ENSURING TERMINATION BY TYPABILITY

Yuxin Deng[1] and Davide Sangiorgi[2]
[1]*INRIA and University Paris 7,* [2]*University of Bologna*

Abstract A term terminates if all its reduction sequences are of finite length. We show four type systems that ensure termination of well-typed π-calculus processes. The systems are obtained by successive refinements of the types of the simply typed π-calculus. For all (but one of) the type systems we also present upper bounds to the number of steps well-typed processes take to terminate. The termination proofs use techniques from term rewriting systems.

We show the usefulness of the type systems on some non-trivial examples: the encodings of primitive recursive functions, the protocol for encoding separate choice in terms of parallel composition, a symbol table implemented as a dynamic chain of cells.

Keywords: Concurrency, the π-calculus, type system, termination

1. Introduction

A term terminates if all its reduction sequences are of finite length. As far as programming languages are concerned, termination means that computation in programs will eventually stop. In computer science termination has been extensively investigated in term rewriting systems [5, 3] and λ-calculi [7, 2] (where strong normalization is a synonym more commonly used). Termination has also been discussed in process calculi, notably the π-calculus [12, 17], a formalism widely used to address issues related to concurrency.

Indeed, termination is interesting in concurrency. For instance, if we interrogate a process, we may want to know that an answer is eventually produced (termination alone does not guarantee this, but termination would be the main ingredient in a proof). Similarly, when we load an applet we would like to know that the applet will not run for ever on our machine, possibly absorbing all the computing resources (a 'denial of service' attack). In general, if the lifetime of a process can be infinite, we may want to know that the process does not remain alive simply because of non-terminating internal activity, and that, therefore, the process will eventually accept interactions with the environment.

Languages of terminating processes are proposed in [19] and [16]. In both cases, the proofs of termination make use of logical relations, a well-known technique from functional languages. The languages of terminating processes so obtained are however rather 'functional', in that the structures allowed are similar to those derived when encoding functions as processes. In particular, the languages are very restrictive on nested inputs (that is, the possibility of having free inputs underneath other inputs), and recursive inputs (that is, replications $!a(x).P$ in which the body P can recursively call the guard a of the replication). Such patterns are entirely forbidden in [19]; nested inputs are allowed in [16] but in a very restricted form. For example, the process

$$a(x).!b.\bar{x} \mid \bar{a}c \tag{1}$$

is legal neither for [19] nor for [16]. The restrictions in [19, 16] actually rule out also useful functional processes, for instance

$$F \equiv !a(n,b).\ \texttt{if}\ n = 1\ \texttt{then}\ \bar{b}\langle 1\rangle\ \texttt{else}\ \nu c(\bar{a}\langle n-1, c\rangle \mid c(m).\bar{b}\langle m * n\rangle) \tag{2}$$

which represents the factorial function.

In this paper, we consider several type systems and well-typed processes under each system are ensured to terminate. First, in Section 3, we present a core type system, which adds level information to the types of the simply typed π-calculus. Then, in Sections 4 to 6 we show three refinements of the core system. Nested inputs and recursive inputs are the main patterns we focus on. For all the type systems (except for the second one, which can capture primitive recursive functions) we also present upper bounds to the number of steps well-typed processes take to terminate. Such bounds depend on the structures of the processes and on the types of the names in the processes. We show the usefulness of the type systems on some non-trivial examples: the encodings of primitive recursive functions, the protocol for encoding separate choice in terms of parallel composition from [13, 17], a symbol table implemented as a dynamic chain of cells from [8, 15].

Roughly, for each type system to prove termination we choose a measure which decreases after finite steps of reductions. To compare two measures, we exploit *lexicographic* and *multiset orderings*, well-known techniques in term rewriting systems [5, 4]. For the core type system, the measure is just a vector recording, for each level, the number of outputs (unguarded by replicated inputs) at channels with that level in the type. For the extended type systems, the ideas are similar, but the measures become more sophisticated since we allow them to decrease after some finite (unknown and variable) number of reductions, up-to some commutativities of reductions and process manipulations.

2. The simply typed π-calculus

We begin with a brief overview of the simply typed π-calculus [17]. In this work we only study type systems $\grave{a}\ la\ Church$, and each name is assigned a type a priori. We write $x : T$ to mean that the name x has type T. A judgment $\vdash P$ says that P is a well-typed process, and $\vdash v : T$ says that v is a well-typed value of type T. The syntax of types and processes as well as the typing rules are shown in Table 1. We use the usual constructors of monadic π-calculus. Recall that in the input prefix $a(x)$ and output prefix $\bar{a}v$, name a is the *subject* and x, v are the *objects* of the prefixes. We assume α-conversion implicitly in order to avoid name capture and keep the uniqueness of every bound name. The transition rules are standard, in the early style.

