On the complexity of termination inference for processes

Romain Demangeon[1], Daniel Hirschkoff[1], Naoki Kobayashi[2], and Davide Sangiorgi[3]

[1] ENS Lyon, France
[2] Tohoku University, Japan
[3] Università di Bologna, Italy

Abstract. We study type systems for termination in the π-calculus from the point of view of type inference. We analyse four systems by Deng and Sangiorgi. We show that inference can be done in polynomial time for two of these, but that this is not the case for the two most expressive systems. To remedy this, we study two modifications of these type systems that allow us to recover a polynomial type inference.

1 Introduction

Termination of concurrent systems is an important property. Even if some concurrent systems, like servers, are designed to offer continuously some interaction, subsystems are often expected to terminate. Typical examples include guaranteeing that interaction with a resource will eventually end (in order to avoid denial of service situations), insuring that the participants in a transaction will reach an agreement, or relying on termination to guarantee other properties (such as, e.g., lock freedom [3, 9]). Such example applications are important for distributed frameworks exploiting various forms of mobility. Being able to assert termination for (part of) a system whose topology can change dynamically is challenging. It can be particularly useful if the method includes some form of automation.

In this paper, we focus on the π-calculus, a model of mobile computing based on name passing, and revisit the work by Deng and Sangiorgi [4] from the point of view of type inference. As we explain below, this can in particular be useful in relation with the work on TyPiCal reported in [9]. [4] introduces four type systems for the π-calculus with replicated inputs, which we will call System 1, 2, 3 and 4, in short S1, S2, S3 and S4. S_{i+1} is strictly more expressive than S_i. The main idea behind these systems is to associate an integer *level* to each name, and to enforce that, for each replicated process, the computation that 'fires' the replication has a bigger weight than the computation which is triggered by this firing step.

In system S1, the point of view is that a term of the form $!a(\widetilde{x}).P$ is triggered by offering an output on a. Hence, the weight of P (which is defined as the total weight of outputs that occur in P without occurring under a replication) has to be strictly smaller than the weight of the output on a, i.e., the level associated to a. Weights are compared lexicographically, which entails that several outputs can occur in P, provided they all happen on names whose level is strictly smaller than the level associated to a.

We show (Sec. 2) that type inference for S1 can be done in polynomial time w.r.t. the size of the process being type checked. This entails that S2, a mild adaptation of S1, enjoys the same property. S2 adds to S1 the possibility to analyse the values being communicated on channels, when these are first order. Provided we have a polynomial time procedure to handle constraints about first order expressions, type inference for S2 is polynomial.

We then move to more expressive type systems from [4]. In system S3, replicated processes are written $!\kappa.P$, where κ is a maximal sequence of input prefixes (i.e., P is not an input process). To typecheck such a process, the weight of outputs in P must be smaller than the total weight of κ (weights are computed as vectors of weights for any level, and vectors are compared lexicographically). System S4 extends S3 with the possiblity to use a partial order between names in order to typecheck replications whenever the weight of κ and the weight of the continuation P are equal. For instance, even if a and b have the same level, process $P_0 =!p.a.(\overline{p} \,|\, \overline{b})$ can be typed provided a dominates b in the partial order (here $\kappa = p.a$).

Our first main result is to show that for systems S3 and S4, the type inference problem is NP complete. Our proof relies on a reduction from 3SAT. More precisely, we prove that an instance of 3SAT determines a CCS process such that the existence of a typing derivation for the induced process is equivalent to the existence of a solution of the original instance of 3SAT.

To remedy the NP-completeness of S3 and S4, we propose two type systems. In the first type system, called S3', we renounce the lexicographic ordering on levels, and simply add the weight (that is, the level) associated to each name to compute the weights of κ and P. We establish that for this system, type inference amounts to solve linear programming on rational numbers without constants, which can be done in polynomial time. We moreover show that system S3' is strictly more expressive than S3. This constitutes the second main contribution of this paper.

The main improvement of system S4 w.r.t. S3 in terms of expressiveness is the possibility to type replicated processes in which the triggered continuation has the same weight as the outputs needed to trigger it, such as process P_0 above. In system S4', we retain the partial order ingredient inherent to S4, and simplify type checking for replicated inputs. We show soundness of S4' (every typable process terminates), and describe a sound and complete inference procedure for it. We prove that the type inference problem is polynomial for system S4', and illustrate the expressiveness of S4' by revisiting an example from [4] that cannot be directly typed in that system. The definition and analysis of S4' is the third main contribution we present in this paper.

Related Work. There are many works on type systems for the π-calculus. In addition to [4], type systems to ensure termination of π-calculus processes have been studied in [11, 13]. In these works, the technique of logical relations is used to isolate a class of terminating processes.

After the seminal work of [6] for Milner's sorts, several studies of type inference in the π-calculus have been conducted, adressing richer type systems or variants of the calculus, such as [12, 5, 7]. To our knowledge, type systems for termination in the π-calculus have not been studied from the perspective of type inference so far.

Our results are connected with the work on the TyPiCal tool [8], which implements various type-based analyses of π-calculus processes. Other recent developments on the question of termination are presented in [9]. The focus is different: [9] extends the termination type systems to guarantee a stronger property called robust termination. Robust termination is then used to insure lock-freedom (which means that certain communications will eventually succeed). The present work can be useful for refining the verification proposed in [9].

Another relevant reference is the work on Terminator [1], and its recent extension to prove thread termination [2]. While the general objectives are similar to ours, the approaches are technically rather different. [2] deals with a fixed number of threads (without dynamic thread creation), and proves termination by detecting some variance of states, while in this paper, we deal with programs that create threads and channels dynamically.

Paper outline. In Sec. 2, we introduce the π-calculus and recall the type systems from [4]. Sec. 3 is devoted to the complexity of type inference for these systems. We present two systems for which type inference is polynomial: S3' in Sec. 4, and S4' in Sec. 5. Final remarks are given in Sec. 6.

