Static and dynamic typing for the termination of mobile processes

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

[1] ENS Lyon, Université de Lyon, CNRS, INRIA – France
[2] Dipartimento di Informatica, Università di Bologna – Italia

Abstract. A process terminates if all its reduction sequences are finite. We propose two type systems that ensure termination of π-calculus processes.
Our first type system is purely static. It refines previous type systems by Deng and Sangiorgi by taking into account certain partial order information on names so to enhance the techniques from term rewriting (based on lexicographic and multiset orderings) that underpin the proof of termination. The second system is mixed, in that it combines a static and a dynamic analysis. During the static analysis, processes are annotated with assertions. These are then used at run time to monitor the execution of processes. An exception may be raised if certain conditions that may lead to divergence are met.
We illustrate the expressiveness of the solutions proposed with a few examples of programming idioms that were beyond reach for previous type systems.

1 Termination of Concurrent Processes

Following the introduction of the π-calculus, a lot of research has been put into the study of languages for process mobility in which computing is exchange of messages between processes. Programs of these languages often produce dynamic recursive structures, that is, systems consisting of a variable number of components (at run time, new components may be created, and existing ones may be removed).

In this paper we study the problem of termination for mobile processes. We focus on the π-calculus, the commonly accepted model for them. A process terminates if all its internal runs are finite; that is, the process has no infinite sequence of reductions. Termination is a fundamental property in sequential languages. It is also important in concurrency. For instance, termination can be used to guarantee that interaction with a resource will eventually end (avoiding denial of service situations), or to ensure that the participants in a transaction will eventually reach an agreement. Unfortunately, termination is also a hard property to ensure, both in functional languages and in concurrency. Termination is particularly hard in the π-calculus, due to the expressiveness of this formalism. A number of programming language features can be encoded into the π-calculus, including functions, objects, and state (in the sense of imperative languages) [8]. Thus the notoriously hard problems of termination for these features hit the π-calculus too.

Previous work on termination in the π-calculus relies on *type systems* to guarantee the property. Languages of terminating processes are proposed in [10] and [7]. In

Please use the following format when citing this chapter:

Demangeon, R., Hirschkoff, D. and Sangiorgi, D., 2008, in IFIP International Federation for Information Processing, Volume 273; Fifth IFIP International Conference on Theoretical Computer Science; Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, Luke Ong; (Boston: Springer), pp. 413–427.

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. For this reason, subsequent work [3, 2] has explored type systems in which termination is proved using techniques from term rewriting systems, essentially defining a measure (a 'weight') that decreases with reductions. These type systems maintain however limitations on the form of recursive structures handled.

To explain the kind of problems encountered, consider a tree-like data structure, where search along the tree may involve recursive calls on all the subtrees of a given node. Termination of a call to the search procedure should intuitively follow from the acyclicity of the data structure. However, the tree cannot be type checked in [3], for the following reason. The type systems in [3] are based on an assignment of *levels* to π-calculus names, where a level is a positive integer. Since all nodes in a tree play the same role, names used in different nodes must have the same type and hence also the same level. As a consequence, when a search at a given node triggers several searches in subtrees of the node, the weight of the process (roughly, the multiset of the levels of the names in the "active" nodes) may increase (since the number of active nodes may be bigger). This breaks the reasoning needed in the proof of termination.

The contribution of the present paper is twofold. First, we refine the type systems in [3] so to be able to handle more complex recursive structures. The main improvement is given by the addition, into the type system, of a partial order that is used to compare names with the same level. While this possibility was already suggested in [3], only a very restrictive form of it had been investigated. In the present paper the idea is explored in depth. As we illustrate in Section 3, setting the balance between partial order and levels (or weight) is delicate, as counterexamples easily arise as soon as we abandon a purely lexicographical ordering between the weight and partial order information. Major problems are updating the partial order when new names are created, and ensuring that only well-founded partial orders are generated (a non-terminating computation could be produced following the step-by-step generation of an infinite descending path).

All type systems mentioned above are purely static: the whole type analysis is made before the processes are run. There is a tradeoff between expressiveness and complexity of the type systems (here complexity refers both to the intricacy of the typing rules and to the actual complexity of the type inference problem). We think that our new static type system is justified by the gain in expressiveness. We do not present, in contrast, other static type systems that we have examined, as the overall gain is more dubious.

We discuss, instead, as the second main contribution of the paper, an alternative approach: a simple and efficient static type system enhanced with dynamic (i.e., run time) checks. The static type system we adopt only exploits information about the level of names; however it annotates the positions where extra information (a partial order), is needed to justify termination. At runtime, these annotations are used to perform dynamic checks. Correctness of the resulting *mixed* type system is stated as follows:

if the first phase succeeds on a process, then the resulting annotated process cannot exhibit an infinite computation: its execution either terminates or raises an exception.

The advantage of dynamic typing is that only the parts of a process that are actually executed need to be analysed. This may considerably reduce the type constraints generated, especially in computations with data dependencies and/or non-determinism. For instance, in our case, dynamic typing reduces the risk of generating non-well-founded paths in the partial order over the names. We illustrate the expressive power of the mixed type system on some non-trivial examples. They include recursive structures created through merges of smaller structures, like, for instance, trees created from smaller trees via append operations (where a tree may be appended onto one or more leaves of another tree). In previous type systems, manipulations of this kind are not allowed, intuitively because all names connecting components of a recursive structure must be created locally. This forbids extensions of the structure with components (or just names) coming from the outside.

The price to pay with dynamic typing is the time for the additional checks at run time and the space for the data structure needed in the checks. In our mixed type system, the time for a dynamic check is at most linear in the number of annotated names created (the annotated names are those in positions that have been marked during the initial phase of static typing). The mixed system could be used in cases where the termination property is important and other, purely static, type systems have failed.

