Replication vs. Recursive Definitions in Channel Based Calculi

Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, I-40127 Bologna, Italy.
busi,gabbri,zavattar@cs.unibo.it

Abstract. We investigate the expressive power of two alternative approaches used to express infinite behaviours in process calculi, namely, replication and recursive definitions. These two approaches are equivalent in the full π-calculus, while there is a common agreement that this is not the case when name mobility is not allowed (as in the case of CCS), even if no formal discriminating results have been proved so far.
We consider a hierarchy of calculi, previously proposed by Sangiorgi, that spans from a fragment of CCS (named "the core of CCS") to the π-calculus with internal mobility. We prove the following discrimination result between replication and recursive definitions: the termination of processes is an undecidable property in the core of CCS, provided that recursive process definitions are allowed, while termination turns out to be decidable when only replication is permitted. On the other hand, this discrimination result does not hold any longer when we move to the next calculus in the hierarchy, which supports a very limited form of name mobility.

1 Introduction

The π-calculus has been designed more than ten years ago for specifying mobile systems and reasoning formally about their behaviour. Rather than an established, definitive theory it can be considered as a "workshop to express ideas about mobility and interaction" [Mil01], indeed many variants, sub-calculi and extensions appeared since its original definition. Given such a rich variety it is important to formally compare the existing π-calculus "dialects" in order to understand precisely their relative expressive power. Several notions of expressive power are meaningful in this context: The classical notion based on the ability to express recursive functions can be further refined by considering, for example, compositionality properties for the encoding of a language into another or the ability to express some patterns of behaviours (typically connected to mobility).

An important aspect which, in general, may significantly affect the expressive power of a language is the mechanism adopted for extending finite processes in order to express infinite behaviours. In the π-calculus theory both replication and recursive process definitions are used: The replication operator $!P$ allows to create an unbounded number of parallel copies of a process P, thus providing an "in width" infinite behaviour, since the copies are placed at the same level. On the other hand, by using recursively defined process constants one can obtain an "in depth" infinite behaviour, since process copies in this case can be nested at an arbitrary depth by using constant application. It is well known that these two mechanisms are equivalent (for any reasonable notion of expressive power) for full π-calculus [MPW92], as the ability to communicate free names together with replication and restriction allows to simulate process constants (replication can easily be simulated by recursive definitions, provided one admits enough constants). On the other hand, there is a common agreement on the fact that recursion cannot be replaced by replication when name mobility is not allowed (as in the case of CCS), even though this result has not been formally proved so far.

In this paper we compare replication and explicit recursion in the context of the π-calculus with internal mobility. This is a sub-calculus obtained essentially by disallowing the free output construct, thus only the output of private names is allowed. As argued in [San96] and formally proved in [Bor98], internal mobility allows one to retain the expressiveness of the π-calculus without some of the semantic complications arising for the full language. In particular, the λ-calculus and several agent-passing calculi can be encoded in the π-calculus with internal mobility and recursive agent definitions [San96], denoted by $\pi\mathrm{I}_D$ in this paper. On the other hand, [San96] shows that the π-calculus with internal mobility and repetition (fragment we refer to with $\pi\mathrm{I}_!$) is strictly less expressive than $\pi\mathrm{I}_D$ in the sense that, by using the type system inherited from the π-calculus, recursive types are not needed in $\pi\mathrm{I}_!$. This has the following relevant consequence in terms of mobility: in $\pi\mathrm{I}_D$ one can create a dependency chain of unbounded length among names, where a name depends on another if the latter carries the former (e.g. in x(y) name y depends on x). This is not possible in $\pi\mathrm{I}_!$, where for each process P there exists a finite limit n to the length of dependency chains that can be created in P. Furthermore, [San96] argues that under certain conditions the λ-calculus cannot be encoded in $\pi\mathrm{I}_!$ and leaves open the question if λ-calculus can be encoded in $\pi\mathrm{I}_!$ at all.

We answer positively to this question by providing a deterministic encoding in $\pi\mathrm{I}_!$ of the Random Access Machines (RAMs), which provide another Turing powerful computing formalism. Actually, we show that the fragment $\pi\mathrm{I}_!^2$ is sufficient to this purpose, where following the mobility hierarchy defined in [San96] the calculus $\pi\mathrm{I}_!^n$ with $n > 0$ includes only those processes which can be typed with types of order n or less than n. In terms of mobility, processes in $\pi\mathrm{I}_!^n$ are those which have dependency chains among names of length at most n. In particular, $\pi\mathrm{I}_!^1$ does not allow mobility at all and is the core of CCS.

