On the Expressiveness of CCS with Replication

Jesús A. Aranda[1], Cinzia Di Giusto[2], Mogens Nielsen[3], and Frank D. Valencia[4]

[1] LIX, École Polytechnique, France
jesus.aranda@lix.polytechnique.fr
[2] Dip. Scienze dell'Informazione, Università di Bologna, Italy
digiusto@cs.unibo.it
[3] BRICS - University of Aarhus, Denmark
mn@brics.dk
[4] CNRS - LIX École Polytechnique, France
frank.valencia@lix.polytechnique.fr

Abstract. A remarkable result in [4] shows that in spite of its being strictly less expressive than CCS w.r.t. weak bisimilarity, CCS$_!$ (a CCS variant where infinite behavior is specified by using replication rather than recursion) is Turing powerful. This is done by encoding Random Access Machines (RAM) in CCS$_!$. The encoding is said to be *non-faithful*, in the sense that it may move from a state which can lead to termination into a divergent one which do not correspond to the any configuration of the encoded RAM. I.e., the encoding is not termination preserving.
In this paper we explore the expressiveness of CCS$_!$ w.r.t. the existence of faithful encodings of models of computability *strictly less* expressive than Turing Machines. Namely, grammars of types 1,2 and 3 in the Chomsky Hierarchy. We provide faithful encodings of type 3 grammars (Regular Languages). We then show that it is impossible to provide a faithful encoding of type 2 grammars (Context Free Languages). We show that termination-preserving CCS$_!$ processes can generate languages which are not type 2. We finally show that the languages generated by termination-preserving CCS$_!$ processes are type 1 (Context Sensitive Languages).

1 Introduction

The study of concurrency is often conducted with the aid of process calculi. A common feature of these calculi is that they treat processes much like the λ-calculus treats computable functions. They provide a language in which the structure of *terms* represents the structure of processes together with a *reduction* relation to represent computational steps. Undoubtedly Milner's CCS [9], a calculus for the modeling and analysis of synchronous communication, remains a standard representative of such calculi.

Infinite behaviour is ubiquitous in concurrent systems. Hence, it ought to be represented by process terms. In the context of CCS we can find at least two representations of them: *Recursive definitions* and *Replication*. Recursive process definitions take the form $A(y_1, \ldots, y_n)$ each assumed to have a unique, possibly recursive, *parametric process definition* $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$. The intuition is that $A(y_1, \ldots, y_n)$ behaves as P with each y_i replacing x_i. Replication takes the form $!P$ and it means $P \mid P \mid \cdots$;

an unbounded number of copies of the process P in parallel. An interesting result is that in the π-calculus, itself a generalization of CCS, parametric recursive definitions can be encoded using replication up to weak bisimilarity. This is rather surprising since the syntax of $!P$ and its description are so simple. In fact, in [3] it is stated that in CCS recursive expressions are more expressive than replication. More precisely, it is shown that it is impossible to provide a weak-bisimulation preserving encoding from CCS with recursion, into the CCS variant in which infinite behaviour is specified only with replication. From now on we shall use CCS to denote CCS with recursion and $\text{CCS}_!$ to the CCS variant with replication.

Now, a remarkable expressiveness result in [4] states that, in spite of its being less expressive than CCS in the sense mentioned above, $\text{CCS}_!$ is Turing powerful. This is done by encoding (Deterministic) Random Access Machines (RAM) in $\text{CCS}_!$. Nevertheless, the encoding is not *faithful* (or deterministic) in the sense that, unlike the encoding of RAMs in CCS, it may introduce computations which do not correspond to the expected behaviour of the modeled machine. Such computations are forced to be *infinite* and thus regarded as non-halting computations which are therefore ignored. Only the finite computations correspond to those of the encoded RAM.

A crucial observation from [4] is that to be able to force wrong computation to be infinite, the $\text{CCS}_!$ encoding of a given RAM can, during evolution, move from a state which may terminate (i.e., weakly terminating state) into one that cannot terminate (i.e., strong non-terminating state). In other words, the encoding does not *preserve (weak) termination* during evolution. It is worth pointing that since RAMs are deterministic machines, their faithful encoding in CCS given in [3] does preserve weak termination during evolution. A legitimate question is therefore: What can be encoded with termination-preserving $\text{CCS}_!$ processes ?

This work. We shall investigate the expressiveness of $\text{CCS}_!$ processes which indeed preserve (weak) termination during evolution. This way we disallow the technique used in [4] to unfaithfully encode RAMs.

A sequence of actions s (over a finite set of actions) performed by a process P specifies a sequence of interactions with P's environment. For example, $s = a^n.\bar{b}^n$ can be used to specify that if P is input n a's by environment then P can output n b's to the environment. We therefore find it natural to study the expressiveness of processes wrt sequences (or patterns) of interactions (languages) they can describe. In particular we shall study the expressiveness of $\text{CCS}_!$ w.r.t. the existence of termination preserving encodings of grammars of Types 1 (Context Sensitive grammars), 2 (Context Free grammars) and 3 (Regular grammars) in the Chomsky Hierarchy whose expressiveness corresponds to (non-deterministic) Linear-bounded, Pushdown and Finite-State Automata, respectively. As elaborated later in the related work, similar characterizations are stated in the Caucal hierarchy of transition systems for other process algebras [2].

It should be pointed out that by using the non termination-preserving encoding of RAM's in [3] we can encode Type 0 grammars (which correspond to Turing Machines) in $\text{CCS}_!$.

Now, in principle the mere fact that a computation model fails to generate some particular language may not give us a definite answer about its computation power. For

a trivial example, consider a model similar to Turing Machines except that the machines always print the symbol a on the first cell of the output tape. The model is essentially Turing powerful but fails to generate b. Nevertheless, our restriction to termination-preserving processes is a natural one, much like restricting nondeterministic models to deterministic ones, meant to rule out unfaithful encodings of the kind used in [4]. As matter of fact, Type 0 grammars can be encoded by using the termination-preserving encoding of RAMs in CCS [3].

