Ensuring Termination by Typability ⋆

Yuxin Deng[1] and Davide Sangiorgi[2]

[1] INRIA and University Paris 7, France
[2] University of Bologna, Italy

Abstract. A term terminates if all its reduction sequences are of finite length. We show four type systems that ensure termination of well-typed π-calculus processes. The systems are obtained by successive refinements of the types of the simply typed π-calculus. For all (but one of) the type systems we also present upper bounds to the number of steps well-typed processes take to terminate. The termination proofs use techniques from term rewriting systems.
We show the usefulness of the type systems on some non-trivial examples: the encodings of primitive recursive functions, the protocol for encoding separate choice in terms of parallel composition, a symbol table implemented as a dynamic chain of cells.

1 Introduction

A term terminates if all its reduction sequences are of finite length. As far as programming languages are concerned, termination means that computation in programs will eventually stop. In computer science termination has been extensively investigated in term rewriting systems [7, 5] and λ-calculi [9, 4] (where strong normalization is a synonym more commonly used). Termination has also been discussed in process calculi, notably the π-calculus [17, 23], a formalism widely used to address issues related to concurrency.

Indeed, termination is interesting in concurrency. For instance, if we interrogate a process, we may want to know that an answer is eventually produced (termination alone does not guarantee this since other security properties e.g. deadlock-freedom [12] are also involved, but termination would be the main ingredient in a proof). Similarly, when we load an applet we would like to know that the applet will not run for ever on our machine, possibly absorbing all the computing resources (a 'denial of service' attack). In general, if the lifetime of a process can be infinite, we may want to know that the process does not remain alive simply because of non-terminating internal activity, and that, therefore, the process will eventually accept interactions with the environment.

Languages of terminating processes are proposed in [27] and [22]. In both cases, the proofs of termination make use of logical relations, a well-known technique from functional languages. The languages of terminating processes so obtained are however rather 'functional', in that the structures allowed are similar to those derived when encoding functions as processes. In particular, the languages are very restrictive on nested inputs (that is, the possibility of having free inputs underneath other inputs), and recursive inputs (that is, replications $!a(x).P$ in which the body P can recursively call the guard a of the replication). Such patterns are entirely forbidden in [27]; nested inputs are allowed in [22] but in a very restricted form. For example, the process

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

⋆ Work partially supported by the EU project PROFUNDIS and EU Integrated Project Sensoria. An extended abstract of this paper has been presented at IFIP TCS 2004.

is legal neither for [27] nor for [22]. The restrictions in [27, 22] actually rule out also useful functional processes, for instance

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

which represents the factorial function. See Section 7 for more discussions on related work.

In this paper, we consider several type systems and well-typed processes under each system are ensured to terminate. First, in Section 3, we present a core type system, which adds level information to the types of the simply typed π-calculus. Then, in Sections 4 to 6 we show three refinements of the core system. Nested inputs and recursive inputs are the main patterns we focus on. For all the type systems (except for the second one, which can capture primitive recursive functions) we also present upper bounds to the number of steps well-typed processes take to terminate. Such bounds depend on the structures of the processes and on the types of the names in the processes. We show the usefulness of the type systems on some non-trivial examples: the encodings of primitive recursive functions, the protocol for encoding separate choice in terms of parallel composition from [18, 23], a symbol table implemented as a dynamic chain of cells from [11, 21].

Roughly, for each type system, to prove termination we choose a measure which decreases after finite steps of reduction. To compare two measures, we exploit *lexicographic* and *multiset orderings*, well-known techniques in term rewriting systems [7, 6]. For the core type system, the measure is just a vector recording, for each level, the numbers of outputs (unguarded by replicated inputs) at channels with that level in the type. For the extended type systems, the ideas are similar, but the measures become more sophisticated since we allow them to decrease after some finite (unknown and variable) number of reductions, up-to some commutations of reductions and process manipulations.

2 The simply typed π-calculus

We begin with a brief overview of the simply typed π-calculus [23]. We presuppose an infinite set $\mathcal{N}$ of names and let $a, b, ..., x, y, ...$ range over it. A channel is a name that may be used to engage in communications. As it is customary in the π-calculus, we make no syntactic difference between channels and variables. The values, denoted by v, w, are the objects that can be exchanged along channels.

In this work we only study type systems *à la Church*, and each name is assigned a type a priori. We write $x : T$ to mean that the name x has type T. A judgment $\vdash P$ says that P is a well-typed process, and $\vdash v : T$ says that v is a well-typed value of type T. The syntax of types and processes as well as the typing rules are shown in Table 1. We use the usual constructors of monadic π-calculus: *inaction, input, output, parallel composition, sum, restriction* and *replication*. In the input prefix $a(b)$ name a is the subject and name b is the object of the prefix, similar for the output prefix $\bar{a}b$. Free names, bound names, names of process P and the subject of a prefix α, written $fn(P)$, $bn(P)$, $n(P)$ and $subj(\alpha)$ respectively, are defined in the standard way. We also assume α-conversion implicitly in order to avoid name capture and keep the uniqueness of every bound name. Substitutions, ranged over by $\sigma, \sigma', \sigma_i, \cdots$, are maps from names to names. The transition rules are presented in the so-called early style in Table 2, where α ranges over actions. The symmetric rules of par1, com1 and sum1 are omitted. Sometime we use the notation $\stackrel{\alpha}{\Longrightarrow}$ which is an abbreviation for $\Longrightarrow\stackrel{\alpha}{\longrightarrow}\Longrightarrow$, where $\Longrightarrow$ is the reflexive and transitive closure of $\stackrel{\tau}{\longrightarrow}$.

For simplicity we only consider two basic types: `Bool`, for boolean values, and `Nat`, for natural numbers. Values of basic types are said to be of first-order because, unlike channels (names of link type), they cannot carry other values. We also assume some basic operations

$$
\begin{array}{lll}
S, T ::= V \mid L & & \text{types} \\
V ::= L \mid \texttt{Bool} \mid \texttt{Nat} & & \text{value types} \\
L ::= \sharp V & & \text{link types} \\
\\
v, w ::= x \mid true, false \mid 0, 1, 2, \cdots & & \text{values} \\
P, Q ::= \mathsf{0} \mid a(x).P \mid \bar{a}v.P \mid P \mid P \mid P + P \mid \nu a P \mid\ !a(x).P & & \text{processes}
\end{array}
$$

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

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

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

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

$$
\textsf{in}\ \frac{}{a(x).P \xrightarrow{av} P\{v/x\}} \qquad
\textsf{out}\ \frac{}{\bar{a}v.P \xrightarrow{\bar{a}v} P}
$$

$$
\textsf{par1}\ \frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad
\textsf{com1}\ \frac{P \xrightarrow{(\nu \widetilde{b})\bar{a}v} P' \quad Q \xrightarrow{av} Q' \quad \widetilde{b} \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\tau} (\nu \widetilde{b})(P' \mid Q')}
$$

$$
\textsf{res}\ \frac{P \xrightarrow{\alpha} P' \quad a \notin n(\alpha)}{\nu a P \xrightarrow{\alpha} \nu a P'} \qquad
\textsf{open}\ \frac{P \xrightarrow{(\nu \widetilde{b})\bar{a}v} P' \quad c \in fn(v) - \{\widetilde{b}, a\}}{\nu c P \xrightarrow{(\nu \widetilde{b}c)\bar{a}v} P'}
$$

$$
\textsf{sum1}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad
\textsf{rep}\ \frac{}{!a(x).P \xrightarrow{av} !a(x).P \mid P\{v/x\}}
$$

Table 2. Transition rules

$$
\begin{array}{rlrl}
wt(0) & \stackrel{\text{def}}{=} \mathsf{0} & wt(\bar{a}v.P) & \stackrel{\text{def}}{=} wt(P) + \mathsf{0}_{lv(a)} \\
wt(!a(x).P) & \stackrel{\text{def}}{=} \mathsf{0} & wt(P \mid Q) & \stackrel{\text{def}}{=} wt(P) + wt(Q) \\
wt(a(x).P) & \stackrel{\text{def}}{=} wt(P) & wt(P+Q) & \stackrel{\text{def}}{=} max\{wt(P), wt(Q)\} \\
wt(\nu aP) & \stackrel{\text{def}}{=} wt(P) & &
\end{array}
$$

The next lemma says that weight is a good measure because it decreases at each reduction step. This property leads naturally to the termination theorem of well-typed processes, by the well-foundedness of weight.

Lemma 1. *Suppose $\mathcal{T} \vdash P$ and $P \xrightarrow{\tau} P'$, then $wt(P') \prec wt(P)$.*

Proof. By induction on transitions. The cases are simple. We only need the following results:

1. If $\mathcal{T} \vdash P$ and $P \xrightarrow{av} P'$, then $wt(P') \prec wt(P) + \mathsf{0}_{lv(a)}$;
2. If $\mathcal{T} \vdash P$ and $P \xrightarrow{\bar{a}v} P'$, then $wt(P') \preceq wt(P) - \mathsf{0}_{lv(a)}$.

□

Theorem 2. *If $\mathcal{T} \vdash P$, then P terminates.*

Proof. By induction on the weight of well-typed processes.

- Base case: All processes with weight $\mathsf{0}$ are terminating because they have no active output.
- Inductive step: Suppose all processes with weights less than $wt(P)$ are terminating. We show that P is also terminating. Consider the set $I = \{i \mid P \xrightarrow{\tau} P_i\}$. For each $i \in I$ we know that: (i) $\mathcal{T} \vdash P_i$ by the subjection reduction property of $\mathcal{T}$, (ii) $wt(P_i) \prec wt(P)$ by Lemma 1. So each such P_i is terminating by induction hypothesis, which ensures that P is terminating.

□

The type system $\mathcal{T}$ provides us with a concise way of handling nested inputs. For example, let $a : \sharp^1\sharp^1\mathtt{Nat}, b : \sharp^2\mathtt{Nat}, c : \sharp^1\mathtt{Nat}$, then process (1) is well-typed and therefore terminating. Similarly, process (4) is well-typed if the types of a, b and c are $\sharp^2\mathtt{Nat}, \sharp^3\mathtt{Nat}$ and $\sharp^1\mathtt{Nat}$, respectively.

Lemma 1 implies that the weight of a process gives us a bound on the time that the process takes to terminate. Let the *size* of a process be the whole number of literals in the process expression, then we have the following result.

Proposition 3. *Let n and k be the size and the highest level in a well-typed process P, respectively. Then P terminates in polynomial time $\mathcal{O}(n^k)$.*

Proof. Let $wt(P)$ be $\langle n_k, ..., n_1 \rangle$, thus $\sum_{i=1}^{k} n_i < n$. The worst case is that when an active output of level i is consumed, all (less than n) new active outputs appear at level $i-1$. Hence one output occurrence of level i gives rise to at most $f(i)$ steps of reduction, where

$$
f(i) = \begin{array}{ll} 1 & \text{if } i = 1 \\ 1 + n * f(i-1) & \text{if } i > 1. \end{array}
$$

In other words,

$$
f(i) = \sum_{j=0}^{i-1} n^j = \frac{n^i - 1}{n - 1}.
$$

Since the weight of P is $\langle n_k, ..., n_1 \rangle$, the length of any reduction sequence from P is bounded by $\sum_{i=1}^{k} n_i * f(i)$. As

$$\sum_{i=1}^{k} n_i * f(i) \leq \sum_{i=1}^{k} n_i * f(k) = (\sum_{i=1}^{k} n_i) * f(k) < n * f(k) = \frac{n(n^k - 1)}{n - 1}$$

we know that P terminates in time $\mathcal{O}(n^k)$. □

As a consequence of Proposition 3 we are not able to encode the simply typed λ-calculus into the π-calculus with type system $\mathcal{T}$, according to the known result that computing the normal form of a non-trivial λ-term cannot be finished in elementary time [24, 14]. We shall see in the next section an extension of $\mathcal{T}$ that makes it possible to encode all primitive recursive functions (some of which are not representable in the simply typed λ-calculus).

4 Allowing limited forms of recursive inputs

The previous type system allows nesting of inputs but forbids all forms of recursive inputs (i.e. replications $!a(x).P$ with the body P having active outputs at channel a). In this and the following sections we study how to relax this restriction.