We also prove that $\pi\mathrm{I}_!^1$ is strictly less expressive than $\pi\mathrm{I}_!^2$, since we prove that termination is decidable for a process in $\pi\mathrm{I}_!^1$. This provides a formal proof for the folk theorem mentioned before, as RAMs can be deterministically encoded in $\pi\mathrm{I}_!^1$ when recursive process definitions are allowed.

The remaining of this paper is organized as follows. Section 2 reports the definitions of the syntax and the operational semantics of the process calculi.

In Section 3 and 4 we present the modeling of RAMs and the proof of the decidability result, respectively. Finally, in Section 5 we discuss related work and we report some conclusive remark. Due to space limits, we do not include the proofs of our Theorems: they can be found in [BGZ03].

2 The Calculi

In this section we recall the (variants of the) π–calculus with internal mobility [San96] that we consider in this paper. The main difference between the full π–calculus and its fragment with internal mobility is that output prefixes can only send fresh names. More formally, we have that only outputs of the form $(\nu\tilde{y})\overline{x}\tilde{y}.P$ can be used, where $(\nu\tilde{y})$ is a binder for the names in the sequence $\tilde{y}$ and $\overline{x}\tilde{y}$ is the output prefix. For the sake of simplicity, in the π-calculus with internal mobility, $\overline{x}(\tilde{y}).P$ is written instead of $(\nu\tilde{y})\overline{x}\tilde{y}.P$. This notation emphasizes the symmetry between the prefixes for output and input (the latter, as usual, is denoted with $x(\tilde{y})$): both the prefixes, under internal mobility, are binders. In [Bor98] it is proved that the restriction to internal mobility does not reduce the expressive power of the full π-calculus.

We start by introducing a finite fragment of the π–calculus with internal mobility, and then we define two infinite extensions; the first one obtained by adding constants with (possibly) recursive definitions, and the second one obtained by introducing replication.

Definition 1. (finite πI) *Let $Name$, ranged over by x, y, ..., be a denumerable set of names. We use $\tilde{x}$, $\tilde{y}$, ..., to denote (possibly empty) sequences of names. The class of finite πI processes is described by the following grammar:*

$$P ::= \sum_{i\in I}\alpha_i.P_i \quad | \quad P|P \quad | \quad (\nu x)P \qquad\qquad \alpha ::= \tau \quad | \quad x(\tilde{y}) \quad | \quad \overline{x}(\tilde{y})$$

We assume that all names in $\tilde{y}$ are pairwise different. When $\tilde{y}$ is empty, we omit the surrounding parentheses.

The guarded sum construct $\sum_{i\in I}\alpha_i.P_i$ is used to make choice among the summands $\alpha_i.P_i$: we assume that I is a finite indexing set and, if I is empty, we abbreviate the sum as $\mathbf{0}$. We shall write $\alpha_1.P_1+\ldots+\alpha_n.P_n$ for $\sum_{i\in\{1\ldots n\}}\alpha_i.P_i$. Parallel composition is used to run parallel programs. Restriction $(\nu x)P$ makes the name x local in P. We denote the process $\alpha.\mathbf{0}$ simply with α.

The possible prefixes α are the silent action τ, the input action $x(\tilde{y})$ and the output action $\overline{x}(\tilde{y})$. For input and output actions, we write $\overline{\alpha}$ for the complementary of α; that is, if $\alpha = x(\tilde{y})$ then $\overline{\alpha} = \overline{x}(\tilde{y})$, if $\alpha = \overline{x}(\tilde{y})$ then $\overline{\alpha} = x(\tilde{y})$. We write $P \equiv_A Q$ if P and Q are alpha convertible. We write $fn(P)$, $bn(Q)$ (resp. $fn(\alpha)$, $bn(\alpha)$) for the *free names* and the *bounded names* of P (resp. α). The *names* of P, written $n(P)$ and $n(\alpha)$, are the union of their free and bounded names. We use $cn(\alpha)$ to denote the names carried by an action α, i.e., $cn(x(\tilde{y})) = cn(\overline{x}(\tilde{y})) = \tilde{y}$.

Table 1 contains the set of the transition rules for πI.