Contributions. For simplicity let us use $\text{CCS}_!^{-\omega}$ to denote the set of $\text{CCS}_!$ processes which preserve weak termination during evolution as described above. We first provide a language preserving encoding of Regular grammars into $\text{CCS}_!^{-\omega}$. We also prove that $\text{CCS}_!^{-\omega}$ processes can generate languages which cannot be generated by any Regular grammar. Our main contribution is to show that it is *impossible* to provide language preserving encodings from Context-Free grammars into $\text{CCS}_!^{-\omega}$. Conversely, we also show that $\text{CCS}_!^{-\omega}$ can generate languages which cannot be generated by any Context-free grammar. We conclude our classification by stating that all languages generated by $\text{CCS}_!^{-\omega}$ processes are context sensitive. The results are summarized in Fig. 1.

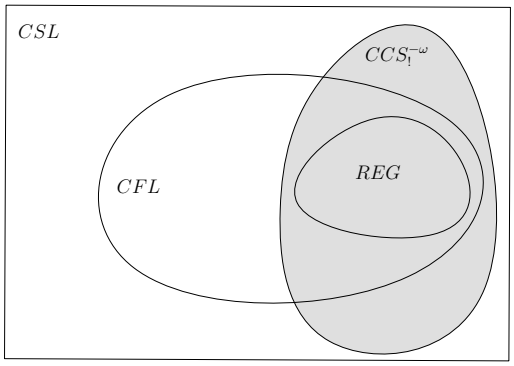

Fig. 1. Termination-Preserving $\text{CCS}_!$ Processes ($\text{CCS}_!^{-\omega}$) in the Chomsky Hierarchy.

Outline of the paper. The rest of the paper is organized as follows. Section 2 introduces the CCS calculi under consideration. We then discuss in Section 3 how unfaithful encodings are used in [4] to provide an encoding of RAM's. We prove the above-mentioned results in Section 4. Finally we give some concluding remarks in Section 5.

2 Preliminaries

In what follows we shall briefly recall the CCS constructs and its semantics as well as the $\text{CCS}_!$ calculus.

2.1 The Calculi

Finite CCS. In CCS, processes can perform actions or synchronize on them. These actions can be either offering port *names* for communication, or the so-called *silent* action τ. We presuppose a countable set $\mathcal{N}$ of port *names*, ranged over by $a, b, x, y \ldots$ and their primed versions. We then introduce a set of *co-names* $\overline{\mathcal{N}} = \{\overline{a} \mid a \in \mathcal{N}\}$ disjoint from $\mathcal{N}$. The set of *labels*, ranged over by l and l', is $\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$. The set of *actions* Act, ranged over by α and β, extends $\mathcal{L}$ with a new symbol τ. Actions a and $\overline{a}$ are thought of as *complementary*, so we decree that $\overline{\overline{a}} = a$. We also decree that $\overline{\tau} = \tau$.

The processes specifying finite behaviour are given by:

$$P, Q \ldots := 0 \;\Big|\; \alpha.P \;\Big|\; (\nu a)P \;\Big|\; P \mid Q \tag{1}$$

Intuitively 0 represents the process that does nothing. The process $\alpha.P$ performs an action α then behaves as P. The restriction $(\nu a)P$ behaves as P except that it can offer neither a nor $\bar{a}$ to its environment. The names a and $\bar{a}$ in P are said to be *bound* in $(\nu a)P$. The *bound names* of P, $bn(P)$, are those with a bound occurrence in P, and the *free names* of P, $fn(P)$, are those with a not bound occurrence in P. The set of names of P, $n(P)$, is then given by $fn(P) \cup bn(P)$. Finally, $P \mid Q$ represents parallelism; either P or Q may perform an action, or they can also synchronize when performing complementary actions.

Notation 1 *We shall write the summation $P + Q$ as an abbreviation of the process $(\nu\, u)(\overline{u} \mid u.P \mid u.Q)$. We also use $(\nu a_1, a_2, \ldots a_n)P$ as a short hand for $(\nu a_1) \ldots (\nu a_n)P$. We often omit the "$0$" in $\alpha.0$.*

The above description is made precise by the operational semantics in Table 1. A transition $P \xrightarrow{\alpha} Q$ says that P can perform α and evolve into Q. In the literature there

$$\text{ACT}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{RES}\ \frac{P \xrightarrow{\alpha} P'}{(\nu\, a)P \xrightarrow{\alpha} (\nu\, a)P'}\ \text{if}\ \alpha \notin \{a, \overline{a}\}$$

$$\text{PAR}_1\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{PAR}_2\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{COM}\ \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\overline{l}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Table 1. An operational semantics for finite processes.

are at least two alternatives to extend the above syntax to express infinite behaviour. We describe them next.

2.2 Parametric Definitions: CCS and CCS$_{\mathtt{P}}$

A typical way of specifying infinite behaviour is by using parametric definitions [10]. In this case we extend the syntax of finite processes (Equation 1) as follows:

$$P, Q, \ldots := \ldots \;\Big|\; A(y_1, \ldots, y_n) \tag{2}$$

Here $A(y_1, \ldots, y_n)$ is an *identifier* (also *call*, or *invocation*) of arity n. We assume that every such an identifier has a unique, possibly recursive, *definition* $A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A$ where the x_i's are pairwise distinct, and the intuition is that $A(y_1, \ldots, y_n)$ behaves as its *body* P_A with each y_i replacing the *formal parameter* x_i. For each $A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A$, we require $fn(P_A) \subseteq \{x_1, \ldots, x_n\}$.

Following [5], we should use CCS_{p} to denote the calculus with parametric definitions with the above syntactic restrictions.

Remark 1. As shown in [5], however, CCS_{p} is equivalent w.r.t. strong bisimilarity to the standard CCS. We shall then take the liberty of using the terms CCS and CCS_{p} to denote the calculus with parametric definitions as done in [10].

The rules for CCS_{p} are those in Table 1 plus the rule:

$$\text{CALL}\ \frac{P_A[y_1, \ldots, y_n/x_1, \ldots, x_n] \stackrel{\alpha}{\longrightarrow} P'}{A(y_1, \ldots, y_n) \stackrel{\alpha}{\longrightarrow} P'}\ \text{if } A(x_1, \ldots, x_n) \stackrel{\texttt{def}}{=} P_A \tag{3}$$

As usual $P[y_1 \ldots y_n/x_1 \ldots x_n]$ results from syntactically replacing every free occurrence of x_i with y_i renaming bound names in P, i.e., α-*conversion*, wherever needed to avoid capture.