4.1 The type system

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

$$\begin{aligned} P &\stackrel{\text{def}}{=} \bar{a}\langle 10 \rangle \mid !a(n).\ \texttt{if}\ n > 0\ \texttt{then}\ (\bar{a}\langle n-1 \rangle \mid \bar{a}\langle n-1 \rangle) \\ &\stackrel{\tau}{\longrightarrow} \bar{a}\langle 9 \rangle \mid \bar{a}\langle 9 \rangle \mid !a(n).\ \texttt{if}\ n > 0\ \texttt{then}\ (\bar{a}\langle n-1 \rangle \mid \bar{a}\langle n-1 \rangle) \end{aligned}$$

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

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

In the typing rule, in any replication $!a(x).P$ we compare the active outputs in P with the input $a(x)$ using the relation $\lhd$ below. We have that $\bar{b}v \lhd a(x)$ holds in two cases: (1) b has a lower level than a; (2) b and a have the same level, but the object v of b is provably smaller than the object x of a. For this, we assume a mechanism for evaluating (possibly open) natural number expressions that allows us to derive assertions such as $x - 29 + 4 * 7 < x$. We should stress that this evaluation mechanism is an orthogonal issue, completely independent from our type system; thus we do not include it in the type system. We adopt an eager reduction strategy, thereby the expression in an output is evaluated before the output fires.

Definition 4. *Let $a : \sharp^n S$ and $b : \sharp^m T$. We write $\bar{b}v \lhd a(x)$ if one of the two cases holds: (i) $m < n$; (ii) $m = n$, $S = T = \texttt{Nat}$ and $v < x$.*

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

$$\textsf{T-rep}\ \frac{a : \sharp^n T \quad x : T \quad \vdash P \quad \forall \bar{b}v \in out(P), \bar{b}v \lhd a(x)}{\vdash !a(x).P}$$

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

To a given process P and level i, with $0 < i \leq k$ and k the highest level in P, we assign a unique Nat-multiset $\mathcal{M}_{P,i} = [n_1, \cdots, n_l]$, with $n_j \in \mathsf{N} \cup \{\infty\}$ for all $j \leq l$. (Here we consider ∞ as the upper bound of the infinite set N.) Intuitively, this multiset is obtained as follows. For each active output $\bar{b}v$ in P with $lv(b) = i$, there are three possibilities. If v is a constant value ($v \in \mathsf{N}$), then v is recorded in $\mathcal{M}_{P,i}$. If v contains variables of type `Nat`, then a ∞ is recorded in $\mathcal{M}_{P,i}$. Otherwise, v is not of type `Nat` and thus contributes nothing to the Nat-multiset. For instance, suppose $a : \sharp^3\texttt{Nat}, b : \sharp^2\texttt{Nat}, c : \sharp^1\texttt{Nat}$ and $P \stackrel{\text{def}}{=} \bar{a}\langle 1\rangle \mid \bar{a}\langle 1\rangle \mid \bar{b}\langle 2\rangle \mid !a(n).\bar{b}\langle n+1\rangle \mid b(n).\bar{c}\langle n\rangle$, then $\mathcal{T}' \vdash P$ and there are three Nat-multisets: $\mathcal{M}_{P,3} = [1,1]$, $\mathcal{M}_{P,2} = [2]$ and $\mathcal{M}_{P,1} = [\infty]$. Formally, we define $\mathcal{M}_{P,i}$ as follows:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{M}_{\mathsf{0},i} & \stackrel{\text{def}}{=} & [\,] & \mathcal{M}_{\nu a P,i} & \stackrel{\text{def}}{=} & \mathcal{M}_{P,i} \\
\mathcal{M}_{!a(x).P,i} & \stackrel{\text{def}}{=} & [\,] & \mathcal{M}_{P|Q,i} & \stackrel{\text{def}}{=} & \mathcal{M}_{P,i} \uplus \mathcal{M}_{Q,i} \\
\mathcal{M}_{a(x).P,i} & \stackrel{\text{def}}{=} & \mathcal{M}_{P,i} & \mathcal{M}_{P+Q,i} & \stackrel{\text{def}}{=} & \mathcal{M}_{P,i} \uplus \mathcal{M}_{Q,i} \\
\mathcal{M}_{\bar{a}v.P,i} & \stackrel{\text{def}}{=} & \begin{cases} \mathcal{M}_{P,i} \uplus [v] & \text{if } a : \sharp^i\texttt{Nat} \text{ and } v \in \mathsf{N} \\ \mathcal{M}_{P,i} \uplus [\infty] & \text{if } a : \sharp^i\texttt{Nat} \text{ and } fvn(v) \neq \emptyset \\ \mathcal{M}_{P,i} & \text{otherwise} \end{cases} & & &
\end{array}$$

where $fvn(v)$ is the set of variables of type `Nat`. We combine a set of Nat-multisets $\{\mathcal{M}_{Q,i} \mid 0 < i \leq k\}$ with the weight of Q (as defined in the previous section), $wt(Q) = \langle n_k, \cdots, n_1\rangle$, so as to get a *compound vector* $t_Q = \langle(\mathcal{M}_{Q,k}; n_k), \cdots, (\mathcal{M}_{Q,1}; n_1)\rangle$. For the above example $wt(P) = \langle 2, 1, 1\rangle$, so $t_P = \langle([1,1]; 2), ([2]; 1), ([\infty]; 1)\rangle$.

The order $\prec$ is extended to compound vectors as follows.

Definition 5. *Suppose $t_P = \langle v_k, \cdots, v_1\rangle$ and $t_Q = \langle u_k, \cdots, u_1\rangle$, where $v_i = \mathcal{M}_{P,i}; n_i$ and $u_i = \mathcal{M}_{Q,i}; n'_i$ for $0 < i \leq k$.*

1. *$v_i \prec u_i$ if $\mathcal{M}_{P,i} <_{mul} \mathcal{M}_{Q,i} \vee (\mathcal{M}_{P,i} = \mathcal{M}_{Q,i} \wedge n_i < n'_i)$*
2. *$t_P \prec t_Q$ if $\exists i \leq k$, $v_i \prec u_i$ and $\forall j > i$, $v_j = u_j$.*

The above definition should be self-explanatory because we have followed the usual way of extending orderings to multisets, products and strings. Using compound vectors as the measure, we can build, with similar proof schemas, the counterparts of Lemma 1 and Theorem 2.

Lemma 6. *If $\mathcal{T}' \vdash P$ and $P \xrightarrow{\tau} P'$ then $t_{P^0} \prec t_P$.*

Proof. By induction on transitions. □

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

Proof. The result follows from Lemma 6. □

Note that the measure used here is much more powerful than that in Section 3. With weights, we can only prove the termination of processes which always terminate in polynomial time. By using compound vectors, however, as we shall see in Section 4.2, we are able to capture the termination property of some processes which terminate in time $\mathcal{O}(f(n))$, where $f(n)$ a is primitive recursive function. For example, we can write a process to encode the *repeated exponentiation*, where $E(0) = 1$, $E(n+1) = 2^{E(n)}$. Once received a number n, the process does internal computation in time $\mathcal{O}(E(n))$ before sending out its result.

Surprisingly, the proof of Theorem 7 is not much more complicated than that of Theorem 2. This is due to the well-designed compound vectors that combine lexicographic and multiset orderings. See e.g. [6, 7] for the usefulness of the two orderings in term rewriting systems.

4.2 Example: primitive recursive functions

For simplicity of presentation, we have concentrated mainly on monadic communications. It is easy to extend our calculus and type system to allow polyadic communications and an if-then-else construct [3] (which are needed in many applications). For instance, with polyadicity, for $\bar{b}\langle \tilde{\mathfrak{e}} \rangle \triangleleft a(\tilde{\mathfrak{e}})$ to hold, there are two possibilities: (i) either the level of a is higher than that of b, or (ii) the two names have the same level but at least one argument of first-order type decreases its value ($y_i < x_i$), and the other first-order arguments do no increase ($y_j \leq x_j$). By appropriately modifying the definition of Nat-multiset, we can show that Theorem 7 still holds. For conditionals, we can extend the definition of weight in this way: $wt(\texttt{ if } b \texttt{ then } P \texttt{ else } Q) = max\{wt(P), wt(Q)\}$.

The advantage of $\mathcal{T}'$ over $\mathcal{T}$ lies in the fact that primitive recursive functions can now be captured, according to the standard encoding of functions as processes [16, 23].

Definition 8. (Primitive recursive functions [1]) *The class of primitive recursive functions consists of those functions that can be obtained by repeated application of composition and primitive recursion starting with (1) the successor function, $S(x) = x+1$, (2) the zero function, $N(x) = 0$, (3) the generalized identity functions $U_i^{(n)}(x_1, \cdots, x_n) = x_i$, with the generating rules for composition and primitive recursion being*

1. *Composition* $\quad h(x_1, \cdots, x_n) = f(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n))$
2. *Primitive recursion*

$$r(0, x_2, \cdots, x_n) = f(x_2, \cdots, x_n)$$
$$r(x_1 + 1, x_2, \cdots, x_n) = g(x_1, r(x_1, \cdots, x_n), x_2, \cdots, x_n)$$

Proposition 9. *All primitive recursive functions can be represented as terminating processes in the π-calculus.*

Proof. The representation follows from Milner's encoding of λ-terms into π-processes [16]. In a similar way (see e.g. [16, 23]) can the correctness of the following five encodings be verified.

We represent a function $f(\tilde{\mathfrak{e}})$ as a process F_p which has replicated input like $!p(\tilde{\mathfrak{e}}, r).R$, where name p is called the port of F, with type $T_{m,n} = \sharp^m(\mathbb{N}\mathtt{at}, \sharp^n\mathtt{Nat})$ where $m > n$. After receiving via p some arguments $\tilde{\mathfrak{e}}$ and a return channel r, process R does some computation, and finally the result is delivered at r. For the three basic functions, the results are returned immediately. Below we write $\bar{r}\langle v \rangle$ instead of $\bar{r}v$ to highlight the output value v.

[3] For convenience of presentation, in the rest of this paper we sometimes use an if-then-else construct. Its typing rules are straightforward (similar to those of the sum construct).

1. The zero function $N_p \stackrel{\text{def}}{=} !p(x,r).\bar{r}\langle 0\rangle$.
2. The successor function $S_p \stackrel{\text{def}}{=} !p(x,r).\bar{r}\langle x+1\rangle$.
3. The identity functions $U_{ip} \stackrel{\text{def}}{=} !p(\tilde{\mathbf{e}},r).\bar{r}\langle x_i\rangle$.
   By assigning to p the type $T_{2,1}$, processes N_p, S_p and U_{ip} defined above are typable in our core type system $\mathcal{T}$, thus typable in $\mathcal{T}'$.
4. Composition Suppose that G_{ip_i} is defined for g_i with the type of p_i being T_{m_i,n_i} for all $1 \le i \le m$, and F_q is defined for f with the type of q being T_{m^0,n^0}. By induction hypothesis, they are well typed in $\mathcal{T}'$. Then we can define H_p for h as:

$$\begin{aligned} H_p \stackrel{\text{def}}{=} !p(\tilde{\mathbf{e}},r).(\nu \tilde{\mathbf{e}}\tilde{\mathbf{e}}q)(G_{1p_1} \mid \bar{p_1}\langle \tilde{\mathbf{e}}, r_1\rangle \mid \cdots \mid G_{mp_m} \mid \bar{p_m}\langle \tilde{\mathbf{e}}, r_m\rangle \\ \mid r_1(y_1).\cdots.r_m(y_m).\bar{q}\langle \tilde{\mathbf{e}}, r\rangle \mid F_q) \end{aligned}$$

   Let $m'' = max\{m_1,\cdots,m_m,m'\}+1$ and give name p the type T_{m^{00},n^0}. It can be easily checked that process H_p is typable in $\mathcal{T}'$.
5. Primitive recursion Suppose that F_q is defined for f with the type of q being T_{m_1,n_1}, and G_{p^0} is defined for g with the type of p' being T_{m_2,n_2}. By induction hypothesis they are well typed in $\mathcal{T}'$. We define R_p as follows.

$$\begin{aligned} R_p \stackrel{\text{def}}{=} !p(\tilde{\mathbf{e}},r).\ &\texttt{if } x_1 = 0 \texttt{ then } (\nu q)(F_q \mid \bar{q}\langle x_2,\cdots,x_n,r\rangle) \\ &\texttt{else } (\nu r')(\bar{p}\langle x_1-1,x_2,\cdots,x_n,r'\rangle \\ &\qquad \mid r'(y).(\nu p')(G_{p^0} \mid \bar{p'}\langle x_1-1,y,x_2,\cdots,x_n,r\rangle)) \end{aligned}$$

   Let $m = max\{m_1,m_2\}+1$ and give type T_{m,n_2} to p. It is easy to see that H_p is well typed in $\mathcal{T}'$. □

For the process F in (2), which represents the factorial function, it is typable if we give name a the type $\sharp^2(\texttt{Nat},\sharp^1\texttt{Nat})$. By contrast, the encoding of functions that are not primitive recursive may not be typable. An example is Ackermann's function. See Appendix A for more detailed discussions.