ALPHA :	$\dfrac{P \equiv_A P' \quad P' \xrightarrow{\alpha} P''}{P \xrightarrow{\alpha} P''}$	PRE :	$\alpha.P \xrightarrow{\alpha} P$
RES :	$\dfrac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}\ x \notin n(\alpha)$	SUM :	$\dfrac{P_i \xrightarrow{\alpha} P_i',\ i \in I}{\sum_{i\in I} P_i \xrightarrow{\alpha} P_i'}$
PAR :	$\dfrac{P \xrightarrow{\alpha} P'}{P\vert Q \xrightarrow{\alpha} P'\vert Q}\ bn(\alpha) \cap fn(Q) = \emptyset$		
COM :	$\dfrac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P\vert Q \xrightarrow{\tau} (\nu x_1)\ldots(\nu x_n)(P'\vert Q')}\ \alpha \neq \tau,\ x_1 \ldots x_n = cn(\alpha)$		

Table 1. The transition system for finite πI (symmetric rules omitted).

Definition 2. **($\pi\mathbf{I}_D$)** *We assume a set of constants, ranged over by D. The class of πI_D processes is defined by adding the production $P ::= D\langle\tilde{x}\rangle$ to the grammar of Definition 1. It is assumed that each constant D has a unique defining equation of the form $D \stackrel{def}{=} (\tilde{x})P$, where $(\tilde{x})$ is a binder for the names $\tilde{x}$. Both in a constant definition $D \stackrel{def}{=} (\tilde{x})P$ and in a constant application $D\langle\tilde{x}\rangle$, the parameter $\tilde{x}$ is a tuple of all distinct names whose length equals the arity of D. As usual, in the case the sequence $\tilde{x}$ is empty, we omit the surrounding parentheses. Moreover, we assume that $fn(P) \subseteq n(\tilde{x})$ where $n(\tilde{x})$ denotes the set of names in the sequence $\tilde{x}$.*

Definition 3. **($\pi\mathbf{I}_!$)** *The class of $\pi I_!$ processes is defined by adding the production $P ::= !P$ to the grammar of Definition 1.*

The transition rules for constant and replication are

$$\frac{P \xrightarrow{\alpha} P'}{D\langle\tilde{x}\rangle \xrightarrow{\alpha} P'} \ \text{ if } D \stackrel{def}{=} (\tilde{y})Q \text{ and } (\tilde{y})Q \equiv_A (\tilde{x})P \qquad\qquad \frac{P \vert\ !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

In [San96], a hierarchy of fragments of $\pi\mathrm{I}_!$ is introduced denoted with $\{\pi\mathrm{I}_!^n\}_{n>0}$. These calculi differ in the form of mobility they support. In particular, the maximal length of *dependency chains* among names is considered: a name depends on another one if the latter carries the former (e.g., in $x(y)$ name y depends on x). By $\pi\mathrm{I}_!^n$ we denote the fragment of $\pi\mathrm{I}_!$ which permits dependency chains with a maximal length less or equal to n.

In order to define the calculi $\{\pi\mathrm{I}_!^n\}_{n>0}$, we introduce an auxiliary type system.

Definition 4. **(typing system for $\pi\mathbf{I}_!$)** *Consider the types S having the form $S ::= (\tilde{S})$ where by $\tilde{S}$ we denote a (possibly empty) sequence of types. We do not consider recursive types because they are not necessary for typing $\pi I_!$ (as*

$$\frac{\Gamma[x] = (\tilde{S}) \quad \Gamma, \tilde{y} : \tilde{S} \vdash P}{\Gamma \vdash x(\tilde{y}).P, \Gamma \vdash \overline{x}(\tilde{y}).P} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash \tau.P}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P|Q} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}$$

$$\frac{\Gamma \vdash P_i, i \in I}{\Gamma \vdash \sum_{i \in I} P_i} \qquad \frac{\Gamma, x : S \vdash P, \text{for some } S}{\Gamma \vdash (\nu x)P}$$

Table 2. The typing rules for the operators of πI.

proved in Lemma 6.9 of [San96]). A typing *is a finite set of assignments of types to names:* $\Gamma \ ::= \ \emptyset \ | \ \Gamma, x : \tilde{S}$. *Names in a typing* Γ *are always taken to be pairwise distinct; this justifies an abuse of notation whereby* Γ *is regarded as a finite function from names to types:* $\Gamma[x]$ *is the type assigned to* x. *A process* P *in* $\pi I_!$ *is* well-typed *for* Γ *if* $\Gamma \vdash P$ *can be inferred from the rules of Table 2. Observe that* $\mathbf{0}$, *corresponding to* $\sum_{i \in \emptyset} P_i$, *is well typed for any* Γ.

Definition 5. (calculi $\{\pi\mathbf{I}_!^n\}_{n>0}$) *The* order *of a type* S *is the maximal level of bracket nesting in the definition of* S. *For example, type* $()$ *has order 1 and type* $((), (()))$ *has order 3. A process* $P \in \pi I_!$ *is in* $\pi I_!^n$, $n > 0$, *if, for some typing* Γ, *there is a derivation proof of* $\Gamma \vdash P$ *in which all types used (including those in* Γ*) have order* n *or less than* n.

