Mobility in the cc-paradigm *

Cosimo Laneve Ugo Montanari

Dip. Informatica, Università di Pisa, C.so Italia 40, Pisa, Italy

1 Introduction

Milner and his collegues [3] have recently showed that higher order power can be obtained without resorting to communication primitives explicitly transmitting agents (*cf.* [8]) but just allowing *mobility* of the linkages among processes (i.e., the channel structure is allowed to change dynamically). Mobility can be achieved by transmitting the name of a channel to another process. Relaxing the staticity of linkages in CCS results into a new calculus, called *π-calculus*, which is powerful enough to encode λ-calculus [4].

However, this approach is not completely satisfactory. The criticism that is moved against it is the use of channels as a standpoint to encode *functional languages*. More precisely: variables in functional languages, and in λ-calculus in particular, are meant in a mathematical sense, that is as entities whose values do not change during their lifes (*place holders*). On the contrary, to describe the control flow of algorithms implementing functions, the above calculi make use of variables (channels) which are reusable resources, altering, therefore, the functional paradigm.

Recently a class of concurrent languages, falling inside the logic programming paradigm, has been developed: the *concurrent constraint* languages (cc, in the following) [6,7]. In this framework, a computation is the result of the interactions among (concurrent) processes which communicate by placing and checking constraints on shared variables. At any step of the computation, the store (the set of common variables) is represented as a consistent set of constraints, i.e. a finite description providing partial information about the possible final values of the variables. As the computation progresses, the store does not decrease. There are two basic operations: *ask* and *tell*. The *ask* operation takes a constraint c and, by using it, probes the store: it succeeds if the store entails c. The *tell*

*Extended Abstract presented at *Mathematical Foundations of Computer Science 1992*, LNCS 629, pp. 336–345, 1992.

operation takes a constraint c and adds (conjoins) it to the store (when the store is consistent with c). As a consequence, variables can be linked once and for all to their value, that is cc-languages are not affected by the previous criticism against process algebras. However, if cc-languages have higher order power or how much of control is described by them (i.e., a comparison with the descriptive power of process algebras) are still open questions. Some phenomena, indeed, suggest a deep relationship between π-calculus and cc-languages: as an example, mobility of channels is reflected in mobility of variables (giving rise to situations of extrusion and intrusion of the scope).

In this paper we will prove that cc-languages yield higher order power even with a very poor underlying system of constraints (the signature through which constraints are defined has a constant and a concatenation operation). More precisely, by rephrasing Milner's encodings of λ-calculus into π-calculus [4], we provide encodings of lazy and call-by-value λ-calculus into the cc-language.

Actually, we have faced the problem of encoding a subset of π-calculus into a cc-language. There is a naive correspondence between the structural operators of the two languages (prefixing, choice, parallelism), but the correspondence fails as soon as communication mechanisms are considered: on one side there are channels with synchronizing primitives and on the other the store with the asynchronous operations of *ask* and *tell*. Therefore the correspondence holds just when the synchronization mechanism is not essential. As a matter of fact this is the case of Milner's encodings of λ-calculus, due to the property of determinacy.

Two considerations deserve to be done. Firstly our encodings distinguish between channels that are used once, just for book-keeping reasons, and those representing variables in [4], thus yielding a clearer distinction between resources. Indeed, the latters are streams. The second observation concerns π-calculus. The encodings in this paper suggest that it is possible to embed both lazy and call-by-value λ-calculus into a more primitive form of π-calculus admitting asynchronous communications. This has been independently observed by Boudol in [1].

2 The cc-language

Here we present the concurrent programming language we use in the following. The main reference is [6]. The language is based over the notion of (simple) *first order constraint system* which describes the constraints managed by the processes. We use a simple constraint system: the *Herbrand universe* of formulas over some signature. Formally, given a signature Σ and a denumerable set of variables $\mathcal{V}$, a constraint in the *Herbrand constraint system* is a conjunction of equalities in $T_\Sigma(\mathcal{V})$.

In this view, the *store* is simply a set of equalities (thus the empty store will be denoted by $\emptyset$). In order to execute programs of the constraint language, the implementation needs to ensure that *consistency* is preserved, and that closure with respect to *entailment* is performed on constraints. Here constraints will be ranged over by c and stores by σ. In the case of Herbrand systems, the consistency is checked by executing the first order unification algorithm, whilst the entailment by executing a variation of the matching algorithm. Consistency of a store σ and a constraint c will be denoted by $(\exists)(\sigma \wedge c)$; the entailment of c by σ will be written $\sigma \Rightarrow c$ (read σ entails c).

The *agents*, ranged over by A, possibly indexed, are built up by the following syntax:

$$A \quad ::= \quad nil \quad | \quad ask(c) \rightarrow A \quad | \quad tell(c) \rightarrow A \quad | \quad \exists x.\, A \quad | \quad A+A \quad | \quad A|A \quad | \quad P(\tilde{x})$$

where c is a constraint, x is a variable and $\tilde{x}$ is a vector of variables in $\mathcal{V}$.

Intuitively, *nil* is the terminated agent. An agent can perform the action $ask(c)$ in a store σ if $\sigma \Rightarrow c$. The action $tell(c)$ can be executed only if $(\exists)(\sigma \wedge c)$, otherwise the process cannot progress. The tailing agent A in $b \rightarrow A$ is performed as soon as b is executed. The agent $A_1 + A_2$ behaves like A_1 or A_2: the choice is done when one of the two can progress. The agent $A_1|A_2$ models the parallel composition: it behaves interleaving the actions of A_1 and A_2. The agent $\exists x.\, A$ evolves like A except that the variable x is bound in A and must be replaced at run-time by a new variable, local to A, which is 'renamed apart' from the others in the environment. Thus the environment cannot affect A through x and viceversa. $P(\tilde{x})$ means *procedure call* (hence the terms live in an environment rich enough of procedure declarations). In the following, for the sake of brevity, we shall omit the tailings '$\rightarrow$ *nil*' from agents.

The formal definition of the operational semantics exploits a reduction relation '$\longrightarrow$' between elements of $\mathcal{P}$, where $\mathcal{P} = \{\langle A, \sigma\rangle \mid A \text{ is an agent}, \ \sigma \text{ is a store}\}$. Elements of $\mathcal{P}$ will be called *configurations* or *processes* and will be ranged over by Q. A store σ is a pair $c : V_c$ where c is a constraint and V_c is a set of variables such that $\mathbf{var}(c) \subseteq V_c$, i.e. the free variables in the constraint c belong to V_c[1].