Table 1. Processes, types and typing rules of the simply typed π-calculus

S, T	$::=$	$V \mid L$	types
V	$::=$	$L \mid$ `bool` $\mid$ `Nat`	value types
L	$::=$	$\sharp V$	link types
v, w	$::=$	$x \mid true, false \mid 0, 1, 2, \cdots$	values
P, Q	$::=$	$0 \mid a(x).P \mid \bar{a}v.P \mid P \mid P \mid P + P \mid \nu aP \mid !a(x).P$	processes

$$\text{T-in}\ \frac{\vdash a : \sharp T \quad x : T \quad \vdash P}{\vdash a(x).P} \qquad \text{T-out}\ \frac{\vdash a : \sharp T \quad \vdash v : T \quad \vdash P}{\vdash \bar{a}v.P} \qquad \text{T-nil}\ \frac{}{\vdash 0}$$

$$\text{T-par}\ \frac{\vdash P \quad \vdash Q}{\vdash P \mid Q} \qquad \text{T-sum}\ \frac{\vdash P \quad \vdash Q}{\vdash P + Q} \qquad \text{T-res}\ \frac{a : L \quad \vdash P}{\vdash \nu aP}$$

$$\text{T-rep}\ \frac{\vdash a : \sharp T \quad x : T \quad \vdash P}{\vdash !a(x).P}$$

For simplicity we only consider two basic types: `bool`, for boolean values, and `Nat`, for natural numbers. Values of basic types are said to be of first-order because, unlike channels (names of link type), they cannot carry other values. We also assume some basic operations on first-order values. For example, we may use addition $(n + m)$, subtraction $(n - m)$, multiplication $(n * m)$ for `Nat` expressions. To avoid being too specific, we do not give a rigid syntax and typing rules for first-order expressions. We just assume a separate mechanism for evaluating expressions of type `Nat`.

Next we introduce some notations about vectors, partial orders and multisets. We write $\mathbf{0}_i$ as the abbreviation of a vector $\langle n_k, \cdots, n_1 \rangle$ where $n_i = 1$ and $n_j = 0$ for all $j \neq i$, and $\mathbf{0}$ for a vector with all 0 components. The binary operator *sum* can be defined between two vectors. Let $\varphi_1 \equiv \langle n_k, n_{k-1}, \cdots, n_1 \rangle$, $\varphi_2 \equiv \langle m_l, m_{l-1}, \cdots, m_1 \rangle$ and $k \geq l$. First we extend the length of φ_2 to k by inserting $(k - l)$ zeros to the left of m_l to get an equivalent vector φ_2'. Then we do pointwise addition over two vectors with equal length. We also define an order between two vectors of equal length as follows: $\langle n_k, n_{k-1}, \cdots, n_1 \rangle \prec \langle m_k, m_{k-1}, \cdots, m_1 \rangle$ iff $\exists i \leq k$ with $n_j = m_j$ for $j > i$ and $n_i < m_i$.

Let S be a set and $>$ a strict partial order on S. Following [1], we write a multiset $\mathcal{M}$ *over* S in the form $\mathcal{M} = [x_1, \ldots, x_n]$, where $x_i \in S$ for $1 \leq$

$i \leq n$; we use $(\mathcal{M} \uplus \mathcal{M}')$ for the *union* of $\mathcal{M}$ and $\mathcal{M}'$, and write $>_{mul}$ for the multiset ordering (on multisets over S) induced by $>$. A multiset becomes smaller, in the sense of $>_{mul}$, by replacing one or more of its elements by any finite number (including zero) of smaller elements. It can indeed be shown that $>_{mul}$ is well-founded [1].

In this paper we restrict our attention to the termination property of closed processes, i.e., processes without free names of `bool` or `Nat` types.

3. The core system: the simply typed π-calculus with levels

Our first type system for termination is obtained by making mild modifications to the types and typing rules of the simply typed π-calculus. We assign a level, which is a natural number, to each channel name and incorporate it into the type of the name. Now the syntax of link type takes the new form:

$$\begin{array}{lcll} L & ::= & \sharp^n V & \text{link types} \\ n & ::= & 1, 2, \cdots & \text{levels} \end{array}$$

The typing rules in Table 1 are still valid (by obvious adjustments for link types), with the exception of rule T-rep, which takes the new form:

$$\text{T-rep}\ \frac{\vdash a : \sharp^n T \quad x : T \quad \vdash P \quad \forall b \in os(P), lv(b) < n}{\vdash !a(x).P}$$

where $os(P)$ is a set collecting all names in P which appear as subjects of those outputs that are not underneath any replicated input (we say this kind of outputs are *active*). The function $lv(b)$ calculates the *level* of channel b from its type. If $b : \sharp^n T$ then $lv(b) = n$.

The purpose of using levels is to rule out recursive inputs as, for instance, in the process $\bar{a} \ |!a.\bar{b} \ |!b.\bar{a}$ the two replicated processes can call each other thus producing a divergence. It is ruled out by our type system because $!a.\bar{b}$ requires $lv(a) > lv(b)$ while $!b.\bar{a}$ requires $lv(b) > lv(a)$. With levels, we also have a concise way of handling nested inputs. For example, let $a : \sharp^1\sharp^1\mathtt{Nat}, b : \sharp^2\mathtt{Nat}, c : \sharp^1\mathtt{Nat}$, then process (1) is well-typed. We call $\mathcal{T}$ this type system and write $\mathcal{T} \vdash P$ to mean that P is a well-typed process under $\mathcal{T}$. The subject reduction theorem of the simply typed π-calculus can be easily adapted to $\mathcal{T}$.