In type systems, the idea of extending static analyses with forms of dynamic checking is certainly not new. Works on the addition of dynamic types to statically typed languages include [1, 4]. We can also mention stack inspection, as, e.g., in [5], where checks on the access to resources are made at run time. Similar mechanisms are also employed in incremental garbage collectors, through read- or write-barriers, to prevent the program from accessing data that need to be processed by the collector (see [9]). In the present work, we use the phrase 'dynamic typing' by analogy with the aforementioned approaches, although the analysis that we make at runtime boils down to some lightweight sanity checks on partial orders.

2 The π-Calculus

We let $a,b,c,\ldots,p,q,\ldots,x,y,z$ range over an infinite set of *names*. Processes, ranged over using P,Q, are described by the following grammar (we use notation $\widetilde{n}$ to range over possibly empty tuples of names):

$$P ::= \mathbf{0} \mid P_1|P_2 \mid (\nu c)P \mid P_1+P_2 \mid a(\widetilde{x}).P \mid \overline{a}\langle\widetilde{v}\rangle.P \mid !a(\widetilde{x}).P \ .$$

The constructs of input, replicated input and restriction are binding. We sometimes call a bound name a *variable* and a free name a *channel*. We implicitly suppose that in all processes bound names are pairwise distinct and distinct from all free names. In an input $a(\widetilde{x}).P$ and an output $\overline{a}\langle\widetilde{v}\rangle.P$ we call a the *subject* name. As usual, trailing occurrences of $\mathbf{0}$ are omitted, and emissions and receptions of empty tuples of names

along a are respectively abbreviated $a.P$ and $\overline{a}.P$. The reduction relation of the calculus is standard (see Appendix 6).

Strict partial orders. As the rules of our type system heavily rely on partial orders, we first introduce some notations for partial orders on names. We use $\mathscr{R}$ to range over *strict* partial orders on names, and $\mathrm{dom}(\mathscr{R})$ is the domain of the order (the set of related elements). In the sequel, the phrase 'partial order' will always be used to denote a strict partial order, since we are only interested in these. Partial orders will be represented as the set of all pairs of related elements. However, for writing convenience, we usually indicate only a subset of the pairs, namely a subset whose transitive closure gives the induced partial order. For instance, the set $\{(a,b),(b,c)\}$ stands for the (strict) partial order $\{(a,b),(b,c),(a,c)\}$.

3 A purely static type system

3.1 Previous Type Systems: a Motivating Example

We recall here the basic ideas behind the type systems of [3, 2], using an example that also illustrates some of the limitations of these systems on recursive structures. Here and in the sequel, we shall use extensions of the π-calculus of Section 2 for the presentation of examples: the additional constructs are standard, and do not raise any particular difficulty for type checking termination. The example is about the implementation of a symbol table as a binary tree. Each node in the tree is a simple π-calculus process. The process T_0 below is the generator of nodes. An output $\overline{node}\langle a,l,r,s,e\rangle$ produces a node that stores a string s whose key is e, that is connected to its parent node (or to the environment, in case of the root node) with name a, and to its children nodes with names l and r. A tree at a (that is, a tree whose root uses a for interactions with the outside) is searched for a value v via requests of the form $\overline{a}\langle \mathsf{search}, v, ans\rangle$ where ans is a return channel. When the search reaches a node, if the value is found in the node, then the corresponding key is sent back on ans; otherwise the request is concurrently propagated to both subtrees of the node. (We omit the details of a search operation that fails.)

$$
\begin{array}{lcl}
T_0 & \stackrel{\text{def}}{=} & !node(a,l,r,s,e).a(mode,v,ans). \\
& & \quad \text{if } mode = \mathsf{search} \text{ then} \\
& & \quad\quad \text{if } v = s \text{ then } \ \overline{ans}\langle e\rangle \mid \overline{node}\langle a,l,r,s,e\rangle \\
& & \quad\quad \text{else } \ \overline{l}\langle mode,v,ans\rangle \mid \overline{r}\langle mode,v,ans\rangle \mid \overline{node}\langle a,l,r,s,e\rangle \\
& & \quad \text{else } \ldots
\end{array}
$$

The type systems in [3, 2] recognise a system as terminating if the continuations activated in an interaction (i.e., the processes underneath the interacting prefixes) have a smaller "weight" than that of the output that has been consumed to trigger the interaction. This notion of weight is formalised with an assignment of levels (positive

$$\frac{}{\mathscr{R} \vdash \mathbf{0}} \qquad \frac{\mathscr{R}_1 \vdash P_1 \qquad \mathscr{R}_2 \vdash P_2}{\mathscr{R}_1 + \mathscr{R}_2 \vdash P_1 \mid P_2} \qquad \frac{\mathscr{R}_1 \vdash P_1 \qquad \mathscr{R}_2 \vdash P_2}{\mathscr{R}_1 + \mathscr{R}_2 \vdash P_1 + P_2} \qquad \frac{\mathscr{R} \vdash P}{\mathscr{R} \Downarrow_c \vdash (\nu c) P}$$

$$\frac{\mathscr{R} \vdash P \qquad \vdash a : \sharp^{l_a}_{SS} \widetilde{T} \qquad \vdash \widetilde{v} : \widetilde{T} \qquad SS * \widetilde{v} \subseteq \mathscr{R}}{\mathscr{R} \vdash \overline{a}\langle \widetilde{v} \rangle . P} \qquad \frac{\mathscr{R} + (SS * \widetilde{x}) \vdash P \qquad \vdash a : \sharp^{l_a}_{SS} \widetilde{T} \qquad \vdash \widetilde{x} : \widetilde{T}}{\mathscr{R} \vdash a(\widetilde{x}) . P}$$

