the previous case.

- $P = \sum_i M_i$. (Thus $P : \diamond$)

  We have to show that $Q = \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid \sum_i M_i) \in$ TER. Using the Replication Theorems we have

  $$Q \sim \sum_i \boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid M_i)\,.$$

  By Lemma 7.15 this process is in TER iff each component is so. The latter is true by the induction on the structure. Hence also $Q \in$ TER.

- $P = M_1 \mid M_2$. (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 7.16, $Q \in \text{TER}$ iff each Q_i is so. The latter is true by the induction on the structure.

- $P = {!b(x)}.\,M$. Then $Q : b\_{\sharp}\,\langle T\rangle$. 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

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

and

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

(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 $R' \in \text{TER}$ (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:

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

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 $J_x \in \mathcal{L}$ by Lemmas 7.10 and 7.18. 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 $Q'' \in \text{TER}$.

$\square$

Corollary 7.23 If $P \in \mathcal{P}^-$ then $P \in \mathcal{L}$. $\square$

8 Arbitrary first-order types and polyadicity

The language $\mathcal{P}^-$ of Section 2 has `unit` as the only first-order type, and is monadic. Extending the definition of logical relations and the proof of termination to allow arbitrary first-order values (subject to Conditions 2.1, 2.2, and 3.1) and polyadicity is easy. We only show the modifications to Definition 6.3 of logical relations. We call $\mathcal{P}_0$ such an extension of $\mathcal{P}^-$.

First, the case of a pre-process $P : a\_{\sharp}\,\langle t\rangle$ where t is a arbitrary first-order type.

- $P \in \mathcal{L}^{a\_\sharp \langle t \rangle}$, where t is a first-order type, if $P : a\_\sharp \langle t \rangle$, and for all $v : t$,

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

Now we consider polyadicity. We show the clause of a process $P : a\_\sharp \langle \widetilde{T} \rangle$ where all T_i are connection types. The case when all T_i are first order is simpler, and the case where $\widetilde{T}$ is a mixture of first-order and connection types is the expected combination of clauses.

- $P \in \mathcal{L}^{a\_\sharp \langle \widetilde{T} \rangle}$, where $\widetilde{T} = T_1, \ldots, T_n$, if $P : a\_\sharp \langle \widetilde{T} \rangle$, and, for all $\widetilde{b} = b_1, \ldots, b_n$ fresh, and for all $I_{b_1} : b_1\_T_1, \ldots, I_{b_n} : b_n\_T_n$, with $I_{\widetilde{b}}$ relatively independent,

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

The hypothesis of relative independence can be dropped, but simplifies the proofs of a few lemmas.

9 Proofs based on simulation

The language $\mathcal{P}_0$ of Section 8 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 $\mathcal{P}_0$ implies that of a much broader language. This is what we are going to show now.

We consider below a number of extensions. 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 $\mathcal{P}_i \subset \mathcal{P}_{i+1}$ 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. In process calculi, 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 9.1 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 $(M\sigma, N\sigma) \in \mathcal{R}$. $\square$

Relation $\preceq$ is reflexive and transitive, and is preserved by all operators of the π-calculus (as well the other operators of Section 2.2). Hence $\preceq$ is a precongruence in all languages $\mathcal{P}_i$ we shall consider. An important property of $\preceq$ for us is:

Lemma 9.2 If $M \preceq M'$ then $M' \in \mathrm{TER}$ implies $M \in \mathrm{TER}$. $\square$

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; $\mathcal{P}_0$ is the language of Section 8 (whose normal forms are those of Table 4 with the extension to arbitrary first-order values and polyadicity). Processes and resources are then obtained by closing the corresponding normal forms under $\equiv$, in the same way as we did for $\mathcal{P}^-$ and $\mathcal{P}_0$. From now on, we do not need pre-processes anymore – pre-processes had been introduced only to define logical relations in Section 6.

For each $\mathcal{P}_i$, it is sufficient to prove that the normal forms of the processes in $\mathcal{P}_i$ terminate. By Lemma 9.3 this implies the termination of all $\mathcal{P}_i$ processes.

Lemma 9.3 $M \equiv M'$ and $M' \in \mathrm{TER}$ imply $M \in \mathrm{TER}$. $\square$

9.1 Synchronous outputs

Language $\mathcal{P}_1$

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

Proof of termination of $\mathcal{P}_1$ The transformation $[\![\cdot]\!]_0 : \mathcal{P}_1 \to \mathcal{P}_0$ acts on outputs thus:

$$[\![\overline{b}\langle\widetilde{v}\rangle.\, M]\!]_0 \stackrel{\mathrm{def}}{=} \overline{b}\langle\widetilde{v}\rangle \mid [\![M]\!]_0\,.$$

The transformation is an homomorphism elsewhere. Its correctness is given by the law

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

and Lemma 9.2. $\square$

9.2 Non-functional resources

In $\mathcal{P}_1$ all resources are functional. We now consider extensions of the language that allow us to have non-functional resources.

Language $\mathcal{P}_2$

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

Proof of termination of $\mathcal{P}_2$ The transformation $[\![\cdot]\!]_1 : \mathcal{P}_2 \to \mathcal{P}_1$ replaces a process $\boldsymbol{\nu} a\,(M_1 \mid M_2)$ with $\boldsymbol{\nu} a\,(!a(x).\,\mathbf{0} \mid M_1 \mid M_2)$. The correctness of the transformation is given by the property

$$\boldsymbol{\nu} a\,(M_1 \mid M_2) \preceq \boldsymbol{\nu} a\,(!a(x).\,\mathbf{0} \mid M_1 \mid M_2)$$

where a is any name. $\square$

Language $\mathcal{P}_3$

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

Proof of termination of $\mathcal{P}_3$ The difference between $\mathcal{P}_3$ and $\mathcal{P}_2$ is that in the former there can be several input-replicated processes at the same link. The correctness of $\mathcal{P}_3$ is then inferred from that of $\mathcal{P}_2$ and the law

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

$\square$

Now we consider the simultaneous creation of resources on different links.