Observe that $\pi\mathrm{I}_!^1$ permits only process synchronization and does not support the possibility to communicate names; it corresponds to the fragment of CCS [Mil89] without relabeling, with guarded choice instead of free choice, and with replication instead of recursive definition. For this reason, this calculus is also called “the core of CCS”.

Definition 6. ($\pi\mathbf{I}_D^1$) *A process* $P \in \pi I_D$ *is in* πI_D^1 *if* $\tilde{y}$ *is empty for any* $x(\tilde{y})$ *and* $\overline{x}(\tilde{y})$ *in* P.

Observe that $\pi\mathrm{I}_D^1$ corresponds to $\pi\mathrm{I}_!^1$, where constants are considered instead of replication.

Given a process Q, its internal runs $Q \longrightarrow Q_1 \longrightarrow \ldots$ are given by those transitions $\longrightarrow$ that the process can perform in isolation. As usual, internal transitions $\longrightarrow$ correspond to the transitions labeled with τ, i.e. $P \longrightarrow P'$ iff $P \xrightarrow{\tau} P'$. We say that P *terminates* if all its internal runs terminate, i.e. the process P cannot give rise to an infinite computation: formally,

P *terminates* iff there exist no $\{P_i\}_{i \in \mathbb{N}}$, s.t. $P_0 = P$ and $P_j \longrightarrow P_{j+1}$ for any j

Observe that *process termination* does not corresponds to *process convergence*: a process converges when it has at least one finite (complete) internal run; while it terminates if all its internal runs are finite.

3 Modeling RAMs in $\pi\mathrm{I}^1_D$ and $\pi\mathrm{I}^2_!$

In this section we prove that the calculi $\pi\mathrm{I}^1_D$ and $\pi\mathrm{I}^2_!$ are expressive enough for modeling Turing powerful formalisms. This is proved by showing how to model Random Access Machines(RAMs), a well known register based Turing powerful formalism.

As a consequence of the fact that the calculi $\pi\mathrm{I}^1_D$ and $\pi\mathrm{I}^2_!$ are expressive enough to model RAMs in a deterministic manner, we have that termination is undecidable in these calculi. Formally, P *terminates* is an undecidable property for both $\pi\mathrm{I}^1_D$ and $\pi\mathrm{I}^2_!$. On the other hand, in the following section, we will prove that this is not the case for the calculus $\pi\mathrm{I}^1_!$, for which the termination turns out to be decidable.

A RAM (denoted in the following with R) is a computational model composed of a finite set of registers $r_1, \ldots, r_n$, that can hold arbitrary large natural numbers, and by a program composed by indexed instructions $(1 : I_1), \ldots, (m : I_m)$, that is a sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps. An internal state of a RAM is given by $(i, c_1, \ldots, c_n)$ where i is the program counter indicating the next instruction to be executed, and $c_1, \ldots, c_n$ are the current contents of the registers $r_1, \ldots, r_n$, respectively.

Without loss of generality, we assume that the registers contain the value 0 at the beginning of the computation and that the execution of the program begins with the first instruction $(1 : I_1)$. In other words, the initial configuration is $(1, 0, \ldots, 0)$. The computation continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached. More formally, we indicate by $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ the fact that the configuration of the RAM R changes from $(i, c_1, \ldots, c_n)$ to $(i', c'_1, \ldots, c'_n)$ after the execution of the i-th instruction. If a configuration with program counter i different from any instruction index is reached, then the computation terminates.

The following two instructions are sufficient to model every recursive function:

- $(i : Succ(r_j))$: adds 1 to the contents of register r_j;
- $(i : DecJump(r_j, s))$: if the contents of register r_j is not zero, then decreases it by 1 and go to the next instruction, otherwise jumps to instruction s.

3.1 Modeling RAMs in $\pi\mathrm{I}^1_D$

In this subsection, we will exploit only a limited form of constant applications. More precisely, every time a term $D\langle\tilde{x}\rangle$ is used, we assume that in the corresponding defining equation $D \stackrel{def}{=} (\tilde{y})P$ we have that $\tilde{x} = \tilde{y}$. In other words, the *actual* names used in constant applications always correspond to the *formal* names used in constant definitions. This permits us to use a simplified notation, by omitting $\langle\tilde{x}\rangle$ (resp. $(\tilde{y})$) in each constant application $D\langle\tilde{x}\rangle$ (resp. constant definition $D \stackrel{def}{=} (\tilde{y})P)$. This simplified notation does not introduce ambiguity because $\tilde{x}$ and $\tilde{y}$ exactly correspond to the free names appearing in P.