Reductions are induced by a finite set of variables which consists of all the free variables in the main program.

Definition 2.1 Reduction, written $\longrightarrow$, is the smallest relation over $\mathcal{P}$ satisfying the following rules:

[1]The function $\mathbf{var}$ is generalized to agents, i.e. $\mathbf{var}(nil) = \emptyset$, $\mathbf{var}(ask(c) \rightarrow A) = \mathbf{var}(tell(c) \rightarrow A) = \mathbf{var}(c) \cup \mathbf{var}(A)$, $\mathbf{var}(A_1 + A_2) = \mathbf{var}(A_1|A_2) = \mathbf{var}(A_1) \cup \mathbf{var}(A_2)$ and $\mathbf{var}(\exists x.\, A) = \mathbf{var}(A) \backslash \{x\}$. Of course, $\mathbf{var}(P(x_1, \cdots, x_n))$ is defined as the set $\{x_1, \cdots, x_n\}$.

$$Ask)\quad \frac{\sigma \Rightarrow c}{V \vdash \langle ask(c) \rightarrow A,\, \sigma\rangle \longrightarrow \langle A,\, \sigma\rangle}$$

$$Tell)\quad \frac{(\exists)(c' : V' \wedge c)}{V \vdash \langle tell(c) \rightarrow A,\, c' : V'\rangle \longrightarrow \langle A,\, (c' \wedge c) : V' \cup \mathbf{var}(c)\rangle}$$

$$Sum_l)\quad \frac{V \vdash \langle A_1,\, \sigma\rangle \longrightarrow \langle A_1',\, \sigma'\rangle}{V \vdash \langle A_1 + A_2,\, \sigma\rangle \longrightarrow \langle A_1',\, \sigma'\rangle} \qquad Sum_r)\quad \frac{V \vdash \langle A_2,\, \sigma\rangle \longrightarrow \langle A_2',\, \sigma'\rangle}{V \vdash \langle A_1 + A_2,\, \sigma\rangle \longrightarrow \langle A_2',\, \sigma'\rangle}$$

$$Par_l)\quad \frac{V \vdash \langle A_1,\, \sigma\rangle \longrightarrow \langle A_1',\, \sigma'\rangle}{V \vdash \langle A_1 | A_2,\, \sigma\rangle \longrightarrow \langle A_1' | A_2,\, \sigma'\rangle} \qquad Par_r)\quad \frac{V \vdash \langle A_2,\, \sigma\rangle \longrightarrow \langle A_2',\, \sigma'\rangle}{V \vdash \langle A_1 | A_2,\, \sigma\rangle \longrightarrow \langle A_1 | A_2',\, \sigma'\rangle}$$

$$Res)\quad \frac{V \vdash \langle A\{^y/_x\},\, c : V' \cup \{y\}\rangle \longrightarrow Q \qquad y \notin V \cup V'}{V \vdash \langle \exists x.\, A,\, c : V'\rangle \longrightarrow Q}$$

■

Operational semantics of the procedure call is modelled by means of the semantics of the body, that is, if $P(\tilde{x}) = A(\tilde{x})$ then

$$\frac{V \vdash \langle A(\tilde{s}),\, \sigma\rangle \longrightarrow Q}{V \vdash \langle P(\tilde{s}),\, \sigma\rangle \longrightarrow Q}$$

Before going on, some remarks about the operational semantics are needed. Perhaps the most cumbersome rule is *Res*. Let us explain briefly it. Let V be the set of global variables in the environment in which an existentially quantified process $\langle \exists x.\, A,\, c : V'\rangle$ lives. The variable x is a new local variable for the above process, so it must be renamed apart from the global variables V and from the local variables V' in the store. Formally we have to choose a variable y which does not belong to $V \cup V'$, and this is (a part of) what is stated in the premise of *Res*. The rest of the premise checks for the existence of some reduction of the process $\langle A\{^y/_x\},\, c : V' \cup \{y\}\rangle$ where the new variable y is substituted for every free occurrence of x in A and the local store is augmented by y. Let Q be the reduced process. Under these hypotheses, the process $\langle \exists x.\, A,\, c : V'\rangle$ can transit into Q.

We assume familiarity of the reader with the notion of *syntactic substitution* (notation $p\{^{p'}/_x\}$ meaning that the term p' is substituted for the variable x into p) possible extended to n-tuples or sets and with the notions of free and bound variables.

We remark that the above rules only model *internal* moves of concurrent constraint languages (*cf.* [6]). Furthermore we have no process representing successful termination (again, *cf.* [6]) because we are interested just in the on-going behaviour, rather than in the final value of the store.

(**Our Herbrand universe**) The Herbrand universe we consider has the signature consisting of a constant symbol 0 and a binary *pairing* $\langle\ ,\ \rangle$. Elements of the term algebra will be ranged over by m, k. We will call 1 the pair $\langle 0,0\rangle$ and sometimes we will write $x.y$ instead of $\langle x,\, y\rangle$.

The example below will show a *mobility* of variables: it has been our trigger for embarking in this research. Variables play in cc-languages a role similar to that of channels in π-calculus. The reader familiar with π-calculus will find strict analogies.

Example 2.2 Consider the process $\langle A|B,\, \emptyset\rangle$, where

$$\begin{array}{rcl} A & = & \exists z.\, \mathit{tell}(y = \langle 0,\, z\rangle) \rightarrow A' \\ B & = & \exists y'.\, \exists w.\, \mathit{ask}(y = \langle y',\, w\rangle) \rightarrow \mathit{tell}(y = \langle y',\, w\rangle)) \rightarrow B' \end{array}$$

Informally the agent A constraints a local variable z to a global one y whilst the agent B waits for this constraint in order to make further computations. Here we have *implemented* an "imperative style" variable as a *stream* recording its values. Thus, for example, $y = \langle 1,\, \langle 0,\, y'\rangle\rangle$ is a variable whose first value was 1, the actual value is 0 and it can be modified by constraining y'. Furthermore notice the nesting of the *ask* and *tell* operations in the process B. The *ask* operation is used to *wait* till a constraint holds, the *tell* performs the pattern-matching (hence the linked variables can be referred and modified in the following of the computation). We emphasize the role of the *ask* construct to accomplish the "communication" between A and B over the variable y. Indeed if we assume simply $B = \exists y'.\, \exists w.\, \mathit{tell}(y = \langle y',\, w\rangle)) \rightarrow B'$, then B may progress before A. Eventually it may happen that the operation $\mathit{tell}(y = \langle 0,\, z\rangle)$ in A is forbidden by the firing of a *tell* in B' which is inconsistent with $y = \langle 0,\, z\rangle$.