5 Asynchronous names

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

$$\begin{aligned} P &\stackrel{\text{def}}{=} \bar{a} \mid \bar{b} \mid !a.b.\bar{a} \\ &\stackrel{\tau}{\longrightarrow} \bar{b} \mid b.\bar{a} \mid !a.b.\bar{a} \\ &\stackrel{\tau}{\longrightarrow} \bar{a} \mid !a.b.\bar{a} \end{aligned}$$

The weight of P does not decrease after the first step of reduction (we consume a copy of $\bar{a}$ but liberate another one). Only after the second reduction does the weight decrease. Further, P might run in parallel with another process, say $Q \stackrel{\text{def}}{=} !b.b.b.(\bar{b} \mid \bar{b})$, that interferes with P by requiring a communication at b and prevents the second reduction from happening:

$$
\begin{array}{l}
\quad P \mid Q \\
\xrightarrow{\tau} \bar{b} \mid b.\bar{a} \mid !a.b.\bar{a} \mid Q \\
\xrightarrow{\tau} b.\bar{a} \mid !a.b.\bar{a} \mid b.b.(\bar{b} \mid \bar{b}) \mid Q.
\end{array}
$$

This example illustrates two new problems that we have to consider: the weight of a process may not decrease at every step; because of interferences and interleaving among the activities of concurrent processes, consecutive reductions may not yield "atomic blocks" after which the weight decreases. (The weight of $P \mid Q$ increases after two steps of reduction.)

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

5.1 Proving termination with asynchronous names

A name a is *asynchronous* if all outputs with subject a are followed by $\mathsf{0}$. That is, if $\bar{a}v.P$ appears in a process then $P = \mathsf{0}$. A convenient way of distinguishing between synchronous and asynchronous names is using Milner's sorts [15]. Thus we assume two sorts of names, AN and SN, for asynchronous and synchronous names respectively, with the requirement that all names in AN are syntactically used as asynchronous names. We assume that all processes are well-sorted in this sense and will not include the requirements related to sorts in our type systems. (We stick to using both asynchronous and synchronous names instead of working on asynchronous π-calculus, because synchronous π-calculus is sometimes useful – see for instance the example in Section 6.2. However, all the results in this paper are valid for asynchronous π-calculus as well.)

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

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

This addition is not necessary – it only simplifies the presentation. It is partly justified by the usefulness of input sequences in applications. (It also strongly reminds us of the input pattern construct of the Join-calculus [8]). We call κ an input pattern. Note that all but the last name in κ are required to be asynchronous. As far as termination is concerned, we believe that the constraint – and therefore the distinction between asynchronous and synchronous names – can be lifted. However, we do not know how to prove Theorem 10 without it.

To avoid problems of ambiguity in input patterns (for example, in the process $!a.b.\mathsf{0}$, the ambiguity is whether the input pattern is a or $a.b$ [4]), some extra bracketing can be used. For instance, $!(a.b).\mathsf{0}$ would indicate that the input pattern is $a.b$. We will not do so, because in our uses the input pattern will always be clear from the context.

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

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

[4] Note that the choice of input prefixes is relevant for typing. For example, the process $!(a.b).\bar{a}$ is typable in our type system, but $!(a).b.\bar{a}$ is not.

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

Theorem 10. *If $\mathcal{T}'' \vdash P$ then P terminates.*

Below we briefly explain the structure of the proof and proceed in four steps. Firstly, we decorate processes and transition rules with tags, which indicate the origin of each reduction: whether it is caused by calling a replicated input, a non-replicated input or it comes from an if-then-else structure. This information helps us to locate some points, called *landmarks*, that are reached by consuming the last input prefix of an input pattern in a reduction path. If a process performs a sequence of reductions that are locally ordered (that is, all and only the input prefixes of a given input pattern are consumed), then the process goes from a landmark to the next one and decreases its weight (Lemma 12). (This is not sufficient to guarantee termination, since in general the reductions of several input patterns may interleave and some input patterns may be consumed only partially.) Secondly, by taking advantage of the constraint about asynchronous names, we show a limited form of commutation of reductions (Lemma 13). Thirdly, by commuting consecutive reductions, we adjust a reduction path and establish on it some locally ordered sequences separated by landmarks. Moreover, when an input pattern is not completely consumed, we perform some manipulations on the derivatives of processes and erase some inert subprocesses. Combining all of these with the result of Step 1, we are able to prove the termination property of tagged processes (Lemma 14). Finally, the termination of untagged processes follows from the operational correspondence between tagged and untagged processes (Lemma 11), which concludes our proof of Theorem 10.

We begin with introducing the concepts of atomic tag, tag and tagged process. *Atomic tags* are names from a separate infinite set $\mathcal{N}'$, which is disjoint from the set $\mathcal{N}$ used for constructing untagged processes. We use the function $\rho : \mathcal{N}' \mapsto \mathsf{N}$ to associate every atomic tag with a natural number. Note that we require $\mathcal{N}'$ to be an infinite set so that it can always supply fresh atomic tags as we need. We let $l, l', l_1, \cdots$ range over atomic tags and ϵ stand for a special atomic tag by setting $\rho(\epsilon) = 0$. A *tag* is a pair (l, n) where l is an atomic tag and n is an integer with $n \leq \rho(l)$. We let $t, t', \cdots$ range over tags and write ϵ as the abbreviation of the special tag $(\epsilon, 0)$. The only difference between *tagged processes* and untagged ones is that the former gives tags for all non-replicated inputs.

$$P ::= \cdots \mid a^t(x).P$$

Note that we do not give tags to input patterns. A tagged process P is *regular* if the only tag that appears in P is the special tag ϵ. On the contrary, if there is a tag t with $t \neq \epsilon$ in P, then P is irregular. We reserve the tag ϵ' for the transition rules if-t and if-f (see Table 3). Unlike ϵ, ϵ' only appears in transitions, not in tagged processes. We define the operator $erase(\cdot)$ to erase all tags in a tagged process so as to get an untagged process. Let P be a tagged process. We define $wt(P)$ as $wt(erase(P))$, and we write $\mathcal{T}'' \vdash P$ if $\mathcal{T}'' \vdash erase(P)$. The transition rules for tagged processes are the same as in Table 2 except for rules in, com1, rep, if-t and if-f, which are displayed in Table 3. In the rule rep, a fresh atomic tag l is introduced to witness the invocation of the replicated input $!\kappa.P$. The result of invoking $!\kappa.P$ is the generation of a new process $(a_2^{(l,2)}(x_2).\cdots.a_n^{(l,n)}(x_n).P)\{v/x_1\}$. The condition $\rho(l) = n$ relates l to κ by

$$\textsf{if-t}\ \frac{}{\texttt{if}\ \mathit{true}\ \texttt{then}\ P\ \texttt{else}\ Q \xrightarrow{\epsilon^0} P} \qquad \textsf{if-f}\ \frac{}{\texttt{if}\ \mathit{false}\ \texttt{then}\ P\ \texttt{else}\ Q \xrightarrow{\epsilon^0} Q}$$

$$\textsf{com1}\ \frac{P \xrightarrow{(\nu\widetilde{b})\bar{a}v} P' \quad Q \xrightarrow{a^{\mathrm{t}}v} Q' \quad \widetilde{b} \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{t} (\nu\widetilde{b})(P' \mid Q')} \qquad \textsf{in}\ \frac{}{a^t(x).P \xrightarrow{a^{\mathrm{t}}v} P\{v/x\}}$$

$$\textsf{rep}\ \frac{\kappa = a_1(x_1).\cdots.a_n(x_n) \quad l \text{ fresh} \quad \rho(l) = n}{!\kappa.P \xrightarrow{a_1^{(\mathrm{l};\,1)}v} !\kappa.P \mid (a_2^{(l,2)}(x_2).\cdots.a_n^{(l,n)}(x_n).P)\{v/x_1\}}$$

Table 3. Transition rules for tagged processes

requiring the number of input prefixes in κ to be $\rho(l)$. So if an input prefix has tag $(l, \rho(l))$ then it originates from the last input prefix in κ.

Note that substitutions of names do not affect tags. More precisely, we have

$$(a^t(x).P)\{c/b\} = (a\{c/b\})^t(x).P\{c/b\}.$$

From the transition rules it can be seen that tags are never used as values to be transmitted between processes and that there is no substitution for tags.

Tags give us information about the transitions of tagged processes. For example, if P is regular and $P \xrightarrow{t} P'$, then at least we know the following information:

- if $t = \epsilon'$ then an if-then-else structure in P disappears when P evolves into P';
- if $t = \epsilon$ then the reduction results from an internal communication between an active output and a non-replicated input;
- if $t = (l, 1)$ then the reduction results from an internal communication between an active output and a replicated input of the form $!a_1(x_1).\cdots.a_{\rho(l)}(x_{\rho(l)}).Q$; moreover, if $\rho(l) > 1$ then P' has a subprocess $a_2(x_2).\cdots.a_{\rho(l)}(x_{\rho(l)}).Q$.

We define the operator $(\cdot)^\circ$, which is complementary to $erase(\cdot)$, to translate untagged processes into regular processes by giving all non-replicated inputs the special tag ϵ.

$$\begin{array}{rlrl}
\mathsf{0}^\circ &\stackrel{\text{def}}{=} \mathsf{0} & (a(x).P)^\circ &\stackrel{\text{def}}{=} a^\epsilon(x).P^\circ \\
(\bar{a}v.P)^\circ &\stackrel{\text{def}}{=} \bar{a}v.P^\circ & (\nu aP)^\circ &\stackrel{\text{def}}{=} \nu aP^\circ \\
(P \mid Q)^\circ &\stackrel{\text{def}}{=} P^\circ \mid Q^\circ & (!\kappa.P)^\circ &\stackrel{\text{def}}{=} !\kappa.P^\circ \\
(\ \texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q)^\circ &\stackrel{\text{def}}{=} \ \texttt{if}\ b\ \texttt{then}\ P^\circ\ \texttt{else}\ Q^\circ
\end{array}$$

Note that $erase(P^\circ) = P$ holds but $(erase(P))^\circ = P$ may not be valid. For example $!a.b.\bar{c} \mid \bar{a} \xrightarrow{(l,1)} !a.b.\bar{c} \mid b^{(l,2)}.\bar{c} \equiv P'$, and thus $(erase(P'))^\circ = !a.b.\bar{c} \mid b^\epsilon.\bar{c} \neq P'$. However, there exists operational correspondence between tagged and untagged processes since tags do not have semantic meaning and the purpose of using tags is to identify every newly created process from some replicated process. This is precisely what the next lemma shows.

Lemma 11. *Let P be a tagged process and Q an untagged one.*

1. *If $P \xrightarrow{t} P'$ then $erase(P) \xrightarrow{\tau} erase(P')$.*
2. *If $Q \xrightarrow{\tau} Q'$ and $erase(P) = Q$, then $P \xrightarrow{t} P'$ and $erase(P') = Q'$ for some t.*

Proof. These results follow from the definition of $erase(\cdot)$. □

As we shall see soon, (well-typed) tagged processes have some interesting properties such as decrement of weight after some specific steps of reduction and commutation of reductions.

Lemma 12. *1. If $P \xrightarrow{\epsilon} P'$ then $wt(P) \succ wt(P')$.*
2. If $P \xrightarrow{\epsilon^0} P'$ then $wt(P) \succeq wt(P')$
3. If $P \xrightarrow{(l,1)} P_1 \xrightarrow{(l,2)} \cdots P_{n-1} \xrightarrow{(l,n)} P'$ and $n = \rho(l) > 0$ then $wt(P) \succ wt(P')$.