Let R be a RAM with registers $r_1, \ldots, r_n$, and instructions $(1 : I_1), \ldots, (m : I_m)$; we model R as described in the following. For each $1 \leq i \leq m$, we model the i-th instruction $(i : I_i)$ of R with a program constant $Inst_i$ defined as follows.

$$
\begin{array}{lll}
Inst_i \stackrel{def}{=} \overline{inc_j}.Inst_{i+1} & & \text{if } I_i = Succ(r_j) \\
Inst_i \stackrel{def}{=} \overline{dec_j}.ack.Inst_{i+1} \; + \; \overline{zero_j}.Inst_s & & \text{if } I_i = DecJump(r_j, s)
\end{array}
$$

In the first case, the process $Inst_i$ simply increments the register r_j (by firing the $\overline{inc_j}$ prefix) and activates the subsequent instruction; in the second case the process $Inst_i$ tries either to decrement or to test whether the register r_j is empty. According to the prefix which is fired ($\overline{dec_j}$ or $\overline{zero_j}$, respectively) the corresponding subsequent instruction is activated ($Inst_{i+1}$ or $Inst_s$, respectively). In the case of decrement, the process waits for an acknowledgement ack before activating the next instruction; this is necessary in order to activate the next instruction only after the actual update of the register.

As far as the modeling of the registers is concerned, we represent each register r_j, which is initially empty, with a constant Z_j. The constant Z_j is defined in terms of two other constants O_j and E_j:

$$
\begin{array}{l}
Z_j \stackrel{def}{=} zero_j.Z_j \; + \; inc_j.(\nu x)(O_j | \; x.\overline{ack}.Z_j) \\
O_j \stackrel{def}{=} dec_j.\overline{x} \; + \; inc_j.(\nu y)(E_j \mid y.\overline{ack}.O_j) \\
E_j \stackrel{def}{=} dec_j.\overline{y} \; + \; inc_j.(\nu x)(O_j \mid x.\overline{ack}.E_j)
\end{array}
$$

The idea behind this modeling of the registers is to exploit a chain of nested restrictions with a length corresponding to the content of the register. More precisely, the term Z_j represents the register when empty, while O_j and E_j model the register when it has an odd or an even content, respectively. Each time the register is incremented, the length of the chain of restrictions augments due to the creation of a new name. Observe that in order to avoid name collisions, the two names x and y are alternatively exploited. The use of two different names requires also the exploitation of the two different constants O_j and E_j.

Definition 7. *Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given the initial configuration $(1, 0, \ldots, 0)$ of R we define*

$$[\![(1, 0, \ldots, 0)]\!]_D \; = \; Inst_1 | Z_1 | \ldots | Z_n$$

with $Inst_i$ and Z_j defined as above.

As we have already discussed, the computation of the encoding of RAMs proceeds deterministically, and corresponds exactly to the computation of the corresponding RAM; thus the encoding terminates if and only if the considered RAM terminates, as stated by the following theorem.

Theorem 1. *Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given the initial configuration $(1, 0, \ldots, 0)$ of R we have that R terminates if and only if the process $[\![(1, 0, \ldots, 0)]\!]_D$ terminates.*

3.2 Modeling RAMs in $\pi\mathrm{I}^2_!$

The modeling of RAMs we have presented in the previous subsection exploits recursive definitions in two ways: (i) in the definition of the instructions (where an instruction $Inst_i$ may be directly or indirectly defined in terms of itself) and (ii) in the definition of the constants Z_j, O_j, and E_j modeling the registers.

Here we show that we can rewrite the modeling of RAMs in terms of replication, at the price of introducing a limited form of mobility of names, namely, the mobility supported by the calculus $\pi\mathrm{I}^2_!$.

Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. As far as (i) is concerned, we control the flow of execution of the program instructions simply by representing explicitly the program counter. We use $\overline{p_i}$ in order to indicate that $(i : I_i)$ is the next instruction to execute.

According to this approach, each instruction $(i : I_i)$ is represented by a process which is guarded by an input operation p_i, subsequently performs the corresponding operation on the registers, then waits for an acknowledgement indicating that the operation on the registers have been performed, and finally updates the program counter by producing $\overline{p_{i+1}}$ (or $\overline{p_s}$ in the case of jump).