To prove the termination property of well-typed processes, we need to define a measure for processes. The measure that we choose in this section is the *weight*, $wt(P)$, of a process P. It is a vector determined by the levels of subject names which appear in active outputs. Specifically,

$$\begin{array}{rcl@{\qquad}rcl} wt(0) & = & \mathbf{0} & wt(\bar{a}v.P) & = & wt(P) + \mathbf{0}_{lv(a)} \\ wt(!a(x).P) & = & \mathbf{0} & wt(P \mid Q) & = & wt(P) + wt(Q) \\ wt(a(x).P) & = & wt(P) & wt(P + Q) & = & max\{wt(P), wt(Q)\} \\ wt(\nu a P) & = & wt(P) & & & \end{array}$$

In the next theorem, clause (i) says that weight is a good measure because it decreases at each reduction step, which leads naturally to the termination property of well-typed processes (clause (ii)), by the well-foundedness of weight.

THEOREM 1 *(i) Suppose* $\mathcal{T} \vdash P$ *and* $P \xrightarrow{\tau} P'$, *then* $wt(P') \prec wt(P)$.
(ii) If $\mathcal{T} \vdash P$, *then* P *terminates.*

It is easy to see that the weight of a process gives us a bound on the time that the process takes to terminate.

PROPOSITION 2 *Let* n *and* k *be the size and the highest level in a well-typed process* P, *respectively. Then* P *terminates in polynomial time* $\mathcal{O}(n^{k-1})$.

As a consequence we are not able to encode the simply typed λ-calculus, according to the known result that computing the normal form of a non-trivial λ-term cannot be finished in elementary time [18, 9]. We shall see in the next section an extension of $\mathcal{T}$ that makes it possible to encode all primitive recursive functions (some of which are not representable in the simply typed λ-calculus).

4. Allowing limited forms of recursive inputs

The previous type system allows nesting of inputs but forbids all forms of recursive inputs. In this and the following sections we study how to relax this restriction.

4.1 The type system

Let us consider a simple example. Process P below has a recursive input: underneath the replication at a there are two outputs at a itself. However, the values emitted at a are "smaller" than the value received. This, and the fact that the "smaller than" relation on natural numbers is well-founded, ensures the termination of P. In other words, the termination of P is ensured by the relation among the subjects and objects of the prefixes – rather the subjects alone as it was in the previous system.

$$\begin{array}{rl} P \equiv & \bar{a}\langle 10\rangle \mid !a(n).\,\texttt{if}\ n > 0\ \texttt{then}\ (\bar{a}\langle n-1\rangle \mid \bar{a}\langle n-1\rangle) \\ \longrightarrow & \bar{a}\langle 9\rangle \mid \bar{a}\langle 9\rangle \mid !a(n).\,\texttt{if}\ n > 0\ \texttt{then}\ (\bar{a}\langle n-1\rangle \mid \bar{a}\langle n-1\rangle) \end{array}$$

For simplicity, the only well-founded values that we consider are naturals. But the arguments below apply to any data type on whose values a well-founded relation can be defined.

We use function $out(P)$ to extract all active outputs in P. The definition is similar to that of $os(P)$ in Section 3. The main difference is that each element of $out(P)$ is a complete output prefix, including both subject and object names. For example, we have $out(!a(x).P) = \emptyset$ and $out(\bar{a}v.P) = \{\bar{a}v\} \cup out(P)$.

In the typing rule, in any replication $!a(x).P$ we compare the active outputs in P with the input $a(x)$ using the relation $\lhd$ below. We have that $\bar{b}v \lhd a(x)$ holds in two cases: (1) b has a lower level than a; (2) b and a have the same level, but the object v of b is provably smaller than the object x of a. For this, we assume a mechanism for evaluating (possibly open) integer expressions that allows us to derive assertions such as $x - i < x$ if $i > 0$, or $x + 2 - 3 + 1 \leq x$. We adopt an eager reduction strategy, thereby the expression in an output is evaluated before the output fires.

DEFINITION 3 *Let $a : \sharp^n S$ and $b : \sharp^m T$. We write $\bar{b}v \lhd a(x)$ if one of the two cases holds: (i) $m < n$; (ii) $m = n,\ S = T = \mathtt{Nat}$ and $v < x$.*

By substituting the following rule for T-rep in Table 1, we get the extended type system $\mathcal{T}'$. The second condition in the definition of $\lhd$ allows us to include some recursive inputs and gives us the difference from $\mathcal{T}$.

$$\textsf{T-rep}\qquad \frac{\vdash a : \sharp^n T \quad x : T \quad \vdash P \quad \forall \bar{b}v \in \mathit{out}(P'), \bar{b}v \lhd a(x)}{\vdash !a(x).P}$$

The termination property of $\mathcal{T}'$ can also be proved with a schema similar to the proof in last section. However, the details are more complex because we need to be clear about how the first-order values in which we are interested evolve with the reduction steps. So we use a measure which records, for each output prefix, the value of the object and the level information of the subject. More precisely, the measure is a *compound vector*, which consists of two parts: the *Nat-multiset* and the weight, corresponding to each aspect of information that we wish to record.