2.3 Replication: $\text{CCS}_!$

One simple way of expressing infinite is by using replication. Although, mostly found in calculus for mobility such as the π-calculus and mobile ambients, it is also studied in the context of CCS in [3,5].

For replication the syntax of finite processes (Equation 1) is extended as follows:

$$P, Q, \ldots := \ldots \mid\ !P \tag{4}$$

Intuitively the process $!P$ behaves as $P \mid P \mid \ldots \mid P \mid !P$; unboundedly many P's in parallel. We call $\text{CCS}_!$ the calculus that results from the above syntax [5]. The operational rules for $\text{CCS}_!$ are those in Table 1 plus the following rule:

$$\text{REP}\ \frac{P \mid\ !P \stackrel{\alpha}{\longrightarrow} P'}{!P \stackrel{\alpha}{\longrightarrow} P'} \tag{5}$$

3 The Role of Strong Non-Termination

In this section we shall single out the fundamental non-deterministic strategy for the Turing-expressiveness of $\text{CCS}_!$. First we need a little notation.

[5] The work [4] considers also guarded summation for $\text{CCS}_!$. The results about the encodability of RAM's our work builds on can easily be adapted to our summation-free fragment.

Notation 2 *Define* $\stackrel{s}{\Longrightarrow}$*, with* $s = \alpha_1 \ldots \alpha_n \in \mathcal{L}^*$*, as*

$$(\stackrel{\tau}{\longrightarrow})^* \stackrel{\alpha_1}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^* \ldots (\stackrel{\tau}{\longrightarrow})^* \stackrel{\alpha_n}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^*.$$

For the empty sequence $s = \epsilon$*,* $\stackrel{s}{\Longrightarrow}$ *is defined as* $(\stackrel{\tau}{\longrightarrow})^*$*.*

We shall say that a process generates a sequence of non-silent actions s if it can perform the actions of s in a finite maximal sequence of transitions. More precisely:

Definition 1 (Sequence and language generation). *The process* P generates *a sequence* $s \in \mathcal{L}^*$ *if and only if there exists* Q *such that* $P \stackrel{s}{\Longrightarrow} Q$ *and* $Q \stackrel{\alpha}{\not\longrightarrow}$ *for any* $\alpha \in Act$*. Define the language of (or generated by) a process* P*,* $L(P)$*, as the set of all sequences* P *generates.*

The above definition basically states that a sequence is generated when no reduction rule can be applied. It is inspired by language generation of the model of computations we are comparing our processes with. Namely, formal grammars where a sequence is generated when no rewriting rule can be applied.

As we shall see below (strong) non-termination plays a fundamental role in the expressiveness of CCS$_!$. We borrow the following terminology from rewriting systems:

Definition 2 (Termination). *We say that a process* P *is* (weakly) terminating *(or that it can terminate) if and only if there exists a sequence* s *such that* P *generates* s*. We say that* P *is* (strongly) non-terminating*, or that it* cannot terminate *if and only if* P *cannot generate any sequence.*

The authors in [4] show the Turing-expressiveness of CCS$_!$, by providing a CCS$_!$ encoding $[\![\cdot]\!]$ of Random Access Machines (RAMs) a well-known Turing powerful deterministic model [11]. The encoding is said to be *unfaithful* (or nondeterministic) in the following sense: Given M, during evolution $[\![M]\!]$ may make a transition, by performing a τ action, from a weakly terminating state (process) into a state which do not correspond to any configuration of M. Nevertheless such states are strongly non-terminating processes. Therefore, they may be thought of as being configurations which cannot lead to a halting configuration. Consequently, the encoding $[\![M]\!]$ does not *preserve (weak) termination* during evolution.

Now rather than recalling the full encoding of RAMs in CCS$_!$, let us use a much simpler example which uses the same technique in [4]. Below we encode a typical context sensitive language in CCS$_!$.

Example 1. Consider the following processes:

$$\begin{aligned} P &= (\nu\, k_1, k_2, k_3, u_b, u_c)(\ \overline{k_1}\ \mid\ \overline{k_2}\ \mid\ Q_a\ \mid\ Q_b\ \mid\ Q_c) \\ Q_a &= !k_1.a.(\overline{k_1}\ \mid\ \overline{k_3}\ \mid\ \overline{u_b}\ \mid\ \overline{u_c}) \\ Q_b &= k_1.!k_3.k_2.u_b.b.\overline{k_2} \\ Q_c &= k_2.(!u_c.c\ \mid\ u_b.DIV) \end{aligned}$$

where $DIV = !\tau$. It can be verified that $L(P) = \{a^n b^n c^n\}$. Intuitively, in the process P above, Q_a performs (a sequence of actions) a^n for an arbitrary number n (and also produces n u_b's). Then Q_b performs b^m for an arbitrary number $m \leq n$ and each time it produces b it consumes a u_b. Finally, Q_c performs c^n and diverges if $m < n$ by checking if there are u_b's that were not consumed. □

The Power of Non-Termination. Let us underline the role of strong non-termination in Example 1. Consider a run

$$P \stackrel{a^n b^m}{\Longrightarrow} \ldots$$

Observe that the name u_b is used in Q_c to test if $m < n$, by checking whether some u_b were left after generating b^m. If $m < n$, the non-terminating process DIV is triggered and the extended run takes the form

$$P \stackrel{a^n b^m c^n}{\Longrightarrow} \stackrel{\tau}{\longrightarrow} \stackrel{\tau}{\longrightarrow} \ldots$$

Hence the sequence $a^n b^m c^n$ arising from this run (with $m < n$) is therefore not included in $L(P)$.

The tau move. It is crucial to observe that there is a τ transition arising from the moment in which $\overline{k_2}$ chooses to synchronize with Q_c to start performing the c actions. One can verify that if $m < n$ then the process just before that τ transition is weakly terminating while the one just after is strongly non-terminating. □

Formally the class of termination-preserving processes is defined as follows.