Language $\mathcal{P}_4$

$$\begin{array}{lcll}
M^{\mathrm{NF}} & ::= & \ldots \;\Big|\; \boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}} \mid M^{\mathrm{NF}}) & \\
I^{\mathrm{NF}}_{\widetilde{a}} & ::= & \ldots & \\
 & \Big| & !a_i(\widetilde{x}).\,M^{\mathrm{NF}} & \text{if } a_i \in \widetilde{a}, \text{ and } a_j \notin \mathsf{fn}(M^{\mathrm{NF}}) \text{ for all } j \geq i \\
 & \Big| & I^{\mathrm{NF}}_{\widetilde{a}} \mid I^{\mathrm{NF}}_{\widetilde{a}} & \\
 & \Big| & M^{\mathrm{NF}} &
\end{array}$$

The side condition '$a_j \notin \mathsf{fn}(M^{\mathrm{NF}})$ for all $j \geq i$' says that $\widetilde{a}$ is an ordered set of names and the body of an input at a_i can only use names that are below a_i in the order. This condition is a generalisation of the condition on non-recursive replications in the previous languages.

Since multiple resources ($I^{\mathrm{NF}}_{\widetilde{a}}$) include single resources (I^{NF}_{a}), all productions for single resources can now be dropped from the grammar.

Proof of termination of $\mathcal{P}_4$ A process $\boldsymbol{\nu}\widetilde{a}\,(I_{\widetilde{a}} \mid M)$ is structurally congruent to a process of the form

$$\boldsymbol{\nu}a_1\,(I_{a_1} \mid \boldsymbol{\nu}a_2\,(I_{a_2} \mid \ldots \mid \boldsymbol{\nu}a_n\,(I_{a_n} \mid M))\,.$$

Therefore for each term R in $\mathcal{P}_4$ there is a term R' in $\mathcal{P}_3$ with $R \equiv R'$. $\square$

We now continue with additions to $I^{\mathrm{NF}}_{\widetilde{a}}$.

Language $\mathcal{P}_5$

$$I^{\mathrm{NF}}_{\widetilde{a}} \quad ::= \quad \ldots \;\Big|\; a_i(\widetilde{x}).\,M^{\mathrm{NF}} \quad \text{with } a_i \in \widetilde{a} \text{ and } a_j \notin \mathsf{fn}(M^{\mathrm{NF}}) \text{ for all } j \geq i$$

Proof of termination of $\mathcal{P}_5$ Follows from that of $\mathcal{P}_4$ and the law $a(\widetilde{x}).\,R \preceq !a(\widetilde{x}).\,R$. $\square$

Language $\mathcal{P}_6$

$$I^{\mathrm{NF}}_{\widetilde{a}} \quad ::= \quad \ldots \;\Big|\; \overline{b}\langle\widetilde{v}\rangle.\,I^{\mathrm{NF}}_{\widetilde{a}}$$

Proof of termination of $\mathcal{P}_6$ We use the law $\overline{a}\langle\widetilde{v}\rangle.\,R \preceq \overline{a}\langle\widetilde{v}\rangle \mid R$. $\square$

$$
\begin{array}{rcl}
I^{\mathrm{NF}}_{\widetilde{a}} & ::= & !\mathcal{I}^{\widetilde{a}} \\
& \big| & \mathcal{I}^{\widetilde{a}} \\
& \big| & M^{\mathrm{NF}} \\
& \big| & I^{\mathrm{NF}}_{\widetilde{a}} \mid I^{\mathrm{NF}}_{\widetilde{a}} \\
& \big| & I^{\mathrm{NF}}_{\widetilde{a}} + I^{\mathrm{NF}}_{\widetilde{a}} \\
& \big| & \overline{b}\langle \widetilde{v} \rangle .\, I^{\mathrm{NF}}_{\widetilde{a}} \\
\mathcal{I}^{\widetilde{a}} & ::= & a_i(\widetilde{x}).\, M^{\mathrm{NF}} \quad \text{with } a_i \in \widetilde{a} \text{ and } a_j \notin \mathsf{fn}(M^{\mathrm{NF}}) \text{ for all } j \geq i
\end{array}
$$

Table 5: Normal forms for the resources in $\mathcal{P}_7$

Language $\mathcal{P}_7$

$$
I^{\mathrm{NF}}_{\widetilde{a}} \quad ::= \quad \ldots \;\Big|\; I^{\mathrm{NF}}_{\widetilde{a}} + I^{\mathrm{NF}}_{\widetilde{a}}
$$

Proof of termination of $\mathcal{P}_7$ We use the law $R_1 + R_2 \preceq R_1 \mid R_2$. □

Table 5 shows the final grammar for resources in normal form.

9.3 If-then-else

We recall that B is a generic boolean expression on values, subject to Conditions 2.1, 2.2, and 3.1.

Language $\mathcal{P}_8$

$$
\begin{array}{rcll}
M^{\mathrm{NF}} & ::= & \ldots & \Big|\; \texttt{if } B \texttt{ then } M^{\mathrm{NF}} \texttt{ else } M^{\mathrm{NF}} \\
I^{\mathrm{NF}}_{\widetilde{a}} & ::= & \ldots & \Big|\; \texttt{if } B \texttt{ then } I^{\mathrm{NF}}_{\widetilde{a}} \texttt{ else } I^{\mathrm{NF}}_{\widetilde{a}}
\end{array}
$$

Proof of termination of $\mathcal{P}_8$ Consider a term `if` B `then` R_1 `else` R_2; let $\widetilde{x}$ be the free variables of B, and $V \stackrel{\mathrm{def}}{=} \{\widetilde{v} \mid \widetilde{x} : \widetilde{v} \text{ and } \widetilde{v} \text{ is closed}\}$.
For each $\widetilde{v}$, there is a bound $n_{\widetilde{v}}$ on the number of reduction steps that an expression $B\{\widetilde{v}/\widetilde{x}\}$ can perform. For each $n_{\widetilde{v}}$, let $R_{\widetilde{v}}$ be a term such that

$$
R_{\widetilde{v}} \,(\stackrel{\tau}{\longrightarrow})^{n_{\widetilde{v}}} \;\sim\; \mathbf{0}.
$$