Fig. 1 Static system: typing rules (see main text for the typing of replication)

integers) to the types of the names. Now, consider the system composed by a tree at a and a search request $\overline{a}\langle \mathsf{search}, v, ans \rangle$. Names a, l and r play the same role in the structure, and therefore must have the same level. As the consumption of the output at a may produce outputs at l and r (the 'else' branch in T_0), the overall weight of the system increases (indeed, ensuring termination essentially boils down to controlling the outputs that can be generated along computation, since outputs may be used to trigger new copies of replicated processes). Due to this increase, T_0 is not typable. (The systems of [3, 2] allow the weight of the derivatives of an interaction to be at most the same as that of the initial process, and for this they rely on a rudimentary partial order information on names; however, the weight may never increase, as is instead the case for T_0.)

In the new type system that we propose below, replications in which the weight increases may be typed (indeed T_0 is typable, see Section 3.3). The greater expressiveness is achieved by enforcing a tight coupling between weight and a well-founded partial order. Increases in weight through reductions are possible, provided they are appropriately compensated in the partial order. This schema, while intuitively simple, is rather delicate. As an example of the possible problems (other examples will be given later in the paper), consider the system

$$T_1 \stackrel{\text{def}}{=} \overline{u} \mid \overline{v} \mid U_1 \mid U_2 \qquad \text{with} \qquad \begin{cases} U_1 \stackrel{\text{def}}{=} !p(a,b,c).a.(\overline{b} \mid \overline{c}) \\ U_2 \stackrel{\text{def}}{=} !u.v.(\overline{w} \mid \overline{p}\langle w,u,v \rangle) \end{cases}$$

where names w, u, v have the same level k and p has level $k' < k$ (this can be imposed, e.g., by adding extra processes in parallel). In U_1, the weight increases underneath the initial inputs at p and a; but the new outputs are smaller in the partial order, if we set a above b and c. In U_2, the weight decreases underneath the top two inputs. The system seems to meet the termination conditions; however, it does not terminate (the outputs at u and v trigger U_2, which in turn triggers U_1 and we are back to T_1).

3.2 The Type System

Figure 1 presents the typing rules for our new system. As in [3, 2], the type system follows the Church style, in the sense that each name is assigned a type a priori. We write $\vdash a : T$ if T is the type so assigned to name a. We add the termination analysis on top of the simply-typed polyadic π-calculus. Accommodating other standard type constructs would be straightforward; indeed, in examples, we sometimes use primitive types for values (integers and booleans, with the related if-then-else operator in the syntax of processes) and, in one case, recursive types. The grammar for types is given by

$$T \quad ::= \quad \sharp^{l}_{SS}\widetilde{T} \ .$$

In $\vdash p : \sharp^{l}_{SS}\widetilde{T}$, integer l is called the *level* of p, written $\mathrm{lvl}(p) = l$; and SS is the partial order associated to tuples of names carried along p, in which the i-th component is represented by integer i. For instance, if $SS = \{(2,3)\}$, then the second component of a tuple should be above the third one; thus an output $\overline{p}\langle u,v,w\rangle.P$ is typable only if v is above w in the partial order with which the output is typed. We let SS range over partial orders on integers of this kind, and use operator $*$ to 'project' them onto a relation on names. For example, if $SS = \{(1,2),(4,3)\}$, then, if $\mathscr{R} = SS * (u,v,w,t)$, we have $u\mathscr{R}v$ and $t\mathscr{R}w$.

We need some further notations to define the type system. Two partial orders $\mathscr{R}_1$ and $\mathscr{R}_2$ are compatible if $\mathscr{R}_1 \cup \mathscr{R}_2$ yields a partial order. If $\mathscr{R}_1$ and $\mathscr{R}_2$ are compatible then $\mathscr{R}_1 + \mathscr{R}_2$ is the partial order (induced by) $\mathscr{R}_1 \cup \mathscr{R}_2$; if they are not compatible, then $\mathscr{R}_1 + \mathscr{R}_2$ is undefined. $\mathscr{R} \Downarrow_c$ stands for the relation obtained by removing all pairs involving c after closing $\mathscr{R}$ by transitivity.

The typing judgements for processes are of the form $\mathscr{R} \vdash P$, where $\mathscr{R}$ is a partial order. They are defined by the rules of Figure 1, plus the rules for replication below. The rules of Figure 1 are similar to those in [3]. The typing of replication, however, is different. We comment on the main typing rules. In the rule for output, the partial order $\mathscr{R}$ must include $SS * \widetilde{v}$, which is the partial order derived from the type of the subject a. Similarly, in the rule for input, the partial order is extended with constraints on bound names of the input as derived from the type of its subject. We now present the two rules for replication. They are defined on processes of the form $!\kappa.P$, where κ is a sequence of inputs, such as $a_1(\widetilde{x_1}).a_2(\widetilde{x_2})\ldots a_n(\widetilde{x_n})$; moreover the sequence is maximal, in the sense that the outermost process operator in P is not an input. If $\kappa = a_1(\widetilde{x_1}).a_2(\widetilde{x_2})\ldots a_n(\widetilde{x_n})$, then M_κ is the multiset of the names $a_1,\ldots,a_n$ that occur in subject position in κ. Moreover, if $\sharp^{l_i}_{SS_i}\widetilde{T_i}$ is the type of a_i, for $i = 1,..,n$, then $\mathscr{R}_\kappa$ stands for $(SS_1 * \widetilde{x_1}) \cup \cdots \cup (SS_n * \widetilde{x_n})$. For a given multiset M of names, $M|_l$ is the multiset of names in M whose level is equal to l, and $\mathrm{card}(M)$ is the cardinality of M.