To a given process P and level i, with $0 < i \leq k$, we assign a unique Nat-multiset $\mathcal{M}_{P,i} = [n_1, \cdots, n_l]$, with $n_j \in \mathbb{N} \cup \{\infty\}$ for all $j \leq l$. (Here we consider ∞ as the upper bound of the infinite set $\mathbb{N}$.) Intuitively, this multiset is obtained as follows. For each active output $\bar{b}v$ in P with $lv(b) = i$, there are two possibilities. If v is a constant value ($v \in \mathbb{N}$), then v is recorded in $\mathcal{M}_{P,i}$. If v contains variables of type $\mathtt{Nat}$, then a ∞ is recorded in $\mathcal{M}_{P,i}$. For instance, suppose $a : \sharp^3\mathtt{Nat}, b : \sharp^2\mathtt{Nat}, c : \sharp^1\mathtt{Nat}$ and $P \equiv \bar{a}\langle 1\rangle \mid \bar{a}\langle 1\rangle \mid \bar{b}\langle 2\rangle \mid !a(n).\bar{b}\langle n+1\rangle \mid b(n).\bar{c}\langle n\rangle$, then $\mathcal{T}' \vdash P$ and there are three Nat-multisets: $\mathcal{M}_{P,3} = [1,1],\ \mathcal{M}_{P,2} = [2]$ and $\mathcal{M}_{P,1} = [\infty]$. We define an operator $\searrow$ to combine a set of Nat-multisets $\{\mathcal{M}_{Q,i} \mid 0 < i \leq k\}$ with the weight of Q (as defined in the previous section), $wt(Q) = \langle n_k, \cdots, n_1\rangle$, so as to get a *compound vector* $t_Q = \langle(\mathcal{M}_{Q,k}; n_k), \cdots, (\mathcal{M}_{Q,1}; n_1)\rangle$. The order $\prec$ is extended to compound vectors as follows:

DEFINITION 4 *Suppose $t_P = \langle(v_k), \cdots, (v_1)\rangle$ and $t_Q = \langle(u_k), \cdots, (u_1)\rangle$, where $v_i = \mathcal{M}_{P,i}; n_i$ and $u_i = \mathcal{M}_{Q,i}; n'_i$ for $0 < i \leq k$.*
(i) $v_i \prec u_i$ if $\mathcal{M}_{P,i} <_{mul} \mathcal{M}_{Q,i} \vee (\mathcal{M}_{P,i} = \mathcal{M}_{Q,i} \wedge n_i < n'_i)$
(ii) $t_P \prec t_Q$ if $\exists i \leq k, v_j = u_j$ for $j > i$ and $v_i \prec u_i$

THEOREM 5 *(i) If $\mathcal{T}' \vdash P$ and $P \xrightarrow{\tau} P'$ then $t_{P'} \prec t_P$.*
(ii) If $\mathcal{T}' \vdash P$ then P terminates.

The measure used here is more powerful than that in Section 3. With weights, we only prove the termination of processes which always terminate in polynomial time. By using compound vectors, however, as we shall see immediately, we are able to capture the termination property of some processes which terminate in time $\mathcal{O}(f(n))$, where $f(n)$ a is primitive recursive function. For example, we can write a process to encode the *repeated exponentiation*, where $E(0) = 1$, $E(n+1) = 2^{E(n)}$. Once received a number n, the process does internal computation in time $\mathcal{O}(E(n))$ before sending out its result.

4.2 Example: primitive recursive functions

For simplicity of presentation, we have concentrated mainly on monadic communication. It is easy to extend our calculus and type system to allow polyadic communications and an if-then-else construct, which are needed in this example.

PROPOSITION 6 *All primitive recursive functions can be represented as terminating processes in the π-calculus.*

We represent each function $f(\widetilde{x})$ as a process with replicated inputs like $!p(\widetilde{x}, r).R$, where name p has type $T_{m,n} = \sharp^m(\widetilde{\mathtt{Nat}}, \sharp^n\mathtt{Nat})$ with $m > n$. After receiving via p some arguments $\widetilde{x}$ and a return channel r, process R does some computation, and finally the result is delivered at r. This style of encoding is a straightforward adaptation of Milner's encoding of λ-terms into π-processes [10]. Furthermore, the resulting processes are well typed in $\mathcal{T}'$. For instance, the process F in (2) is typable if we give name a the type $\sharp^2(\mathtt{Nat}, \sharp^1\mathtt{Nat})$. By contrast, the encoding of functions that are not primitive recursive may not be typable. An example is Ackermann's function.