Definition 3 (Termination Preservation). *A process P is said to be (weakly) termination-preserving if and only if whenever $P \stackrel{s}{\Longrightarrow} Q \stackrel{\tau}{\longrightarrow} R$:*

- *if Q is weakly terminating then R is weakly terminating.*

We shall sometimes use $CCS_!^{-\omega}$ to denote the set of those $CCS_!$ processes which are termination-preserving.

One may wonder why only τ actions are not allowed in Definition 3 when moving from a weakly terminating state into a strongly non-terminating one. The next proposition answers to this.

Proposition 1. *For every $P, P', \alpha \neq \tau$ if $P \stackrel{\alpha}{\longrightarrow} P'$ and P is weakly terminating then P' must be weakly terminating.*

In the following sections we shall investigate how far we can get with termination preserving processes.

4 $CCS_!$ and Chomsky Hierarchy

In this section we shall study the expressiveness of termination-preserving $CCS_!$ processes in the Chomsky hierarchy of grammars. We recall that, in a strictly decreasing expressive order, Types 0, 1, 2 and 3 in the Chomsky hierarchy correspond, respectively, to unrestricted-grammars (Turing Machines), Context Sensitive Grammars (Non-Deterministic Linear Bounded Automata), Context Free Grammars (Non-Deterministic PushDown Automata), and Regular Grammars (Finite State Automata).

We assume that the reader is familiar with the notions and notations of formal grammars. A grammar is a quadruple $G = (\Sigma, N, S, P)$ where Σ are the terminal symbols, N the non-terminals, S the initial symbol, P the set of production rules. The language of (or generated by) a formal grammar G, denoted as $L(G)$, is defined as all those strings in Σ^* that can be generated by starting with the start symbol S and then applying the production rules in P until no more non-terminal symbols are present.

4.1 Encoding Regular Languages

Regular Languages (REG) are those generated by grammars whose production rules can only be of the form $A \rightarrow a$ or $A \rightarrow a.B$. Recall that they can be alternatively characterized as those recognized by regular expressions.

Regular expressions are given by the following syntax:

$$e = \emptyset \mid \epsilon \mid a \mid e_1 + e_2 \mid e_1.e_2 \mid e^*$$

where a is a terminal symbol.

Definition 4. *Given a regular expression e, we define $[\![e]\!]$ as the CCS$_!$ process $(\nu\, m)([\![e]\!]_m \mid m)$ where $[\![e]\!]_m$, with $m \notin fn([\![e]\!])$, is inductively defined as follows:*

$$
\begin{array}{ll}
[\![\emptyset]\!]_m = & DIV \\
[\![\epsilon]\!]_m = & \overline{m} \\
[\![a]\!]_m = & a.\overline{m} \\
[\![e_1 + e_2]\!]_m = & \begin{cases} [\![e_1]\!]_m & \text{if } L(e_2) = \emptyset \\ [\![e_2]\!]_m & \text{if } L(e_1) = \emptyset \\ [\![e_1]\!]_m + [\![e_2]\!]_m & \text{otherwise} \end{cases} \\
[\![e_1.e_2]\!]_m = & (\nu\, m_1)([\![e_1]\!]_{m_1} \mid m_1.[\![e_2]\!]_m) \text{ with } m_1 \notin fn(e_1) \\
[\![e^*]\!]_m = & \begin{cases} \overline{m} & \text{if } L(e) = \emptyset \\ (\nu\, m')(\overline{m'} \mid\, !m'.[\![e]\!]_{m'} \mid m'.\overline{m}) \text{ with } m' \notin fn(e) & \text{otherwise} \end{cases}
\end{array}
$$

where $DIV = !\tau$.

Remark 2. The conditionals on language emptiness in Definition 4 are needed to make sure that the encoding of regular expressions always produce termination preserving processes. To see this consider the case $a+\emptyset$. Notice that while $[\![a]\!] = a$ and $[\![\emptyset]\!] = DIV$ are termination-preserving, $a + DIV$ is not. Hence $[\![e_1 + e_2]\!]$ cannot be defined as $[\![e_1]\!] + [\![e_2]\!]$. Since the emptiness problem is decidable for regular expressions, it is clear that given e, $[\![e]\!]$ can be effectively constructed.

The following proposition, which can be proven by using induction on the structure of regular expressions, states the correctness of the encoding.

Proposition 2. *Let $[\![e]\!]$ as in Definition 4. We have $L(e) = L([\![e]\!])$ and furthermore $[\![e]\!]$ is termination-preserving.*

From the standard encoding from Type 3 grammars to regular expressions and the above proposition we obtain the following result.

Theorem 3. *For every Type 3 grammar G, we can construct a termination-preserving CCS$_!$ process P_G such that $L(G) = L(P_G)$.*

The converse of the theorem above does not hold; Type 3 grammars are strictly less expressive.

Theorem 4. *There exists a termination-preserving CCS$_!$ process P such that $L(P)$ is not Type 3.*

The above statement can be shown by providing a process which generates the typical $a^n b^n$ context-free language. Namely, let us take

$$P = (\nu\, k, u)(\overline{k} \mid !(k.a.(\overline{k} \mid \overline{u})) \mid k.!(u.b)).$$

One can verify that P is termination-preserving and that $L(P) = a^n b^n$.

4.2 Impossibility Result: Context Free Languages

Context-Free Languages (CFL) are those generated by Type 2 grammars: grammars where every production is of the form $A \rightarrow \gamma$ where A is a non-terminal symbol and γ is a string consisting of terminals and/or non-terminals.

We have already seen that termination-preserving CCS$_!$ process can encode a typical CFL language such as $a^n b^n$. Nevertheless, we shall show that they cannot in general encode Type 2 grammars.

The nesting of restriction processes plays a key role in the following results CCS$_!$.