2 Processes and Type Systems

We let $a, b, c, \ldots, p, q, \ldots, x, y, z$ range over an infinite set of *names*. Processes, ranged over using $P, Q, \ldots$, are defined by the following syntax:

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

The constructs of input, replicated input and restriction are binding; we shall often use $x, y, z, \ldots$ for *variables* – names bound by input – and c for *channels* – names bound by restriction. a is called the *subject* of the prefixes in the grammar above. We shall sometimes extend the calculus with first-order values (integers, booleans, ...). This kind of extension is standard (the reader can refer e.g. to [4]), and we shall use it implicitly when necessary. We let $\texttt{os}(P)$ stand for the *multiset* of subjects of outputs that occur in P and do not occur under a replication. Similarly, $\texttt{rs}(P)$ stands for the multiset of names that are restricted in P and such that the restriction does not occur under a replication.

The standard operational semantics of the calculus is omitted. The reduction relation is written $P \longrightarrow P'$.

Type systems. We recall here briefly the definitions of systems S1 to S4. We refer to [4] for detailed explanations and motivating examples accompanying the definitions. To remain close to [4], we give a presentation of the type systems à la Church: each name has a given type a priori, and hence we could also omit mentioning the typing context (ranged over using Γ) in the typing rules. [9] proposes a version à la Curry of these type systems. We keep typing contexts in typing rules in order to ease reading. All systems assign *levels* to names: a typing hypothesis has the form $a : \#^k\widetilde{T}$, to specify that name a transmits tuples of type $\widetilde{T}$, and that the level of a is k, which we write $\mathtt{lvl}(a) = k$ (k is a natural number).

System S1. Below are the typing rules for S1. With respect to simple types, the differences worth mentioning are that level information decorates types, and that the rule for replicated inputs is adapted to control termination.

$$\frac{\Gamma(a) = \#^k\widetilde{x} \qquad \Gamma \vdash P}{\Gamma \vdash a(\widetilde{x}).P} \qquad\qquad \frac{\Gamma(a) = \#^k\widetilde{T} \qquad \Gamma(\widetilde{p}) = \widetilde{T} \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}\langle\widetilde{p}\rangle.P}$$

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1|P_2} \qquad \frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 + P_2} \qquad \frac{}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash (\boldsymbol{\nu} a)\, P}$$

$$\frac{\Gamma(a) = \#^k\widetilde{x} \qquad \Gamma \vdash a(\widetilde{x}).P \qquad \forall n \in \mathtt{os}(P).\, \mathtt{lvl}(n) < k}{\Gamma \vdash !a(\widetilde{x}).P}$$

As explained in the introduction, the control on replications consists in verifying that all names in $\mathtt{os}(P)$ (the multiset of subjects of outputs that occur in P without being guarded by a replication) have a level strictly smaller than $\mathtt{lvl}(a)$.

System S2. System S2 is of minor interest for the purposes of this paper, because type inference can be done almost as for system S1. The only typing rule that differs w.r.t. S1 is the rule for replication:

$$\frac{\Gamma \vdash a(\widetilde{x}).P \qquad \forall \overline{b}\langle\widetilde{v}\rangle \in \mathrm{out}(P).\, \overline{b}\langle\widetilde{v}\rangle \lhd a(\widetilde{x})}{\Gamma \vdash !a(\widetilde{x}).P}$$

$\overline{b}\langle\widetilde{v}\rangle \lhd a(\widetilde{x})$ holds if either $\mathtt{lvl}(b) < \mathtt{lvl}(a)$, or $\mathtt{lvl}(b) = \mathtt{lvl}(a)$ and $\widetilde{v}$, $\widetilde{x}$ are tuples of first-order expressions that can be compared according to some well-founded order. For instance, this is the case if $\widetilde{x} = \langle x_1, x_2\rangle$, $\widetilde{v} = \langle x_1 - 1, x_2 + 2\rangle$, and if tuples of expressions are compared lexicographically (the x_is are natural numbers, and we suppose we can prove $x_1 > 0$).

S2 makes it possible to allow outputs on a in a term of the form $!a(\widetilde{x}).P$: a process like $!a(x).\mathtt{if}\ x > 0\ \mathtt{then}\ \overline{a}\langle x - 2\rangle\ \mathtt{else}\ \overline{b}\langle x\rangle$ is typable in S2 provided $\mathtt{lvl}(b) < \mathtt{lvl}(a)$ (x is a natural number), despite the emission on a.

System S3. The typing rule for replication in S3 is:

$$\frac{\Gamma \vdash \kappa.P \qquad \mathrm{wt}(\kappa) \succ \mathrm{wt}(P)}{\Gamma \vdash !\kappa.P}$$

κ is a maximal sequence of input prefixes (i.e., in $\kappa.P$, P is not an input). The meaning of condition $\mathrm{wt}(\kappa) \succ \mathrm{wt}(P)$ is the following: $\mathrm{wt}(\kappa)$ is defined as a vector of natural numbers $(I_k, \ldots, I_1)$, where I_j is equal to the number of occurrences of names at level j occurring in subject position in κ (k is the level of biggest weight). Similarly, $\mathrm{wt}(P)$ is $(O_k, \ldots, O_1)$, and O_j is the number of occurrences of names at level j in $\mathtt{os}(P)$. Relation $\succ$ is defined as the lexicographical comparison of the weight vectors. For instance, $!p.q.(\overline{p}\,|\,\overline{p})$ is well-typed if $\mathtt{lvl}(p) = 1$, $\mathtt{lvl}(q) = 2$ (the vectors corresponding to κ and P are $(1,1)$ and $(0,2)$ respectively).

In [4], S3 additionally imposes that the name being used as last input subject in κ should be *asynchronous*, that is, no continuation can appear after outputs on this name. This constraint can actually be removed and the proof of soundness adapted rather easily, so we omit it here.