5. Asynchronous names

In this section we start a new direction for extending our core type system of Section 3: we prove termination by exploiting the structure of processes instead of the well-foundedness of first-order values. The goal of the new type systems (in this and in the next section) is to gain more flexibility in handling nested inputs. In the previous type systems, we required that in a replicated process $!a(x).P$, the highest level should be given to a. This condition appears rigid when we meet a process like $!a.b.\bar{a}$ because we do not take advantage of the level of b. This is the motivation for relaxing the requirement. The basic idea is to take into account the sum of the levels of two input subjects a, b, and compare it with the level of the output subject a. However, this incurs another problem. Observe the following reductions:

$$\begin{array}{rl} P \equiv & \bar{a} \mid \bar{b} \mid !a.b.\bar{a} \\ \longrightarrow & \bar{b} \mid b.\bar{a} \mid !a.b.\bar{a} \\ \longrightarrow & \bar{a} \mid !a.b.\bar{a} \end{array}$$

The weight of P does not decrease after the first step of reduction (we consume a copy of $\bar{a}$ but liberate another one). Only after the second reduction does the weight decrease. Further, P might run in parallel with another process, say Q, that interferes with P and prevents the second reduction from happening. This example illustrates two new problems that we have to consider: the weight of a process may not decrease at every step; because of interferences and interleaving among the activities of concurrent processes, consecutive reductions may not yield "atomic blocks" after which the weight decreases.

In the new type system we allow the measure of a process to decrease after a finite number of steps, rather than at every step, and up-to some commutativities of reductions and process manipulations. This difference has a strong consequence in the proofs. For technical reasons related to the proofs, we require certain names to be asynchronous.

5.1 Proving termination with asynchronous names

A name a is *asynchronous* if all outputs with subject a are followed by $\mathsf{0}$. That is, if $\bar{a}v.P$ appears in a process then $P \equiv \mathsf{0}$. A convenient way of distinguishing between synchronous and asynchronous names is using Milner's sorts [11]. Thus we assume two sorts of names, AN and SN, for asynchronous and synchronous names respectively, with the requirement that all names in AN are syntactically used as asynchronous names. We assume that all processes are well-sorted in this sense and will not include the requirements related to sorts in our type systems. (We stick to using both asynchronous and synchronous names instead of working on asynchronous π-calculus, because synchronous π-calculus is sometimes useful – see for instance the example in Section 6.2 – and it is more expressive [14]. However, all the results in this paper are valid for asynchronous π-calculus as well.)

We make another syntactic modification to the calculus by adding a construct to represent a sequence of inputs underneath a replication:

$$\begin{array}{rcll} \kappa & ::= & a_1(x_1).\cdots.a_n(x_n) & \qquad n \geq 1 \text{ and } \forall i < n, a_i \in AN \\ P & ::= & \ldots \mid !\kappa.P & \end{array}$$

This addition is not necessary – it only simplifies the presentation. It is partly justified by the usefulness of input sequences in applications. (It also strongly reminds us of the input pattern construct of the Join-calculus [6]). We call κ an input pattern. Note that all but the last name in κ are required to be asynchronous. As far as termination is concerned, we believe that the constraint – and therefore the distinction between asynchronous and synchronous names – can be lifted. However, we do not know how to prove Theorem 7 without it.

The usual form of replication $!a(x).P$ is now considered as a special case where the input pattern has length 1, i.e., it is composed of just one input prefix. We extend the definition of weight to input patterns by taking account of the levels of input subjects: $wt(a_1(x_1).\cdots.a_n(x_n)) = \mathbf{0}_{k_1} + \cdots + \mathbf{0}_{k_n}$ where $lv(a_i) = k_i$. The typing rule T-rep in Table 1 is replaced by the following one.

$$\textsf{T-rep}\ \frac{\vdash \kappa.P \quad wt(\kappa) \succ wt(P)}{\vdash !\kappa.P}$$

Intuitively, this rule means that we consume more than what we produce. That is, to produce a new process P, we have to consume all the prefixes from $a_1(x_1)$ to $a_n(x_n)$ on the left of P, which leads to the consumption of corresponding outputs at $a_1, \cdots, a_n$. Since the sum of weights of all the outputs is larger than the weight of P, the whole process has a tendency to decrease its weight. Although the idea behind this type system ($\mathcal{T}''$) is simple, the proof of termination is non-trivial because we need to find out whether and when a whole input pattern is consumed and thus the measure decreases.

THEOREM 7 *If $\mathcal{T}'' \vdash P$ then P terminates.*

Below we briefly explain the structure of the proof and proceed in four steps. Firstly, we decorate processes and transition rules with tags, which indicate the origin of each reduction: whether it is caused by calling a replicated input, a non-replicated input or it comes from an if-then-else structure. This information helps us to locate some points, called *landmarks*, in a reduction path. If a process performs a sequence of reductions that are locally ordered (that is, all and only the input prefixes of a given input pattern are consumed), then the process goes from a landmark to the next one and decreases its weight. (This is not sufficient to guarantee termination, since in general the reductions of several input patterns may interleave and some input patterns may be consumed only partially.) Secondly, by taking advantage of the constraint about asynchronous names, we show a limited form of commutativity of reductions. Thirdly, by commuting consecutive reductions, we adjust a reduction path and establish on it some locally ordered sequences separated by landmarks. Moreover, when an input pattern is not completely consumed, we perform some manipulations on the derivatives of processes and erase some inert subprocesses. Combining all of these with the result of Step 1, we are able to prove the termination property of tagged processes. Finally, the termination of untagged processes follows from the operational correspondence between tagged and untagged processes, which concludes our proof of Theorem 7.