Therefore we have

$$
\texttt{if } B \texttt{ then } R_1 \texttt{ else } R_2 \preceq R_1 \mid R_2 \mid \sum_{\widetilde{v} \in V} R_{\widetilde{v}}\,.
$$

Using this transformation, each if-then-else can be removed. □

9.4 Nested inputs

We show that nesting of inputs is possible for (certain) first-order links. In a later section we show that, surprisingly, input nesting in general destroys termination. In the new production below, $\mathcal{I}^{\widetilde{a}}$ is the grammar symbol of Table 5; $\mathtt{on}(R)$ are the names that appear free in R in output prefixes; we recall that a *replicated input* is an input that is underneath a replication (in following languages, which will have also the recursion operator, an input is considered replicated also if it is in the body of a recursion).

Language $\mathcal{P}_9$

$$\mathcal{I}^{\widetilde{a}} \quad ::= \quad \ldots \;\Big|\; a_i(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}}$$

where

- $a_i \in \widetilde{a}$
- $a_j \notin \mathtt{on}(I^{\mathrm{NF}}_{\widetilde{a}})$ for all $j \geq i$
- if a_i is higher order, then $I^{\mathrm{NF}}_{\widetilde{a}}$ does not contain: free higher-order inputs; free replicated first-order inputs.

The condition on the occurrence of names in $\widetilde{a}$ in output position in the body of the input was also present in the previous language $\mathcal{P}_8$. The distinction between first-order and higher-order links in the last condition says that a first-order input gives no constraints on nesting, whereas underneath a higher-order input we can only have non-replicated first-order inputs. The proof of the correctness of $\mathcal{P}_9$ requires some preliminary work.

Let $\mathrm{A}\pi_\Sigma$ be π-calculus defined in Section 7 (the simply-typed π-calculus, without recursion and if-then-else, and with only asynchronous outputs, but with indexed sum and replication). We use a few results for this calculus in the correctness proof of $\mathcal{P}_9$.

Lemma 9.4 (in $\mathrm{A}\pi_\Sigma$**)** Suppose C is any context where the hole is not in the scope of a binder, and R any process. We have:

$$C[R] \preceq C[\mathbf{0}] \mid !R.$$

Proof By induction on C. $\square$

For a generic process R of $\mathrm{A}\pi_\Sigma$ and first-order names $\widetilde{x}$, we define the process $R\{\widetilde{x}\}$; we write W for the set $\{\widetilde{w} \mid \widetilde{w}$ is closed and $\widetilde{x} : \widetilde{w}\}$.

$$\begin{array}{rcl}
\overline{a}\langle\widetilde{v}\rangle\{\widetilde{x}\} & \stackrel{\mathrm{def}}{=} & \sum_{\widetilde{w}\in W} \overline{a}\langle\widetilde{v}\{\widetilde{w}/\widetilde{x}\}\rangle \\
(R_1 \mid R_2)\{\widetilde{x}\} & \stackrel{\mathrm{def}}{=} & R_1\{\widetilde{x}\} \mid R_2\{\widetilde{x}\} \\
(\sum_i R_i)\{\widetilde{x}\} & \stackrel{\mathrm{def}}{=} & \sum_i (R_i\{\widetilde{x}\}) \\
(\boldsymbol{\nu}\widetilde{a}\, R)\{\widetilde{x}\} & \stackrel{\mathrm{def}}{=} & \boldsymbol{\nu}\widetilde{a}\, (R\{\widetilde{x}\}) \\
(b(\widetilde{y}).\, R)\{\widetilde{x}\} & \stackrel{\mathrm{def}}{=} & b(\widetilde{y}).\, R\{\widetilde{x}\} \\
(!R)\{\widetilde{x}\} & \stackrel{\mathrm{def}}{=} & !R\{\widetilde{x}\}
\end{array}$$

Intuitively, $R\{\widetilde{x}\}$ eliminates $\widetilde{x}$ from the free names of R, by replacing all output expressions in which $\widetilde{x}$ could appear free, with a summation on all possible closed values that $\widetilde{x}$ could take.

Lemma 9.5 (in $\mathrm{A}\pi_{\Sigma}$) Suppose $\widetilde{x}$ are first-order names, and C is any context where the hole is not in the scope of a binder, and R is any process. We have:

$$a(\widetilde{x}).\, C[R] \preceq a(\widetilde{x}).\, C[\mathbf{0}] \mid !R\{\widetilde{x}\}.$$

Proof Below V is the set $\{\widetilde{v} \mid \widetilde{v}$ is closed and $\widetilde{x} : \widetilde{v}\}$.

$$\begin{array}{rcl} a(\widetilde{x}).\, C[R] & \preceq & a(\widetilde{x}).\, C[\sum_{\widetilde{v}\in V} R\{\widetilde{v}/\widetilde{x}\}] \\ & \preceq & a(\widetilde{x}).\, (C[\mathbf{0}] \mid !\sum_{\widetilde{v}\in V} R\{\widetilde{v}/\widetilde{x}\}) \\ & \preceq & a(\widetilde{x}).\, (C[\mathbf{0}] \mid !\sum_{\widetilde{v}\in V} (R\{\widetilde{x}\})) \\ & \preceq & a(\widetilde{x}).\, (C[\mathbf{0}] \mid !(R\{\widetilde{x}\})) \\ & \preceq & a(\widetilde{x}).\, C[\mathbf{0}] \mid !(R\{\widetilde{x}\}) \end{array}$$

where we use, in the sequence, the laws:

- $P \preceq \sum_{\widetilde{v}\in V} P\{\widetilde{v}/\widetilde{x}\}$
- Lemma 9.4
- $P\{\widetilde{v}/\widetilde{x}\} \preceq P\{\widetilde{x}\}$
- $\sum_i P \preceq P$
- $a(\widetilde{x}).\, (P \mid Q) \preceq a(\widetilde{x}).\, P \mid Q$, if $\widetilde{x}$ not free in Q.