System S4. The typing judgement for S4 is of the form $\Gamma \vdash_{\mathcal{R}} P$, where $\mathcal{R}$ is a strict partial order on the free names of P. Only names having the same simple type can be compared using $\mathcal{R}$.

The syntax of types is extended to include partial order information. If $\mathcal{S}$ is a set of pairs of natural numbers, $p : \#^k_{\mathcal{S}}\widetilde{T}$ specifies that p is of level k, carries a tuple of names of type $\widetilde{T}$, and imposes that whenever $(k,l) \in \mathcal{S}$, (i) the kth and lth components of $\widetilde{T}$ exist and have the same simple type; and (ii) for any tuple of names emitted on p, the kth component of the tuple must dominate the lth component according to the partial order. For instance, if $p : \#^k_{\{(2,3)\}}\langle T_1, T_2, T_2\rangle$ and if the process contains a subterm of the form $\overline{p}\langle u, v, w\rangle.\mathbf{0}$, where u, v, w are free names, then typability imposes that v and w have type T_2 and $v\mathcal{R}w$. Checking this kind of constraints is enforced by the typing rule for outputs. The typing rules for restriction and input are modified w.r.t. S3 in order to extend $\mathcal{R}$ appropriately in the premise (see [4]).

Intuitively, the role of $\mathcal{R}$ is to insure termination in replicated processes for which $\mathrm{wt}(\kappa) = \mathrm{wt}(P)$. In such situations, there is a risk to generate infinite computations by extending relation $\mathcal{R}$ via newly created names. S4 therefore imposes a form of control over restricted names. An occurrence of a restriction is *unguarded* if it does not occur under an input or output prefix. RN stands for the set of names n such that if n appears in prefix subject position, then the continuation process has no unguarded restrictions.

In S4, the condition of S3 in the rule for replication is replaced with $\kappa :\succ P$. $\kappa :\succ P$ holds iff either (i) $\mathrm{wt}(\kappa) \succ \mathrm{wt}(P)$ (as in S3), or (ii) $\mathrm{wt}(\kappa) = \mathrm{wt}(P)$, $\kappa\widehat{\mathcal{R}_\kappa}P$ and the last input subject of κ belongs to RN. For the needs of this paper, we can avoid entering the technical details of the definition of $\widehat{\mathcal{R}_\kappa}$, as we shall use a simplified version of this relation in S4' (and, in analysing the complexity of S4, we shall not resort to (ii) above). Let us just say that this relation is based on a multiset extension of the order $\mathcal{R}$ on free names.

The problem of type inference. In the sequel, we shall always implicitly consider a process P, from which we want to infer an explictly typed process, where inputs and restrictions are decorated with type information. We suppose that P obeys the Barendregt convention, i.e., all its bound names are pairwise distinct and distinct from all the free names of P. Typing constraints between (bound or free) names of P will be generated regardless of scope – we will of course then take scope into account to assert whether a process is typable.

We shall say that a type inference procedure is *polynomial* to mean that it can be executed in polynomial time w.r.t. the size of P. We shall sometimes simply call a type system 'polynomial' to mean that it admits a polynomial time inference procedure.

Type inference for simple types is standard (see, e.g., [12]), and can be done in polynomial time. In the remainder of the paper, we shall implicitly assume that each process we want to type admits a simple typing, and we will concentrate on the question of finding annotations (levels, and, possibly, partial order information) that allow us to ensure typability for the systems we study.

3 Type Inference for Deng and Sangiorgi's Type Systems

3.1 Inference for Systems S1 and S2 is in P

Proposition 1 *Type inference for system S1 is polynomial.*

Proof. We adapt the standard type inference procedure for simple types [12]. We associate to each type a *level variable*. Based on the typing rules, we can generate a set C of constraints consisting of unification constraints on types and inequality constraints (of the form $l_1 < l_2$) on level variables, such that C is satisfiable if and only if P is typable, and the size of C is linear in the size of P. Using the standard unification algorithm, we can transform C into a set C' of inequality constraints on level variables in polynomial time. The satisfiability of C' is equivalent to the acyclicity of the graph induced from C', which can again be checked in polynomial time. Thus, the type inference problem for S1 is polynomial. $\square$

We can adapt this proof to derive a similar result for S2: whenever we find a cycle in the graph, if the cycle only contain names carrying first-order values, instead of failing, we invoke $\lhd$ to check for typability (otherwise, we fail).

Proposition 2 *Suppose we are given relation $\lhd$ together with a procedure to decide $\lhd$ in polynomial time. Then type inference for S2 is polynomial.*

3.2 Hardness of Systems S3 and S4

Theorem 3 *The type inference problem for system S3 is NP-complete.*

Proof. Let z be the number of names occurring in P. The problem is in NP because trying one of the z^z different ways of distributing names into z levels can be done in polynomial time w.r.t. the size of the process and the number of names. It is easy to prove that no more than z levels are required.

We now show that we can reduce 3SAT to the problem of finding a mapping of levels. We consider an instance $\mathcal{I}$ of 3SAT: we have n clauses $(C_i)_{i\leq n}$ of three literals each, $C_i = l_i^1, l_i^2, l_i^3$. Literals are possibly negated propositional variables taken from a set $V = \{v_1, \dots, v_m\}$. The problem is to find a mapping from V to $\{True, False\}$ such that, in each clause, at least one literal is set to True.

All names we use to build the processes below will be CCS names. We fix a name true. To each variable $v_k \in V$, we associate two names x_k and x'_k, and define the process

$$P_k \quad \stackrel{\text{def}}{=} \quad !\mathsf{true}.\mathsf{true}.\overline{x_k}.\overline{x'_k} \mid !x_k.x'_k.\overline{\mathsf{true}} \enspace .$$

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

$$Q_i \quad \stackrel{\text{def}}{=} \quad !n_i^1.n_i^2.n_i^3.\overline{\mathsf{true}} \enspace .$$

We call $\mathcal{I}_{\mathrm{t}}$ the problem of finding a typing derivation in S3 for the process $P \stackrel{\text{def}}{=} P_1 \mid \dots \mid P_m \mid Q_1 \mid \dots \mid Q_n$. Note that the construction of P is polynomial in the size of $\mathcal{I}$.