PROPOSITION 8 *For a process P well-typed under $\mathcal{T}''$, let n and k be its size and the highest level, respectively. Then P terminates in polynomial time $\mathcal{O}(n^k)$.*

5.2 Example: the protocol of encoding separate choice

Consider the following protocol which is used for encoding separate choice by parallel composition [13], [17, Section 5.5.4]. One of the main contributions in [13] is the proof that the protocol does not introduce divergence. Here we prove it using typability.

$$
\begin{array}{l}
[\Sigma_{i=1}^{n}\bar{x}_i d_i.P_i] \equiv \nu s\ (\ \bar{s}\langle true\rangle \\
\qquad\qquad\qquad\quad \mid\ \Pi_{i=1}^{n}\nu a\bar{x}_i\langle d_i, s, a\rangle.a(x).\ \texttt{if}\ x\ \texttt{then}\ [P_i]\ \texttt{else}\ 0) \\
[\Sigma_{i=1}^{m}y_i(z).Q_i] \equiv \\
\nu r\ (\ \bar{r}\langle true\rangle \\
\quad\ \mid\ \Pi_{i=1}^{m}\nu g\ (\ \bar{g} \\
\qquad\qquad\quad \mid !g.y_i(z,s,a).r(x).\ \texttt{if}\ x\ \texttt{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\ s(y).\ \texttt{if}\ y\ \texttt{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\ \bar{r}\langle false\rangle \mid \bar{s}\langle false\rangle \mid \bar{a}\langle true\rangle \mid [Q_i]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\ \bar{r}\langle true\rangle \mid \bar{s}\langle false\rangle \mid \bar{a}\langle false\rangle \mid \bar{g})) \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \bar{r}\langle false\rangle \mid \bar{y}_i\langle z,s,a\rangle))
\end{array}
$$

where $r,\ s$ and a are fresh and $\Pi_{i=1}^{n}P_i$ means $P_1 \mid \cdots \mid P_n$.

The protocol uses two locks s and r. When one input branch meets a matching output branch, it receives a datum together with lock s and acknowledge channel a. Then the receiver tests r and s sequentially. If r signals failure, because another input branch has been chosen, the receiver is obliged to resend the value just received. Otherwise, it continues to test s. When s also signals success, the receiver enables the acknowledge channel and let the sender proceed. At the same time, both r and s are set to $false$ to prevent other branches from proceeding. If the test of s is negative, because the current output branch has committed to another input branch, the receiver should restart from the beginning and try to catch other send-requests. This backtracking is implemented by recursively triggering a new copy of the input branch.

Usually when a protocol employs a mechanism of backtracking, it has a high probability to give rise to divergence. The protocol in this example is an exception. However, to figure out this fact is non-trivial, one needs to do careful reasoning so as to analyze the possible reduction paths in all different cases. With the aid of type system $\mathcal{T}''$, we reduce the task to a routine type-checking problem. Simply taking $g.y_i(z,s,a)$ as an input pattern, one can check that the typability of $[P_i]$ and $[Q_i]$ implies that of $[\Sigma_{i=1}^{n}\bar{x}_i d_i.P_i]$ and $[\Sigma_{i=1}^{m}y_i(z).Q_i]$, which means that the protocol does not have infinite loops.

6. Partial orders

The purpose of our final type system is to type processes even if they contain replications whose input and output parts have the same weight. Of course not all such processes can be accepted. For instance, $!a.b.(\bar{a} \mid \bar{b})$ should not be accepted, since it does not terminate when running together with $\bar{a} \mid \bar{b}$. However, we might want to accept

$$!p(a,b).a.(\bar{p}\langle a,b\rangle \mid \bar{b}) \tag{3}$$

where a and b have the same type. Processes like (3) are useful. For instance they often appear in systems composed of several "similar" processes (an example is the chain of cells in Section 6.2). In (3) the input pattern $p(a,b).a$ and the continuation $\bar{p}\langle a,b\rangle \mid \bar{b}$ have the same weight, which makes rule T-rep of $\mathcal{T}''$ inapplicable. In the new system, termination is proved by incorporating partial orders into certain link types. For instance, (3) will be accepted if the partial order extracted from the type of p shows that b is below a.

6.1 The type system

We present the new type system $\mathcal{T}'''$. The general structure of the associated termination proof goes along the same line as the proof in Section 5.1. But now we need a measure which combines lexicographic and multiset orderings.

To begin with, we introduce some preliminary notations. Let $\mathcal{A}$ be a set and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ be a partial order on elements of $\mathcal{A}$. The set of names appearing in elements of $\mathcal{R}$ is $n(\mathcal{R}) = \{a \mid a\mathcal{R}b \vee b\mathcal{R}a \text{ for some } b\}$. Let $\widetilde{x}$ be a tuple of names $x_1, \cdots, x_n$. The partial order $\mathcal{S}$ on the index set $\{1, \ldots, n\}$ induces a partial order on $\widetilde{x}$, defined as $\mathcal{S} * \widetilde{x} = \{(x_i, x_j) \mid i\mathcal{S}j\}$.