$\square$

Lemma 9.6 Suppose M, M_1, M_2, M_i $(i \in L)$ are in $\mathcal{P}_9$, and $\widetilde{x} : \widetilde{v}$; then also `if` $\widetilde{x} = \widetilde{v}$ `then` M_1 `else` M_2, and $M\{\widetilde{v}/\widetilde{x}\}$, and $\sum_{i\in L} M_i$ are in $\mathcal{P}_9$. $\square$

Lemma 9.7 Suppose $R \in \mathcal{P}_9$, where R has no if-then-else construct and uses only asynchronous outputs. Then also $R\{\widetilde{x}\} \in \mathcal{P}_9$, for all first-order names $\widetilde{x}$. Moreover, if R is a resource, then also $R\{\widetilde{x}\}$ is a resource.

Proof Induction on the structure of R. $\square$

Proof of termination of $\mathcal{P}_9$ By appealing to the transformations that we used to prove the correctness of $\mathcal{P}_1$ and $\mathcal{P}_8$, it suffices to consider normal forms without if-then-else constructs and with only asynchronous outputs. These are processes that are also in $\mathrm{A}\pi_{\Sigma}$. We prove that each such normal form Q (resource or process) in $\mathcal{P}_9$ is simulated by another process of $\mathcal{P}_9$ that has no nesting of free inputs. The latter processes is also in $\mathcal{P}_8$, because nesting of free inputs is the only difference between $\mathcal{P}_9$ and $\mathcal{P}_8$, and is in $\mathrm{A}\pi_{\Sigma}$.
We proceed by induction on the max depth n of nesting of free inputs. For $n = 1$ there is nothing to prove. For $n > 1$ consider an input $!a(\widetilde{x}).\, R$ in Q that has depth n (the case where the input is not replicated is similar). By induction, we can transform R into a process R' in which there is no nesting of free inputs, and such that R' is both in $\mathcal{P}_9$ and in $\mathrm{A}\pi_{\Sigma}$, and $R \preceq R'$.
We distinguish 2 cases, depending on whether a is a first-order or a higher-order link.

- Suppose that a is first order.

  We show how to eliminate any free input underneath the input at a. First, we consider the case of a replicated input. Thus the continuation R' is of the form $C[!b(\widetilde{y}).\,N]$, where C is a context in which the hole is not in the scope of a binder. Using Lemma 9.5,

  $$!a(\widetilde{x}).\,C[!b(\widetilde{y}).\,N] \preceq !a(\widetilde{x}).\,C[\mathbf{0}] \mid !(b(\widetilde{y}).\,N)\{\widetilde{x}\}\,.$$

  The case when the input at b is not replicated is similar.

- Suppose now that a is a higher-order name; according to the grammar of $\mathcal{P}_9$, all free inputs underneath a are first order and are not replicated. We consider one of these inputs and we show how to eliminate it. Thus let $R' = C[b(\widetilde{y}).\,N]$, where C is a context that does not bind the hole. We have, for $V \stackrel{\text{def}}{=} \{\widetilde{v} \mid \widetilde{v}$ is closed and $\widetilde{y} : \widetilde{v}\}$:

  $$\begin{array}{rcl} !a(\widetilde{x}).\,C[b(\widetilde{y}).\,N] & \preceq & !a(\widetilde{x}).\,C[b(\widetilde{y}).\,\sum_{\widetilde{v}\in V} N\{\widetilde{v}/\widetilde{y}\}] \\ & \preceq & !a(\widetilde{x}).\,C[b(\widetilde{y}).\,\mathbf{0} \mid \sum_{\widetilde{v}\in V} N\{\widetilde{v}/\widetilde{y}\}] \\ & \preceq & !a(\widetilde{x}).\,C[\sum_{\widetilde{v}\in V} N\{\widetilde{v}/\widetilde{y}\}] \mid !b(\widetilde{y}).\,\mathbf{0} \end{array}$$

  where we use

  - $P \preceq \sum_{\widetilde{v}\in V} [\widetilde{x} = \widetilde{v}] P\{\widetilde{v}/\widetilde{x}\}$
  - $\pi.\,P \preceq \pi \mid P$ if π does not bind P
  - Lemma 9.5

The final result of the transformation is a process in $\mathcal{P}_9$, which can be proved from Lemmas 9.6 and 9.7, and the fact that the grammar of resources $\mathcal{P}_9$ has parallel composition, and that resources include processes. Moreover, this process is also in $\mathrm{A}\pi_\Sigma$.
Iterating this procedure, all free inputs underneath the input at a can be eliminated, and the result is a process both in $\mathcal{P}_9$ and in $\mathrm{A}\pi_\Sigma$. ☐

9.5 Recursion

We show that we can add some forms of recursion, with first-order state.

Language $\mathcal{P}_{10}$

$$\begin{array}{rcl} I^{\mathrm{NF}}_{\widetilde{a}} & ::= & \ldots \;\big|\; (\mu X(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}})\lfloor\widetilde{v}\rfloor \\ M^{\mathrm{NF}} & ::= & \ldots \;\big|\; X\lfloor\widetilde{v}\rfloor \end{array}$$

where, in $\mu X(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}}$:

- $\widetilde{x}$ are first-order names;
- $I^{\mathrm{NF}}_{\widetilde{a}}$ does not have unguarded outputs and unguarded if-then-else.

The conditions on recursion are related to those for input nesting in $\mathcal{P}_9$: in both cases severe constraints are placed on higher-order state.

Recursion implicitly allows us to write forms of input nesting that go beyond those allowed in $\mathcal{P}_9$. For instance, consider the process

$$R \stackrel{\text{def}}{=} \mu X.\, a(y,z).\, \overline{y}\langle z\rangle.\, X$$

where a is a higher-order link. Unfolding the recursion we obtain

$$a(y, z).\, \overline{y}\langle z\rangle.\, a(y, z).\, \overline{y}\langle z\rangle.\, R\,,$$

which has a nesting of two free higher-order inputs – a structure not allowed in $\mathcal{P}_9$. Note also that R is not behaviourally equivalent to a replicated process $!a(y, z).\, R'$, for in the latter process the input at a is always available. (The recursion defining R has no parameters, but R has a state.)