Definition 5. *The maximal number of nesting of restrictions $|P|_\nu$ can be inductively given as follows:*

$$|(\nu\, x)P|_\nu = 1 + |P|_\nu \qquad |P \mid Q|_\nu = max(|P|_\nu, |Q|_\nu)$$
$$|\alpha.P|_\nu = |!P|_\nu = |P|_\nu \qquad |0|_\nu = 0$$

A very distinctive property of CCS$_!$ is that the maximal nesting of restrictions is invariant during evolution.

Proposition 3. *Let P and Q be CCS$_!$ processes. If $P \stackrel{s}{\Longrightarrow} Q$ then $|P|_\nu = |Q|_\nu$.*

Remark 3. In CCS because of the *unfolding* of recursive definitions the nesting of restrictions can increase unboundedly during evolution [6]. E.g., consider $A(a)$ where $A(x) \stackrel{\mathtt{def}}{=} (\nu\, y)(x.\bar{y}.R \mid y.A(x))$ (see Section 2.2) which has the following sequence of transitions

$$A(a) \stackrel{aaa\ldots}{\Longrightarrow} (\nu y)(R \mid (\nu y)(R \mid (\nu y)(R \mid \ldots)))$$

□

Another distinctive property of CCS$_!$ is that if a CCS$_!$ process can perform a given action β, it can always do it by performing a number of actions bounded by a value that depends only on the size of the process. In fact, as stated below, for a significant class

[6] Also in the π-calculus [15], an extension of CCS$_!$ where names are communicated, the nesting of restrictions can increase during evolution due its name-extrusion capability.

of processes, the bound can be given solely in terms the maximal number of nesting of restrictions.

Now, the above statement may seem incorrect since as mentioned earlier CCS$_!$ is Turing expressive. One may think that β above could represent a termination signal in a TM encoding, then it would seem that its presence in a computation cannot be determined by something bounded by the syntax of the encoding. Nevertheless, recall that the Turing encoding in [4] may wrongly signal β (i.e., even when the encoded machine does not terminate) but it will diverge afterwards.

The following section is devoted to some lemmas needed for proving our impossibility results for CCS$_!$ processes.

Trios-Processes.

For technical reasons we shall work with a family of CCS$_!$ processes, namely *trios-processes*. These processes can only have prefixes of the form $\alpha.\beta.\gamma$. The notion of trios was introduced for the π-calculus in [14] . We shall adapt trios and use them as a technical tool for our purposes.

We shall say that a CCS$_!$ process T is a *trios-process* iff all prefixes in T are *trios*; i.e., they all have the form $\alpha.\beta.\gamma$ and satisfy the following: If $\alpha \neq \tau$ then α is a *name* bound in P, and similarly if $\gamma \neq \tau$ then γ is a *co-name* bound in P. For instance $(\nu l)(\tau.\tau.\bar{l} \mid l.a.\tau)$ is a trios-process. Intuitively, we will view a trio $l.\beta.\bar{l}$ as linkable node with incoming link l from another trio, outgoing link $\bar{l}$ to another trio, and contents β.

Interestingly, the family of trios-processes can capture the behaviour of arbitrary CCS$_!$ processes via the following encoding:

Definition 6. *Given a CCS$_!$ process P, $[\![P]\!]$ is the trios-process $(\nu\, l)(\tau.\tau.\bar{l} \mid [\![P]\!]_l)$ where $[\![P]\!]_l$, with $l \notin n(P)$, is inductively defined as follows:*

$$
\begin{array}{ll}
[\![0]\!]_l = & 0 \\
[\![\alpha.P]\!]_l = & (\nu\, l')(l.\alpha.\overline{l'} \mid [\![P]\!]_{l'}) \text{ where } l' \notin n(P) \\
[\![P \mid Q]\!]_l = & (\nu\, l', l'')(l.\overline{l'}.\overline{l''} \mid [\![P]\!]_{l'} \mid [\![P]\!]_{l''}) \text{ where } l', l'' \notin n(P) \cup n(Q) \\
[\![!P]\!]_l = & (\nu\, l')(!l.\overline{l'}.\bar{l} \mid ![\![P]\!]_{l'}) \text{ where } l' \notin n(P) \\
[\![(\nu\, x)P]\!]_l = & (\nu\, x)[\![P]\!]_l
\end{array}
$$

Notice that the trios-process $[\![\alpha.P]\!]_l$ encodes a process $\alpha.P$ much like a linked list. Intuitively, the trio $l.\alpha.\overline{l'}$ has an outgoing link l to its continuation $[\![P]\!]'_l$ and incoming link l from some previous trio. The other cases can be explained analogously. Clearly the encoding introduces additional actions but they are all silent—i.e., they are synchronizations on the bound names l, l' and l''.

Unfortunately the above encoding is not invariant wrt language equivalence because the replicated trio in $[\![!P]\!]_l$ introduces divergence. E.g, $L((\nu x)!x) = \{\epsilon\}$ but $L([\![(\nu x)!x]\!]) = \emptyset$. Nevertheless, it has a pleasant invariant property, namely weak bisimilarity.

Definition 7 (Weak Bisimilarity). *A* (weak) simulation *is a binary relation $\mathcal{R}$ satisfying the following: $(P, Q) \in \mathcal{R}$ implies that:*

– *if* $P \stackrel{s}{\Longrightarrow} P'$ *where* $s \in \mathcal{L}^*$ *then* $\exists Q' : Q \stackrel{s}{\Longrightarrow} Q' \wedge (P', Q') \in \mathcal{R}$.

The relation $\mathcal{R}$ *is a* bisimulation *iff both* $\mathcal{R}$ *and its converse* $\mathcal{R}^{-1}$ *are -simulations. We say that* P *and* Q *are* (weak) bisimilar*, written* $P \approx Q$ *iff* $(P, Q) \in \mathcal{R}$ *for some bisimulation* $\mathcal{R}$.

Proposition 4. *For every CCS*$_!$ *process* P, $P \approx [\![P]\!]$ *where* $[\![P]\!]$ *is the trios-process constructed from* P *as in Definition 6.*

Another property of trios is that if a trios-process T can perform an action α, i.e., $T \stackrel{s.\alpha}{\Longrightarrow}$, then $T \stackrel{s'.\alpha}{\Longrightarrow}$ where s' is a sequence of actions whose length bound can be given solely in terms of $|T|_\nu$.