Formally, the instruction $(i : I_i)$ is modeled by $[\![(i : I_i)]\!]$ which is a shorthand notation for the following processes.

$$
\begin{array}{lcll}
[\![(i : I_i)]\!] & : & !p_i.\overline{inc_j}.ack.\overline{p_{i+1}} & \text{if } I_i = Succ(r_j) \\
[\![(l : I_l)]\!] & : & !p_l.(\overline{dec_j}.ack.\overline{p_{l+1}} \ + \ \overline{zero_j}.ack.\overline{p_s}) & \text{if } I_i = DecJump(r_j, s)
\end{array}
$$

In this case the acknowledgement is always necessary because, as it will be clear in the following, the update of the register requires several internal steps.

As far as (ii) is concerned, we use the two channel names z_j and s_j. The name z_j is used to trigger a processes which represents the register r_j when empty, while s_j is used to trigger terms that represent the register when it is not empty. We model each register r_j, when it is empty, with the following process, simply denoted with $[\![r_j = 0]\!]$ in the following:

$$
\begin{array}{lll}
[\![r_j = 0]\!] \ : & & (zero_j.\overline{z_j} \ + \ inc_j.\overline{s_j}(x).x.\overline{z_j}) \mid \\
& & !z_j.\overline{ack}.(zero_j.\overline{z_j} \ + \ inc_j.\overline{s_j}(x).x.\overline{z_j}) \mid \\
& & !s_j(x).(\nu z)(\overline{z} \mid !z.\overline{ack}.(dec_j.\overline{x} \ + \ inc_j.\overline{s_j}(y).y.\overline{z}))
\end{array}
$$

Also in this case, the idea is to exploit a chain of nested restrictions; however, the chain is obtained here with a different technique. In the previous encoding we have used recursively defined processes; in this case, we exploit replicated processes and the possibility to extend the scope of local names using name mobility. More precisely, we use replication to have an unbounded amount of processes, each one representing a unit inside the register; these processes are activated one after the other each time the register is incremented. The chain of restrictions is as follows: each of these processes has a local name y, and when an increment occurs, the last activated process triggers a new instance of these terms, and passes to the new term its local name y.

Definition 8. *Let R be a RAM with program instructions $(1 : I_1), \ldots, (m : I_m)$ and registers $r_1, \ldots, r_n$. Given the initial configuration $(1, 0, \ldots, 0)$ of R we define*

$$[\![(1,0,\ldots,0)]\!]_! \;=\; \overline{p_1}|[\![(1 : I_1)]\!]|\ldots|[\![(m : I_m)]\!]|[\![r_1 = 0]\!]|\ldots|[\![r_n = 0]\!]$$

with $[\![(i : I_i)]\!]$ and $[\![r_j = 0]\!]$ as above.

Also in this case the encoding faithfully simulates the computation of the corresponding RAM. Indeed, the Theorem 1 holds also for the encoding $[\![\,]\!]_!$.

4 Decidability of Termination in $\pi\mathrm{I}^1_!$

In both of the RAM encodings presented in the previous section natural numbers are represented by chains of nested restrictions, that are constructed by exploiting either constant definitions or name passing in the calculus with replication. In this section we show that in $\pi\mathrm{I}_!$ name passing (at least the limited form of name passing of $\pi\mathrm{I}^1_!$) is really needed to obtain Turing completeness. In fact we prove that termination is decidable for $\pi\mathrm{I}^1_!$ processes.

This result is based on the theory of well-structured transition systems [FS01]; first of all, we provide an alternative semantics for $\pi\mathrm{I}^1_!$ that is equivalent w.r.t. termination to the one presented in Section 2, but is based on a finitely branching transition system. Then, by exploiting the theory developed in [FS01], we show that termination is decidable for $\pi\mathrm{I}^1_!$ processes.

We start recalling some basic definitions and results of [FS01], concerning well-structured transition systems, that will be used in the following.

A *quasi-ordering* is a reflexive and transitive relation.

Definition 9. *A* well-quasi-ordering *(wqo) is a quasi-ordering $\leq$ over a set X such that, for any infinite sequence $x_0, x_1, x_2, \ldots$ in X, there exist indexes $i < j$ such that $x_i \leq x_j$.*

Note that, if $\leq$ is a wqo, then any infinite sequence $x_0, x_1, x_2, \ldots$ contains an infinite increasing subsequence $x_{i_0}, x_{i_1}, x_{i_2}, \ldots$ (with $i_0 < i_1 < i_2 < \ldots$).

Transition systems can be formally defined as follows.

Definition 10. *A* transition system *is a structure $TS = (S, \rightarrow)$, where S is a set of* states *and $\rightarrow \subseteq S \times S$ is a set of* transitions.