Proof. See Appendix B. □

Generally speaking, commutativity of reductions does not hold in the π-calculus. For instance, the process $P = a.b \mid \bar{a} \mid \bar{b}$ has reduction path $P \xrightarrow{\tau_a} \xrightarrow{\tau_b}$ but not $\xrightarrow{\tau_b} \xrightarrow{\tau_a}$, where $\xrightarrow{\tau_c}$ means that an internal communication happens on channel c. As we shall see in the next two lemmas, this property does hold in the presence of certain constraints. We write $P \stackrel{\widetilde{t}}{\Longrightarrow} R$ for $P \xrightarrow{t_1} \cdots \xrightarrow{t_n} R$, where $\widetilde{t} = t_1 \cdots t_n$.

Lemma 13. *1. If P is regular and $P \stackrel{\widetilde{t}}{\Longrightarrow} R \xrightarrow{(l,i)} R_1 \xrightarrow{t} R'$, $t \in \{\epsilon, \epsilon'\}$ and $i < \rho(l)$, then there exists R_1' such that $R \xrightarrow{t} R_1' \xrightarrow{(l,i)} R'$.*
2. If P is regular and $P \stackrel{\widetilde{t}}{\Longrightarrow} R \xrightarrow{(l^0,j)} R_1 \xrightarrow{(l,i)} R'$, $l \neq l'$, $j < \rho(l')$ and $i \leq \rho(l)$, then there exists some R_1' such that $R \xrightarrow{(l,i)} R_1' \xrightarrow{(l^0,j)} R'$.

Proof. See Appendix B. □

In the following lemma, we make full use of commutativity and reorganize a reduction path in a way easy of pinpointing landmarks, which witness the decrement of the measure that we choose for the beginning process of the path.

Lemma 14. *All the regular tagged processes terminate.*

Proof. We sketch the idea of the proof; more details are given in Appendix B.

Let P be a regular tagged process. We show that P terminates by induction on its weight $wt(P)$.

• Base case: All processes with weight $\mathsf{0}$ must be terminating because they have no active outputs.

• Inductive step: Suppose P is non-terminating and thus has an infinite reduction sequence

$$P \equiv P_0 \xrightarrow{t_1} P_1 \xrightarrow{t_2} \cdots \xrightarrow{t_i} P_i \xrightarrow{t_{i+1}} \cdots$$

Now the tag t_1 takes one of the three forms: ϵ', ϵ or (l, i). By doing case analysis we can prove that in every case there always exists some Q such that: (i) Q is also non-terminating; (ii) Q is regular; (iii) $wt(P) \succ wt(Q)$. When Q is found, we get a contradiction since by induction hypothesis all processes with weights less than $wt(P)$ are terminating. So the supposition is false and P should be terminating.

In seeking this Q, we carefully manipulate the reduction path of P by commuting reductions (Lemma 13) in order to put all tags belonging to the same input pattern in contiguous positions. Then we can use Lemma 12 to prove (iii). If an input pattern cannot be completed, which means that its continuation does not contribute to the subsequent reductions of P, we can substitute $\mathsf{0}$ for the continuation. For example, suppose $P \stackrel{\text{def}}{=} \nu a_2(\bar{a}_1 \mid !a_1.a_2.R_1) \mid R_2$ and there is a reduction sequence like:

$$P \xrightarrow{(l,1)} P_1 \xrightarrow{t_2} P_2 \xrightarrow{t_3} \cdots$$

with $P_1 \equiv \nu a_2(a_2^{(l,2)}.R_1 \mid !a_1.a_2.R_1) \mid R_2$. Since $a_2^{(l,2)}.R_1$ is never consumed in the reduction sequence, it contributes nothing to the subsequent reductions starting from P_1. So we can safely take Q to be $\nu a_2(\mathsf{0} \mid !a_1.a_2.R_1) \mid R_2$, and the same transition sequence can still be made, with $\mathsf{0}$ in place of the top level $a_2^{(l,2)}.R_1$ in all derivatives.

Consequently, for each new atomic tag l with $\rho(l) > 0$ created by the derivatives of P, either we have found the complete input pattern corresponding to l, or the input pattern cannot be completed but no l appears in the infinite reduction path starting from Q. As a result, no new tag appears in Q, i.e. (ii) is satisfied. □

Theorem 10 follows from the above lemma and the following observation (given by Lemma 11):

Let P and Q be untagged and tagged processes respectively. If $erase(Q) = P$, then P is non-terminating iff Q is non-terminating.

We also show an upper bound to the number of reduction steps for each well-typed process.

Proposition 15. *For a process P well-typed under $\mathcal{T}''$, let n and k be its size and the highest level, respectively. Then P terminates in polynomial time $\mathcal{O}(n^{k+1})$.*

Proof. From the proof Lemma 14 (where Lemma 12 and Lemma 13 are used) we know that: (i) commutation of reductions does not change the length of a reduction sequence; (ii) the measure diminishes from one landmark to the next one; (iii) the distance between two neighboring landmarks is less than n. In addition, by similar arguments as in the proof of Proposition 3 it can be shown that in each locally ordered reduction path there are at most $\frac{n(n^k-1)}{n-1}$ landmarks. Therefore the whole length of each reduction path is bounded by $\frac{n^2(n^k-1)}{n-1}$. □

5.2 Example: the protocol of encoding separate choice

Consider the protocol in Table 4 which is used for encoding separate choice (the summands of the choice are either all inputs or all outputs) by parallel composition [18], [23, Section 5.5.4]. One of the main contributions in [18] is the proof that the protocol does not introduce divergence. Here we prove it using typability.

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

Usually when a protocol employs a mechanism of backtracking, it has a high probability to give rise to divergence. The protocol in this example is an exception. However, to figure out this fact is non-trivial, one needs to do careful reasoning so as to analyze the possible reduction paths in all different cases. With the aid of type system $\mathcal{T}''$, we reduce the task to a routine type-checking problem. We show that the protocol does not add any infinite loop by proving that the typability of $[P_i]$ and $[Q_i]$ implies that of $[\Sigma_i \bar{x}_i d_i.P_i]$ and $[\Sigma_i y_i(z).Q_i]$. Then we conclude by Theorem 10. Here we take the i-th branch of input guarded choice as an example and assume that y_i does not appear in Q_i. Suppose that $[Q_i]$ is typable by $\mathcal{T}''$ and

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

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

Table 4. The protocol of encoding separate choice

the highest level of names in $[Q_i]$ is n with $n > 1$. Let us give type $\sharp^1\texttt{Bool}$ to r, type $\sharp^{n+1}\texttt{Unit}$ to g (here we use a new basic type `Unit`, which is straightforward to handle, otherwise g can also be considered of type $\sharp^{n+1}\texttt{Nat}$) and type $\sharp^2(T_z, \sharp^1\texttt{Bool}, \sharp^1\texttt{Bool})$ to y_i where T_z is the type of the datum z. Take $g.y_i(z,s,a).r(x)$ as the input pattern, noted as κ, and abbreviate its continuation as P. Then $!\kappa.P$ is well typed under $\mathcal{T}''$ because $wt(\kappa) = \langle 1, \cdots, 1, 1\rangle$ and $wt(P) = \langle 1, \cdots, 0, 3\rangle$ (the dots stand for a 0-sequence of length $(n-2)$), thus $wt(\kappa) \succ wt(P)$.

6 Partial orders

The purpose of our final type system is to type processes even if they contain replications whose input and output parts have the same weight. Of course not all such processes can be accepted. For instance, $!a.b.(\bar{a} \mid \bar{b})$ should not be accepted, since it does not terminate when running together with $\bar{a} \mid \bar{b}$[5]. However, we might want to accept

$$!p(a,b).a.(\bar{p}\langle a,b\rangle \mid \bar{b}) \tag{5}$$

where a and b have the same type. Processes like (5) are useful. For instance they often appear in systems composed of several "similar" processes (an example is the chain of cells in Section 6.2). In (5) the input pattern $p(a,b).a$ and the continuation $\bar{p}\langle a,b\rangle \mid \bar{b}$ have the same weight, which makes rule T-rep of $\mathcal{T}''$ inapplicable. In the new system, termination is proved by incorporating partial orders into certain link types. For instance, (5) will be accepted if the partial order extracted from the type of p shows that b is below a (both b and a being names that are received along p).

6.1 The type system

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

[5] The reader might argue that the process $!a.b.(\bar{a} \mid \bar{b})$ itself is terminating, so it is parallel composition that is to blame. Given the importance of the parallel composition operator, we choose to impose the restrictions on the input operator and thus discard that process.

To begin with, we introduce some preliminary notations. Let $\mathcal{A}$ be a set and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ be a partial order on elements of $\mathcal{A}$. The set of names (that can be channel names or natural numbers) appearing in elements of $\mathcal{R}$ is $n(\mathcal{R}) = \{a \mid a\mathcal{R}b \vee b\mathcal{R}a \text{ for some } b\}$. Let $\tilde{x}$ be a tuple of names $x_1, \cdots, x_n$, we write the length n of the tuple as $|\ \tilde{x}\ |$. Without risk of confusion, sometimes we consider $\tilde{x}$ as the set $\{x_1, ..., x_n\}$. In the following, we define some operators for partial orders. They will be used for simplifying the presentation of our typing rules in Table 5.

Definition 16. *Let $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{N}$ and $\mathcal{S} \subseteq \mathtt{Nat} \times \mathtt{Nat}$ be two partial orders and $\tilde{x}$ be a tuple of names in $\mathcal{N}$. We define two operators $/$ and $*$ to transform one partial order into the other.*

1. $\mathcal{R}/\tilde{x} \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } n(\mathcal{R}) \cap \tilde{x} = \emptyset \\ \{(i,j) \mid x_i \mathcal{R} x_j\} & \text{if } n(\mathcal{R}) \subseteq \tilde{x} \\ \text{undefined} & \text{otherwise} \end{cases}$
2. $\mathcal{S} * \tilde{x} \stackrel{\text{def}}{=} \{(x_i, x_j) \mid i\mathcal{S}j\}$ if $max\{n(\mathcal{S})\} \leq |\ \tilde{x}\ |$

As shown by the following lemma, the two operators are complementary to each other to some extent.

Lemma 17. 1. $(\mathcal{R}/\tilde{x}) * \tilde{x} = \mathcal{R}$ *if* $n(\mathcal{R}) \subseteq \tilde{x}$
2. $(\mathcal{S} * \tilde{x})/\tilde{x} = \mathcal{S}$ *if* $max\{n(\mathcal{S})\} \leq |\ \tilde{x}\ |$

Proof. By the definition of $/$ and $*$ directly. □

For example, let $\mathcal{R} = \{(a,b),(a,c)\}$, $\mathcal{S} = \{(1,2),(1,3)\}$, and $\tilde{x}$ be the tuple of four names *abcd*, then we have $\mathcal{R}/\tilde{x} = \mathcal{S}$ and $\mathcal{S} * \tilde{x} = \mathcal{R}$.

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

Let $\mathcal{R}$ be a partial order. We extract the sub-partial order defined on $n(\mathcal{R}) \setminus \tilde{x}$ by $\mathcal{R}\Downarrow_{\tilde{x}} = \{(a,b) \mid a, b \notin \tilde{x} \text{ and } a\mathcal{R}c_1\mathcal{R}\cdots\mathcal{R}c_n b \text{ for some } \tilde{\mathbf{e}} \subseteq \tilde{x} \text{ and } n \geq 0\}$. Given two partial orders $\mathcal{R}_1$, $\mathcal{R}_2$ with $T_{\mathcal{R}_1} = T_{\mathcal{R}_2}$, we let $\mathcal{R}_1 + \mathcal{R}_2$ be $\mathcal{R}_1 \cup \mathcal{R}_2$ if such a union is a partial order. Otherwise, $\mathcal{R}_1 + \mathcal{R}_2$ is undefined.