Proposition 5. *Let* T *be a trios-process such that* $T \stackrel{s\cdot\beta}{\Longrightarrow}$*. There exists a sequence* s'*, whose length is bounded by a value depending only on* $|T|_\nu$*, such that* $T \stackrel{s'\cdot\beta}{\Longrightarrow}$.

We conclude this technical section by outlining briefly the main aspects of the proof of the above proposition. Roughly speaking, our approach is to consider a minimal sequence of visible actions $t = \beta_1 \ldots \beta_m$ performed by T leading to β (i.e., $P \stackrel{t}{\Longrightarrow}$ and $\beta_m = \beta$) and analyze the *causal dependencies* among the (occurrences of) the actions in this t. Intuitively, β_j depends on β_i if T, while performing t, could not had performed β_j without performing β_i first. For example in

$$T = (\nu l)(\nu l')(\nu l'')(\tau.a.\bar{l} \mid \tau.b.\overline{l'} \mid l.l'.\overline{l''} \mid l''.c.\tau)$$

$\beta = c$, $t = abc$, we see that c depends on a and b, but b does not depend on a since T could had performed b before a.

We then consider the unique directed acyclic graph G_t arising from the transitive reduction[7] of the partial ordered induced by the dependencies in t. Because t is minimal, β is the only sink of G_t.

We write $\beta_i \rightsquigarrow_t \beta_j$ (β_j depends directly on β_i) iff G_t has an arc from β_i to β_j. The crucial observation from our restrictions over trios is that if $\beta_i \rightsquigarrow_t \beta_j$ then (the trios corresponding to) β_i and β_j must occur in the scope of a restriction process R_{ij} in T.

To give an upper bound on the number of nodes of G_t (i.e., the length of t), we give an upper bound on its length and maximal in-degree. Take a path $\beta_{i_1} \rightsquigarrow_t \beta_{i_2} \ldots \rightsquigarrow_t \beta_{i_u}$ of size u in G_t. With the help of the above observation, we consider sequences of restriction processes $R_{i_1 i_2} R_{i_2 i_3} \ldots R_{i_{u-1} i_u}$ such that for every $k < u$ the actions β_{i_k} and $\beta_{i_{k+1}}$ (i.e., the trios where they occur) must be under the scope of $R_{i_k i_{k+1}}$. Note that any two different restriction processes with a common trio under their scope (e.g. $R_{i_1 i_2}$ and $R_{i_2 i_3}$) must be nested, i.e., one must be under the scope of the other. This induces tree-like nesting among the elements of the sequence of restrictions. E.g., for the restrictions corresponding to $\beta_{i_1} \rightsquigarrow_t \beta_{i_2} \rightsquigarrow_t \beta_{i_3} \rightsquigarrow_t \beta_{i_4}$ we could have a tree-like situation with $R_{i_1 i_2}$ and $R_{i_3 i_4}$ being under the scope of $R_{i_2 i_3}$ and thus inducing a nesting of at least two. We show that for a sequence of restriction processes, the number m of

[7] The transitive reduction of a binary relation r on X is the smallest relation r' on X such that that the transitive closure of r' is the same as the transitive closure of r.

nesting of them satisfies $u \leq 2^m$. Since the nesting of restrictions remains invariant during evolution (Proposition 3) then $u \leq 2^{|T|_\nu}$. Similarly, we give an upper bound $2^{|T|_\nu}$ on the indegree of each node β_j of G_t (by considering sequences $R_{i_1 j}, \ldots, R_{i_m j}$ such that $\beta_{i_k} \rightsquigarrow \beta_j$, i.e having common trio corresponding to β_j under their scope). We then conclude that the number of nodes in G_t is bounded by $2^{|T|_\nu \times 2^{|T|_\nu}}$.

Main Impossibility Result.

We can now prove our main impossibility result.

Theorem 5. *There exists a Type 2 grammar G such that for every termination-preserving CCS$_!$ process P, $L(G) \neq L(P)$.*

Proof Outline. It suffices to show that no process in CCS$_!^{-\omega}$ can generate the CFL $a^n b^n c$. Suppose, as a mean of contradiction, that P is a CCS$_!^{-\omega}$ process such that $L(P) = a^n b^n c$.

Pick a sequence $\rho = P \stackrel{a^n}{\Longrightarrow} Q \stackrel{b^n c}{\Longrightarrow} T \not\rightarrow$ for a sufficiently large n. From Proposition 4 we know that for some R, $[\![P]\!] \stackrel{a^n}{\Longrightarrow} R \stackrel{b^n c}{\Longrightarrow}$ and $R \approx Q$. Notice that R may not be a trios-process as it could contain prefixes of the form $\beta.\gamma$ and γ . However, by mapping such prefixes into $\tau.\beta.\gamma$ and $\tau.\tau.\gamma$, we obtain a trios-process R' such that $R \approx R'$ and $|R|_\nu = |R'|_\nu$. We then have $R' \stackrel{b^n c}{\Longrightarrow}$ and, by Proposition 5, $R' \stackrel{s'.c}{\Longrightarrow}$ for some s' whose length is bounded by a constant k that depends only on $|R'|_\nu$. Therefore, $R \stackrel{s'.c}{\Longrightarrow}$ and since $R \approx Q$, $Q \stackrel{s'.c}{\Longrightarrow} D$ for some D. With the help of Proposition 3 and from Definition 6 it is easy to see that $|R'|_\nu = |R|_\nu = |[\![P]\!]|_\nu \leq 1 + |P| + |P|_\nu$ where $|P|$ is the size of P. Consequently the length of s' must be independent of n, and hence D must be strongly non-terminating since for any $s'' \in \mathcal{L}^*$, $a^n s' c s'' \notin L(P)$.

Notice that Q above is a member of the (non-empty) set of processes in ρ which can evolve into a strongly non-terminating process D. Let Q' be the last process in $\rho = P \stackrel{s_1}{\Longrightarrow} Q' \stackrel{s_2}{\Longrightarrow} T \not\rightarrow$ with $s_1 s_2 = a^n b^n c \in L(P)$ such that $Q' \stackrel{\alpha}{\longrightarrow} D'$ where D' is a strongly non-terminating process (see Fig. 2).