$$[\texttt{Rep1}]\ \frac{\mathscr{R} \vdash \kappa.P \qquad \exists l > 0 \text{ s.t. } \begin{cases} (i)\ \forall j > l, M_\kappa|_j = os(P)|_j \\ (ii)\ \forall j \geq l, rs(P)|_j = \emptyset \\ (iii)\ os(P)|_l \subsetneq M_\kappa|_l \end{cases}}{\mathscr{R} \vdash !\kappa.P}$$

$$[\texttt{Rep2}]\ \frac{\mathscr{R} \vdash \kappa.P \qquad \exists l > 0 \text{ s.t. } \begin{cases} (i)\ \forall j > l, M_\kappa|_j = os(P)|_j \\ (ii)\ \forall j \geq l, rs(P)|_j = \emptyset \\ (iii)\ \mathrm{card}(M_\kappa|_l) \leq \mathrm{card}(os(P)|_l) \\ (iv)\ M_\kappa|_l\, (\mathscr{R}_\kappa)_{\mathrm{mul}}\, os(P)|_l \end{cases}}{\mathscr{R} \vdash\, !\kappa.P}$$

Although the definition of rules [Rep1] and [Rep2] is complex, the checks made are fairly simple. The rules differ only in conditions (iii) and (iv). Besides the expected condition on the typing of $\kappa.P$, the most important aspect is the comparison between M_κ, the multiset of the subjects of the inputs in κ, and $os(P)$, the multiset of the subjects of the outputs in P not occurring under a replication. In the two rules, l is the maximal level on which the weights of M_κ and $os(P)$ differ (at higher levell they are the same: condition (i)). In [Rep1], intuitively, levels are sufficient to guarantee termination. We indeed check in (iii) that, at level l, M_κ *strictly* contains $os(P)$, as a multiset. This condition enforces two properties: first, the weight decreases at level l when consuming the sequence κ; second, the partial order cannot be used to produce diverging computations, by compensating the loss in weight with an increase in the partial order.

In rule [Rep2] (that uses the same notations as [Rep1]), condition (iii) says that at level l the weight of M_κ is not bigger. Hence weight alone is not sufficient to guarantee termination, and the partial order becomes crucial: we check in (iv) that, at level l, M_κ dominates $os(P)$ according to the strict partial order associated to the multiset extension of $\mathscr{R}_\kappa$. Precisely, $M_\kappa|_l\, (\mathscr{R}_\kappa)_{\mathrm{mul}}\, os(P)|_l$ holds if $M_\kappa|_l \neq os(P)|_l$, and there is a multiset C included in M_κ and $os(P)$ s.t. for all $b \in os(P)|_l \setminus C$, there is $a \in M_\kappa|_l \setminus C$ with $a \mathscr{R} b$.

In both rules, the remaining condition (ii) ensures that no name is created at level l or higher; the need for this technical condition will be shown in the second example of Section 3.3.

3.3 Examples

We present two examples that illustrate some of the technicalities of the type system. The first example explains the need for condition (iv) in rule [Rep1]. Consider the system T_1 in Section 3.1. The system diverges. We explain why it is rejected by our system, supposing, as we did in Section 3.1, that we must have $\mathrm{lvl}(a) > \mathrm{lvl}(p)$; e.g., p has level 1 and all other names have level 2. This way, we can type U_2 using [Rep1] (two inputs at level 2 'weight more' than one at level 2 and one at level 1). For U_1, since the weight is increasing, we must resort to the partial order, and impose that the first component of tuples transmitted on p dominates the two other components (so that name a dominates b and c), and we can use [Rep2]. However, this renders the typing of U_2 invalid, because condition (iv) of [Rep1] is not satisfied: $M_\kappa|_2 = \{u, v\}$ does not contain $os(P)|_2 = \{w\}$. It can be shown, more generally, that for any assignment of levels to names, T_1 cannot be typed.

Another delicate aspect of the type system is the control of the creation of new names. In a replication $!\kappa.P$, a new name that is created should have a level smaller than the maximal level that decreases when moving from κ to P (this is imposed by condition (ii) in [Rep1] and [Rep2]). The need for this constraint is illustrated by the following process.

$$T_2 \stackrel{\text{def}}{=} !p(a,e,f).a.(\overline{e} \mid \overline{f} \mid \overline{p}\langle a,e,f\rangle) \;\mid\; !p(a,e,f).e.f.(\nu c)(\overline{c} \mid \overline{p}\langle c,e,f\rangle) \;\; .$$

Without the constraint on the creation of new names, T_2 could be typed, by setting the level of p to 1, the level of all the other names to 2, and annotating the type of p with a partial order that forces a to be above e and f. But T_2, when put in parallel with $\overline{u} \mid \overline{p}\langle u,v,w\rangle$ (which would also be typable), diverges:

$$T_2 \mid \overline{u} \mid \overline{p}\langle u,v,w\rangle \;\longrightarrow\!\!\!\!\rightarrow\; T_2 \mid \overline{v} \mid \overline{w} \mid \overline{p}\langle u,v,w\rangle \;\longrightarrow\!\!\!\!\rightarrow\; (\nu c)\left(T_2 \mid \overline{c} \mid \overline{p}\langle c,v,w\rangle\right)$$

At the end, c plays the role played by u in the initial state. In the second replication in T_2, where the new name c is created, the maximal level that decreases is 2 (at level 2, two outputs are consumed to reach the body of the replication, namely e and f, and only one output is produced, namely c). The newly created name has precisely level 2, hence typing fails.