To prove the correctness $\mathcal{P}_{10}$, we need some auxiliary π-calculus results. Let $\pi^\star$ be the π-calculus of Section 2, simply-typed, and with the addition of indexed sum and replication. If R is a process that contains at most the recursion variable X free in it, then $R[X]$ is the process obtained from R by replacing all calls $X\lfloor\widetilde{v}\rfloor$ with $\mathbf{0}$.

Lemma 9.8 (in $\pi^\star$) Suppose X is free in R and is not underneath a replication, $H \stackrel{\text{def}}{=} \mu Y(\widetilde{y}).\, M$, and $\mathsf{bn}(R) \cap \mathsf{fn}(H) = \emptyset$. Let also $V \stackrel{\text{def}}{=} \{\widetilde{v} \mid \widetilde{v}$ is closed and $\widetilde{y} : \widetilde{v}\}$. Then $R\{^{H}\!/X\} \preceq R[X] \mid !\sum_{\widetilde{v}\in V} H\lfloor\widetilde{v}\rfloor$.

Proof Induction on R. $\square$

Definition 9.9 A relation $\mathcal{R}$ on closed processes is a *simulation up to $\preceq$ and up to context* if $M\ \mathcal{R}\ N$ implies:

- whenever $M \xrightarrow{\alpha} M'$ there is N' such that $N \xrightarrow{\alpha} N'$ and there are a context C (with possibly infinitely-many holes) and tuples $\widetilde{M}$ and $\widetilde{N}$ (of possibly infinite length) such that
  - $M' \preceq C[\widetilde{M}]$,
  - $C[\widetilde{N}] \preceq N'$,
  - $\widetilde{M}\ \mathcal{R}\ \widetilde{N}$,

  where the grammar for the context C is:

$$C ::= [\cdot] \;\Big|\; C_1 \mid C_2 \;\Big|\; \sum_{i\in L} C_i \;\Big|\; !C\,.$$

$\square$

Lemma 9.10 If $\mathcal{R}$ is closed under substitutions (that is, $M\sigma\ \mathcal{R}\ N\sigma$, for all M, N, and closing substitutions σ with $M\ \mathcal{R}\ N$) and is a simulation up to $\preceq$ and up to context, then $\mathcal{R} \subseteq \preceq$.

Proof Similar to analogous proofs of soundness of 'up-to context' techniques [San98]. $\square$

Lemma 9.11 (in $\pi^\star$) Suppose R does not contain replications. Then $(\mu X(\widetilde{x}).\, R)\lfloor\widetilde{v}\rfloor \preceq\ !R[X]\{\widetilde{x}\}$.

Proof We prove that the relation $\mathcal{R}$ consisting of all pairs of closed processes of the form

$$((\mu X(\widetilde{x}).\, R)\lfloor\widetilde{v}\rfloor, !R[X]\{\widetilde{x}\})$$

where R does not contain replications, is a a simulation up to $\preceq$ and up to context. By Lemma 9.10, this proves the result. Take any pair of processes in $\mathcal{R}$, say (M_1, M_2) with $M_1 \stackrel{\text{def}}{=} (\mu X(\widetilde{x}).\, R)\lfloor\widetilde{v}\rfloor$, $M_2 \stackrel{\text{def}}{=} !R[X]\{\widetilde{x}\}$, and suppose $M_1 \xrightarrow{\alpha} M_1'$. This means that

$$R\{\widetilde{v}/\widetilde{x}\}\{\mu X(\widetilde{x}).\, R/X\} \xrightarrow{\alpha} M_1' .$$

Since recursions are guarded, this also implies that there is R' such that

$$\begin{array}{rcl} M_1' & = & R'\{\mu X(\widetilde{x}).\, R/X\} \\ R\{\widetilde{v}/\widetilde{x}\}[X] & \xrightarrow{\alpha} & R'[X] \end{array}$$

where $\mathsf{fn}(\mu X(\widetilde{x}).\, R) \cap \mathsf{bn}(R') = \emptyset$ and R' contains no replications. By Lemma 9.8,

$$M_1' \preceq R'[X] \mid !\sum_{\widetilde{v} \in V} (\mu X(\widetilde{x}).\, R)\lfloor\widetilde{v}\rfloor$$

where $V \stackrel{\text{def}}{=} \{\widetilde{v} \mid \widetilde{v} \text{ is closed and } \widetilde{x} : \widetilde{v}\}$. Moreover, since $R\{\widetilde{v}/\widetilde{x}\}[X] \preceq R[X]\{\widetilde{x}\}$,

$$R[X]\{\widetilde{x}\} \xrightarrow{\alpha} R'' \succeq R'[X] .$$

Hence

$$\begin{array}{rcl} M_2 = !R[X]\{\widetilde{x}\} & \xrightarrow{\alpha} & R'' \mid !R[X]\{\widetilde{x}\} \\ & \succeq & R'' \mid !!R[X]\{\widetilde{x}\} \\ & \succeq & R'' \mid !\sum_{\widetilde{v} \in V} !R[X]\{\widetilde{x}\} . \end{array}$$

Concluding, we have:

$$M_1 \xrightarrow{\alpha}\preceq R'[X] \mid !\sum_{\widetilde{v} \in V} (\mu X(\widetilde{x}).\, R)\lfloor\widetilde{v}\rfloor$$

and

$$M_2 \xrightarrow{\alpha}\succeq R'[X] \mid !\sum_{\widetilde{v} \in V} !R[X]\{\widetilde{x}\}$$

and now we are done, up to context. □

Proof of termination of $\mathcal{P}_{10}$ We show how to eliminate all recursions, in such a way that the resulting process still belongs to $\mathcal{P}_{10}$. The result therefore also belongs to $\mathcal{P}_9$. A resource $(\mu X(\widetilde{x}).\, I_{\widetilde{a}})\lfloor\widetilde{v}\rfloor$ of $\mathcal{P}_{10}$ is also a process of $\pi^\star$. Applying Lemma 9.11 we have

$$(\mu X(\widetilde{x}).\, I_{\widetilde{a}})\lfloor\widetilde{v}\rfloor \preceq !I_{\widetilde{a}}[X]\{\widetilde{x}\}$$

The process $!I_{\widetilde{a}}[X]\{\widetilde{x}\}$ may not be an input-guarded replication (for instance, the outermost operator in $!I_{\widetilde{a}}[X]\{\widetilde{x}\}$ may be a parallel composition), and therefore may not be in $\mathcal{P}_9$. The process can be transformed into one with only input-guarded replication reasoning by induction on $I_{\widetilde{a}}$. We consider all the cases below.