Remark: In this paper we use partial order in a very narrow sense. Formally, for a partial order on names to be well defined, we require that it satisfies the following two conditions: (i) mathematically it is a strict partial order (irreflexive, antisymmetric and transitive); (ii) all names in $n(\mathcal{R})$ are of the same type (this type is written $T_{\mathcal{R}}$).

The operator $os(\cdot)$ of Section 3 is now refined to be $mos_{\mathcal{R}}(\cdot)$, which defines a multiset recording all subject occurrences of names in active outputs and with type $T_{\mathcal{R}}$. The operator $mos_{\mathcal{R}}(\cdot)$ can be extended to input patterns by defining: $mos_{\mathcal{R}}(\kappa) = mos_{\mathcal{R}}(\bar{a}\widetilde{x}_1 \mid \cdots \mid \bar{a}\widetilde{x}_n)$ if $\kappa = a_1(\widetilde{x}_1). \cdots .a_n(\widetilde{x}_n)$.

Let $\mathcal{R}$ be a partial order and $\mathcal{R}_{mul}$ be the induced multiset ordering on multisets over $n(\mathcal{R})$. The binary relation $\widehat{\mathcal{R}}$ defined below will act as the second component of our measure, which is a lexicographic ordering with weight of processes as its first component.

DEFINITION 9 *Suppose that $\mathcal{R}$ is a partial order, Q is a process, P is either an input pattern or a process. It holds that $P\ \widehat{\mathcal{R}}\ Q$ if the following three conditions are satisfied, for some multisets $\mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{M}$: (i) $mos_{\mathcal{R}}(P) = \mathcal{M} \uplus \mathcal{M}_1$; (ii) $mos_{\mathcal{R}}(Q) = \mathcal{M} \uplus \mathcal{M}_2$; (iii) $\mathcal{M}_1\ \mathcal{R}_{mul}\ \mathcal{M}_2$.*

Essentially the relation $\widehat{\mathcal{R}}$ is an extension of the multiset ordering $\mathcal{R}_{mul}$. So it is also well-founded: if $\mathcal{R}$ is finite, then there exists no infinite sequence $P_0\ \widehat{\mathcal{R}}\ P_1\ \widehat{\mathcal{R}}\ P_2\ \widehat{\mathcal{R}} \cdots$

Now we are well-prepared to present our types and type system. Here we consider polyadic π-calculus and redefine link type as follows.

$$L ::= \sharp^n_{\mathcal{S}} \widetilde{T} \quad \text{where} \quad \forall i, j \in n(\mathcal{S}),\ \ T_i = T_j$$

where $\mathcal{S} \subseteq \mathtt{Nat} \times \mathtt{Nat}$ is a partial order on the indexes of $\widetilde{T}$. If i and j are two indexes related by $\mathcal{S}$, then the i-th and j-th components of $\widetilde{T}$ have the same type. Suppose $\kappa = a_1(\widetilde{x}_1).\cdots.a_n(\widetilde{x}_n)$ and each a_i has type $\sharp^{m_i}_{\mathcal{S}_i}\widetilde{T}$. We extract a partial order from κ by defining $\mathcal{R}_\kappa = \mathcal{S}_1 * \widetilde{x}_1 \cup \cdots \cup \mathcal{S}_n * \widetilde{x}_n$. It is well defined as all the bound names are assumed to be different from each other.

If $\nu a P$ is a subprocess of Q, we say that the restriction νa is *unguarded* if $\nu a P$ is not underneath any input or output prefix. Besides the two sorts AN and SN introduced in the beginning of Section 5.1, now we need another sort RN. It requires that if a name of sort RN appears in the subject position of a prefix, then the continuation process has no unguarded restrictions. This technical condition facilitates the presentation of the definition below.

DEFINITION 10 *Let $\kappa = a_1(\widetilde{x}_1).\cdots.a_n(\widetilde{x}_n)$. The relation $\kappa \mathrel{:\succ} P$ holds if one of the following two cases holds: (i) $wt(\kappa) \succ wt(P)$; (ii) $wt(\kappa) = wt(P)$, $\kappa\, \widehat{\mathcal{R}_\kappa}\, P$ and $a_n \in RN$.*

The second condition indicates the improvement of $\mathcal{T}'''$ over $\mathcal{T}''$. We allow the input pattern to have the same weight as that of the continuation, as long as there is some partial order to reflect a tendency of decrement. The constraint imposed on a_n prohibits dangerous extension of partial orders underneath an input pattern and also simplifies our proof of Theorem 11. For the new type system $\mathcal{T}'''$, the most important rule is the following one:

$$\textsf{T-rep}\ \frac{\mathcal{R} \vdash \kappa.P \qquad \kappa \mathrel{:\succ} P}{\mathcal{R} \vdash !\kappa.P}$$

Now the judgment $\mathcal{R} \vdash P$ means that P is a well-typed process under $\mathcal{T}'''$ and the free names in P respect the (possibly empty) partial order $\mathcal{R}$. All other rules are easily adapted from $\mathcal{T}''$ by adding some appropriate partial order information to the type environment. Finally we have the following termination theorem for $\mathcal{T}'''$. The proof heavily relies on the well-foundedness of $\widehat{\mathcal{R}}$.