We write $Succ(s)$ to denote the set $\{s' \in S \mid s \rightarrow s'\}$ of immediate successors of S.

TS is finitely branching *if all $Succ(s)$ are finite. We restrict to finitely branching transition systems.*

Well-structured transition system, defined as follows, provide the key tool to decide properties of computations.

Definition 11. *A* well-structured transition system with strong compatibility *is a transition system $TS = (S, \rightarrow)$, equipped with a quasi-ordering $\leq$ on S, such that the two following conditions hold:*

1. **well-quasi-ordering**: *$\leq$ is a well-quasi-ordering, and*
2. **strong compatibility**: *$\leq$ is (upward) compatible with $\rightarrow$, i.e., for all $s_1 \leq t_1$ and all transitions $s_1 \rightarrow s_2$, there exists a state t_2 such that $t_1 \rightarrow t_2$ and $s_2 \leq t_2$.*

The following theorem (a special case of a result in [FS01]) will be used to obtain our decidability result.

Theorem 2. *Let $TS = (S, \rightarrow, \leq)$ be a finitely branching, well-structured transition system with strong compatibility, decidable $\leq$ and computable Succ. The existence of an infinite computation starting from a state $s \in S$ is decidable.*

4.1 A finitely branching transition system for $\pi\mathbf{I}^1_!$

As the results on well-structured transition systems apply to finitely branching transition systems, first of all we need to define an alternative semantics for $\pi\mathrm{I}^1_!$, that is based on a finitely branching transition system and that is equivalent w.r.t. termination to the semantics presented in Section 2.

The existence of infinitely branching processes is due to the rules for alpha conversion and for replication. We define a new semantics by removing the rule for alpha conversion and by reformulating the semantics of replication.

The new transition relation $\overset{\alpha}{\mapsto}$ over $\pi\mathrm{I}^1_!$ processes is the least relation satisfying all the axioms and rules of Table 1 (where $\overset{\alpha}{\mapsto}$ is substituted for $\xrightarrow{\alpha}$) but `ALPHA`, plus the following rules `REPL1` and `REPL2`.

$$\texttt{REPL1}: \frac{P \overset{\alpha}{\mapsto} P'}{!P \overset{\alpha}{\mapsto} P' |\ !P} \qquad\qquad \texttt{REPL2}: \frac{P \overset{\alpha}{\mapsto} P' \quad P \overset{\overline{\alpha}}{\mapsto} P''}{!P \overset{\tau}{\mapsto} P' |\ P'' |\ !P}$$

As done for the standard transition system, we assume that the reductions $\mapsto$ of the new semantics corresponds to the τ–labeled transitions $\overset{\tau}{\mapsto}$. Also for the new semantics, we say that a process P terminates if and only if all its computations are finite, i.e. it cannot give rise to an infinite sequence of reductions $\mapsto$.

Proposition 1. *Let $P \in \pi I^1_!$. Then P terminates according to the semantics $\longrightarrow$ iff P terminates according to the new semantics $\mapsto$.*

4.2 Termination is decidable in $(\pi\mathbf{I}^1_!, \mapsto)$

In this section we equip the transition system $(\pi\mathrm{I}^1_!, \mapsto)$ with a preorder on processes which turns out to be a well-quasi-ordering compatible with $\mapsto$. Thus, exploiting the Theorem 2 we show that termination is decidable.

Definition 12. *Let $P \in \pi I^1_!$. With $Deriv(P)$ we denote the set of processes reachable from P with a sequence of reduction steps:*

$$Deriv(P) = \{Q \mid P \mapsto^* Q\}$$

To define the wqo on processes we need the following structural congruence, that turns out to be compatible with $\mapsto$.

Definition 13. *We define* $\equiv$ *as the least congruence relation satisfying the following axioms:* $P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)|R \quad P|\mathbf{0} \equiv P$

Proposition 2. *Let* $P, Q \in \pi I^1_!$. *If* $P \equiv Q$ *and* $Q \overset{\alpha}{\mapsto} Q'$ *then there exists* P' *such that* $P \overset{\alpha}{\mapsto} P'$ *and* $P' \equiv Q'$.

Now we are ready to define the preorder on processes:

Definition 14. *Let* $P, Q \in \pi I^1_!$. *We write* $P \preceq Q$ *iff there exist* n, $x_1, \ldots, x_n$, P', R, $P_1, \ldots, P_n$, $Q_1, \ldots, Q_n$ *such that* $P \equiv P'|\prod_{i=1}^{n}(\nu x_i)P_i$, $Q \equiv P'|R|\prod_{i=1}^{n}(\nu x_i)Q_i$, *and* $P_i \preceq Q_i$ *for* $i = 1, \ldots, n$.