The operator $os(\cdot)$ of Section 3 is now refined to be $mos_{\mathcal{R}}(\cdot)$, which defines a multiset recording all subject occurrences of names in active outputs and with type $T_{\mathcal{R}}$.

$$
\begin{aligned}
mos_{\mathcal{R}}(\mathsf{0}) &\stackrel{\text{def}}{=} [\,] \\
mos_{\mathcal{R}}(!a(\tilde{x}).P) &\stackrel{\text{def}}{=} [\,] \\
mos_{\mathcal{R}}(a(\tilde{x}).P) &\stackrel{\text{def}}{=} mos_{\mathcal{R}}(P) \\
mos_{\mathcal{R}}(\nu a P) &\stackrel{\text{def}}{=} mos_{\mathcal{R}}(P) \\
mos_{\mathcal{R}}(\bar{a}\tilde{\mathbf{e}}.P) &\stackrel{\text{def}}{=} \begin{cases} [a] \uplus mos_{\mathcal{R}}(P) & \text{if } a : T_{\mathcal{R}} \\ mos_{\mathcal{R}}(P) & \text{otherwise} \end{cases} \\
mos_{\mathcal{R}}(P \mid Q) &\stackrel{\text{def}}{=} mos_{\mathcal{R}}(P) \uplus mos_{\mathcal{R}}(Q) \\
mos_{\mathcal{R}}(\ \mathtt{if}\ b\ \mathtt{then}\ P\ \mathtt{else}\ Q) &\stackrel{\text{def}}{=} mos_{\mathcal{R}}(P) \uplus mos_{\mathcal{R}}(Q)
\end{aligned}
$$

The operator $mos_{\mathcal{R}}(\cdot)$ can be extended to input patterns by defining: $mos_{\mathcal{R}}(\kappa) \stackrel{\text{def}}{=} mos_{\mathcal{R}}(\bar{a}_1\tilde{\mathbf{e}}_1 \mid \cdots \mid \bar{a}_n\tilde{\mathbf{e}}_n)$ if $\kappa = a_1(\tilde{\mathbf{e}}_1).\cdots.a_n(\tilde{\mathbf{e}}_n)$.

Let $\mathcal{R}$ be a partial order and $\mathcal{R}_{mul}$ be the induced multiset ordering on multisets over $n(\mathcal{R})$ as defined in Section 2. The binary relation defined below will act as the second component of our measure, which is a lexicographic ordering with weight of processes as its first component.

Definition 18. *Let $\mathcal{R}$ be a partial order on names, Q be a process, P be either an input pattern or a process. It holds that $P \not\mathcal{R} Q$ if the following three conditions are satisfied, for some multisets on names $\mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{M}$: (i) $mos_{\mathcal{R}}(P) = \mathcal{M} \uplus \mathcal{M}_1$; (ii) $mos_{\mathcal{R}}(Q) = \mathcal{M} \uplus \mathcal{M}_2$; (iii) $\mathcal{M}_1 \ \mathcal{R}_{mul} \ \mathcal{M}_2$.*

Essentially the relation $\not\mathcal{R}$ is an extension of the multiset ordering $\mathcal{R}_{mul}$. One can easily prove that $\not\mathcal{R}$ is also well-founded: if $\mathcal{R}$ is finite, then there exists no infinite sequence like $P_0 \not\mathcal{R} P_1 \not\mathcal{R} P_2 \not\mathcal{R} \cdots$

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

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

where $\mathcal{S} \subseteq \mathtt{Nat} \times \mathtt{Nat}$ is a partial order on the indexes of $\tilde{\mathbf{P}}$ (that is, if $|\ \tilde{T}\ | = m$ then $\mathcal{S}$ is a partial order on the set $\{1, ..., m\}$). The condition in the definition says that if i and j are two indexes related by $\mathcal{S}$, then the i-th and j-th components of $\tilde{\mathbf{P}}$ have the same type.

If $\nu a P$ is a subprocess of Q, we say that the restriction νa is *unguarded* if $\nu a P$ is not underneath any input or output prefix. More precisely, we define a set $ur(P)$ to collect all unguarded restrictions in P.

$$\begin{aligned}
ur(\mathsf{0}) &\stackrel{\text{def}}{=} \emptyset & ur(a(\tilde{\mathbf{e}}).P) &\stackrel{\text{def}}{=} \emptyset \\
ur(!a(\tilde{\mathbf{e}}).P) &\stackrel{\text{def}}{=} \emptyset & ur(\bar{a}\tilde{\mathbf{e}}.P) &\stackrel{\text{def}}{=} \emptyset \\
ur(\nu a P) &\stackrel{\text{def}}{=} \{a\} \cup ur(P) & ur(P \mid Q) &\stackrel{\text{def}}{=} ur(P) \cup ur(Q) \\
ur(\ \mathtt{if}\ b\ \mathtt{then}\ P\ \mathtt{else}\ Q) &\stackrel{\text{def}}{=} ur(P) \cup ur(Q)
\end{aligned}$$

If we pull all unguarded restrictions of Q to the outmost positions, the resulting process $\nu\tilde{\mathbf{e}}Q'$ has the same behavior as Q. In the literature this property is often characterized by a sequence of structural rules describing scope extrusion, see for example [19]. Since we assume that bound names are different from free names, the side conditions of those rules are met automatically. We use this property implicitly and often write Q as $\nu\tilde{\mathbf{e}}Q'$ without unguarded restrictions in Q'.

Besides the two sorts AN and SN introduced in the beginning of Section 5.1, now we need another sort RN. It requires that

$$\text{if } \alpha.P \text{ is a process with } subj(\alpha) \in RN \text{ then } ur(P) = \emptyset.$$

In other words, if a name of sort RN appears in the subject position of a prefix (either input or output), then the continuation process has no unguarded restrictions. This technical condition facilitates the presentation of Definition 19.

Suppose $\kappa = a_1(\tilde{\mathbf{e}}_1).\cdots.a_n(\tilde{\mathbf{e}}_n)$ and each a_i has type $\sharp^{m_i}_{\mathcal{S}_i} \tilde{\mathbf{P}}$. We extract a partial order from κ by defining $\mathcal{R}_\kappa = \mathcal{S}_1 * \tilde{\mathbf{e}}_1 \cup \cdots \cup \mathcal{S}_n * \tilde{\mathbf{e}}_n$. It is well defined because all the bound names are assumed to be different from each other. For example, if $\kappa = a_1(x_{11}, x_{12}, x_{13}).a_2(x_{21}, x_{22}, x_{23})$, $\mathcal{S}_1 = \{(1, 2)\}$ and $\mathcal{S}_2 = \{(2, 1)\}$, then we have $\mathcal{R}_\kappa = \{(x_{11}, x_{12}), (x_{22}, x_{21})\}$.

Definition 19. *Let $\kappa = a_1(\tilde{\mathbf{e}}_1).\cdots.a_n(\tilde{\mathbf{e}}_n)$. The relation $\kappa \succcurlyeq P$ holds if one of the following two cases holds: (i) $wt(\kappa) \succ wt(P)$; (ii) $wt(\kappa) = wt(P)$, $\kappa \not\mathcal{R}_\kappa P$ and $a_n \in RN$.*

The second condition indicates the improvement of $\mathcal{T}'''$ over $\mathcal{T}''$. We allow the input pattern to have the same weight as that of the continuation, as long as there is some partial order to reflect a tendency of decrement.

The typing rules of $\mathcal{T}'''$ are presented in Table 5. Now the judgment $\mathcal{R} \vdash P$ means that P is a well-typed process and the free names in P respect the (possibly empty) partial order $\mathcal{R}$.

$$\textsf{T-in}\ \frac{a:\sharp^n_{\mathcal{S}}\widetilde{T}\quad \widetilde{x}:\widetilde{T}\quad \mathcal{R}\vdash P\quad \mathcal{S}=\mathcal{R}/\widetilde{x}}{\mathcal{R}\Downarrow_{\widetilde{x}}\vdash a(\widetilde{x}).P} \qquad \textsf{T-nil}\ \frac{}{\emptyset\vdash \mathsf{0}}$$

$$\textsf{T-out}\ \frac{a:\sharp^n_{\mathcal{S}}\widetilde{T}\quad \widetilde{v}:\widetilde{T}\quad \mathcal{R}\vdash P}{\mathcal{R}+\mathcal{S}*\widetilde{v}\vdash \bar{a}\widetilde{v}.P} \qquad \textsf{T-par}\ \frac{\mathcal{R}_1\vdash P\quad \mathcal{R}_2\vdash Q}{\mathcal{R}_1+\mathcal{R}_2\vdash P\mid Q}$$

$$\textsf{T-if}\ \frac{b:\texttt{Bool}\quad \mathcal{R}_1\vdash P\quad \mathcal{R}_2\vdash Q}{\mathcal{R}_1+\mathcal{R}_2\vdash\ \texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q} \qquad \textsf{T-res}\ \frac{a:L\quad \mathcal{R}\vdash P}{\mathcal{R}\Downarrow_a\vdash \nu aP}$$

$$\textsf{T-rep}\ \frac{\mathcal{R}\vdash \kappa.P\quad \kappa\ \succcurlyeq P}{\mathcal{R}\vdash !\kappa.P}$$

Table 5. Typing rules of $\mathcal{T}'''$

In the premise of rule T-in, if there exists some non-empty partial order relation on $\mathbf{e}$, then it is exactly captured by $\mathcal{R}$, the partial order built upon free names of P. In rule T-out, for $\mathcal{R}+\mathcal{S}*\mathbf{e}$ to be well defined, the partial order on $\mathbf{e}$ should not conflict with the partial order exhibited by P. Similarly in rules T-par and T-if the partial orders contributed by P and Q should be compatible in the sense that $\mathcal{R}_1+\mathcal{R}_2$ is well defined. Otherwise, $\mathcal{R}_1+\mathcal{R}_2\vdash P\mid Q$ is not a legal judgment. As we only consider the partial order on free names of νaP, in rule T-res all pairs concerning a are deleted from $\mathcal{R}$ while the relative partial order relation on other names are kept intact. In rule T-rep the appearance of the replication operator does not affect the existing partial order, but it requires the validity of the condition $\kappa\ \succcurlyeq P$, which plays an important role in Lemma 21 and gives us the possibility of doing termination proof.

In Definition 19 the constraint imposed on a_n is used to prohibit potential extension of partial orders caused by the restriction operator. Let us consider two examples, concerning two different occurrences of restricted names.

(i) Underneath an input pattern

$$\begin{aligned} P &\stackrel{\text{def}}{=} !p(a,b).a.\nu c(\bar{p}\langle b,c\rangle \mid \bar{b}) \mid \bar{p}\langle a,b\rangle \mid \bar{a} \mid \bar{p}\langle a,b\rangle \\ &\xrightarrow{\tau} !p(a,b).a.\nu c(\bar{p}\langle b,c\rangle \mid \bar{b}) \mid a.\nu c(\bar{p}\langle b,c\rangle \mid \bar{b}) \mid \bar{a} \mid \bar{p}\langle a,b\rangle \\ &\xrightarrow{\tau} !p(a,b).a.\nu c(\bar{p}\langle b,c\rangle \mid \bar{b}) \mid \nu c(\bar{p}\langle b,c\rangle \mid \bar{b}) \mid \bar{p}\langle a,b\rangle \\ &\equiv \nu d(!p(a,b).a.\nu c(\bar{p}\langle b,c\rangle \mid \bar{b}) \mid \bar{p}\langle b,d\rangle \mid \bar{b} \mid \bar{p}\langle a,b\rangle) \\ &\stackrel{\text{def}}{=} \nu dP' \end{aligned}$$

(ii) Outside an input pattern

$$\begin{aligned} Q &\stackrel{\text{def}}{=} !p(a,b).a.(\bar{p}\langle a,b\rangle \mid \bar{b}) \mid \bar{p}\langle a,b\rangle \mid \bar{a}.\nu c\bar{p}\langle b,c\rangle \\ &\xrightarrow{\tau} !p(a,b).a.(\bar{p}\langle a,b\rangle \mid \bar{b}) \mid a.(\bar{p}\langle a,b\rangle \mid \bar{b}) \mid \bar{a}.\nu c\bar{p}\langle b,c\rangle \\ &\xrightarrow{\tau} !p(a,b).a.(\bar{p}\langle a,b\rangle \mid \bar{b}) \mid \bar{p}\langle a,b\rangle \mid \bar{b} \mid \nu c\bar{p}\langle b,c\rangle \\ &\equiv \nu d(!p(a,b).a.(\bar{p}\langle a,b\rangle \mid \bar{b}) \mid \bar{p}\langle a,b\rangle \mid \bar{b} \mid \bar{p}\langle b,d\rangle) \\ &\stackrel{\text{def}}{=} \nu dQ' \end{aligned}$$

Let the type of name p be $\sharp^2_{\{(1,2)\}}(\sharp^1_\emptyset T,\sharp^1_\emptyset T)$. Assume $\mathcal{R}=\{(a,b)\}$ and $\mathcal{R}'=\{(a,b),(b,d)\}$. If the condition $a_n\in RN$ in Definition 19 was lifted, then both P and Q would be well typed: in the first example, it could be derived that $\mathcal{R}\vdash P$ and $\mathcal{R}'\vdash P'$; in the second example, $\mathcal{R}\vdash Q$ and $\mathcal{R}'\vdash Q'$. In both cases the new name d extends the partial order $\mathcal{R}$ to be $\mathcal{R}'$.