We now analyse the constraints induced by the processes we have defined.

The level associated to name true is noted t.

- The constraint associated to $!\mathsf{true}.\mathsf{true}.\overline{x_k}.\overline{x'_k}$ is equivalent to

  $$\big(t \geq \mathtt{lvl}(x_k) \wedge t \geq \mathtt{lvl}(x'_k)\big) \ \wedge \ \big(t > \mathtt{lvl}(x_k) \vee t > \mathtt{lvl}(x'_k)\big) \enspace .$$

  The constraint associated to $!x_k.x'_k.\overline{\mathsf{true}}$ is equivalent to

  $$t \leq \mathtt{lvl}(x_k) \ \vee \ t \leq \mathtt{lvl}(x'_k) \enspace .$$

  Hence, the constraint determined by P_k is equivalent to

  $$\big(\mathtt{lvl}(x_k) = t \wedge \mathtt{lvl}(x'_k) < t\big) \quad \vee \quad \big(\mathtt{lvl}(x'_k) = t \wedge \mathtt{lvl}(x_k) < t\big) \enspace . \qquad (1)$$

- The constraint associated to $!n_{i_1}.n_{i_2}.n_{i_3}.\overline{\mathsf{true}}$ is equivalent to

  $$t \leq \mathtt{lvl}(n_i^1) \ \vee \ t \leq \mathtt{lvl}(n_i^2) \ \vee \ t \leq \mathtt{lvl}(n_i^3) \enspace . \qquad (2)$$

We now prove that '$\mathcal{I}_{\mathrm{t}}$ has a solution' is equivalent to '$\mathcal{I}$ has a solution'.

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

Conversely, if $\mathcal{I}_{\mathrm{t}}$ has a solution, then we deduce a boolean mapping for the literals in the original 3SAT problem. Since constraint (1) is satisfied, we can set v_k to True if $\mathtt{lvl}(x_k) = t$, and False otherwise. We thus have that v_k is set to True iff $\mathtt{lvl}(x_k) = t$, iff $\mathtt{lvl}(x'_k) < t$. Hence, because constraint (2) is satisfied, we have that in each clause C_i, at least one of the literals is set to True, which shows that we have a solution to $\mathcal{I}$. $\square$

This proof can be easily adapted to establish the same result for S4: the idea is to 'disable' the use of the partial order, e.g. by adopting a different type for true. We thus get:

Corollary 4 *The type inference problem for System S4 is NP-complete.*

The cause of NP-difficulty. The crux in the proof of Thm. 3 is to use the 'κ component' of S3 to introduce a form of choice: to type process $!a.a'.P$, we cannot know a priori, for $b \in \mathtt{os}(P)$, whether to set $\mathtt{lvl}(a) \geq \mathtt{lvl}(b)$ or $\mathtt{lvl}(a') \geq \mathtt{lvl}(b)$. Intuitively, we exploit this to encode the possibility for booleans to have two values, as well as the choice of the literal being set to True in a clause. By removing the κ component from S3, we get system S1, which is polynomial.

However, it appears that NP-completeness is not only related to κ: indeed, it is possible to define a polynomial restriction of S3 where the choice related to the κ component is still present. Let us call S3" the type system obtained from S3 by imposing *distinctness of levels*: two names can have the same level only if their types are unified when resolving the unification constraints. Note that this is more demanding than having the same simple type: in $\overline{p} \mid \overline{q}$, p and q have the same simple type, but must be given different levels in S3" because their types are not unified during inference.

Although typing process $!a(x).a'(y,z).P$ seems to introduce the same kind of choice as in S3, it can be shown that type inference is polynomial in S3". Intuitively, the reason for this is that there exists a level variable, say α, such that for every constraint on weight vectors determined by the process being typed, the cardinal of α in the continuation process is not greater than the cardinal in κ. We call α a *root level variable*: it can be shown that if no such α exists, then the process is not typable.

This gives a strategy to compute a level assignment for names, and do so in polynomial time: set α to the maximum level, and consider a weight vector constraint $\mathrm{wt}(\kappa) \succ \mathrm{wt}(P)$: if there are as many αs in $\mathrm{wt}(\kappa)$ as in $\mathrm{wt}(P)$, replace the constraint with the equivalent constraint where the αs are removed. Otherwise, the number of αs strictly decreases, which means we can simply get rid of this constraint. We thus obtain an equivalent, smaller problem, and we can iterate this reasoning (if there are no more constraints to satisfy, we pick a random assignment for the remaining levels).

System S3" retains the lexicographical comparison and the κ component from S3, but is polynomial. By Prop. 7 below, since S3" is a restriction of S3, it is less expressive than S3'. In some sense, S3" *'respects the identity of names'*: while in S3' levels are added, and we rely on algebraic calculations on natural numbers, only comparisons between levels are used in S3"; this means that, intuitively, we cannot trade a name a for one or several names whose role in the given process is completely unrelated to the role of a.

4 Summing the Levels Assigned to Names

We now study system S3', in which we renounce the lexicographical comparison between names through levels, and instead add levels to compute the weight of κ and P in a term of the form $!\kappa.P$.

Definition 5 (System S3') *We let* $\mathrm{subj}(\kappa)$ *stand for the multiset of names occurring in subject position in* κ.

System S3' is defined by the same rules as system S3, except that the condition for the replication rule is $\Sigma_{n\in\mathrm{subj}(\kappa)}\mathtt{lvl}(n) > \Sigma_{n\in\mathtt{os}(P)}\mathtt{lvl}(n)$ *(for all* n, $\mathtt{lvl}(n)$ *is a natural number).*

Note that $\mathrm{subj}(\kappa)$ and $\mathtt{os}(P)$ are *multisets*, so that the weight of names having multiple occurrences is counted several times.