Theorem 3. *Let* $P \in \pi I^1_!$. *Then the transition system* $(Deriv(P), \mapsto, \preceq)$ *is a well-structured transition system with strong compatibility, decidable* $\preceq$ *and computable* $Succ$.

Corollary 1. *Let* $P \in \pi I^1_!$. *The termination of process* P *is decidable.*

5 Related Work and Conclusion

We have studied the expressive power of repetition and recursive process definition in the context of π-calculus with internal mobility. We have considered a mobility hierarchy $\pi\mathrm{I}^1_!$, $\pi\mathrm{I}^2_!$, ... , $\pi\mathrm{I}^n_!$, ... , $\pi\mathrm{I}_!$ [San96] which leads from the core of CCS to the π-calculus with internal mobility via a sequence of calculi with strictly increasing expressive power: each calculus $\pi\mathrm{I}^n_!$ allows mobility of order n, that is, it allows dependency chains among names of length at most n. We have proved that repetition and recursive process definition have the same expressive power, provided that the minimal form of mobility in the hierarchy is allowed. More precisely, we have shown that Random Access Machines (and therefore Turing machines) can be deterministically encoded in the language $\pi\mathrm{I}^2_!$ which uses repetition instead of recursive definitions.

On the other hand, such an equivalence does not hold when mobility is not allowed, as we have proved that termination is decidable in $\pi\mathrm{I}^1_!$. Since Turing machines can be encoded deterministically in $\pi\mathrm{I}^1_D$ (core of CCS with recursive definitions) this implies that $\pi\mathrm{I}^1_D$ is strictly more expressive than $\pi\mathrm{I}^1_!$ and therefore provides a formal account for the common agreement which considers recursive process definition more powerful than repetition when no name mobility is allowed. As previously mentioned $\pi\mathrm{I}^1_!$ can be seen as a core CCS (with replication), as $\pi\mathrm{I}^1_!$ does not allow relabeling and uses guarded choice rather than general choice. Nevertheless, we claim that our discriminating result, which shows that recursive process definition is more powerful than repetition, holds also for full CCS, since the arguments that we have used in the proofs can be extended to deal with relabeling and general choice.

Our results have the following intuitive explanation. In order to obtain a Turing powerful formalism one needs to express and manipulate the natural numbers, modeled in terms of some suitable representation. In the case of $\pi\mathrm{I}^1_D$ such a representation is provided by the nesting of processes, as we have shown with our encoding of the RAMs: the successor of n can be obtained by constant application in the scope of the term representing n. Such a representation is not possible when replication is used, since the copies of a process are all at the same level and the different copies can communicate only via pure synchronization messages. Therefore, in presence of replication, we express natural numbers by exploiting name mobility, since the exchange of new names allows to hierarchically structure the various copies of a process. We have shown that a minimal form of mobility is enough, as it is sufficient to link two processes by making a name dependent on another one.

Related to the present work is also the paper [NPV02] where the authors investigate the expressive power of several timed concurrent constraint languages obtained by using different extensions of finite processes. In particular, one of the results in that paper shows that the language with replication is strictly less expressive than the language with recursive definitions of processes (in case process constants have parameters). Because of the very different underlying computational model, these results cannot be applied directly to the π-calculus.

Acknowledgments – We thank the referees for their comments and Catuscia Palamidessi and Frank Valencia for fruitful discussions on a preliminary version of the paper.

References

[Bor98] M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science*, 195(2): 205-226, 1998.

[BGZ03] N. Busi, M. Gabbrielli, and G. Zavattaro. Replication vs. Recursive Definitions in Channel Based Calculi (extended version). Available at `http://cs.unibo.it/~zavattar/papers.html`.

[CG00] L. Cardelli and A.D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[FS01] A. Finkel and Ph. Schnoebelen. Well-Structured Transition Systems Everywhere ! *Theoretical Computer Science*, 256:63–92, 2001.

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

[Mil01] R. Milner. Foreword of *The pi-calculus: a Theory of Mobile Processes*, by D. Sangiorgi and D. Walker. Cambridge University Press, 2001.

[MPW92] R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77. Academic Press, 1992.

[NPV02] M. Nielsen, C. Palamidessi, and F. D. Valencia. On the Expressive Power of Temporal Concurrent Constraint Programming Languages. In *Proc. of 4th International Conference on Principles and Practice of Declarative Programming (PPDP 2002)*. ACM Press, 2002.

[San96] D. Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.