However, the process P does not terminate because it can make cyclic reduction and the two steps from P to $\nu dP'$ form a cycle. Therefore the structure in (i) is dangerous and should be disallowed. The process Q always terminates in at most 6 steps, but ruling out the

structure in (ii) simplifies our proof of Lemma 22. (We believe that it is possible to allow the structure in (ii) at the expense of a more complicated proof of Lemma 22.)

For this type system, we have the following subject reduction property.

Theorem 20. (Subject reduction) *Suppose* $\mathcal{R} \vdash P$ *and* $P \xrightarrow{\alpha} P'$.

1. *If* $\alpha = \tau$ *due to a communication then* $\mathcal{R} \vdash P'$.
2. *If* $\alpha = \tau$ *due to a conditional then* $\mathcal{R}' \vdash P'$ *with* $\mathcal{R} = \mathcal{R}' + \mathcal{R}''$ *for some* $\mathcal{R}'$ *and* $\mathcal{R}''$.
3. *If* $\alpha = a\mathbf{e}$ *then there exists* $n, \mathcal{S}$ *and* $\mathbf{F}$ *such that*
   (a) $a : \sharp^n_{\mathcal{S}}\mathbf{F}$ *and* $\mathbf{e} : \mathbf{F}$
   (b) *if* $\mathcal{S} * \mathbf{e}$ *is a partial order then* $\mathcal{R} + \mathcal{S} * \mathbf{e} \vdash P'$.
4. *If* $\alpha = (\nu \mathbf{f})\bar{a}\mathbf{e}$ *then there exists* $n, \mathcal{S}, \mathcal{R}'$ *and* $\mathbf{F}$ *such that*
   (a) $a : \sharp^n_{\mathcal{S}}\mathbf{F}$ *and* $\mathbf{e} : \mathbf{F}$
   (b) $\mathcal{R}' \vdash P'$
   (c) $\mathcal{R} = (\mathcal{R}' + \mathcal{S} * \mathbf{e})\Downarrow_{\widetilde{b}}$

Proof. See Appendix C. Most efforts are made to check the consistency of partial orders in the type environments. □

The following lemma is the counterpart of Lemma 12.

Lemma 21. *Suppose that* $ur(P) = \emptyset$, $\mathcal{R} \vdash P$, $P \xrightarrow{(l,1)} P_1 \xrightarrow{(l,2)} \cdots P_{n-1} \xrightarrow{(l,n)} P'$ *and* $n = \rho(l) > 0$. *Then one of the following two cases holds.*

1. $wt(P) \succ wt(P')$
2. $P \not\mathcal{R}\ P'$ *and* $ur(P') = \emptyset$.

Proof. See Appendix C. □

With the last lemma we are able to prove Lemma 22, whose role in $\mathcal{T}'''$ is the same as that of Lemma 14 in $\mathcal{T}''$.

Lemma 22. *All the regular tagged processes (well-typed under* $\mathcal{T}'''$*) terminate.*

Proof. Compared with the proof of Lemma 14, the main difference is that when we have completed some input patterns and get a reduction sequence like

$$P_0 \stackrel{\widetilde{t_1}}{\Longrightarrow} P_1 \xrightarrow{\epsilon^0} P_2 \stackrel{\widetilde{t_2}}{\Longrightarrow} \cdots \xrightarrow{\epsilon^0} P_{i-1} \stackrel{\widetilde{t_i}}{\Longrightarrow} P_i \cdots$$

it may be possible that $\forall j < i, wt(P_j) = wt(P_{j+1})$. Let $\mathcal{R} \vdash P_0$, we can show by contradiction that the sequence of processes of equal weight is finite, by the well-foundedness of $\mathcal{R}_{mul}$. See Appendix C for more details. □

Finally we have the following termination theorem for $\mathcal{T}'''$, due to the operational correspondence between tagged and untagged process and Lemma 22.

Theorem 23. *If* $\mathcal{R} \vdash P$ *then* P *terminates. Moreover, let* n *and* k *be its size and the highest level, then* P *terminates in time* $\mathcal{O}(n^{k+3})$.

Proof. The proof of termination is straightforward. Let us look at the time complexity. Clearly the sizes of the two sets $n(\mathcal{R})$ and $mos_{\mathcal{R}}(P)$ are less than n. It follows that for any sequence $P \not\mathcal{R}\ P_1 \not\mathcal{R}\ \cdots \not\mathcal{R}\ P_m$, we have $m < n^2$. By similar arguments as in the proof of Proposition 15 it can be shown that in each locally ordered reduction path there are at most $\frac{n(n^k-1)}{n-1}$ landmarks and the distance between two neighboring landmarks is less than n^3. Therefore the whole length of each reduction path is bounded by $\frac{n^4(n^k-1)}{n-1}$. □

6.2 Example: symbol table

This example comes from [11, 21]. It is about the implementation of a symbol table as a chain of cells. In Table 6 G is a generator for cells; ST_0 is the initial state of the symbol table with only one cell; ST_m is the system in which the symbol table has m pending requests.

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

$$
\begin{array}{l}
G \stackrel{\text{def}}{=} !p(a,b,n,s).a(t,r). \\
\qquad\qquad \texttt{if } t = s \texttt{ then} \\
\qquad\qquad\quad \bar{r}\langle n\rangle.\bar{p}\langle a,b,n,s\rangle \\
\qquad\qquad \texttt{else if } b = nil \texttt{ then} \\
\qquad\qquad\qquad\quad \bar{r}\langle n+1\rangle.\nu c(\bar{p}\langle c, nil, n+1, t\rangle \mid \bar{p}\langle a,c,n,s\rangle) \\
\qquad\qquad\qquad \texttt{else } \bar{b}\langle t,r\rangle.\bar{p}\langle a,b,n,s\rangle \\
ST_0 \stackrel{\text{def}}{=} \nu p(G \mid \bar{p}\langle a, nil, 1, s_0\rangle) \\
ST_m \stackrel{\text{def}}{=} ST_0 \mid \bar{a}\langle t_1, r_1\rangle \mid \cdots \mid \bar{a}\langle t_m, r_m\rangle
\end{array}
$$

Table 6. The implementation of a symbol table

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

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

7 Concluding remarks

In this paper we have proposed a core type system and three extensions of it to ensure termination of processes in the π-calculus. The system in Section 5 exploits the structure of processes, so does the system in Section 6, but the latter is more expressive because it is a conservative extension of the former. The system in Section 4 exploits the well-foundedness of

first-order values. Since it is parametric w.r.t. a binary relation on first-order value expressions, we are not able to delimit its exact expressiveness, though we know that it can capture primitive recursion and that it is incomparable with the systems in Sections 5 and 6 (for example, the encoding of the factorial function in (2) is not typable under the systems of Sections 5 and 6, while the examples in Tables 4 and 6 are not typable under the system of Section 4). Based on the type systems we are able to prove the termination property of some non-trivial applications: the encodings of primitive recursive functions, the protocol for encoding separate choice in terms of parallel composition, a symbol table implemented as a dynamic chain of cells. For all (but one of) the type systems we also present upper bounds to the number of steps well-typed processes take to terminate.

We believe that the idea of using levels can be applied to other name-passing calculi. For instance, we have checked that in the Join-calculus [8] the type system presented in Section 5 can be simplified. Intuitively, this is because the Join-calculus can be encoded into a sublanguage of the asynchronous π-calculus with each input channel being unique, thus our assumption about asynchronous names in Section 5 is automatically met and recursive inputs are easier to be handled.

We have already discussed related work on termination, notably [22] and [27]. The systems proposed in this paper are incomparable with those in [22] and [27]. Roughly, in [22] and [27] processes are mainly "functional" and indeed include the standard encodings of the λ-calculus into the π-calculus. These processes are not typable in the systems of this paper. In this work the processes are mainly "imperative". For instance, the examples in sections 5.2 and 6.2 are not typable in [22] and [27]. One way of interpreting the results of this paper is to consider combinatory approach (on which this paper is based) as a complementary technique to logical relations (on which [22] and [27] are based) for showing termination of processes. It would be interesting to see whether the two approaches can be successfully combined.

A typical way of increasing the expressive power of a type system is to use polymorphism. Turner [25] has extended the simply typed π-calculus with impredicative polymorphism and has given a type-preserving encoding of System F [10, 20]. Berger *et al.* [2] have incorporated polymorphism into their linear typing [27], which enables them to embed System F fully abstractly in linear polymorphic processes. In this way they are able to prove the termination property of the terms in System F. However, in spite of its power of guaranteeing termination of processes that are functional, the polymorphic linear system of [2] suffers from the same drawback as the linear system of [27]: it rules out many useful processes that are imperative, such as the examples in sections 5.2 and 6.2.

A (less important) difference between our type systems and that in [27] is about finite processes. In our type systems we impose type-checking restrictions on replicated inputs. Since finite processes are constructed without using replicated inputs, it is easy to see that if a process is accepted in the simply typed π-calculus, then it is also accepted by our type systems. This is not the case for [27], where, for example, the process $a.\bar{b} \mid b.\bar{a} \mid \bar{a}$ is disallowed.

Kobayashi [13] has introduced a type system to guarantee non-interference of processes in the π-calculus. He uses natural numbers in types as obligation levels and capability levels. An obligation level expresses the degree of an obligation to do an action, while a capability level expresses the degree of a capability to successfully complete an action. The level information is used to detect deadlocks, so his type system is incompatible to the systems presented in this paper. For example, the deadlocked process $a.\bar{b} \mid b.\bar{a}$ is ruled out by [13], but accepted by our type systems as a terminating process. On the other hand, the process $!a.\bar{a}$ is dangerous for termination (when put in parallel with $\bar{a}$), thus discarded by our systems, but it is typable in [13].

For simplicity we have given our type systems in the Church version. It is not difficult to transform them into the Curry version. For the Curry version of $\mathcal{T}$ and $\mathcal{T}'$, it is possible to

check automatically whether a program is well-typed by using type inference, following for instance Vasconcelos and Honda's type inference algorithm for polyadic π-calculus [26]. Here one needs an extra constraint, which is a partial order between levels of names. By inspecting the structure of a process, this task can be done in linear time w.r.t. the size of the process. For $\mathcal{T}''$ and $\mathcal{T}'''$, however, type inference is not straightforward. We leave it as future work to investigate efficient type inference algorithms for them.

Acknowledgements

We would like to thank the anonymous referees for their valuable comments on a preliminary version of the paper.

References

[1] F. S. Beckman. *Mathematical Foundations of Programming.* Addison-Wesley Publishing Company, Inc., 1980.

[2] M. Berger, K. Honda, and N. Yoshida. Genericity and the pi-calculus. In *Proceedings of 6th International Conference on Foundations of Software Science and Computational Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 103–119. Springer, 2003.

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

[4] G. Boudol. On strong normalization in the intersection type discipline. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 2003.

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

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

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

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

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

[10] J.-Y. Girard. *Interprétation Fonctionnelle et Éliminations des Coupures de l'Arithmétique d'Ordre Supérieur.* PhD thesis, Université de Paris VII, 1972.

[11] C. Jones. A π-calculus semantics for an object-based design notation. In *Proceedings of 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1993.

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

[13] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

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

[15] R. Milner. The polyadic π-calculus: A tutorial. Technical Report 180, LFCS, University of Edinburgh, 1991.

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

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

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

[19] J. Parrow. An introduction to the pi-calculus. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.
[20] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.
[21] D. Sangiorgi. The typed π-calculus at work: A proof of Jones's parallelisation theorem on concurrent objects. *Theory and Practice of Object-Oriented Systems*, 5(1):25–33, 1999.
[22] D. Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 2006. To appear.
[23] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
[24] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, 1979.
[25] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.
[26] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In *Proceedings of the 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 1993.
[27] N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the pi-calculus. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 311–322. IEEE Computer Society, 2001.