After three reductions, any change of w in B', even if it is local to B, will, as a side effect, modify z, too. That is, a process (B) which is not in the scope of a variable z intrudes into the scope as the computation progresses (*scope intrusion*). The possible presence of a private variable z in B does not matter because the operation of existential quantification $\exists w. -$ avoids any clash. Notice that, in this case, the behaviour of agents in π-calculus is slightly different. The symmetry of π-calculus communications make the scope of the above variable z be augmented only by the process B after the communication. This does not happen in the cc-paradigm, because, after the *tell*-operation, any process who knows y will be able to access z.

A dual phenomenon is emphasized by the process $\langle (\exists x.(A|B))|C,\, \emptyset\rangle$, where

$$\begin{array}{rcl} A & = & \mathit{tell}(y = \langle 0,\, x\rangle) \rightarrow A' \\ B & = & \exists u.\, \exists w.\, \mathit{ask}(x = \langle u,\, w\rangle) \rightarrow \mathit{tell}(x = \langle u,\, w\rangle) \rightarrow B' \\ C & = & \exists y'.\, \exists z.\, \mathit{ask}(y = \langle y',\, z\rangle) \rightarrow \mathit{tell}(y = \langle y',\, z\rangle) \rightarrow \mathit{tell}(z = \langle a,\, b\rangle) \rightarrow C' \end{array}$$

and suppose that $x \notin \mathbf{var}(A')$. In this case A delegates C to accomplish the task of modifying x and extrudes itself from the scope of x (*scope extrusion*). ■

Finally, agents will be identified if they are *structurally equivalent*, according to the following definition.

Definition 2.3 The *structural congruence* $\equiv$ is the smallest congruence over processes satisfying the set of equations:

$\langle A, \sigma\rangle \equiv \langle A, \sigma'\rangle$, if $\sigma \Rightarrow \sigma'$ and $\sigma' \Rightarrow \sigma$;

$\langle A \star nil, \sigma\rangle \equiv \langle A, \sigma\rangle$, $\langle A_1 \star A_2, \sigma\rangle \equiv \langle A_2 \star A_1, \sigma\rangle$, $\langle A_1 \star (A_2 \star A_3), \sigma\rangle \equiv \langle (A_1 \star A_2) \star A_3, \sigma\rangle$

$\langle \exists x.\, nil, \sigma\rangle \equiv \langle nil, \sigma\rangle$ $\langle \exists x.\, \exists y.\, A, \sigma\rangle \equiv \langle \exists y.\, \exists x.\, A, \sigma\rangle$ $\langle \exists x.\, (A_1|A_2), \sigma\rangle \equiv \langle A_1|\exists x.\, A_2, \sigma\rangle$, *if* $x \notin \mathbf{var}(A_1)$

$\langle P(\tilde{x}), \sigma\rangle \equiv \langle A(\tilde{x}), \sigma\rangle$, if $P(\tilde{x}) \stackrel{def}{=} A(\tilde{x})$.

where $\star \in \{+, |\}$. ∎

3 The encode of lazy λ-calculus

Terms $\mathcal{L}$ of the λ-calculus, ranged by M, N are inductively defined by the following rules:

$$x \quad | \quad \lambda x.M \quad | \quad MN$$

where x ranges over an enumerable set of variables $\mathcal{V}$. The operation $\lambda x. -$ is called *abstraction* and the operation $--$ is called *application*. Application is *left associative*. The function $fv(M)$ gives the set of free variables in M.

Definition 3.1 The *lazy reduction* relation $\rightarrow$ over $\mathcal{L}$ is the smallest relation which satisfies the following rules:

$$\beta)\ (\lambda x.M)N \rightarrow M\{^N/_x\} \qquad\qquad ap)\ \frac{M \rightarrow M'}{MN \rightarrow M'N}$$ ∎

A nice property of $\rightarrow$ is that it is *deterministic* (or r-determinate), that is, for every M there is at most one M' such that $M \rightarrow M'$. Let $\rightarrow^*$ be the transitive and refexive closure of $\rightarrow$. If $M \rightarrow^* M'$ and $M' \not\rightarrow$, that is M *converges* to M' (notation $M \downarrow M'$), then M' can be an *abstraction* $\lambda x.N$ or of the form $xN_1 \ldots N_n$. When M is a *closed term* (notation $M \in \mathcal{L}^0$) then M' is closed, too, and is an abstraction.

By exploiting convergence, it is possible to define a preorder $\preceq_\lambda$ over $\mathcal{L}^0$.

Definition 3.2 Let L, $M \in \mathcal{L}^0$. Then $L \preceq_\lambda M$ if for all sequences $\tilde{N}$ in $\mathcal{L}^0$

$$L\tilde{N}\downarrow \quad implies \quad M\tilde{N}\downarrow$$

Moreover, if L, $M \in \mathcal{L}$, with free variables $\tilde{x}$, we define $L \preceq_\lambda M$ to mean that $L\{^{\tilde{N}}/_{\tilde{x}}\} \preceq_\lambda M\{^{\tilde{N}}/_{\tilde{x}}\}$ for all $\tilde{N}$ in $\mathcal{L}^0$. Let $\simeq_\lambda$ be $\preceq_\lambda \cap \succeq_\lambda$. ∎

The preorder $\preceq_\lambda$ is a precongruence. In particular, the following statement holds:

$$L \preceq_\lambda M \quad \textit{iff} \quad \textit{for every closed context } \mathcal{C}[-] : \mathcal{C}[L]\downarrow \Rightarrow \mathcal{C}[M]\downarrow .$$

We will extend the definition of r-determinate and the one of *convergence* given over λ-terms to (r-determinate) cc-agents.