- $I_{\widetilde{a}} = !I'_{\widetilde{a}}$.
  Use the law $!!R \preceq !R$ and induction.

- $I_{\widetilde{a}} = I'_{\widetilde{a}} \mid I''_{\widetilde{a}}$ or $I_{\widetilde{a}} = I'_{\widetilde{a}} + I''_{\widetilde{a}}$
  Use the laws $I'_{\widetilde{a}} + I''_{\widetilde{a}} \preceq I'_{\widetilde{a}} \mid I''_{\widetilde{a}}$ and $!(I'_{\widetilde{a}} \mid I''_{\widetilde{a}}) \preceq !I'_{\widetilde{a}} \mid !I''_{\widetilde{a}}$, plus induction.
- Otherwise $I_{\widetilde{a}} = M$, for some M. Since M cannot have unguarded outputs or conditionals, $M \preceq \mathbf{0}$.

No other cases are possible, because recursions are guarded and because recursions in $\mathcal{P}_{10}$ have no unguarded outputs or conditionals. □

9.6 Restrictions on resources

Language $\mathcal{P}_{11}$

$$M^{\mathrm{NF}} \quad ::= \quad \ldots \;\Big|\; \boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}})$$

Proof of termination of $\mathcal{P}_{11}$ Any term $\boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}})$ obtained with the new production can be replaced by a term $\boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}} \mid \mathbf{0})$ obtained with the production

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

This way a term in $\mathcal{P}_{11}$ can be transformed into a term in $\mathcal{P}_{10}$ and the latter simulates the former. □

10 Proof of Theorem 4.4

Here is the complete grammar of the normal forms of the final language $\mathcal{P}_{11}$.

Resources

$$
\begin{array}{rcll}
I^{\mathrm{NF}}_{\widetilde{a}} & ::= & !\mathcal{I}^{\widetilde{a}} & \\
& | & \mathcal{I}^{\widetilde{a}} & \\
& | & M^{\mathrm{NF}} & \\
& | & I^{\mathrm{NF}}_{\widetilde{a}} \mid I^{\mathrm{NF}}_{\widetilde{a}} & \\
& | & I^{\mathrm{NF}}_{\widetilde{a}} + I^{\mathrm{NF}}_{\widetilde{a}} & \\
& | & \overline{b}\langle\widetilde{v}\rangle.\, I^{\mathrm{NF}}_{\widetilde{a}} & \\
& | & \texttt{if } B \texttt{ then } I^{\mathrm{NF}}_{\widetilde{a}} \texttt{ else } I^{\mathrm{NF}}_{\widetilde{a}} & \\
& | & (\mu X(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}})\lfloor\widetilde{v}\rfloor & \\
\mathcal{I}^{\widetilde{a}} & ::= & a_i(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}} & \text{with } a_i \in \widetilde{a} \text{ and } a_j \notin \mathtt{on}(I^{\mathrm{NF}}_{\widetilde{a}}) \text{ for all } j \geq i
\end{array}
$$

Processes

$$
\begin{array}{rcl}
M^{\mathrm{NF}} & ::= & \boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}} \mid M^{\mathrm{NF}}) \\
& | & \boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}}) \\
& | & \sum_{i\in L} M^{\mathrm{NF}}_i \\
& | & M^{\mathrm{NF}} \mid M^{\mathrm{NF}} \\
& | & \overline{v}\langle\widetilde{w}\rangle.\, M^{\mathrm{NF}} \\
& | & \mathbf{0} \\
& | & \texttt{if } B \texttt{ then } M^{\mathrm{NF}} \texttt{ else } M^{\mathrm{NF}} \\
& | & X\lfloor\widetilde{v}\rfloor
\end{array}
$$

with the conditions:

- in an higher-order input $a(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}}$, the continuation $I^{\mathrm{NF}}_{\widetilde{a}}$ does not contain free higher-order inputs, and does not contain free replicated first-order inputs;
- the parameters of a recursion are first order;
- the body of a recursion has no unguarded outputs and no unguarded if-then-else.

We prove that the language $\mathcal{P}$ of Theorem 4.4 is contained in the resources of $\mathcal{P}_{11}$. The following modifications to the grammar of $\mathcal{P}_{11}$ may only decrease the set of defined resources.

1. Remove the production $M^{\mathrm{NF}} ::= \boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}} \mid M^{\mathrm{NF}})$.
2. Replace indexed sum with binary sum.