A The λ-calculus and Ackermann's function

This section consists of two parts. In the first part, we discuss why the standard encodings of the simply typed λ-calculus cannot be typed in the type systems presented in this paper. In the second part, we show that if we encode Ackmermann's function in the same way as we did for recursive functions (cf. Proposition 9), the resulting process is not typable by our type systems.

First we recall the standard encoding of untyped call-by-value λ-calculus [16, 23]. Each λ-term M is encoded as π-process $[\![M]\!]p$, where p is the channel along which the process sends out a value (or a link to the value) represented by M.

$$\begin{aligned}
[\![\lambda x.M]\!]p &\stackrel{\text{def}}{=} \bar{p}(y).!y(x,q).[\![M]\!]q \\
[\![x]\!]p &\stackrel{\text{def}}{=} \bar{p}x \\
[\![c]\!]p &\stackrel{\text{def}}{=} \bar{p}c \\
[\![MN]\!]p &\stackrel{\text{def}}{=} (\nu qr)([\![M]\!]q \mid [\![N]\!]r \mid q(y).r(z).\bar{y}\langle z,p\rangle)
\end{aligned}$$

where c is a constant.

Consider the term $M \stackrel{\text{def}}{=} \lambda x.(\lambda y.y)c$. According to the above encoding, we have that

$$\begin{aligned}
[\![M]\!]p &= \bar{p}(p_x).!p_x(x,q).[\![(\lambda y.y)c]\!]q \\
&= \bar{p}(p_x).!p_x(x,q).(\nu q_1 q_2)([\![\lambda y.y]\!]q_1 \mid [\![c]\!]q_2 \mid q_1(p_y).q_2(z).\bar{p_y}\langle z,q\rangle) \\
&= \bar{p}(p_x).!p_x(x,q).(\nu q_1 q_2)(\bar{q_1}(p_y).!p_y(y,q_3).\bar{q_3}y \mid \bar{q_2}c \mid q_1(p_y).q_2(z).\bar{p_y}\langle z,q\rangle).
\end{aligned}$$

In the typed setting, the encoding of terms is similar to that in untyped setting, but we need to encode types and annotate bound names with types (see e.g. [23]). Each type in the λ-calculus is encoded as a unique type in the π-calculus. Let us consider the term M again. If x, y and c have the same type S, then the term M is well typed and has type S in the simply typed λ-calculus. In the process $[\![M]\!]p$, the type of p_x is determined by that of x, similar for

p_y and y. Since the types of x and y should be the same, say $[\![S]\!]$, the two names p_x and p_y must have a common type constructed from $[\![S]\!]$, thus they have the same level. Now the structure

$$!p_x(x,q).\ \ldots\ .\bar{p_y}\langle z,q\rangle$$

in $[\![M]\!]p$ is not typable in any type system reported in this paper. The reason is that the two prefixes $p_x(x,q)$ and $\bar{p_y}\langle z,q\rangle$ have the same weight, but there is no way to establish any kind of "greater than" relation between x and z. Similar problems exist in the encoding of call-by-name λ-calculus as well.

Below we have a look at Ackermann's function, which is defined as follows:

$$f(x,y) = \begin{cases} y+1 & \text{for } x=0 \text{ and } y \geq 0 \\ f(x-1,1) & \text{for } x>0 \text{ and } y=0 \\ f(x-1,f(x,y-1)) & \text{for } x,y>0. \end{cases}$$

If we follow the encodings of primitive functions given in Proposition 9, we get the process A_p.

$$\begin{aligned} A_p \stackrel{\text{def}}{=} !p(x,y,r).\ & \texttt{if } x=0 \text{ and } y \geq 0 \texttt{ then } \bar{r}\langle y+1\rangle \\ & \texttt{else if } x>0 \text{ and } y=0 \texttt{ then } \bar{p}\langle x-1,1,r\rangle \\ & \qquad \texttt{else } (\nu r')(\bar{p}\langle x,y-1,r'\rangle \mid r'(z).\bar{p}\langle x-1,z,r\rangle) \end{aligned}$$

In the above process, the structure

$$!p(x,y,r).\ \ldots\ .\bar{p}\langle x-1,z,r\rangle$$

is not typable by our type systems because we cannot establish any clear relation between y and z. Note that the condition $x-1<x$ is not sufficient to make the assertion

$$\bar{p}\langle x-1,z,r\rangle \lhd p(x,y,r)$$

be true. So far we have not been able to work out an alternative encoding that is typable by our type systems. We leave this problem as future work.

B Proofs from Section 5

Proof of Lemma 12

Proof. 1. There is a communication performed between a non-replicated input and an output message. That is, $P \equiv (\nu \tilde{\mathfrak{b}})(a^{\epsilon}(x).P_1 \mid \bar{a}v.Q_1 \mid Q_2)$ for some a, P_1, Q_1, Q_2, v and $\tilde{\mathfrak{b}}$, and $P' \equiv (\nu \tilde{\mathfrak{b}})(P_1\{v/x\} \mid Q_1 \mid Q_2)$. Therefore we have that

$$\begin{aligned} wt(P) &= wt(P_1) + wt(\bar{a}v) + wt(Q_1) + wt(Q_2) \\ &\succ wt(P_1) + wt(Q_1) + wt(Q_2) = wt(P') \end{aligned}$$

2. If rule if-t is used, then we have that $P \equiv (\nu \tilde{\mathfrak{b}})((\ \texttt{if}\ \mathit{true}\ \texttt{then}\ P_1\ \texttt{else}\ Q_1) \mid Q_2)$ and $P' \equiv (\nu \tilde{\mathfrak{b}})(P_1 \mid Q_2)$ for some $\tilde{\mathfrak{b}}, P_1, Q_1$ and Q_2. Depending on the relation between $wt(P_1)$ and $wt(Q_1)$ we have $wt(P) \succ wt(P')$ if $wt(P_1) \prec wt(Q_1)$ and $wt(P) = wt(P')$ if $wt(P_1) \succeq wt(Q_1)$. The symmetric case for if-f is similar.
3. By the transition rule rep, each time a replicated process is invoked a fresh tag is produced. So there is no replicated process invoked in P_i for $1 \leq i \leq n-1$. Then there are two possibilities:

(a) No replicated process invoked in P either. Therefore all communications take place between non-replicated inputs and outputs. Reasoning as in Clause 1, one can derive that

$$wt(P) \succ wt(P_1) \succ \cdots \succ wt(P')$$

(b) A replicated process $!\kappa.Q$, with $\kappa = a_1(x_1).\cdots.a_n(x_n)$, is invoked in P and a new process $(a_2^{(l,2)}(x_2).\cdots.a_n^{(l,n)}(x_n).Q)\sigma$, for some σ, is spawned. The subsequent reductions consume the input prefixes from $a_2^{(l,2)}\sigma(x_2)$ to $a_n^{(l,n)}\sigma(x_n)$ and their corresponding outputs. Thus we have the relation

$$wt(P') + wt(\kappa) = wt(P) + wt(Q\sigma').$$

Substitution of names does not affect the weight of a process, so $wt(Q\sigma') = wt(Q)$. The side condition of rule rep requires that $wt(\kappa) \succ wt(Q)$. Hence we have the conclusion that $wt(P) \succ wt(P')$.

□

Proof of Lemma 13

Proof. Let $n = \rho(l)$.

1. Since P is regular, the transition with tag (l, i) must originate from a communication between an active output and a replicated input. So R must be of the form:

$$\begin{array}{ll}(\nu \tilde{e})(!a_1(x_1).\cdots.a_n(x_n).P \mid \bar{a_1}v \mid Q) & \text{if } i = 1\\ (\nu \tilde{e})(!a_1(x_1).\cdots.a_n(x_n).P \mid (a_i^{(l,i)}(x_i).\cdots.a_n^{(l,n)}(x_n).P)\sigma \mid \bar{a_i'}v \mid Q) & \text{if } 1 < i < n\end{array}$$

with $a_i\sigma = a_i'$. To have a subsequent transition with tag ϵ, Q must be of the form: $c^{\epsilon}(x).Q_1 \mid \bar{c}w.Q_2 \mid Q_3$ for some c, w, Q_1, Q_2 and Q_3. It is evident that R also have the reduction path $R \xrightarrow{\epsilon} R_1' \xrightarrow{(l,i)} R'$. The case for $t = \epsilon'$ is also straightforward.

2. Let $m = \rho(l')$. As in the proof of Clause 1 we know that the transitions with non-special tags come from replicated inputs. Depending on whether l and l' come from the same input pattern or not, we have the following two cases:
   (a) They are generated by two different input patterns, that is, there exist at least two replicated inputs in P, say $!a_1(x_1).\cdots.a_n(x_n).P_1$ and $!b_1(x_1).\cdots.b_m(x_m).P_2$ respectively. There are four possibilities. Let us consider the typical case that $j \neq 1$ and $i \neq 1$. Then R should be of the form

$$\begin{array}{l}R \equiv (\nu \tilde{e})(!b_1(y_1).\cdots.b_n(y_n).P_2 \mid !a_1(x_1).\cdots.a_n(x_n).P_1\\ \quad \mid (b_j^{(l^0,j)}(y_j).\cdots.b_m^{(l^0,m)}(y_m).P_2)\sigma_1 \mid (a_i^{(l,i)}(x_i).\cdots.a_n^{(l,n)}(x_n).P_1)\sigma_2\\ \quad \mid \bar{b_j'}w \mid Q)\end{array}$$

with $b_j\sigma_1 = b_j'$. Since $j < \rho(l')$ the consumption of $b_j\sigma_1(y_j)$ does not liberate any output, and an output on $a_i\sigma_2$ should be directly available in Q so as to make the subsequent communication on $a_i\sigma_2$ possible, which means that

$$Q \equiv \begin{array}{ll}\bar{a_i'}v \mid Q_2 & \text{if } i < n\\ \bar{a_i'}v.Q_1 \mid Q_2 & \text{if } i = n\end{array}$$

with $a_i\sigma_2 = a_i'$. Obviously in both cases R can take another reduction path: $R \xrightarrow{(l,i)} R_1' \xrightarrow{(l^0,j)} R'$ for some R_1'.

(b) l and l' originate from the same input pattern $!a_1(x_1).\cdots.a_n(x_n).P_1$, which has been invoked two times. The arguments are similar to Case (a).

□

Proof of Lemma 14

Proof. We consider the inductive step. Suppose P has an infinite reduction sequence $P \equiv P_0 \xrightarrow{t_1} P_1 \xrightarrow{t_2} \cdots \xrightarrow{t_i} P_i \xrightarrow{t_{i+1}} \cdots$. We shall do case analysis to find some process Q satisfying the three conditions: (i) Q is also non-terminating; (ii) Q is regular; (iii) $wt(P) \succ wt(Q)$.

It is clear that if $t_j = (l, i)$ and $i < \rho(l)$, then the atomic tag l is generated by invoking an input pattern, since in P there are only special tags.

Case 1: If $t_1 = \epsilon'$, by Lemma 12 there are two possibilities. If $wt(P) \succ wt(P_1)$ we can set $Q = P_1$. If $wt(P) = wt(P_1)$, we need to start the search from t_2. Note that any reduction sequence by consecutively using rules if-t or if-f is finite since the size of the starting process decreases step by step. So we will find either a tag ϵ' that decreases weight or a tag of the form ϵ or (l, i), which directs the analysis to Case 2 or Case 3 accordingly.

Case 2: If $t_1 = \epsilon$, then by Lemma 12 we know that $wt(P) \succ wt(P_1)$. P_1 is just the process Q we are looking for.

Case 3: If $t_1 = (l, i)$ and $\rho(l) > 0$, then $i = 1$ since P is regular. Let $n = \rho(l)$.
−If $n = 1$, then by Lemma 12 it holds that $wt(P) \succ wt(P_1)$. So we can set $Q = P_1$.
− If $n > 1$ and hence a new process $R \stackrel{\text{def}}{=} (a_2^{(l,2)}(x_2).\cdots.a_n^{(l,n)}(x_n).R_0)\sigma$ appears in P_1.