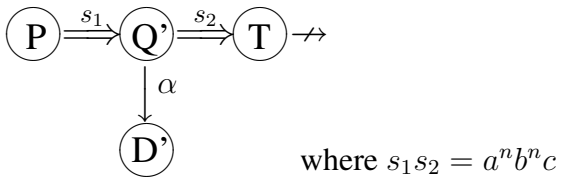

Fig. 2. The sequence ρ and alternative evolution of Q'

Now from the definition of CCS$_!^{-\omega}$ the action α above must be different from τ. Since α is a visible action and $Q' \stackrel{\alpha}{\longrightarrow}$, then α is not guarded in Q' by any prefix. We can then verify that there must be a Q'' such that $\rho = P \stackrel{s_1}{\Longrightarrow} Q' \stackrel{t_1}{\Longrightarrow} Q'' \stackrel{t_2}{\Longrightarrow} T \not\rightarrow$ with $s_2 = t_1 t_2$ and $Q'' \stackrel{\alpha}{\longrightarrow}$.

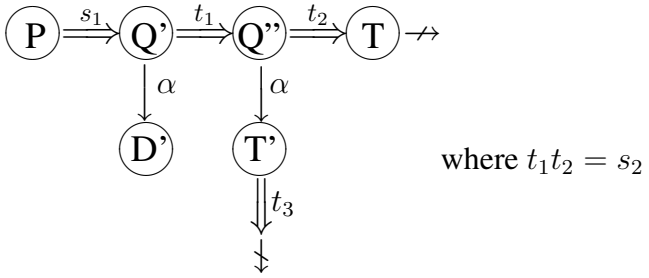

Fig. 3. Sequence ρ and alternative evolutions of Q' and Q''

Since we assumed that Q' is the last process in ρ which can evolve into a strongly nonterminating process, we know that after α, Q'' can only evolve into a weakly terminating process T': e.g. $Q'' \xrightarrow{\alpha} T' \stackrel{t_3}{\Longrightarrow} \not\rightarrow$ for some t_3 (Fig. 3).

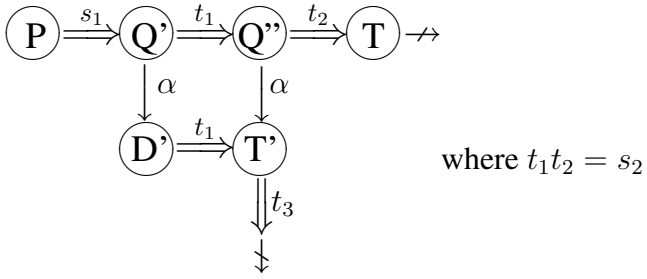

Fig. 4. Confluence for Q'

Nevertheless, from $Q' \stackrel{t_1}{\Longrightarrow} Q'' \xrightarrow{\alpha} T'$ and $Q' \xrightarrow{\alpha} D'$, we can show that $Q' \xrightarrow{\alpha} D' \stackrel{t_1}{\Longrightarrow} T'$ (Fig. 4) This contradicts the fact that D' is a strongly non-terminating process.

□

It turns out that the converse of Theorem 5 also holds: Termination-preserving CCS$_!$ processes can generate non CFL's. Take

$$P = (\nu\, k, u)(\overline{k} \mid !k.a.(\overline{k} \mid \overline{u})) \mid k.!u.(b \mid c))$$

One can verify that P is termination-preserving. Furthermore, $L(P) \cap a^*b^*c^* = a^nb^nc^n$, hence $L(P)$ is not a CFL since CFL's are closed under intersection with regular languages. Therefore:

Theorem 6. *There exists a termination-preserving CCS$_!$ process P such that $L(P)$ is not a CFL.*

Now, notice that if we allow the use of CCS$_!$ processes which are not termination-preserving, we can generate a^nb^nc straightforwardly by using a process similar to that of Example 1.

Example 2. Consider the process P below:

$$\begin{aligned} P &= (\nu\, k_1, k_2, k_3, u_b)(\ \overline{k_1}\ \mid\ \overline{k_2}\ \mid\ Q_a\ \mid\ Q_b\ \mid\ Q_c) \\ Q_a &= !k_1.a.(\overline{k_1}\ \mid\ \overline{k_3}\ \mid\ \overline{u_b}) \\ Q_b &= k_1.!k_3.k_2.u_b.b.\overline{k_2} \\ Q_c &= k_2.(c\ \mid\ u_b.DIV) \end{aligned}$$

where $DIV = !\tau$. One can verify that $L(P) = \{a^n b^n c\}$. □

Termination-Preserving CCS. Type 0 grammars can be encoded by using the termination-preserving encoding of RAMs in CCS given in [3]. However, the fact that preservation of termination is not as restrictive for CCS as it is for CCS$_!$ can also be illustrated by giving a simple termination-preserving encoding of Context-Free grammars.

Theorem 7. *For every type 2 grammar G, there exists a termination-preserving CCS process P_G, such that $L(P_G) = L(G)$.*

Proof Outline. For simplicity we restrict ourselves to Type 2 grammars in Chomsky normal form. All production rules are of the form $A \rightarrow B.C$ or $A \rightarrow a$. We can encode the productions rules of the form $A \rightarrow B.C$ as the recursive definition $A(d) \stackrel{\texttt{def}}{=} (\nu\, d')(B(d')\ \mid\ d'.C(d))$ and the terminal production $A \rightarrow a$ as the definition $A(d) \stackrel{\texttt{def}}{=} a.\overline{d}$. Rules with the same head can be dealt using the summation $P + Q$. One can verify that, given a Type 2 grammar G, the suggested encoding generates the same language as G.

Notice, however, that there can be a grammar G with a non-empty language exhibiting derivations which do not lead to a sequence of terminal (e.g., $A \rightarrow B.C$, $A \rightarrow a$, $B \rightarrow b$, $C \rightarrow D.C$,$D \rightarrow d$). The suggested encoding does not give us a termination-preserving process. However one can show that there exists another grammar G', with $L(G) = L(G')$ whose derivations can always lead to a final sequence of terminals . The suggested encoding applied to G' instead, give us a termination-preserving process. □

4.3 Inside Context Sensitive Languages (CSL)

Context-Sensitive Languages (CSL) are those generated by Type 1 grammars. We shall state that every language generated by a termination-preserving CCS$_!$ process is context sensitive.