3. Having the production $I^{\mathrm{NF}}_{\widetilde{a}} ::= M^{\mathrm{NF}}$, we can add some of the productions for processes to the productions for resources, in particular

$$I^{\mathrm{NF}}_{\widetilde{a}} ::= \mathbf{0} \;\Big|\; X\lfloor\widetilde{v}\rfloor \;\Big|\; \boldsymbol{\nu}\widetilde{a}\, I^{\mathrm{NF}}_{\widetilde{a}} \,.$$

4. Remove the production

$$I^{\mathrm{NF}}_{\widetilde{a}} ::= M \,.$$

5. Remove the production

$$I^{\mathrm{NF}}_{\widetilde{a}} ::= !\mathcal{I}^{\widetilde{a}} \,.$$

At the end of all these steps, the grammar for resources is as follows

$$\begin{array}{rcl}
I^{\mathrm{NF}}_{\widetilde{a}} & ::= & a_i(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}} \\
& \Big| & I^{\mathrm{NF}}_{\widetilde{a}} \mid I^{\mathrm{NF}}_{\widetilde{a}} \\
& \Big| & I^{\mathrm{NF}}_{\widetilde{a}} + I^{\mathrm{NF}}_{\widetilde{a}} \\
& \Big| & \overline{b}\langle\widetilde{v}\rangle.\, I^{\mathrm{NF}}_{\widetilde{a}} \\
& \Big| & \ \texttt{if}\ B\ \texttt{then}\ I^{\mathrm{NF}}_{\widetilde{a}}\ \texttt{else}\ I^{\mathrm{NF}}_{\widetilde{a}} \\
& \Big| & (\mu X(\widetilde{x}).\, I^{\mathrm{NF}}_{\widetilde{a}})\lfloor\widetilde{v}\rfloor \\
& \Big| & \boldsymbol{\nu}\widetilde{a}\,(I^{\mathrm{NF}}_{\widetilde{a}}) \\
& \Big| & X\lfloor\widetilde{v}\rfloor \\
& \Big| & \mathbf{0}
\end{array}$$

subject to the conditions on nesting of inputs and recursion as before. If we now replace the symbols $I^{\mathrm{NF}}_{\widetilde{a}}$ with M we have the same grammar and the same conditions as in the processes of $\mathcal{P}$.

11 Counterexamples for nested inputs and recursion

The counterexamples in this section show the importance of the clauses (2) and (3) of Condition 4.2, on nesting of inputs and on state, for the termination of the processes of $\mathcal{P}$. For readability we use input-guarded replications rather than recursion. Recall that an input-guarded replication $!a(\widetilde{x}).\,M$ can be written with recursion as $\mu X.\,a(\widetilde{x}).\,(X \mid M)$. First we show why nesting of higher-order inputs can be dangerous. Consider the process

$$\begin{array}{rl}
R_1 \stackrel{\mathrm{def}}{=} & !a(x,-).\,\Big(\overline{x} \mid a(-,p).\,\overline{p}\langle x\rangle\Big) \\
& \mid\, !c.\,\boldsymbol{\nu}p\,(\overline{a}\langle -,p\rangle.\,p(x).\,\overline{a}\langle x,-\rangle) \\
& \mid\, \overline{a}\langle c,-\rangle
\end{array}$$

(We use a hyphen in places where the value emitted or received is not important; moreover we abbreviate $\overline{a}\langle v\rangle.\,R$ and $a(x).\,R$ as $\overline{a}.\,R$ and $a.\,R$ when v and x are of unit type.) This process is typable in the simply-typed π-calculus, with these types for the names, abbreviating $\sharp\,\langle\mathtt{unit}\rangle$ as T

$$\begin{array}{rl} x, c: & T \\ p: & \sharp\,\langle T\rangle \\ a: & \sharp\,\langle T \times \sharp\,\langle T\rangle\rangle \end{array}$$

R_1 has one divergent computation; letting $A \stackrel{\text{def}}{=} !a(x,-).\,(\overline{x} \mid a(-,p).\,\overline{p}\langle x\rangle)$ and $V \stackrel{\text{def}}{=} !c.\,\boldsymbol{\nu} p\,\overline{a}\langle -,p\rangle.\,p(x).\,\overline{a}\langle x,-\rangle$, we have:

$$\begin{array}{rlr} R_1 & \xrightarrow{\tau}\equiv \;\; \overline{c} \mid a(-,p).\,\overline{p}\langle c\rangle \mid A \mid V & [a\langle c,-\rangle] \\ & \xrightarrow{\tau}\equiv \;\; a(-,p).\,\overline{p}\langle c\rangle \mid \boldsymbol{\nu} p\,(\overline{a}\langle -,p\rangle.\,p(x).\,\overline{a}\langle x,-\rangle \mid A \mid V & [c] \\ & \xrightarrow{\tau} \;\; \boldsymbol{\nu} p\,(\overline{p}\langle c\rangle \mid p(x).\,\overline{a}\langle x,-\rangle) \mid A \mid V & [a\langle -,p\rangle] \\ & \xrightarrow{\tau}\equiv \;\; \overline{a}\langle c,-\rangle \mid A \mid V & [pc] \\ & \sim \;\; R_1 & \end{array}$$

where the column on the right indicates the action performed in the reduction.

In R_1, the two inputs at a are not 'simply nested': the second input runs in parallel with an output that depends on the first input. Replacing R_1 with R_2 below we obtain a counterexample that uses simple nesting:

$$\begin{array}{rl} R_2 \stackrel{\text{def}}{=} & !a(x,-).\,a(-,p).\,\overline{p}\langle x\rangle \\ & \mid\, !c.\,\boldsymbol{\nu} p\,(\overline{a}\langle -,p\rangle.\,p(x).\,(\overline{x} \mid \overline{a}\langle x,-\rangle)) \\ & \mid\, \overline{a}\langle c,-\rangle \mid \overline{c} \end{array}$$

The types for the names of R_2 are the same as those of R_1. Here is a divergent computation of R_2, where $A \stackrel{\text{def}}{=} !a(x,-).\,a(-,p).\,\overline{p}\langle x\rangle$ and $V \stackrel{\text{def}}{=} !c.\,\boldsymbol{\nu} p(\overline{a}\langle -,p\rangle.\,p(x).\,(\overline{x} \mid \overline{a}\langle x,-\rangle))$:

$$\begin{array}{rlr} R_2 & \xrightarrow{\tau} \;\; a(-,p).\,\overline{p}\langle c\rangle \mid A \mid V \mid \overline{c} & [\overline{a}\langle c,-\rangle] \\ & \xrightarrow{\tau}\equiv \;\; a(-,p).\,\overline{p}\langle c\rangle \mid \boldsymbol{\nu} p\,(\overline{a}\langle -,p\rangle.\,p(x).\,(\overline{x} \mid \overline{a}\langle x,-\rangle)) \mid A \mid V & [\overline{c}] \\ & \xrightarrow{\tau} \;\; \boldsymbol{\nu} p\,(\overline{p}\langle c\rangle \mid p(x).\,(\overline{x} \mid \overline{a}\langle x,-\rangle)) \mid A \mid V & [\overline{a}\langle -,p\rangle] \\ & \xrightarrow{\tau}\equiv \;\; \boldsymbol{\nu} p\,((\overline{c} \mid \overline{a}\langle c,-\rangle)) \mid A \mid V & [\overline{p}\langle c\rangle] \\ & \sim \;\; R_2 & \end{array}$$