Let us turn to the encoding of $\mathcal{L}$ into $\mathcal{A}$. The application operator MN is interpreted as parallel composition between two processes encoding M and N respectively; the β-reduction is seen as a kind of *remote procedure call*, i.e. the process encoding the function $\lambda x.\, M'$, as soon as a variable x is encountered in the computation of the body M', sends a message carrying an environment, say u, to the process encoding the argument N. This message makes the process encoding N fire by asking arguments to the environment u. However, before computing N, in order to face possible future calls of N, a previous replication is needed. Here, to manage communications between processes, *streams* are substituted for channels. A stream is a list of pairs: the first component of an element stores the identifier of the process which has sent the message and the message exchanged through the channel it represents at a given moment, the second component is a pointer to the following element. The pointer of the last pair of the list will be called *end of the stream.* Each $M \in \mathcal{L}$ is encoded as $[\![M,\, m]\!]$, a map from $\mathcal{V}$ to $\mathcal{P}$. The variable u in $[\![M,\, m]\!]u$ will allow M to receive arguments; the *string* m has to be considered as a sort of process identifier, as showed by Proposition 3.4 below. The presence of process identifiers is needed in order to guarantee that when a process send a message *msg* through a stream σ, the stream is properly lengthened, even if another message *msg* has been previously exchanged through σ.

In the following we will use the shortening below:

$$\mathbf{ASK}_X(x = f(X)) \;=\; \exists X.\, \mathit{ask}(x = f(X)) \rightarrow \mathit{tell}(x = f(X))$$

where $X = \{x_1, \cdots, x_n\}$ and f is a function over X. The notation $\exists X$ stays for $\exists x_1. \cdots \exists x_n$. The following set of procedure declarations is used:

$$\begin{array}{lcl} \mathit{Arg}_N(x,\, m) & \stackrel{def}{=} & \mathbf{ASK}_{\{m',\, u',\, x'\}}(x = \langle\langle m',\, u'\rangle,\, x'\rangle) \rightarrow ([\![N,\, 0.m]\!]u' \mid \mathit{Arg}_N(x',\, 1.m)) \\ \mathit{Test}(x,\, u,\, m) & \stackrel{def}{=} & \mathbf{ASK}_{\{y,\, x'\}}(x = \langle y,\, x'\rangle) \rightarrow \mathit{Test}(x',\, u,\, m) \\ & & + \\ & & \exists x'.\, \mathit{tell}(x = \langle\langle m,\, u\rangle,\, x'\rangle) \end{array}$$

The processes $\mathit{Arg}_N(x,\, m)$ will wait for the binding (a message) of the variable x (statement *ask* - unfold the **ASK**!) and react by evaluating the term N into an *environment* u. Furthermore, the procedure replicates itself (over a different argument, of course) to face possible future calls. The processes $Test(x, u, m)$ test the *stream* x in order to select the end of the stream; when the end of the stream is selected they add a message to the stream.

$$
\begin{array}{rcl}
[\![\lambda x.\, M,\, m]\!]u & \stackrel{def}{=} & \exists v.\, \exists x.\, tell(u = \langle x,\, v\rangle) \rightarrow \mathbf{ASK}_{\{w\}}(v = \langle 0,\, w\rangle) \rightarrow [\![M,\, m]\!]w \\
[\![x,\, m]\!]u & \stackrel{def}{=} & Test(x,\, u,\, m) \\
[\![MN,\, m]\!]u & \stackrel{def}{=} & \exists v.\, ([\![M,\, 0.m]\!]v \mid (\exists x.\, \exists y.\, ask(v = \langle x,\, y\rangle) \rightarrow \\
 & & \qquad tell(v = \langle x,\, \langle 0,\, u\rangle\rangle) \rightarrow Arg_N(x,\, 1.m)))
\end{array}
$$

where, in the third row, $x \notin fv(MN)$. Below we will use the following shortenings:

$$
\begin{array}{lcl}
A_N(u, v, m) & = & \exists x.\, \exists y.\, ask(v = \langle x,\, y\rangle) \rightarrow tell(v = \langle x,\, \langle 0,\, u\rangle\rangle) \rightarrow Arg_N(x,\, 1.m) \\
A^{\star}_N(v, m) & = & \mathbf{ASK}_{\{w\}}(v = \langle 0,\, w\rangle) \rightarrow [\![M,\, m]\!]w
\end{array}
$$

The reader that knows the corresponding encoding in π-calculus will find some simplifications which consist in considering the channels used just once (those named u, v) as simple variables. An example will help in the understanding of the encoding.

Example 3.3 Consider the λ-term $L = [(\lambda x.\, x)(\lambda y.\, y)]z$. Its lazy computation is:

$$[(\lambda x.\, x)(\lambda y.\, y)]z \;\rightarrow\; (\lambda y.\, y)z \;\rightarrow\; z$$

Below is the run of the cc-process $[\![L,\, 0]\!]u$ into an empty store. Notice that $\mathbf{var}([\![L,\, 0]\!]u) = \{u,\, z\}$, thus reductions are induced by this set (which will be omitted).

$$
\begin{array}{rcll}
\langle [\![L,\, 0]\!]u,\, \emptyset\rangle & \equiv & \langle \exists r.\, (\exists s.\, ([\![\lambda x.\, x,\, 0^3]\!]s \mid A_{\lambda y.\, y}(r,\, s,\, 1.0^2)) \mid A_z(u, r, 1.0)),\, \emptyset\rangle & \\
 & \longrightarrow^5 & \langle [\![x,\, 0^3]\!]r \mid\; Arg_{\lambda y.\, y}(x, 1.0^2) \mid A_z(u, r, 1.0),\, \sigma\rangle & \\
 & \longrightarrow^3 & \langle [\![\lambda y.\, y,\, 0.1.0^2]\!]r \mid Arg_{\lambda y.\, y}(x',\, 1^2.0^2) \mid A_z(u, r, 1.0),\, \sigma'\rangle & (1) \\
 & \sim & \langle [\![\lambda y.\, y,\, 0.1.0^2]\!]r \mid A_z(u, r, 1.0),\, \sigma'\rangle & (2) \\
 & \longrightarrow^5 & \langle [\![y,\, 0.1.0^2]\!]u \mid Arg_z(y,\, 1.0),\, \sigma''\rangle & \\
 & \longrightarrow^3 & \langle [\![z,\, 0.1.0]\!]u \mid Arg_z(y',\, 1^2.0),\, \sigma'''\rangle & \\
 & \sim & \langle [\![z,\, 0.1.0]\!]u,\, \sigma'''\rangle & (3)
\end{array}
$$

where $X = \{r, s\}$, $X' = X \cup \{x'\}$, $Y = X' \cup \{y\}$, $\sigma = (s = \langle x,\, \langle 0,\, r\rangle\rangle) : X$, $\sigma' = \sigma \wedge (x = \langle\langle 0^3,\, r\rangle,\, x'\rangle) : X'$, $\sigma'' = \sigma' \wedge r = \langle y,\, \langle 0,\, u\rangle\rangle : Y$ and $\sigma''' = \sigma'' \wedge y = \langle\langle 0.1.0^2,\, u\rangle,\, y'\rangle : Y \cup \{y'\}$.