Process T_0 presented in Section 3.1 can be type-checked, by assigning type T_a to names a,l,r, type T_{ans} to *ans*, and type T_{node} to *node*, with

$$T_a = \sharp^3(\mathsf{M},\mathsf{S},T_{ans}), \quad T_{ans} = \sharp^2(\mathsf{K}), \quad T_{node} = \sharp^1_{\{(1,2),(1,3)\}}(T_a,T_a,T_a,\mathsf{S},\mathsf{K}),$$

where S is the type of the value v (strings in the example), K the type of the key associated to a value, M the type of tags indicating the method that is invoked on the tree. In the typing, the critical part is the 'else' branch in T_0; here the input on a at level 1 is traded for two outputs, on names l and r, at the same level, and we rely on the partial order derived from p to conclude the typing ([Rep2] – a dominates both l and r). Note that at the higher level, level 3, the weight does not change, as the input at *node* is followed by an output on the same channel.

3.4 Soundness of the Type System

Theorem 1. *If $\mathscr{R} \vdash P$ then P terminates.*

Proof (Sketch). Suppose P is non-terminating, i.e. there exists an infinite derivation $\mathscr{D} : P_1 = P \to P_2 \to \ldots$ We write $\kappa_1,\ldots,\kappa_n$ for the (finitely many) prefix sequences occurring in P. The typing for P determines, for each κ_i, a level, written $\mathrm{lvl}(\kappa_i)$, which is the maximal level at which either the order or the weight decreases (this is integer l in [Rep1] and [Rep2]).

Some steps in $\mathscr{D}$ correspond to a communication that erases the last input prefix of one of the κ_is – we call such steps *gaps*; there are necessarily infinitely

many gaps, otherwise no divergence could arise. Since there are finitely many κ_is, at least one of the κ_i is involved in an infinite number of gaps in $\mathscr{D}$. We let $k = \max\{\mathrm{lvl}(\kappa_i).\ \kappa_i \text{ is fired an infinite number of times in } \mathscr{D}\}$. We focus on reductions that involve gaps at level k to derive a contradiction.

By definition of k, and because P is typable, there exists a step in $\mathscr{D}$ after which: (i) no new name is created at a level $\geq k$ (and hence the support of the partial order involving free names remains the same at level k); (ii) no output occurring at a level strictly greater than k is triggered. After that step, there are necessarily infinitely many gaps involving some κ at level k along which the order decreases: if this was not the case, there would exist a step after which all such gaps would correspond to a strictly decreasing weight, which is impossible. Since for such gaps the partial order cannot grow (condition (iv) in [Rep1]), and since the support of the partial order remains the same, we derive a contradiction ($\mathscr{R}_{\mathrm{mul}}$ is well-founded whenever R is). □

The proof of Theorem 1 departs considerably from the correctness proof of the systems in [3]. The strategy of the latter proof is less robust, because it exploits additional syntactical hypotheses about prefixes in processes to rearrange reductions in an infinite computation. In the present proof, we extract some ordering information from a diverging computation in order to derive a contradiction.

4 The Mixed Type System

In this section we discuss another approach to typing termination. We present a mixed system in which the type checks are performed in two separated phases: a phase that precedes execution, and the phase of execution itself. Below, these two phases are referred to as *static* and *dynamic*, respectively; correspondingly we distinguish between static and dynamic typing.

The static typing, besides making the type checks, inserts into the processes *assertions* on names of the form $[a > b]$. We call a process with assertions an *annotated process*. The grammar for annotated processes is the same as that of ordinary processes in Section 2, with the addition of the production $[a > b]P$ for assertions. We use $A, B, \ldots$ to range over annotated processes.

The assertions are needed in the dynamic typing. Precisely, at run time we check that the transitive closure of the assertions encountered during execution is well-founded. Thus the operational semantics is defined on pairs $(A, \mathscr{R})$ where A is an annotated process and $\mathscr{R}$ a partial order (as usual, represented by a set of pairs whose reflexive and transitive closure induces the partial order).

Failure in the dynamic checks occurs when the addition of a new assertion introduces a cycle; in this case the special term $\perp$ is produced, meaning that an exception has been raised. We call $\perp$ and the pairs $(A, \mathscr{R})$ *configurations*. We first define the dynamic system, and then the static system.

4.1 The dynamic system

The operational semantics on ordinary processes is extended to configurations as expected, and we write $\mapsto$ for the reduction relation on configurations. The only new rule is the following (see Appendix 6.2 for a complete presentation).

$$[a > b]A, \mathscr{R} \mapsto \begin{cases} A, (\mathscr{R} \cup \{(a,b)\}) & \text{if } \mathscr{R} \cup \{(a,b)\} \text{ is a partial order} \\ \bot & \text{otherwise} \end{cases}$$

An annotated process A is *divergent* if there is an infinite sequence of reductions emanating from $(A, \emptyset)$ (where $\emptyset$ is the empty relation).

4.2 The static system

The static type system takes an ordinary process, performs some type checks on it, and returns an annotated process. Judgements for processes are of the form $\vdash P \rightsquigarrow A$, meaning that P is well typed and A is the annotated version of P that is produced.

The rules are presented in Figure 2. As in Figure 1, the main termination analysis is performed in the rule for replication. To type a replication $!a(\widetilde{x}).P$, we insert an assertion whenever we encounter an output in P that is not under a replication and whose subject has the same level as a; in this situation, levels alone are not sufficient to guarantee termination, and further checks, via the assertions, are postponed at run time.