Soundness of S3' can be established by adapting the proof for S3 in [4]:

Proposition 6 *System S3' ensures termination.*

Proposition 7 *System S3' is strictly more expressive than S3.*

Proof. We first show that S3' is at least as expressive as S3. We consider a process of the form $P_0 = !\kappa.P$, that can be typed in S3 (κ is a maximal input prefix). We write $(I_k, \dots, I_1)$ and $(O_k, \dots, O_1)$ for the vectors of levels associated to κ and $\mathtt{os}(P)$ respectively (the I_js are natural numbers, and I_j is the number of subject occurrences of names of level j in κ — and similarly for the O_js). We fix an integer b such that $\forall j \in [1 \dots k].\, |O_j - I_j| < b$, and build a S3' typing context for P_0 by assigning level $b^{L(n)}$ to name n, where $L(n)$ denotes the level of n according to the S3-typing of P_0.

Let us show that this induces a correct typing for P_0 in S3'. Because P_0 is typed in S3, there exists u such that $I_k = O_k, I_{k-1} = O_{k-1}, \dots, I_{u+1} = O_{u+1}$ and $I_u > O_u + 1$. We compute the difference of weights between κ in P according to S3': $\mathrm{wt}(\kappa) - \mathrm{wt}(P) = \Sigma_{1\le j\le k}(I_j - O_j)b^j \ge b^u + \Sigma_{1\le j<u}(I_j - O_j)b^j$. The latter quantity is strictly positive by definition of b, which shows that P_0 is S3'-typable.

We can generalize this reasoning by remarking that an arbitrary process Q has a finite number of replications, which allows us to fix a b which is suitable for all replicated subterms of Q.

To show that there are processes which can be typed by system S3' but not by S3, consider $P_1 \stackrel{\mathrm{def}}{=} !a.\overline{b} \mid !b.b.\overline{a}$. P_1 is ill-typed according to S3: the first subterm imposes $\mathtt{lvl}(a) > \mathtt{lvl}(b)$, and the vectors associated to the second subterm are hence of the form $(0, 2)$ and $(1, 0)$, and we do not have $(0, 2) \succ (1, 0)$. By setting $\mathtt{lvl}(a) = 3$ and $\mathtt{lvl}(b) = 2$, we can check that P_1 is typable for S3'. $\square$

Theorem 8 *Type inference for system S3' is polynomial.*

Proof. By inspecting the process to be typed, type inference amounts to find a solution to a system of inequalities of the form $\Sigma_j a_{i,j}.u_j > 0$, where the $a_{i,j}$s are integers and the solution is the vector of the u_js, which are natural numbers. This system has a solution if and only if the system consisting of the inequalities $\Sigma_j a_{i,j}.u_j \geq 1$ has one. We resort to linear programming in rationals to solve the latter problem (we can choose to minimize $\Sigma_j u_j$), which can be done in polynomial time. Because of the shape of inequalities generated by the typing problem, there exists a rational number solution to the inequalities if and only if there exists an integer solution. $\square$

5 Exploiting Partial Orders on Names

5.1 System S4': Definition and Properties

System S4 from [4] is built on top of S3, and improves its expressiveness by allowing the use of partial orders. To define S4', we restrict ourselves to the partial order component of S4, and do not analyse sequences of input prefixes (κ) as in S3: in a term of the form $!a(\widetilde{x}).P$, name a must dominate every name in $\mathtt{os}(P)$, either because it is of higher level, or via the partial order relation.

We now introduce S4'. Let $\mathcal{R}$ be a relation on names, $\mathcal{S}$ a relation on natural numbers, and $\widetilde{x}$ a tuple of names. We define two operators $/$ and $*$ as follows:

$$\mathcal{R} \,/\, \widetilde{x} \;=\; \begin{cases} \emptyset & \text{if } n(\mathcal{R}) \cap \widetilde{x} = \emptyset \\ \{(i,j) \mid x_i \mathcal{R} x_j\} & \text{if } n(\mathcal{R}) \subseteq \widetilde{x} \\ \text{undefined} & \text{otherwise} \end{cases} \qquad \begin{array}{c} \mathcal{S} * \widetilde{x} \;=\; \{(x_i, x_j) \mid i\mathcal{S}j\} \\ \text{if } \max(n(\mathcal{S})) \leq |\widetilde{x}| \end{array}$$

Above, $n(\mathcal{R}) = \{a.\, \exists b.\, a\mathcal{R}b \vee b\mathcal{R}a\}$, $n(\mathcal{S}) = \{i.\, \exists j.\, i\mathcal{S}j \vee j\mathcal{S}i\}$, and $|\widetilde{x}|$ denotes the number of names in $\widetilde{x}$. We also define
$\mathcal{R} \Downarrow_{\widetilde{x}} \;=\; \{(a,b) \mid a, b \notin \widetilde{x} \text{ and } a\mathcal{R}c_1\mathcal{R}\cdots\mathcal{R}c_n\mathcal{R}b \text{ for some } \widetilde{c} \subseteq \widetilde{x} \text{ and } n \geq 0\}$.

The typing rules for S4' are given on Fig. 1. Again, although the type system is defined à la Church, we mention the typing context to ease readability. When writing a judgment of the form $\Gamma \vdash_{\mathcal{R}} P$, we implicitly require that $\mathcal{R}$ does not contain a cycle. Note that w.r.t. system S4 in [4], we relax the constraint that $\mathcal{R}$ should only relate names having the same simple type.

In the rule for replication, $\Gamma \vdash_{\mathcal{R}} a :\succ (N_1, N_2)$ holds if either of the following conditions holds:

(i) $\forall v \in N_1.\mathtt{lvl}(v) < \mathtt{lvl}(a) \wedge \forall v \in N_2.\mathtt{lvl}(v) \leq \mathtt{lvl}(a)$
(ii) $\forall v \in N_2.\mathtt{lvl}(v) < \mathtt{lvl}(a)$
$\wedge \exists b \in N_1.\mathtt{lvl}(b) = \mathtt{lvl}(a) \wedge a\mathcal{R}b \wedge \forall v \in N_1 - \{b\}.\mathtt{lvl}(v) < \mathtt{lvl}(a)$.