Notice that we lose any correspondence between the *number* of reductions in the λ-calculus and those in the cc-language. Two reasons must be charged of this: the first is that substitutions, which are primitive in the λ-calculus, are here computed by interactions (and, so, by performing reductions). The second is due to the presence of streams and hence their scan in order to read or write at their end.

Let us explain the above computation. In (1) we have avoided, for simplicity sake, the introduction of aliases. In the following, as far as it is possible, aliases will be always avoided and, similarly, for the sake of readability, we will forget about local variables which will never be used in the computation. In (2) we have simplified the process resulting from the previous reduction. The simplification consists in garbage collecting the deadlocked processes (in this case $Arg_{\lambda y.\,y}(x',\, 1^2.0^2)$) and it is formally expressed by the notion of *bisimulation*[2] (see [6]). We remark that the process in (2) resembles $\langle [\![(\lambda y.\,y)w,\, 0]\!]u,\, \sigma' \rangle$ except for a renaming of process identifiers. In a sense, the above last configuration corresponds to the λ-term $(\lambda y.\,y)z$ reached after one reduction. That is, we have simulated the first β-reduction through eight reductions of the cc-process encoding L. A similar observation concerns the process in (3). ■

Although any correspondence between the number of reductions of $[\![M,\, 0]\!]u$ in the cc-language and those of M in the λ-calculus is lost, we will show in a moment that we can relate the two reduction in a very close way. The relationship wich follows exploits a property of our encodings, due to the role played by process identifiers: every *Test* operation always lengthen the involved stream. For the sake of clarity, we will allow ourselves a bit of informality in stating Proposition 3.4. Here process identifiers are considered different when their flattening to sequences of 0 and 1 are different.

Proposition 3.4 Let $L \in \mathcal{L}$, every sequential process generated during the computation of $\langle [\![L,\, 0]\!]u, \emptyset \rangle$ has a process identifier which is distinct from every other. ■

The correspondence consists in defining, for every $L \in \mathcal{L}^0$, a relation $\Re_L$ between the set of terms reachable from L and (a subset of) the agents reachable from $\langle [\![L,\, 0]\!]u,\, \bot \rangle$. Every closed λ-term L will be paired with an agent *equal* to $\langle [\![L,\, 0]\!]u,\, \sigma \rangle$ *except for a renaming of process identifiers.* Pairings are modelled by defining three closure operators $\Im_1$, $\Im_2$ and $\Im_3$, each of them taking care of different situations.

[2]Here, since transitions are not labelled, two agents $\langle A_1,\, \sigma_1 \rangle$ and $\langle A_2,\, \sigma_2 \rangle$ are (strong) *bisimilar* if, whenever $\langle A_1,\, \sigma_1 \rangle \longrightarrow \langle A_1',\, \sigma_1' \rangle$, there exists an agent $\langle A_2',\, \sigma_2' \rangle$ such that $\langle A_2,\, \sigma_2 \rangle \longrightarrow \langle A_2',\, \sigma_2' \rangle$ and viceversa.

Below $M\{^{\tilde{N}}/_{\tilde{x}}\}$ will stay for $(\cdots(M\{^{N_1}/_{x_1}\})\cdots)\{^{N_k}/_{x_k}\}$ and $Arg_{\tilde{N}}(\tilde{x},\, \tilde{m})$ will stand for $Arg_{N_1}(x_1,\, m_1) \mid \ldots \mid \mid Arg_{N_k}(x_k,\, m_k)$ (the natural k will be clear from the context).

Definition 3.5 Let $L \in \mathcal{L}^0$ and let $k \geq 0$; $M, N_1, \cdots, N_k \in \mathcal{L}$; $\tilde{x} = \{x_1, \cdots, x_k\} \subseteq \mathcal{V}$ all distinct, with $fv(M) \subseteq \tilde{x}$, $fv(N_i) \subseteq \{x_{i+1}, \cdots, x_k\}$ $(1 \leq i \leq k)$, such that $L = M\{^{\tilde{N}}/_{\tilde{x}}\}$. Let $\mathcal{L}^0_L$ be the set $\{M' \mid L \rightarrow^* M'\}$ and $\Re_L \subseteq \mathcal{L}^0_L \times \mathcal{P}$ be the minimal relation such that $(L,\, \langle [\![L,\, 0]\!]u,\, \perp\rangle) \in \Re_L$ and $\Re_L$ is closed by the operations $\Im_1$, $\Im_2$ and $\Im_3$ in Definitions 3.6, 3.7 and 3.8 below. ■

The first closure operator $\Im_1$ takes care of pairing λ-terms which results from the execution of the β-reduction in terms as $((\lambda x.\, M)M_1 \cdots M_n)\{^{\tilde{N}}/_{\tilde{x}}\}$.