We explain the notations used in the rule for replication. If A is an annotated process and a a name, then $C(A,a)$ stands for the annotated process obtained from A by inserting an assertion $[a > b]$ in front of each output (not guarded by replication) whose subject name b has the same level as a. Intuitively, $[a > b]$ is a sanity check: a has to dominate b according to the partial order to guarantee that the process does not loop (see examples in Section 4.4). We write $\mathrm{lvl}(os(P))$ and $\mathrm{lvl}(rs(P))$ for the sets of the levels of the names in $os(P)$ and $rs(P)$, respectively. Thus $l_a \succeq \mathrm{lvl}(os(P))$ means that l_a is greater than, or equal to, the level of each name in $os(P)$; and $l_a \succ \mathrm{lvl}(rs(P))$ means that l_a is strictly greater than the level of each name in $rs(P)$.

Remark 1. In the rule for replication, only the initial input of the replication is examined. The system can be made more powerful by taking into account sequences of inputs, along the lines of the type system of Section 3 (where sequences are indicated by the κ prefix). We have not done so for simplicity of presentation and for efficiency: as discussed in Section 4.5, inference for the present system is polynomial. It would become NP-complete with sequences (as a matter of fact, it can be proved along the lines of [2] that type inference is NP-complete for the inference problem for the type system of Section 3).

Since we do not take sequences into account, the systems of Sections 3 and 4 are incomparable: none of them captures more processes than the other.

$$\frac{\vdash P \rightsquigarrow A \qquad \vdash a : \sharp^{l_a}\widetilde{T} \qquad \vdash \widetilde{v} : \widetilde{T}}{\vdash \overline{a}\langle\widetilde{v}\rangle.P \rightsquigarrow \overline{a}\langle\widetilde{v}\rangle.A} \qquad \frac{\vdash P \rightsquigarrow A \qquad \vdash a : \sharp^{l_a}\widetilde{T} \qquad \vdash \widetilde{x} : \widetilde{T}}{\vdash a(\widetilde{x}).P \rightsquigarrow a(\widetilde{x}).A}$$

$$\frac{\vdash P \rightsquigarrow A \qquad \vdash Q \rightsquigarrow B}{\vdash P \mid Q \rightsquigarrow A \mid B} \qquad \frac{\vdash P \rightsquigarrow A \qquad \vdash Q \rightsquigarrow B}{\vdash P + Q \rightsquigarrow A + B} \qquad \frac{\vdash P \rightsquigarrow A}{\vdash (\nu c)P \rightsquigarrow (\nu c)A} \qquad \frac{}{\vdash \mathbf{0} \rightsquigarrow \mathbf{0}}$$

$$\frac{\begin{array}{c}\vdash P \rightsquigarrow A \qquad \vdash a : \sharp^{l_a}\widetilde{T} \qquad \vdash \widetilde{x} : \widetilde{T} \qquad a \notin os(A) \\ l_a \succeq \mathrm{lvl}(os(P)) \quad l_a \succ \mathrm{lvl}(rs(P)) \qquad A' = C(A,a)\end{array}}{\vdash !a(\widetilde{x}).P \rightsquigarrow !a(\widetilde{x}).A'}$$

Fig. 2 The static type analysis in the mixed system

4.3 Soundness

Theorem 2. *If* $\vdash P \rightsquigarrow A$, *then* A *has no diverging computation.*

For lack of space, we omit the proof of this result, that follows the same general strategy as the proof of Theorem 1.

The following proposition says that a process and its annotated version perform the same reductions, unless the annotated one raises an exception. Relation $\rightarrow^*$ (resp. $\mapsto^*$) is the reflexive transitive closure of $\rightarrow$ (resp. $\mapsto$). We write $\mathrm{erase}(\mathrm{A})$ for the process obtained by removing the assertions from A.

Proposition 1. *Suppose* $\vdash P \rightsquigarrow A$. *If* $P \rightarrow^* P'$, *then either* $A, \emptyset \mapsto^* A', \mathscr{R}$ *with* $\mathrm{erase}(\mathrm{A}') = P'$ *for some* $\mathscr{R}$, *or* $A, \emptyset \mapsto^* \bot$. *Conversely, if* $A, \emptyset \mapsto^* A', \mathscr{R}$, *then* $P \rightarrow^* P'$ *for some* P' *with* $\mathrm{erase}(\mathrm{A}') = P'$.

4.4 Examples

The first example shows a divergent process that passes the static phase of the mixed system and produces a failure exception at run time. Let

$$R \stackrel{\text{def}}{=} !p(a,b,c).(!a.\overline{b} \mid !b.\overline{c} \mid !c.\overline{a}) \mid \overline{p}\langle u,v,w\rangle \mid \overline{u} \ .$$

R is typable: we have a derivation for

$$\vdash R \rightsquigarrow A \stackrel{\text{def}}{=} !p(a,b,c).(!a.[a>b].\overline{b} \mid !b.[b>c].\overline{c} \mid !c.[c>a].\overline{a}) \mid \overline{p}\langle u,v,w\rangle \mid \overline{u}$$

by assigning the same level to a, b, c. At run time we have the following (deterministic) sequence of reductions:

$$
\begin{array}{l}
!build(a,s_0,e_0).\ (\nu state)\ (\\
\quad \overline{state}\langle nil, nil, s_0, e_0\rangle \\
\quad \mid\ !a(chan,mode).state(l,r,s,e).chan(v,ans,n). \\
\qquad \text{if } mode = \mathsf{merge} \text{ then} \\
\qquad\quad \text{if } l = \mathsf{nil} \text{ then } \overline{state}\langle n,r,s,e\rangle \\
\qquad\quad \text{else if } r = \mathsf{nil} \text{ then } \overline{state}\langle l,n,s,e\rangle \\
\qquad\quad \text{else } (\nu chan')\ \ (\overline{l}\langle chan', \mathsf{merge}\rangle.\overline{chan'}\langle v,ans,n\rangle.\overline{state}\langle l,r,s,e\rangle \\
\qquad\qquad\qquad\qquad + \overline{r}\langle chan', \mathsf{merge}\rangle.\overline{chan'}\langle v,ans,n\rangle.\overline{state}\langle l,r,s,e\rangle\ ) \\
\qquad \text{else} \qquad \ldots)
\end{array}
$$