(notice that N_1 is a multiset).

The last rule in Fig. 1 is optional; it does not change typability, but makes the correspondence with the constraint generation algorithm more clear. Accordingly, in the rules for parallel composition and choice, we could mention the same relation $\mathcal{R}$ in both premises and in the conclusion — the version of the rules we present is closer to the type inference procedure (see Sec. 5.2).

$$\Gamma \vdash_{\mathcal{R}} \mathtt{0} \qquad \frac{\Gamma \vdash_{\mathcal{R}_1} P \qquad \Gamma \vdash_{\mathcal{R}_2} Q}{\Gamma \vdash_{\mathcal{R}_1+\mathcal{R}_2} P|Q} \qquad \frac{\Gamma \vdash_{\mathcal{R}_1} P \qquad \Gamma \vdash_{\mathcal{R}_2} Q}{\Gamma \vdash_{\mathcal{R}_1+\mathcal{R}_2} P+Q}$$

$$\frac{\Gamma(a) = \sharp^n_{\mathcal{S}}\Gamma(\widetilde{x}) \qquad \Gamma \vdash_{\mathcal{R}} P \qquad \mathcal{S} \supseteq \mathcal{R} \,/\, \widetilde{x}}{\Gamma \vdash_{\mathcal{R}\Downarrow_{\widetilde{x}}} a(\widetilde{x}).P}$$

$$\frac{\Gamma(a) = \sharp^n_{\mathcal{S}}\Gamma(\widetilde{v}) \qquad \Gamma \vdash_{\mathcal{R}} P \qquad \mathcal{R} \supseteq \mathcal{S} * \widetilde{v}}{\Gamma \vdash_{\mathcal{R}} \overline{a}\langle\widetilde{v}\rangle.P} \qquad \frac{\Gamma(c) = \sharp^n_{\mathcal{S}}\widetilde{T} \qquad \Gamma \vdash_{\mathcal{R}} P}{\Gamma \vdash_{\mathcal{R}\Downarrow_c} (\nu c)P}$$

$$\frac{\Gamma(a) = \sharp^n_{\mathcal{S}}\Gamma(\widetilde{x}) \qquad \Gamma \vdash_{\mathcal{R}} P \qquad \mathcal{S} \supseteq \mathcal{R} \,/\, \widetilde{x} \qquad \Gamma \vdash_{\mathcal{R}} a :\succ (\mathtt{os}(P), \mathtt{rs}(P))}{\Gamma \vdash_{\mathcal{R}\Downarrow_{\widetilde{x}}} !a(\widetilde{x}).P}$$

$$\frac{\Gamma \vdash_{\mathcal{R}'} P \qquad \mathcal{R}' \subseteq \mathcal{R}}{\Gamma \vdash_{\mathcal{R}} P}$$

Fig. 1. System S4': Typing Rules

Notice that the partial order can be used for *at most one output* in the continuation process to typecheck a replication. Indeed, by omitting this constraint in case (ii) above, we could typecheck the following divergent process:

$$P_2 \quad \stackrel{\text{def}}{=} \quad !p(a,b,c,d).(!a.\overline{c}.\overline{d} \;|\; !b.(\boldsymbol{\nu}e,f)\,\overline{p}\langle c,d,e,f\rangle) \quad | \quad \overline{p}\langle u,v,w,t\rangle.(\overline{u}\,|\,\overline{v}),$$

by setting $a\mathcal{R}c$ and $a\mathcal{R}d$. In P_2, the subterm replicated at b makes a recursive call to p with two new fresh names; the subterm replicated at a is typed using the partial order twice, and the outputs it triggers feed the loop (a similar example can be constructed to show that we must also forbid using $\mathcal{R}$ twice with the same pair of names).

Proposition 9 *System S4' ensures termination.*

Proof. We suppose that there exists a process P admitting a diverging sequence $\mathcal{D}$: $P = P_1 \longrightarrow P_2 \longrightarrow P_3 \longrightarrow \ldots$, and that P is well-typed according to S4'. Let k be the maximum level assigned to names in the typing of P.

We call I the set of integers i such that the reduction step from P_i to P_{i+1} is obtained by triggering a replicated input whose subject is of level k. We let $S_i \stackrel{\text{def}}{=} \{n \in \mathtt{os}(P_i).\,\mathtt{lvl}(n) = k\}$ (S_i is a multiset).

We remark that the size of S_i cannot grow. Indeed, if the reduction from P_i to P_{i+1} does not trigger a replicated input, this obviously holds. If on the contrary the reduction does, there are two cases: either $i \notin I$, and by maximality of k, no output at level k can be unleashed by triggering an input at level strictly smaller than k; or $i \in I$, and there are two cases again. If the replicated input has been typed using clause (i) of the definition of $:\succ$, then S_{i+1} has one element less than S_i. If clause (ii) has been used, then S_{i+1} has been obtained from S_i by