Definition 3.6 Let $\Re \subseteq \mathcal{L}^0_L \times \mathcal{P}$ and let $(M,\, \langle A,\, c\, :\, X\rangle) \in \Re$ where $M = ((\lambda x.\, M')M_1 \cdots M_n)\{^{\tilde{N}}/_{\tilde{x}}\}$ $(n \geq 1)$, $fn(((\lambda x.\, M')M_1 \cdots M_n)) \subseteq \tilde{x}$ and $\tilde{N}, \tilde{x}$ are as in Definition 3.5. Furthermore, let

$$A \;=\; ([\![\lambda x.\, M',\, m_0]\!]v_n \mid A_{M_1}(v_{n-1},\, v_n,\, m_1)) \mid \cdots \mid A_{M_n}(u,\, v_0,\, m_n)) \mid Arg_{\tilde{N}}(\tilde{x}',\, \tilde{m}')$$

where $\tilde{x}' = \{x'_1,\, \ldots\, ,\, x'_k\}$ and every x'_i is the end of the stream x_i in c [3]. Then

$$(\; ((M'M_2 \cdots M_n)\{^{M_1}/_x\})\{^{\tilde{N}}/_{\tilde{x}}\},\, \langle A',\, c' : X'\rangle \;) \;\in\; \Im_1(\Re)$$

where $X' = X \cup \{v_n,\, x\}$, $c' = c \wedge v_n = \langle x,\, \langle 0,\, v_{n-1}\rangle\rangle$ and

$$\begin{aligned} A' \;\; = \;\; & [\![M',\, m_0]\!]v_{n-1} \mid A_{M_2}(v_{n-2},\, v_{n-1},\, m_1) \mid \cdots \mid A_{M_n}(u,\, v_0,\, m_n) \mid Arg_{M_1}(x,\, m_1) \\ & \mid Arg_{\tilde{N}}(\tilde{x}',\, \tilde{m}') \end{aligned}$$

■

The closure operator $\Im_2$ manages those pairs having $(xM_1 \cdots M_n)\{^{\tilde{N}}/_{\tilde{x}}\}$ as λ-terms. Here, by definition, x must be linked to a term N (notice that $(xM_1 \cdots M_n)\{^{\tilde{N}}/_{\tilde{x}}\} = (NM_1 \cdots M_n)\{^{\tilde{N}}/_{\tilde{x}}\}$).

Definition 3.7 Let $\Re \subseteq \mathcal{L}^0_L \times \mathcal{P}$ and let $(M,\, \langle A,\, c\, :\, X\rangle) \in \Re$ where $M = (xM_1 \cdots M_n)\{^{\tilde{N}}/_{\tilde{x}}\}$ $(n \geq 0)$, $fn((xM_1 \cdots M_n)) \subseteq \tilde{x}$ and $\tilde{N}, \tilde{x}$ are as in Definition 3.5. Let N be the term linked to x in the substitution $\{^{\tilde{N}}/_{\tilde{x}}\}$. Furthermore, let

$$A \;=\; [\![x,\, m_0]\!]v_n \mid A_{M_1}(v_{n-1},\, v_n,\, m_1)) \mid \cdots \mid A_{M_n}(u,\, v_0,\, m_n)) \mid Arg_{\tilde{N}}(\tilde{x}',\, \tilde{m}')$$

[3] Notice that the agent p is equal to $[\![M,\, 0]\!]u$ except for a *renaming* of the process identifiers and for the variables in $\tilde{x}'$ which are the ends of the streams $\tilde{x}$ in c. This renaming and those we use in this paper are such that no *clash* of process identifiers will happen in the following of the computation. Such property follows, in this case, by Proposition 3.4.

where every $x'_i \in \tilde{x}'$ is the end of the stream x_i. Then

$$(\ (NM_1 \cdots M_n)\{^{\tilde{N}}/_{\tilde{x}}\},\ \langle A',\ c' : X' \rangle\ ) \ \in\ \Im_2(\Re)$$

where $X' = X \cup \{x''\}$, $c' = c \wedge (x' = \langle\langle m_0,\ u\rangle,\ x''\rangle)$ and

$$\begin{aligned} A' \quad = \quad & [\![N,\ 0.m'_i]\!]v_n \mid A_{M_1}(v_{n-1},\ v_n,\ m_1)) \mid \cdots \mid A_{M_n}(u,\ v_0,\ m_n)) \mid Arg_{N_1}(x'_1,\ m'_1) \mid \cdots \\ & \cdots \mid Arg_N(x'',\ 1.m'_i) \mid \cdots \mid Arg_{N_k}(x'_k,\ m'_k) \end{aligned}$$

■

The last closure operator $\Im_3$ takes care of terminal situations. As soon as a term $(\lambda x.\, M)\{^{\tilde{N}}/_{\tilde{x}}\}$ is reached, whilst the λ-term has reached a final state, the corresponding cc-agent can perform a further transition (see the encoding). The operator $\Im_3$ pairs the above λ-term with the agent reached after this transition.

Definition 3.8 Let $\Re \subseteq \mathcal{L}^0_L \times \mathcal{P}$ and let $(M,\ \langle A,\ c\ :\ X\rangle) \in \Re$ where $M = (\lambda x.\, M')\{^{\tilde{N}}/_{\tilde{x}}\}$, $fn(\lambda x.\, M') \subseteq \tilde{x}$ and $\tilde{N}, \tilde{x}$ are as in Definition 3.5. Let $A = [\![\lambda x.\, M',\ m]\!]u \mid Arg_{\tilde{N}}(\tilde{x}',\ \tilde{m}')$ where every variable x'_i in $\tilde{x}'$ is the end of the stream x_i in φ. Then

$$(\ (\lambda x.\, M')\{^{\tilde{N}}/_{\tilde{x}}\},\ \langle A',\ c' : X' \rangle\ ) \ \in\ \Im_3(\Re)$$

where $X' = X \cup \{x, v\}$, $c' = c \wedge (u = \langle x,\ v\rangle)$ and $A' = A^\star_{M'}(v,\ m) \mid Arg_{\tilde{N}}(\tilde{x}',\ \tilde{m}')$.

■

The notion of convergence defined before over λ-terms can be generalized to cc-agents: the agent $\langle A_1,\ \sigma_1\rangle$ *converges* to $\langle A_2,\ \sigma_2\rangle$ (notation $\langle A_1,\ \sigma_1\rangle \downarrow \langle A_2,\ \sigma_2\rangle$) if $\langle A_1,\ \sigma_1\rangle \longrightarrow^* \langle A_2,\ \sigma_2\rangle$ and $\langle A_2,\ \sigma_2\rangle \not\longrightarrow$.