Fig. 3 Merging tree structures

$$
\begin{array}{rcl}
(A,\emptyset) & \rightarrow\rightarrow & (A \mid !u.[u>v].\overline{v} \mid !v.[v>w].\overline{w} \mid !w.[w>u].\overline{u}) \mid [u>v].\overline{v},\emptyset) \\
& \rightarrow\rightarrow & (A \mid !u.[u>v].\overline{v} \mid !v.[v>w].\overline{w} \mid !w.[w>u].\overline{u}) \mid [v>w].\overline{w},\mathscr{R}_1) \\
& \rightarrow\rightarrow & (A \mid !u.[u>v].\overline{v} \mid !v.[v>w].\overline{w} \mid !w.[w>u].\overline{u}) \mid [w>u].\overline{u},\mathscr{R}_2) \\
& \rightarrow & \bot
\end{array}
$$

where $\mathscr{R}_1$ is $\{(u,v)\}$ and $\mathscr{R}_2$ is $\{(u,v),(v,w)\}$. Process A eventually produces $\bot$ as the three inner replications create a cycle in the relation.

The next example illustrates an advantage of dynamic typing on data-dependent or non-deterministic computations. Let

$$Q \stackrel{\text{def}}{=} (Q_0 \mid \overline{p}\langle u,v\rangle \mid \overline{u} \mid \overline{g}) \qquad \text{where} \qquad Q_0 \stackrel{\text{def}}{=} !p(a,b).(!a.\overline{b} \mid (g.!b.\overline{a} + g.b))\ .$$

When the output at p is consumed we obtain the process

$$Q' \stackrel{\text{def}}{=} Q_0 \mid !u.\overline{v} \mid (g.!v.\overline{u} + g.v) \mid \overline{u} \mid \overline{g}\ .$$

If the output on g synchronises with the left summand, a loop is produced by the two replications. If the right summand is selected, the divergence is avoided. A static type system would necessarily reject Q, due to the potential loop in the two replications. In our mixed system, with appropriate choice of levels, Q passes the static analysis. At run time, one computation of Q will yield $\bot$, the other will not. We omit the details for lack of space.

We now discuss the typing of recursive structures as those in Section 1 and Section 3.1: trees with operations of *remote allocation*, that allow one to merge two trees by attaching the root of a tree to a leaf of another tree. To type the tree T_0 of Section 3.1, we need to take into account sequences of inputs in replications, that is, replications of the form $!\kappa.P$ as we do in the type system of Section 3. (Precisely, in the subterm $!node(a,\ldots).a(\ldots)\ldots$, we need to compare the sum of the levels of names *node* and a against the weight of the continuation.) This can be easily done, as discussed in Remark 1, by strengthening the typing rule for replication in the static phase of the mixed system. Alternatively, we can keep the present typing rules and make some modifications to the programs. We discuss this solution in the remainder. Figure 3 presents the modified tree structure. The topmost replication, $!build(a,s_0,e_0)$, acts as a constructor, invoked for the creation of a new node; this new node carries values e_0, s_0, and interacts with the parent node via channel a. The state of this node is represented by the floating

message on *state* (in which the first two components are the names for accessing the children, and are set to the special value nil if the node is a leaf). We only show the code for the merge operation: the code for a search can be adapted from the example in Section 3.1. When merge is invoked, the transmitted channel should be attached to a leaf; if there is room, this happens in the current node; otherwise the merge is nondeterministically delegated to one of the children. (This is a simplified version of merge: the new tree is attached anywhere in the tree, without, for instance, ensuring that the tree remains well-balanced.) The code above is accepted by the static analysis of the mixed type system[1] modulo the insertion of just a few annotations: the highest level is affected to names a, l, r, and an annotation $[a > l]$ (resp. $[a > r]$) is inserted before the output at l (resp. at r). The resulting annotated process does not lead to failure exceptions at run time.

The mixed system remains, of course, incomplete — there are terminating processes whose annotated version yields $\perp$ — as the problem of the termination of a process is not decidable.

4.5 Efficiency

The static analysis of the mixed system can be made in time that is polynomial w.r.t. the size of the process being checked, by adapting the type inference algorithms in [2] (the modifications are mild). (Our system is more flexible than the one of [2], in that, e.g., we allow constraints relating a name received in some input prefix and a name defined above that prefix, but this does not affect the overall inference procedure for levels, which remains polynomial.) With such an algorithm, the static analysis introduces only the necessary assertions. More precisely, if the termination of a process can be proved by only relying on levels and weights (without referring to a partial order), then the static analysis will introduce no assertions and there will be no dynamic checks at run time.

Note that a trivial (and linear) static analysis would assign the same level to all names, and add assertions in front of all outputs prefixes. This would however mean that: all type checks are performed at run time; useful weight information is lost, so that the final termination analysis is rather rough.

The inference problem would become NP-complete if the system were refined by taking into account sequences of inputs underneath a replication as suggested in Remark 1 (this is proved by adapting the NP-completeness result for the system with sequences of inputs in [2]).