$$
\begin{array}{l}
\mathtt{Tinf}(\Gamma, \mathtt{0}) = (r, \{r \supseteq \emptyset\}) \\
\mathtt{Tinf}(\Gamma, a(\widetilde{x}).P) = \\
\quad \mathtt{let}\ (r, C_1) = \mathtt{Tinf}(\Gamma, P) \\
\quad\quad C_2 = \{\Gamma(a) = \sharp^l_{r_1} \Gamma(\widetilde{x})\}\ (l, r_1 \text{ fresh}) \\
\quad \mathtt{in}\ (r_2, C_1 \cup C_2 \cup \{r_1 \supseteq r \,/\, \widetilde{x}, r_2 \supseteq r \Downarrow_{\widetilde{x}}\})\ (r_2 \text{ fresh}) \\
\mathtt{Tinf}(\Gamma, !a(\widetilde{x}).P) = \\
\quad \mathtt{let}\ (r, C_1) = \mathtt{Tinf}(\Gamma, P) \\
\quad\quad C_2 = \{\Gamma(a) = \sharp^l_{r_1} \Gamma(\widetilde{x})\}\ (l, r_1 \text{ fresh}) \\
\quad\quad C_3 = \{r_1 \supseteq r \,/\, \widetilde{x}, r_2 \supseteq r \Downarrow_{\widetilde{x}}\}\ (r_2 \text{ fresh}) \\
\quad \mathtt{in}\ (r_2, C_1 \cup C_2 \cup C_3 \cup \{\Gamma \vdash_r a :\succ (\mathtt{os}(P), \mathtt{rs}(P))\}) \\
\mathtt{Tinf}(\Gamma, \overline{a}\langle\widetilde{v}\rangle.P) = \\
\quad \mathtt{let}\ (r, C_1) = \mathtt{Tinf}(\Gamma, P) \\
\quad\quad C_2 = \{\Gamma(a) = \sharp^l_{r_1} \Gamma(\widetilde{v})\}\ (l, r_1 \text{ fresh}) \\
\quad \mathtt{in}\ (r, C_1 \cup C_2 \cup \{r \supseteq r_1 * \widetilde{v}\}) \\
\mathtt{Tinf}(\Gamma, (\nu c)P) = \\
\quad \mathtt{let}\ (r_1, C) = \mathtt{Tinf}(\Gamma, P) \\
\quad \mathtt{in}\ (r, C \cup \{r \supseteq r_1 \Downarrow_c\})\ (r \text{ fresh}) \\
\mathtt{Tinf}(\Gamma, P_1|P_2) = \\
\quad \mathtt{let}\ (r_1, C_1) = \mathtt{Tinf}(\Gamma, P_1) \\
\quad\quad (r_2, C_2) = \mathtt{Tinf}(\Gamma, P_2) \\
\quad \mathtt{in}\ (r, C_1 \cup C_2 \cup \{r \supseteq r_1 + r_2\})\ (r \text{ fresh}) \\
\mathtt{Tinf}(\Gamma, P_1 + P_2) = \mathtt{Tinf}(\Gamma, P_1|P_2)
\end{array}
$$

Fig. 2. Constraint Generation

removing an element a and replacing it with b, with $a\mathcal{R}b$ (by abuse of notation, we write this $S_i\mathcal{R}S_{i+1}$).

Let us now show that I is finite. The above reasoning implies that I contains only a finite number of reductions corresponding to a replicated input that has been typed using clause (i). Hence there exists an index after which all reductions of $\mathcal{D}$ on a name of level k involve a replicated input typed using clause (ii). We observe that between two such reductions, no name of level k can be created, and none can be created either by such a reduction. This means that we have an infinite sequence $S_j\mathcal{R}S_{j+1}\mathcal{R}\ldots$ (using the notation introduced above), which contradicts the fact that the support of $\mathcal{R}$ at level k cannot grow.

Since I is finite, $\mathcal{D}$ has a suffix such that the resulting infinite sequence does not contain any reduction involving a replicated input at a name of level k. We can reason as above for $k-1$, and finally obtain a contradiction. □

5.2 Type Inference for S4'

We now present the type inference procedure for S4', which has two phases: in the first part, we generate constraints, that are solved in the second part.

Constraint generation algorithm The rules of Fig. 2 define the constraint generation phase of type inference. The output of this procedure is a pair (r, C) where r is a relation variable and C consists of:

– unification constraints $T_1 = T_2$
– order constraints $\Gamma \vdash_r a :\succ (N_1, N_2)$
– relation constraints $r \supseteq \mathcal{R}$, where R consists of relation variables, pairs of names, operations such as $+$, $*$, $\Downarrow$, and $/$.

The size of C is polynomial in the size of the process. Note that relation variables range over relations between names, or between integers (when they correspond to '$\mathcal{S}$' components). They are hence 'intrinsically typed', as is the case for operators $*$ and $/$.

The following lemma can be proved easily. (Here, by solution of C, we mean an assignment of type variables to valid types, level variables to levels, and relation variables to strict partial orders that satisfy all the constraints in C).

Lemma 10 *Let $\{v_1, \ldots, v_n\}$ be the set of all the names occuring in P, and $\Gamma = v_1 : \alpha_1, \ldots, v_n : \alpha_n$. If $\mathtt{Tinf}(\Gamma, P) = (r, C)$, then θ is a solution of C if and only if $\theta\Gamma \vdash_{\theta r} P$.*

Constraint solving Constraints are solved through several constraint transformation steps, that we now describe.

– Step 1: By solving the unification constraints in C, we obtain a set C_1 of order constraints and relation constraints.
– Step 2: Eliminate level variables
  For each order constraint $\Gamma \vdash_r a :\succ (N_1, N_2)$, generate necessary conditions

$$\{\mathtt{lvl}(v) \leq \mathtt{lvl}(a) \mid v \in N_1 \cup N_2\}.$$

  Thus, we obtain a set of level constraints $C_2 = \{l_1 \leq l'_1, \ldots, l_k \leq l'_k\}$. Compute a solution of C_2 that is *optimal* in the sense that whenever possible, different levels are assigned to different level variables. (That can be computed as follows. Construct a directed graph G whose node set is $\{l_1, l'_1, \ldots, l_k, l'_k\}$, and whose edge set is $\{(l_i, l'_i)\}$. Compute strongly connected components of G, and unify all the level variables in the same component. Then, perform a topological sort on the strongly connected components, and assign a level to each component.) Then, substitute the solution for each $\Gamma \vdash_r a :\succ (N_1, N_2)$.
– Step 3: Eliminate order constraints $\Gamma \vdash_r a :\succ (N_1, N_2)$
  $\Gamma \vdash_r a {:\succ} (N_1, N_2)$ can be reduced as follows. Check whether $\forall v \in N_1.\mathtt{lvl}(v) < \mathtt{lvl}(a)$ holds. If so, then just remove the constraint. Otherwise, check that for only one $b \in N_1$, $\mathtt{lvl}(b) = \mathtt{lvl}(a)$ holds, and that $\forall v \in N_2.\mathtt{lvl}(v) < \mathtt{lvl}(a)$ holds. If this is the case, replace $\Gamma \vdash_r a :\succ (N_1, N_2)$ with $r \supseteq \{(a, b)\}$. Otherwise, report that the constraints are unsatisfiable.