Theorem 3.9 For all $L \in \mathcal{L}^0$, $\langle [\![L,\ 0]\!]u,\ \emptyset\rangle$ is r-determinate and one of the following conditions hold:

i) $L \downarrow L'$ and $\langle [\![L,\ 0]\!]u,\ \emptyset\rangle \downarrow \langle A,\ \sigma\rangle$, where $L' \equiv (\lambda x.\, M)[^{\tilde{N}}/_{\tilde{x}}]$, $p \equiv A^\star_M(v,\ m) \mid Arg_{\tilde{N}}(\tilde{x}',\ \tilde{m})$ and $\sigma \Rightarrow u = \langle x,\ v\rangle$;

ii) $L \uparrow$ and $\langle [\![L,\ 0]\!]u,\ \emptyset\rangle \uparrow$. ■

This theorem is a bit more involved with respect to the one in [4], due to the presence of the store. Anyway the proof technique is the same.

The last part of this section is devoted to the characterizztion of $\simeq_\lambda$ in terms of some equivalence over cc-agents. Recall that, according our encode, β-reductions are simulated through finite sequences of (unobservable) reductions of $[\![L,\ 0]\!]u$. Hence, the equivalences should be insensitive to finite internal moves but sensitive to infinite reductions, since $\simeq_\lambda$ distinguishes between *convergence* and *divergence*. Let us define

$$A_1 \sqsubseteq A_2 \quad \stackrel{def}{=} \quad \begin{array}{c} \langle A_1,\, \emptyset\rangle \longrightarrow^* \langle A_1',\, \sigma_1,\rangle \not\longrightarrow \\ \Downarrow \\ \exists \langle A_2',\, \sigma_2\rangle.\, \langle A_2,\, \emptyset\rangle \longrightarrow^* \langle A_2',\, \sigma_2\rangle \not\longrightarrow \end{array}$$

This preorder has a weak distinguishing power since it equates all agents which start with the empty store and have a terminating trace. Despite of this, the naive candidate congruence $\simeq_{\mathsf{cc}} = \preceq_{\mathsf{cc}} \cap \succeq_{\mathsf{cc}}$, where $\preceq_{\mathsf{cc}}$ is

$$p \preceq_{\mathsf{cc}} q \quad \textit{iff} \quad \textit{for every context } \mathcal{C} \,:\, \mathcal{C}[p] \sqsubseteq \mathcal{C}[q],$$

is stronger than $\preceq_\lambda$, how the proposition below states.

Proposition 3.10 Let $L,\, M \in \mathcal{L}^0$ then $[\![L,\, 0]\!]u \preceq_{\mathsf{cc}} [\![M,\, 0]\!]u \;\Rightarrow\; L \preceq_\lambda M$. ■

We show by a counterexample that the converse is false. To this aim (by rephrasing the proof of Theorem 6.3 in [4]) we exploit the existence of combinators definable in cc-language but not in lazy λ-calculus. Consider the two terms:

$$L_1 = x(\lambda y.\, x\Xi\Omega y)\Xi \quad \textit{and} \quad L_2 = x(x\Xi\Omega)\Xi$$

where $\Xi = \lambda z.\, (\lambda x.\, (\lambda y.\, \lambda z.\, y)(xx))(\lambda x.\, (\lambda y.\, \lambda z.\, y)(xx))$ (note that $\Xi\tilde{N}\downarrow$ for every $\tilde{N}$) and $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ (Ω is a divergent term, that is $\Omega\uparrow$). It results that $L_1 \preceq_\lambda L_2$. Anyway, consider the *convergence testing* combinator $\mathbf{T}$ defined below:

$$\mathbf{T}M \downarrow \lambda x.\, x \quad \textit{if} \quad M\downarrow \qquad\qquad \mathbf{T}M\uparrow \quad \textit{otherwise}$$

This operator is not internally definable in lazy λ-calculus (see [5]) and moreover $L_1\{^{\mathbf{T}}/_x\}\downarrow$ and $L_2\{^{\mathbf{T}}/_x\}\uparrow$. On the other hand, $\mathbf{T}$ can be defined in the cc-language as follows:

$$\begin{array}{lll} \mathbf{T}(u,\, m) & \stackrel{def}{=} & \exists x_1.\, \exists x_2.\, \textit{tell}(u = \langle x_1,\, x_2\rangle) \to \exists x_3.\, \textit{ask}(x_2 = \langle 0,\, x_3\rangle) \to \\ & & \qquad \exists x_1'.\, \textit{tell}(x_1 = \langle\langle m,\, x_3\rangle,\, x_1'\rangle)) \to \\ & & \qquad \exists x_3'.\, \exists x_3''.\, \textit{ask}(x_3 = \langle x_3',\, x_3''\rangle) \to \exists x_4.\, [\![\lambda x.\, x,\, 0.m]\!]x_4 \end{array}$$

Hence, considering the context $\mathcal{C} = \exists v.\, ([\![-,\, 0]\!]v \mid \textit{Arg}_{\mathbf{T}}(x,\, 1))$ where

$$\textit{Arg}_{\mathbf{T}}(x,\, m) \;\stackrel{def}{=}\; \mathbf{ASK}_{\{m',u',x'\}}(x = \langle\langle m',\, u'\rangle,\, x'\rangle) \to (\mathbf{T}(u',\, 0.m) \mid \textit{Arg}_{\mathbf{T}}(x',\, 1.m))$$

it results that $\mathcal{C}(L_1)\downarrow$, while $\mathcal{C}(L_2)\uparrow$.

The question of defining a congruence over cc-agents which exactly characterizes $\simeq_\lambda$ is left open.

4 The call-by-value λ-calculus

Due to the limitations in space, we briefly show the encoding of call-by-value λ-calculus without providing any proof (we refer the reader to the full paper). Contrary to the lazy reduction strategy, call-by-value strategy shows up some parallelism.

Let $\mathcal{W}$, ranged over by V, be the set of *values*, i.e. variables in $\mathcal{V}$ or abstractions $\lambda x.\, M$.

Definition 4.1 The *call-by-value* reduction relation $\rightarrow_v$ over $\mathcal{L}$ is the smallest relation which satisfies the following rules

$$\beta)\quad (\lambda x.M)V \ \rightarrow_v\ M\{{}^{V}/_{x}\} \qquad ap_l)\ \ \frac{M \ \rightarrow_v\ M'}{MN \ \rightarrow_v\ M'N} \qquad ap_r)\ \ \frac{N \ \rightarrow_v\ N'}{MN \ \rightarrow_v\ MN'} \ \blacksquare$$

Note that the reduction strategy $\rightarrow_v$ is no longer determinate, even if it still enjoy the Church-Rosser property. Similarly to Section 3, we can define convergence $\Downarrow_v$ and the derived notion of applicative simulation $\preceq_\lambda^v$ which is still a precongruence even if it is uncomparable with $\preceq_\lambda$ (see [4]).