The next proposition reveals a key property of any given termination-preserving CCS$_!$ process P which can be informally described as follows. Suppose that P generates a sequence s of size n. By using a technique similar to the proof of Theorem 5 and Proposition 5, we can prove that there must be a trace of P that generates s with a total number of τ actions bounded by kn where k is a constant associated to the size of P. More precisely,

Proposition 6. *Let P be a termination-preserving CCS$_!$ process. There exists a constant k such that for every $s = \alpha_1 \ldots \alpha_n \in L(P)$ then there must be a sequence*

$$P(\stackrel{\tau}{\longrightarrow})^{m_0} \stackrel{\alpha_1}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^{m_1} \ldots (\stackrel{\tau}{\longrightarrow})^{m_{n-1}} \stackrel{\alpha_n}{\longrightarrow} (\stackrel{\tau}{\longrightarrow})^{m_n} \not\rightarrow$$

with $\Sigma_{i=0}^{n} m_i \leq kn$.

Now recall that context-sensitive grammars are equivalent to linear bounded non-deterministic Turing machines. That is a non-deterministic Turing machine with a tape with only kn cells, where n is the size of the input and k is a constant associated with the machine. Given P, we can define a non-deterministic machine which simulates the runs of P using the semantics of CCS$_!$ and which uses as many cells as the total number of performed actions, silent or visible, multiplied by a constant associated to P. Therefore, with the help of Proposition 6, we obtain the following result.

Theorem 8. *If P is a termination-preserving CCS$_!$ process then $L(P)$ is a context-sensitive language.*

Notice that from the above theorem and Theorem 5 it follows that the languages generated by termination-preserving CCS$_!$ processes form a proper subset of context sensitive languages.

5 Related and Future Work

The closest related work is that in [3,4] already discussed in the introduction. Furthermore in [3] the authors also provide a discrimination result between CCS$_!$ and CCS by showing that the divergence problem (i.e., given P, whether P has an infinite sequence of τ moves) is decidable for the former calculus but not for the latter.

In [5] the authors study replication and recursion in CCS focusing on the role of name scoping. In particular they show that CCS$_!$ is equivalent to CCS with recursion with static scoping. The standard CCS in [9] is shown to have dynamic scoping. A survey on the expressiveness of replication vs recursion is given in [13] where several decidability results about variants of π, CCS and Ambient calculi can be found. None of these works study replication with respect to computability models less expressive than Turing Machines.

In [12] the authors showed a separation result between replication and recursion in the context of temporal concurrent constraint programming (tccp) calculi. They show that the calculus with replication is no more expressive than finite-state automata while that with recursion is Turing Powerful. The semantics of tccp is rather different from that of CCS. In particular, unlike in CCS, processes interact via the shared-memory communication model and communication is asynchronous.

In the context of calculi for security protocols, the work in [6] uses a process calculus to analyze the class of ping-pong protocols introduced by Dolev and Yao. The authors show that all nontrivial properties, in particular reachability, become undecidable for a very simple recursive variant of the calculus.The authors then show that the variant with replication renders reachability decidable. The calculi considered are also different from CCS. For example no restriction is considered and communication is asynchronous.

There is extensive work in process algebras and rewriting transition systems providing expressiveness hierarchies similar to that of Chomsky as well as results closely related to those of formal grammars. For example work involving characterization of

regular expression w.r.t. bisimilarity include [7,8] and more recently [1]. An excellent description is provided in [2]. These works do not deal with replication nor the restriction operator which are fundamental to our study.

As for future work, it would be interesting to investigate the decidability of the question whether a given $\text{CCS}_!$ process P preserves termination. A somewhat complementary study to the one carried in this paper would be to investigate what extension to $\text{CCS}_!$ is needed for providing faithful encoding of RAMs. Clearly the extension with recursion does the job but there may be simpler process construction from process algebra which also do the job.

References

1. J. C. M. Baeten and F. Corradini. Regular expressions in process algebra. In *LICS '05*, pages 12–19, Washington, DC, USA, 2005. IEEE Computer Society.
2. O. Burkart, D. Caucal, F. Moller, and B. Steffen. *Verification on infinite structures*, chapter 9, pages 545–623. Elsevier, North-Holland, 2001.
3. N. Busi, M. Gabbrielli, and G. Zavattaro. Replication vs. recursive definitions in channel based calculi. In *ICALP'03*, volume 2719 of *Lecture Notes in Computer Science*, pages 133–144. Springer-Verlag, 2003.
4. N. Busi, M. Gabbrielli, and G. Zavattaro. Comparing recursion, replication, and iteration in process calculi. In *ICALP'04*, volume 3142 of *Lecture Notes in Computer Science*, pages 307–319. Springer-Verlag, 2004.
5. P. Giambiagi, G. Schneider, and F. D. Valencia. On the expressiveness of infinite behavior and name scoping in process calculi. In *FoSSaCS 2004*, pages 226–240, 2004.
6. H. Huttel and J. Srba. Recursion vs. replication in simple cryptographic protocols. In *SOFSEM'05*, volume 3381 of *LNCS*, pages 175–184. Springer-Verlag, 2005.
7. P. C. Kanellakis and S. A. Smolka. CCS expressions finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.
8. R. Milner. A complete inference system for a class of regular behaviours. *J. Comput. Syst. Sci.*, 28(3):439–466, 1984.
9. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.
10. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
11. M. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.
12. M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of concurrent constraint programming languages. In *PPDP 2002*, pages 156–167. ACM Press, Oct. 2002.
13. C. Palamidessi and F. D. Valencia. Recursion vs replication in process calculi: Expressiveness. *Bulletin of the EATCS*, 87:105–125, 2005.
14. J. Parrow. Trios in concert. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 621–637. MIT Press, 2000.
15. D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.