The same counterexamples can be repeated using nesting of free inputs at two different higher-order link. For instance, replacing the second input at a with an input at a name b, R_2 becomes:

$$\begin{array}{l} !a(x,-).\,b(-,p).\,\overline{p}\langle x\rangle \\ \mid\, !c.\,\boldsymbol{\nu} p\,(\overline{b}\langle -,p\rangle.\,p(x).\,(\overline{x} \mid \overline{a}\langle x,-\rangle)) \\ \mid\, \overline{a}\langle c,-\rangle \mid \overline{c} \end{array} \qquad (5)$$

The next example has nesting of free higher-order inputs in which the second input, rather than the first, is replicated. Name z has type $\sharp\,\langle\mathtt{unit}\rangle$, and a has type $\sharp\,\langle\sharp\,\langle\mathtt{unit}\rangle\rangle$.

$$\begin{array}{rl} R_3 \stackrel{\text{def}}{=} & a(z).\,!a(-).\,\overline{z} \\ & \mid \boldsymbol{\nu} z\,\overline{a}\langle z\rangle.\,(\overline{a}\langle -\rangle \mid !z.\,\overline{a}\langle -\rangle) \end{array}$$

R_3 reduces to a process bisimilar to

$$R_3' \stackrel{\text{def}}{=} \boldsymbol{\nu} z\,(!a(-).\,\overline{z} \mid !z.\,\overline{a}\langle -\rangle \mid \overline{a}\langle -\rangle)$$

in which the mutual recursion in the replications at a and z causes a loop.

We now show why free first-order inputs underneath higher-order inputs cannot be replicated. We use first-order links b, c, with type $\sharp\,\langle\mathtt{unit}\rangle$, and the higher-order link a with type $\sharp\,\langle\sharp\,\langle\mathtt{unit}\rangle\rangle$. The process

$$R_4 \stackrel{\mathrm{def}}{=} !a(x).\,!c.\,\overline{x} \mid \boldsymbol{\nu} b\,(!b.\,\overline{c} \mid \overline{a}\langle b\rangle \mid \overline{c})$$

can reduce to a process bisimilar to

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

that has the same kind of loop as R_3'.

The above examples of divergences can be repeated using recursive definitions with higher-order state instead of input nesting. For instance, writing A for the recursive definition

$$\mu X(x,p).\,(\overline{x} \mid a(y,q).\,(\overline{q}\langle x\rangle \mid X\lfloor y,q\rfloor))\,,$$

R_1 can be replaced by the process

$$\begin{aligned} R_5 \stackrel{\mathrm{def}}{=} \quad & A\lfloor c,-\rfloor \\ & \mid !c.\,\boldsymbol{\nu} p\,\overline{a}\langle -,p\rangle.\,p(x).\,\overline{a}\langle x,-\rangle \end{aligned}$$

that, in 4 steps, reduces to a process of the form $R \mid R_5$, for some R (up to bisimilarity).

12 Non termination in the Join calculus

The Join calculus [FG96, Fou98] is a variant of the π-calculus in which recursion, input, and restriction are combined in the *join* construct. An example of join is

$$\mathtt{def}\ a(x)\ \&\ b(y)\ \triangleright\ M\ \mathtt{in}\ N\,, \tag{6}$$

that introduces two new names a and b. Its effect is similar to that of an input-guarded replication: a copy of M is activated, with appropriate arguments for x and y, whenever an output at a and an output at b become available in N.

In Join, free inputs and recursion do not exist, therefore the clauses (2) and (3) of Condition 4.2 are trivially satisfied. Moreover, condition (1) of the same definition simply forbids recursive patterns (i.e., a link should not be used in output underneath the unique input occurrence of that link): this is the *non-recursive Join*. It is therefore reasonable to wonder whether the simply-typed non-recursive Join only consists of terminating processes.[2]

Counterexample (5) can be easily adapted to Join to show that the answer to the above question is negative. The reason is that, intuitively, a join pattern like (6) is similar to a process $N \mid !a(x).\,b(y).\,M$, which has nesting of inputs. To have only terminating processes, we therefore need a further condition. A sufficient condition, obtained as a corollary of Theorem 4.4, is that in any join pattern such as (6) at most one of the names a, b is higher order.

[2] Fournet [Fou98] has showed that Join can be encoded into the non-recursive Join, but the encoding needs recursive types, therefore cannot be used to answer the question.

13 Conclusions and future work

In this paper we have proved termination for the simply-typed (localised) π-calculus with first-order values, and subject to three syntactic conditions that constrain recursive inputs and state. We have shown, by means of counterexamples, the need for the conditions as well as the need of types. To prove the termination of this language, $\mathcal{P}$, 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.

[Fou98] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, 1998.

[Gan80a] R. O. Gandy. An early proof of normalization by A.M. Turing. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 453–455. Academic Press, 1980.

[Gan80b] R. O. Gandy. Proof of strong normalisation. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[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.

[JM98] F. Joachimski and R. Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. To appear in *Archive for Mathematical Logic*, 1998.

[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.

[Lév77] J.-J. Lévy. *Reductions Correctes et Optimales dans le Lambda-Calcul*. Ph.D. thesis, Université de Paris, 1977.

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

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

[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.

[San98] D. Sangiorgi. On the bisimulation proof method. *Journal of Mathematical Structures in Computer Science*, 8:447–479, 1998.

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

[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, 2001. IEEE Computer Society.

[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.