Concerning the efficiency of dynamic checks, each time a new constraint is added to the $\mathscr{R}$ component of a configuration, we have to check for acyclicity of the resulting relation. This can be done via a depth-first traversal of $\mathscr{R}$, whose cost is linear in $\#\mathscr{R} + |\mathscr{R}|$, where $\#\mathscr{R}$ (resp. $|\mathscr{R}|$) stands for the size of $\mathrm{dom}(\mathscr{R})$ (resp. the number of

[1] Recursive types are needed for typing, independently from the termination analysis; as mentioned in Section 3.1, recursive types, as well as other common type constructs, are straightforward to accommodate.

pairs in $\mathscr{R}$). In [6], an online algorithm is shown, that allows one to perform the same task in linear amortised time in $\#\mathscr{R}$ only.

5 Conclusion

In this paper we have investigated type systems for termination in which techniques of previous systems based on a lexicographical measure are enhanced with partial orders on names. The first system is purely static, the second mixes static and dynamic typing.

We have illustrated the expressiveness of the mixed system on a remote allocation example in which a recursive structure is extended with names and substructures imported from the environment. It would be difficult to handle this kind of system using a purely static system, due to the mobility of the names involved. The reason is that, intuitively, one cannot statically predict the precise name that is received in an input, and therefore one must make a worst-case approximation, using a set of names, guided by the type system. Further, in this situation, if a name is exported, then one has also to foresee the possibility that the name is sent back, which can easily create cycles that break the partial order on the names.

Acknowledgements. This work has been supported by the french ANR projects "MoDyFiable" and "CHoCo", by European Project FET-GC II IST-2005-16004 SENSORIA, and by Italian MIUR Project n. 2005015785, "Logical Foundations of Distributed Systems and Mobile Code".

References

1. M. Abadi, L. Cardelli, B. C. Pierce, and G. D. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.
2. R. Demangeon, D. Hirschkoff, N. Kobayashi, and D. Sangiorgi. On the complexity of termination inference for processes. In *Proceedings of TGC'07*, LNCS. Springer, 2008. to appear.
3. Y. Deng and D. Sangiorgi. Ensuring termination by typability. *Inf. Comput.*, 204(7):1045–1082, 2006.
4. F. Henglein. Dynamic typing. In *Proc. of ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 233–253. Springer, 1992.
5. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison Wesley, 1997.
6. A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996.
7. D. Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 16(1):1–39, 2006.
8. D. Sangiorgi and D. Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
9. P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. of IWMM'92*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer, 1992.
10. N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the pi-calculus. *Information and Computation*, 191(2):145–202, 2004.

6 Formal Definition of the Operational Semantics

6.1 π-calculus Processes

Structural congruence for the π-calculus is the least equivalence relation that is closed under α-conversion, satisfies the laws of an abelian monoid for $\mid$ and $+$ (with $\mathbf{0}$ as neutral element), and moreover validates the following axioms:

$$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad !P \mid P \equiv !P \qquad (\nu z)\mathbf{0} \equiv \mathbf{0} \qquad P + P \equiv P$$

$$(\nu x)(P \mid Q) \equiv (\nu x)P \mid Q \quad \text{if } x \text{ is not a free name of } Q$$

We moreover let $\equiv$ be closed by parallel composition, restriction, and replication, but *not* under prefixes (this is due to a technical reason related to the handling of κ in Section 3). The reduction relation is defined as follows.

$$\overline{(\overline{x}\langle\widetilde{v}\rangle.P_1 + M_1) \mid (x(\widetilde{z}).P_2 + M_2) \to P_1 \mid P_2\{\widetilde{v}/\widetilde{z}\}}$$

$$\frac{P_1 \to P_1'}{P_1 \mid P_2 \to P_1' \mid P_2} \qquad \frac{P \to P'}{(\nu c)\, P \to (\nu c)\, P'} \qquad \frac{P_1 \equiv P_2 \to P_2' \equiv P_1'}{P_1 \to P_1'}$$

6.2 Annotated Processes

Structural congruence for annotated processes is defined as above. If $\mathscr{R}$ is a relation on names, we write $\mathtt{ok}(\mathscr{R})$ if $\mathscr{R}$ induces a partial order, and $\neg\mathtt{ok}(\mathscr{R})$ otherwise. The reduction relation for configurations, written $\mapsto$, is defined by the following rules.

$$\overline{(\overline{x}\langle\widetilde{v}\rangle.A_1 + M_1) \mid (x(\widetilde{z}).A_2 + M_2), \mathscr{R} \mapsto A_1 \mid A_2\{\widetilde{v}/\widetilde{z}\}, \mathscr{R}}$$

$$\frac{A_1, \mathscr{R} \mapsto A_1', \mathscr{R}}{A_1 \mid A_2, \mathscr{R} \mapsto A_1' \mid A_2, \mathscr{R}} \qquad \frac{A, \mathscr{R} \mapsto A', \mathscr{R}}{\nu c\, A, \mathscr{R} \mapsto \nu c\, A', \mathscr{R}}$$

$$\frac{A_1 \equiv A_2 \qquad A_2, \mathscr{R} \mapsto A_2', \mathscr{R} \qquad A_2' \equiv A_1'}{A_1, \mathscr{R} \mapsto A_1', \mathscr{R}}$$

$$\frac{\mathtt{ok}(\mathscr{R} \cup \{(a,b)\})}{[a > b]A, \mathscr{R} \mapsto A, \mathscr{R} \cup \{(a,b)\}} \qquad \frac{\neg\mathtt{ok}(\mathscr{R} \cup \{(a,b)\})}{[a > b]A, \mathscr{R} \mapsto \bot}$$