– Step 4: Solve relation constraints:
We are now left with a set of relation constraints:

$$\{r_1 \supseteq f_1(r_1, \ldots, r_k), \ldots r_k \supseteq f_k(r_1, \ldots, r_k)\} \ .$$

(We assume here that $\{r_1, \ldots, r_k\}$ contains all the relation variables introduced by `Tinf`; otherwise add a trivial constraint $r \supseteq r$.) Here, $f_1, \ldots, f_k$ are monotonic functions on relations (in particular, $\mathcal{R}\Downarrow_{\widetilde{x}}$ is monotonic if we treat 'undefined' as the biggest element). Thus, we can obtain the least solution in a standard manner [10].
Finally, we check that the transitive closure of the solution for each relation variable r is irreflexive. When this is the case, we have a level assignment and a definition of partial orders (between free names, and to decorate types) which are sufficient to deduce a typing derivation for the process being analysed.

Comments about the constraint solving procedure. Step 1 in the procedure above is standard. In Step 2, each order constraint of the form $\Gamma \vdash_r a :\succ (N_1, N_2)$ generates a set of necessary inequalities between level variables. Cycles in the graph that is constructed in this step correspond to level variables that are necessarily identified. The purpose of Step 3 is to get rid of order constraints by determining whether each corresponding subterm is typed using clause (i) or clause (ii) of the definition of $:\succ$. If all inequalities corresponding to the order constraint are satisfied in a strict sense by the level assignment, by clause (i), there is nothing to do. When this is not the case, we necessarily rely on clause (ii): we check that the corresponding hypotheses are satisfied, and generate a relation constraint. Relation constraints are handled in Step 4.

It can be remarked that type inference gives priority to clause (i) to type replicated terms. For instance, consider process $P_3 = \overline{p}\langle a, b\rangle \mid p(x, y)!x.\overline{y}$. Type inference assigns a type of the form $\#^1\langle \#^2T, \#^1T'\rangle$ to p. Alternatively, we can choose to set $p : \#^1_{\{(1,2)\}}\langle \#^1T, \#^1T'\rangle$, i.e., use clause ($ii$). By construction, Step 2 assigns different levels whenever possible, and hence chooses the former typing.

Theorem 11 *The type inference procedure for S4' is sound and complete w.r.t. the typing rules, and runs in polynomial time.*

Souundness and completeness follow from Lemma 10 and the fact that each of the above steps preserves the satisfiability of constraints. For the complexity result, `Tinf` runs in polynomial time and generates constraints of polynomial size. In turn, each step of the constraint solving part runs in time polynomial in the size of the constraints.

6 Conclusion

We have studied the complexity of type inference for the type systems of [4], and shown how the NP complete type systems can be simplified in order to get a polynomial type inference procedure.

A question that remains to be addressed is how to enrich system S3' with the possibility to use partial orders, in order to get closer to systems S4 or S4' in terms of expressiveness. In S4, the partial order can be used when the vector of weights remains the same, while in S4' the vector of weights can even increase when the partial order is used. How to adapt S4 or S4' to a system where weights are summed (as natural numbers) is not clear to us at the moment.

A natural extension of this work is to experiment with the results we have presented. TyPiCal already implements a type inference algorithm for a type system obtained by combining systems S1 to S4, as reported in [9]. The parts of this combined type system that are related to S3 and S4 are treated using a heuristic, incomplete, polynomial algorithm, because of the NP-completeness result we have shown in Sec. 3. It is left for future work to implement S3' and S4' discussed in the paper. For that purpose, a main remaining issue is how to integrate S3' and S4' with S2. As hinted above, our results could also be useful for the developments presented in [9].

Acknowledgements. We thank Alain Darte for his help in finding the reduction used for Thm. 3.

References

1. The Terminator Project: proof tools for termination and liveness. `http://research.microsoft.com/terminator/`, 2007.
2. B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In *Proc. of PLDI'07*, pages 320–330. ACM, 2007.
3. Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. pages 265–276, 2007.
4. Y. Deng and D. Sangiorgi. Ensuring Termination by Typability. *Information and Computation*, 204(7):1045–1082, 2006.
5. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit Typing à la ML for the Join-Calculus. In *Proc. of CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 196–212. Springer, 1997.
6. S. J. Gay. A Sort Inference Algorithm for the Polyadic Pi-Calculus. In *Proc. of POPL'93*, pages 429–438. ACM Press, 1993.
7. A. Igarashi and N. Kobayashi. Type Reconstruction for Linear Pi-Calculus with I/O Subtyping. *Information and Computation*, 161(1):1–44, 2000.
8. N. Kobayashi. TyPiCal: Type-based static analyzer for the Pi-Calculus. available from `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`, 2007.
9. N. Kobayashi and D. Sangiorgi. From Deadlock-Freedom and Termination to Lock-Freedom. submitted, 2007.
10. J. Rehof and T. Mogensen. Tractable Constraints in Finite Semilattices. *Science of Computer Programming*, 35(2):191–221, 1999.
11. D. Sangiorgi. Termination of Processes. *Mathematical Structures in Computer Science*, 16(1):1–39, 2006.
12. V. T. Vasconcelos and K. Honda. Principal Typing Schemes in a Polyadic pi-Calculus. In *Proc. of CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 1993.

13. N. Yoshida, M. Berger, and K. Honda. Strong Normalisation in the Pi-Calculus. *Information and Computation*, 191(2):145–202, 2004.