THEOREM 11 *If $\mathcal{R} \vdash P$ then P terminates. Moreover, let n and k be its size and the highest level, then P terminates in polynomial time $\mathcal{O}(n^{k+2})$.*

6.2 Example: symbol table

This example comes from [8, 15]. It implements a symbol table as a chain of cells. Below: G is a generator for cells; ST_0 is the initial state of the symbol table with only one cell; ST_m is the system in which the symbol table has m pending requests.

Every cell of the chain stores a pair (n, s), where s is a string and n is a key identifying the position of the cell in the chain. A cell is equipped with two channels so as to be connected to its left and right neighbors. The first cell has a public left channel a to communicate with the environment and the last cell has a right channel nil to mark the end of the chain. Once received a query for string t, the table lets the request ripple down the chain until either t is found in a cell, or the end of the chain is reached, which means that t is a new string and thus a new cell is created to store t. In both cases, the key associated to t is returned as a result. There is parallelism in the system: many requests can be rippling down the chain at the same time.

$$\begin{array}{rcl}
G & \equiv & !p(a,b,n,s).a(t,x). \\
 & & \hspace{3em}\texttt{if}\ t = s\ \texttt{then} \\
 & & \hspace{5em}\bar{x}\langle n\rangle.\bar{p}\langle a,b,n,s\rangle \\
 & & \hspace{3em}\texttt{else}\ \ \texttt{if}\ b = nil\ \texttt{then} \\
 & & \hspace{8em}\bar{x}\langle n+1\rangle.\nu c(\bar{p}\langle c, nil, n+1, t\rangle \mid \bar{p}\langle a,c,n,s\rangle) \\
 & & \hspace{6em}\texttt{else}\ \bar{b}\langle t,x\rangle.\bar{p}\langle a,b,n,s\rangle \\
ST_0 & \equiv & \nu p(G \mid \bar{p}\langle a, nil, 1, s_0\rangle) \\
ST_m & \equiv & ST_0 \mid \bar{a}\langle t_1, x_1\rangle \mid \cdots \mid \bar{a}\langle t_m, x_m\rangle
\end{array}$$

As to termination, the example is interesting for at least two reasons. (1) The chain exhibits a syntactically challenging form. The replicated process G has a sophisticated structure of recursive inputs: the input pattern has inputs at p and a, while the continuation has a few outputs at p and one output at b, which should have the same type as a. (2) Semantically, the chain is a dynamic structure, which can grow to finite but unbounded length, depending on the number of requests it serves. Moreover, the chain has a high parallelism involving independent threads of activities. The number of steps that the symbol table takes to serve a request depends on the length of the chain, on the number of internal threads in the chain, and on the value of the request.

Suppose $T \equiv \sharp^2_{\emptyset}(\mathtt{String}, \sharp^1\mathtt{Nat})$, $\mathcal{S} \equiv \{(1,2)\}$ and let the type of p be $\sharp^1_{\mathcal{S}}(T, T, \mathtt{Nat}, \mathtt{String})$. We consider nil as a constant name of the language studied in this section and take it for the bottom element of any partial order $\mathcal{R} \subseteq \mathcal{N}_2 \times \mathcal{N}_2$ with $T_{\mathcal{R}} = T$. For any $m \in \mathbb{N}$, process ST_m is well typed under $\mathcal{T}'''$ and thus terminating.

7. Final remarks

Since we are not able to encode the simply typed λ-calculus, our systems do not include those of [16] and [19]. Nevertheless, a large class of processes (including all examples analyzed in this paper) are excluded by the above two works. One way of interpreting the results of this paper is to consider combinatory approach (on which this paper is based) as a complementary technique to logical relations (on which [16] and [19] are based) for showing termination of processes. It would be interesting to see whether the two approaches can be successfully combined.

References

[1] M. Bezem. Mathematical background. In M. Bezem, J. Klop, and R. de Vrijer, editors, *Term Rewriting Systems*, pages 790–825. Cambridge University Press, 2003.

[2] G. Boudol. On strong normalization in the intersection type discipline. *LNCS*, 2701:60–74, 2003.

[3] N. Dershowitz and C. Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, 1995.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

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

[6] C. Fournet. *The Join-Calculus: A Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Paris, France, 1998.

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

[8] C. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *LNCS*, pages 158–172. Springer, 1993.

[9] R. Loader. Notes on simply typed lambda calculus. Technical Report 381, LFCS, University of Edinburgh, 1998.

[10] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[11] R. Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.

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

[13] U. Nestmann. What is a 'good' encoding of guarded choice? *Journal of Information and Computation*, 156:287–319, 2000.

[14] C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous π-calculi. *Mathematical Structures in Computer Science*, 13:685–719, 2003.

[15] D. Sangiorgi. The typed π-calculus at work: A proof of Jones's parallelisation theorem on concurrent objects. *Theory and Practice of Object-Oriented Systems*, 5(1), 1999.

[16] D. Sangiorgi. Termination of processes, Dec. 2001. Available from `ftp://ftp-sop.inria.fr/mimosa/personnel/davides`.

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

[18] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, 1979.

[19] N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the pi-calculus. In *Logic in Computer Science*, pages 311–322, 2001.