The call-by-value translation function into the cc-language, named $[\![-]\!]_v$, has the same range of $[\![-]\!]$. It differs from the lazy reduction just for some features to allow parallelism between reductions of the arguments of an application. This is the main purpose of the procedure **ap** below.

$$\begin{aligned}\mathbf{ap}(p,\, q,\, r,\, m) \ \stackrel{def}{=}\ & \exists y.\, ask(q = \langle y,\, 0\rangle) \rightarrow \exists y'.\, \exists v.\, tell(q = \langle y,\, 0\rangle \ \wedge\ y = \langle\langle m,\, v\rangle,\, y'\rangle) \rightarrow \\ & \exists z.\, ask(r = \langle z,\, 0\rangle) \rightarrow tell(r = \langle z,\, 0\rangle \ \wedge\ v = \langle z,\, p\rangle)\end{aligned}$$

Moreover, a set of procedures *Arg*, now ranging over values, will be used; their schema is:

$$\begin{aligned} Arg_{\lambda x.M}(y,\, m) \ &\stackrel{def}{=}\ \mathbf{ASK}_{\{n,z,y',p\}}(y = \langle\langle n,\, \langle z,\, p\rangle\rangle,\, y'\rangle) \rightarrow ([\![M\{{}^{z}/_{x}\},\, 0.m]\!]_v p \mid Arg_{\lambda x.M}(y',\, 1.m)) \\ Arg_x(y,\, m) \ &\stackrel{def}{=}\ \mathbf{ASK}_{\{n,z,y',p\}}(y = \langle\langle n,\, \langle z,\, p\rangle\rangle,\, y'\rangle) \rightarrow (Test(x,\, z,\, p,\, 0.m) \mid Arg_x(y',\, 1.m)) \end{aligned}$$

where $z \notin fv(M)$ and the procedure *Test* is now defined as:

$$\begin{aligned} Test(x,\, z,\, p,\, m) \ \stackrel{def}{=}\ & \mathbf{ASK}_{v,\, x'}(x = \langle v,\, x'\rangle) \rightarrow Test(x',\, z,\, p,\, m) \\ & + \\ & \exists x'.\, tell(x = \langle\langle m,\, \langle z,\, p\rangle,\, x'\rangle) \end{aligned}$$

Finally, the call-by-value encoding function $[\![-]\!]_v$ is defined by structural induction by the rules:

$$[\![V,\, m]\!]_v u \quad \stackrel{def}{=} \quad \exists y.\, tell(u = \langle y,\, 0\rangle) \rightarrow Arg_V(y,\, m)$$

$$[\![MN,\, m]\!]_v u \quad \stackrel{def}{=} \quad \exists r.\, \exists s.\, (\, \mathbf{ap}(u,\, r,\, s,\, 0.0.m) \mid [\![M,\, 0.1.m]\!]_v r \mid [\![N,\, 1.0.m]\!]_v s\,)$$

Again, there is no hope to maintain the correspondence between the number of reductions in the two calculi. Anyway, by rephrasing the machinery of the previous section, it is possible to prove the same correctness result for the call-by-value encoding.

5 Conclusions

We have proved in the paper that cc-languages get higher order power even with a very poor underlying system of constraints (the signature consists of a constant and a concatenation operation). The key observation is that the phenomenon of mobility is already present in the cc-paradigm as *mobility of variables*. Therefore, by simulating π-calculus channels in terms of *streams*, it is possible to rephrase Milner's encodings of *lazy* and *call-by-value* λ-calculus into the π-calculus. Our encodings distinguish between channels that are used once, just for book-keeping reasons, and those representing variables in [4], thus yielding a clearer distinction between resources. Furthermore the encoding suggest that both lazy and call-by-value λ-calculus can be encoded into a calculus more primitive than π-calculus, exploiting asynchronous mechanisms of communications.

We have tried to go beyond λ-calculus, by studing possible encodings of π-calculus into the cc-language discussed in this paper, without too much success. The reason derives essentially from the different abstraction layers in which the two (concurrent) languages stay. Indeed, in order to encode the communication primitives of π-calculus we need sequences of *ask* and *tell* operations, which results into a loss of atomicity of the primitive itself. This is dramatic in the presence of both external and internal nondeterminism. The case of λ-calculus is particularly easy because of the lack of any sort of nondeterminism.

An interesting research is the study of other reduction strategies with more parallelism than the call-by-value. The concurrent semantics of λ-calculus has been studied mainly by Lévy [2] and comes under the name of *permutation equivalence*. It is still an open problem if such semantics can be modelled by encodings of λ-calculus into concurrent languages.

Acknowledgments: We thank Catuscia Palamidessi for the useful comments and the pleasant electronic discussion about the cc-paradigm. Thanks also to Maurizio Gabbrielli for his comments about a preliminary version of the paper.

References

[1] G. Boudol: *A note on asynchronous π-calculus*, preprint, 1992.

[2] J. J. Lévy: *Optimal Reductions in the λ-calculus*, in "To H.B. Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism", Academic Press, 1980.

[3] R. Milner, J. Parrow, D. Walker: *A Calculus of Mobile Processes, part I and II*, Technical Report ECS-LFCS-89-85 and 86, Edinburgh University (1989).

[4] R. Milner: *Functions as Processes*, Info and Co, 1991.

[5] C.H.L. Ong: *Fully Abstract Models of the Lazy Lambda Calculus*, Proc. 29th Symposium on Foundation of Computer Science, pp. 368-376, 1988.

[6] V.A. Saraswat, M. Rinard: *Concurrent Constraint Programming*, in Proc. 17'th ACM Annual Symp. on Princ. of Progr. Lang., 1990.

[7] V.A. Saraswat, M. Rinard, P. Panangaden: *Semantic Foundations of Concurrent Constraint Programming*, in Proc. 18'th ACM Annual Symp. on Princ. of Progr. Lang., 1991.

[8] B. Thomsen: *A Calculus of Higher Order Communicating Systems* , in Proc. 16th ACM Annual Symp. on Princ. of Progr. Lang. (1989), pp.143-154.