1. If R does not participate in any communication among the infinite sequence $P_1 \xrightarrow{t_2} \cdots \xrightarrow{t_i} P_i \xrightarrow{t_{i+1}} \cdots$, then replacing R with $\mathsf{0}$ does not affect the sequence. More precisely, let $P_1 = (\nu \mathbf{e})(!a_1(x_1).\cdots.a_n(x_n).R_0 \mid R \mid R_1)$, for some R_1, and $Q = (\nu \mathbf{e})(!a_1(x_1).\cdots.a_n(x_n).R_0 \mid \mathsf{0} \mid R_1)$. Q can produce the same infinite reduction sequence as that of P_1, but with $wt(Q) \prec wt(P)$ because P consumes an output during the transition $P \xrightarrow{(l,1)} P_1$.
2. If R participates in a communication among the sequence, then there exists i such that $t_i = (l, 2)$. We need to classify all the reductions between P_1 and P_i. There are two subcases to consider.
   (a) If all t_j for $1 < j < i$ are of the forms ϵ or ϵ', then we use Lemma 13 for $(i-2)$ times and push $(l, 1)$ forward until to the proper left of $(l, 2)$. The resulting sequence is of the form:
   $$P \xrightarrow{t_2} P_2' \xrightarrow{t_3} \cdots \xrightarrow{t_{i-1}} P_{i-1}' \xrightarrow{(l,1)(l,2)} P_i' \longrightarrow \cdots$$
   By Lemma 12, we have the relations
   $$wt(P) \succeq wt(P_2') \succeq \cdots \succeq wt(P_{i-1}')$$
   (b) If there is a partition of the set $\{j \mid 1 < j < i\}$ by I_1 and I_2 such that all $t_j \in C_1 = \{t_i \mid i \in I_1\} = \{t_{11}, \cdots, t_{1k}\}$ are of the forms ϵ or ϵ' and all $t_j \in C_2 = \{t_i \mid i \in I_2\} = \{t_{21}, \cdots, t_{2k^0}\}$ are of the form (l_j, n_j) with $\rho(l_j) > 0$.
      i. If $\forall j \in I_2, n_j < \rho(l_j)$, i.e., no input pattern is complete (since for each j not all tags from $(l_j, 1)$ to $(l_j, \rho(l_j))$ are in the set C_2), then by using Lemma 13 for finite many times we can push all tags in C_1 to the left of $(l, 1)$ and preserve their order. The sequence changes into this form:
      $$P \xrightarrow{t_{11}} P_{11} \xrightarrow{t_{12}} \cdots \xrightarrow{t_{1k}} P_{1k} \xrightarrow{(l,1)\, t_{21}} \cdots \xrightarrow{t_{2k^0}\,(l,2)} \cdots$$

Similarly, by using Lemma 13, we can push all tags in C_2 to the right of $(l,2)$.

$$P \xrightarrow{t_{11}} P_{11} \xrightarrow{t_{12}} \cdots \xrightarrow{t_{1k}} P_{1k} \xrightarrow{(l,1)}\xrightarrow{(l,2)} P'_i \xrightarrow{t_{21}} \cdots \xrightarrow{t_{2k'0}} \cdots$$

By Lemma 12 it follows that

$$wt(P) \succeq wt(P_{11}) \succeq \cdots \succeq wt(P_{1k}).$$

ii. If there is a set $I'_2 \subseteq I_2$ such that $\forall j \in I'_2, t_j = (l_j, \rho(l_j))$, i.e., all tags in I'_2 are the tags of ending inputs in some input patterns. These patterns can be completed by tags between $(1,l)$ and $(l,2)$. We shall use Lemma 13 to sort out all complete patterns and push them to the left of $(l,1)$.

A. Starting from $(l,1)$ we scan the sequence forward to find the first tag $(l_1, \rho(l_1))$ for some atomic tag l_1 because we want to make all tags with atomic tag l_1 be in consecutive positions by "squeezing out" other tags to the left of $(l_1,1)$ or to the right of $(l_1, \rho(l_1))$. All tags between $(l_1,1)$ and $(l_1,\rho(l_1))$ are of one of the three forms: ϵ, ϵ' or (l_j, n_j) with $n_j < \rho(l_j)$. As we did in Case i, it is feasible to push all ϵ and ϵ' backward and all (l_j,n_j) forward so that only tags with atomic tag l_1 are left between $(l_1,1)$ and $(l_1,\rho(l_1))$ (these tags are already in ascending order since they come from the same input pattern, say $a_1(x_1).\cdots.a_{\rho(l_1)}(x_{\rho(l_1)})$, and the consumption of these input prefixes goes from left to right). After the operations, we get a reduction sequence like

$$P \xrightarrow{(l,1)} \cdots \xrightarrow{\epsilon}\xrightarrow{\epsilon^0} \cdots \underbrace{\xrightarrow{(l_1,1)}\xrightarrow{(l_1,2)} \cdots \xrightarrow{(l_1,\rho(l_1))}}_{\tau^{l_1}} \cdots \xrightarrow{(l_j,n_j)} \cdots \xrightarrow{(l,2)} \cdots$$

B. Find the next tag $(l_2, \rho(l_2))$ for some atomic tag l_2 and make all tags with atomic tag l_2 in consecutive positions. Now we can treat tags in group τ^{l_1} as a whole and push them backward just as what we do for tag ϵ. We repeat this operation for other group τ^{l_j} as long as $(l_j, \rho(l_j))$ lies between $(l,1)$ and $(l,2)$. At the end of this stage, we have a sequence as follows.

$$P \xrightarrow{(l,1)} \cdots \xRightarrow{\tau^{l_1}} \cdots \xRightarrow{\tau^{l_2}} \cdots \xRightarrow{\tau^{l_j}} \cdots \xrightarrow{(l,2)} \cdots$$

where $\xRightarrow{\tau^{l_j}}$ stands for $\xrightarrow{(l_j,1)}\xrightarrow{(l_j,2)} \cdots \xrightarrow{(l_j,\rho(l_j))}$.

C. For other tags t_j with $j \notin I'_2$ and $j \in I_2$, which do not belong to a complete group, we push them forward to the right of $(l,2)$, keeping their order. At this moment, there are still tags like ϵ and ϵ' between $(l,1)$ and $(l,2)$.

$$P \xrightarrow{(l,1)} \cdots \xrightarrow{t} \cdots \xRightarrow{\tau^{l_1}} \cdots \xrightarrow{t} \cdots \xRightarrow{\tau^{l_j}} \cdots \xrightarrow{(l,2)} \cdots$$

where $t \in \{\epsilon, \epsilon'\}$.

D. Push $(l,1)$ forward until to the proper left of $(l,2)$ so as to yield this sequence:

$$P \xrightarrow{t} P'_{11} \cdots \xrightarrow{t}\xRightarrow{\tau^{l_1}} \cdots \xrightarrow{t}\xRightarrow{\tau^{l_j}} \cdots \xrightarrow{t} P'_{j^0k_j\,0} \xrightarrow{(l,1)}\xrightarrow{(l,2)} P'_i \cdots$$

where $t \in \{\epsilon, \epsilon'\}$. By Lemma 12 it follows that

$$wt(P) \succeq wt(P'_{11}) \succeq \cdots \succeq wt(P'_{j^0k_j\,0})$$

In the above four steps, when we commute reductions like $\xrightarrow{(l_j,n_j)}\xrightarrow{t_i}$, the condition $n_j < \rho(l_j)$ is always satisfied. This ensures the correct use of Lemma 13.

If $n = 2$, by Lemma 12 and the transitivity of $\succ$, we have that $wt(P) \succ wt(P'_i)$ and so Q can be set as P'_i. If $n > 2$ we repeat the operations done for $(l, 1)$ on (l, i) with $1 < i < \rho(l)$. There are two possibilities for the ultimate result:
1) either $(l, i + 1)$ does not appear in the subsequent reductions, then we replace $R \stackrel{\text{def}}{=} (a_{i+1}^{(l,i+1)}(x_{i+1}).\cdots.a_n^{(l,n)}(x_n).R_0)\sigma$ with $\mathsf{0}$ and get a non-terminating process Q such that $wt(P) \succ wt(Q)$;
2) or we complete the input pattern with atomic tag l and have a sequence like

$$P \xrightarrow{t_i} \cdots \xrightarrow{(l,1)}\xrightarrow{(l,2)} \cdots \xrightarrow{(l,n)} Q \xrightarrow{t_j} \cdots$$

In this case we also have $wt(P) \succ wt(Q)$ according to previous operations and Lemma 12.

Note that there are possibly three kinds of tags lying in the ultimate sequence between P and Q:
1) tags ϵ or ϵ';
2) tags belonging to complete input patterns;
3) tags not belonging to complete input patterns, but the continuations of these incomplete input patterns are discarded in Q since we have substituted $\mathsf{0}$ for them.

Therefore each new atomic tag l with $\rho(l) > 0$ created by the derivatives of P is used up when reaching Q. As P is regular, Q must be regular as well. Hence the induction hypothesis applies and it maintains that Q is terminating. At this point contradiction arises. □

C Proofs from Section 6

Lemma 24. *If $n(\mathcal{R}) \cap \mathfrak{e} = \emptyset$ then $(\mathcal{R} + \mathcal{R}')\Downarrow_{\widetilde{x}} = \mathcal{R} + \mathcal{R}'\Downarrow_{\widetilde{x}}$.*

Proof. Let $\mathcal{R}'' = \mathcal{R} + \mathcal{R}'$.

$$\begin{aligned}
&(\mathcal{R} + \mathcal{R}')\Downarrow_{\widetilde{x}} \\
&= \{(a, b) \mid a, b \notin \mathfrak{e} \text{ and } a\mathcal{R}''c_1\mathcal{R}''\cdots\mathcal{R}''c_n\mathcal{R}''b \text{ for some } \mathbf{e} \subseteq \mathfrak{e} \text{ and } n \geq 0\} \\
&= \{(a, b) \mid a, b \notin \mathfrak{e} \text{ and } a\mathcal{R}b\} \\
&\quad \cup\{(a, b) \mid a, b \notin \mathfrak{e} \text{ and } a\mathcal{R}'c_1\mathcal{R}'\cdots\mathcal{R}'c_n\mathcal{R}'b \text{ for some } \mathbf{e} \subseteq \mathfrak{e} \text{ and } n \geq 0\} \\
&= \mathcal{R} \cup \mathcal{R}'\Downarrow_{\widetilde{x}} \\
&= \mathcal{R} + \mathcal{R}'\Downarrow_{\widetilde{x}}
\end{aligned}$$

□

Let $\mathcal{R}$ be a partial order and σ be a substitution of names. We say $\mathcal{R}\sigma$ is well defined if $\mathcal{R}\sigma = \{(x\sigma, y\sigma) \mid (x, y) \in \mathcal{R}\}$ is a partial order. For the multiset $\mathcal{M} = [x_1, \cdots, x_n]$ we write $\mathcal{M}\sigma = [x_1\sigma, \cdots, x_n\sigma]$.

Lemma 25. *If $\mathcal{M}_1\ \mathcal{R}_{mul}\ \mathcal{M}_2$ then*
(1) $\mathcal{M}_1\ \mathcal{R}'_{mul}\ \mathcal{M}_2$ with $\mathcal{R}' = \mathcal{R} + \mathcal{S}$.
(2) $(\mathcal{M}_1 \uplus \mathcal{M})\ \mathcal{R}_{mul}\ (\mathcal{M}_2 \uplus \mathcal{M})$ for any multiset $\mathcal{M}$ over $n(\mathcal{R})$.
(3) $\mathcal{M}_1\sigma\ \mathcal{R}\sigma_{mul}\ \mathcal{M}_2\sigma$ with $\mathcal{R}\sigma$ well defined.

Proof. We only need the definition of multiset ordering. (1) Since $\mathcal{R}'$ is a superset of $\mathcal{R}$, it holds that $x\mathcal{R}y$ implies $x\mathcal{R}'y$. (2) Trivial. (3) Since $\mathcal{R}\sigma$ is well defined, it follows that $x\mathcal{R}y$ implies $x\sigma\ \mathcal{R}\sigma\ y\sigma$. □

Given a multiset $\mathcal{M}$, we can extract from it a sub-multiset in the following way:

$$\mathcal{M}_{\mathcal{R}}(x) = \begin{array}{ll} \mathcal{M}(x) & x \in n(\mathcal{R}) \\ 0 & x \notin n(\mathcal{R}) \end{array}$$