A Stackless Runtime Environment for a Pi-calculus

Frédéric Peschanski
Université Pierre et Marie Curie - Paris 6 - LIP6
Frederic.Peschanski@lip6.fr

Samuel Hym
Université Denis Diderot - Paris 7 - PPS
Samuel.Hym@pps.jussieu.fr

Abstract

The Pi-calculus is a formalism to model and reason about highly concurrent and dynamic systems. Most of the expressive power of the language comes from the ability to pass communication channels among concurrent processes, as any other value. We present in this paper the CubeVM, an interpreter architecture for an applied variant of the Pi-calculus, focusing on its operational semantics. The main characteristic of the CubeVM comes from its stackless architecture. We show, in a formal way, that the resource management model inside the VM may be greatly simplified without the need for nested stack frames. This is particularly true for the garbage collection of processes and channels. The proposed GC, based on a reference counting scheme, is highly concurrent and, most interestingly, does automatically detect and reclaim cycles of disabled processes. We also address the main performance issues raised by the fine-grained concurrency model of the Pi-calculus. We introduce the reactive variant of the semantics that allows, when applicable, to increase the performance drastically by bypassing the scheduler. We define the language subset of processes in so called chain-reaction forms for which the sequential semantics can be proved statically. We illustrate the expressive power and performance gains of such chain-reactions with examples of functional, dataflow and object-oriented systems. Encodings for the pure Pi-calculus are also demonstrated.

Categories and Subject Descriptors D.3.4 [*Processors*]: Interpreters

General Terms Languages, Design, Theory, Performance

Keywords Pi-calculus, Interpreter, Operational Semantics, Garbage Collection

1. Introduction

The Pi-calculus is a computational theory in the family of *process algebras* [10, 5]. It is among the most widely studied theories of concurrency at the moment. The syntax and semantics of the calculus are minimalistic and aim at capturing concisely the foundations of *interactive systems*. The ability to pass communication channels between processes as any other value — a phenomenon called *name passing* or *channel mobility* — provides much expressive power to the language. In particular, it allows the concise encoding of most known computational models and also introduces a category of systems characterized by evolving communication structures, which apply perfectly to mobile networks and ubiquitous computing. While the theory is still worked on, applications of the Pi-calculus begin to emerge in various fields, some of them quite unexpected such as biocomputing or business process modeling. The Pi-calculus is also very intriguing from a programming language perspective. Works such as the Pict and Nomadic Pict programming languages [14, 19], TyCO [9] and JoCaml [6], show the interest of implementations exploiting the expressive constructs of the Pi-calculus. But the field is far from its maturity and a lot more experiments must be conducted. The work presented in this paper is about such an experiment.

The runtime system we propose, named the CubeVM, directly interprets an applied variant of the Pi-calculus called the cube-calculus. The architecture of the virtual machine proposes some *original* features. First, it does not rely on any form of a control stack. The pros and cons of stackless architectures for concurrent programming are discussed in the literature [8, 12, 21]. From our point of view, it is clear that processes are lighter without an attached stack. Moreover, an upper bound for the memory reserved by each process may be computed at compile time. But most importantly, having no stack to deal with simplifies greatly the *resource management* model inside the VM. This is particularly true for garbage collection. The GC scheme we propose is based on a lightweight and highly concurrent logic. It falls under the category of reference counting garbage collectors. Noteworthily, it also detects and reclaims cycles of disabled processes in a very simple way. Last but not least, the stackless runtime makes also a lot easier the capture of process states, a prerequisite for *process migration* [7] that is envisaged as a future work.

The fine-grained concurrency model of the Pi-calculus leaves most of the performance issues into the "hands" of the scheduling code. To address these issues, we introduce the *reactive* variants of the semantics that allow, when applicable, to bypass the scheduler and thus increase performance drastically. We introduce a subset of the cube-calculus that may be used to encode processes whose sequential execution can be proved formally. These processes in so called *chain-reaction* forms are guaranteed to execute identically in either standard or reactive semantics. We illustrate the expressive power of chain-reaction forms with examples of functional, dataflow and object-oriented systems.

Perhaps the most important aspect of the approach is that it is based on a well-studied (but still maturing) theory. This allows us to organize the discussion upon formally defined and analyzable operational semantics. The reduction system we propose may be employed to reason about the runtime environment and its most important properties.

The paper is structured as follows. First, we introduce the cube calculus and the basis of its operational semantics in section 2. The innovative resource management scheme is described in section 3. We put the emphasis on the absence of a control stack and the garbage collection of cyclic and passive computation structures.

VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.

Value	$v, \ldots \in$ VALUE	
Channel	$a, b, c, d, \ldots \in$ NAME	
Variable	$x, \ldots \in$ NAME	
Definition	$A, \ldots$	$::= \textbf{def}\ A(x_1, \ldots, x_n) = P;$
Process	$P, Q, \ldots$	$::=$
(inert)		0
(sum)		$\mid P + Q$
(parallel)		$\mid P \parallel Q$
(prefix)		$\mid \alpha, P$
(alternative)		$\mid \textbf{if}\ v\ \textbf{then}\ P\ \textbf{else}\ Q$
(switch)		$\mid \textbf{case}(v)\{v_1 \Rightarrow P_1 \mid \ldots \mid v_n \Rightarrow P_n\}$
(call)		$\mid A(v_1, \ldots, v_n)$
Prefix	$\alpha, \ldots$	$::=$
(communication)		$c?(x_1, \ldots, x_n) \mid c!(v_1, \ldots, v_n)$
(channel creation)		$\mid \textbf{tnew}(c_1, \ldots, c_n) \mid \textbf{snew}(c_1, \ldots, c_n)$
(reactive process)		$\mid \textbf{react}(c1, \ldots, c_n)$
(scoping)		$\mid \textbf{let}(x_1 = v_1, \ldots, x_n = v_n)$
(primitive)		$\mid \textbf{\#prim}(v_1, \ldots, v_n)$

Table 1. Syntax of the cube-calculus

$$
\left[\begin{array}{l}
P \equiv P\{y/x\} \text{ with } x \text{ bound and } y \text{ fresh in } P \\
P + Q \equiv Q + P \\
P + 0 \equiv P \\
P + (Q + R) \equiv (P + Q) + R \\
P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \\
\textbf{if}\ v\ \textbf{then}\ P \equiv \textbf{if}\ v\ \textbf{then}\ P\ \textbf{else}\ 0 \\
A(\widetilde{v_i}) \equiv \textbf{let}(\widetilde{x_i} = \widetilde{v_i}), P \quad \text{with } \textbf{def}\, A(\widetilde{x_i}) = P;
\end{array}\right.
$$

Table 2. Structural congruence

In section 4 we present the reactive variants of the semantics and the chain-reaction form of processes. Related work, conclusion, and bibliography follow.

2. The cube-calculus

2.1 Syntax

The syntax of the cube-calculus is very close to the pure polyadic Pi-calculus [10]. Its core constituents are presented in table 1.

Programs are composed of definitions and process expressions. A basic sequential process is formed by prefixing a continuation expression with an atomic action. The action prefixes concern channel creation, communication, primitive operations, and so on. The main difference with the pure Pi-calculus is that programs here can manipulate not only names, but also values of so-called *value types* such as integers, booleans, strings and symbols[1]. The match and mismatch prefixes, used to compare names in the Pi-calculus, are generalized by traditional if and case constructs. Also, basic primitives such as output of values and arithmetic operators are supported. They are prefixed by a sharp sign and correspond to silent steps in the semantics. A compound expression can be form by composing two expressions in parallel. The sum operator performs a non-deterministic choice between two expressions. We also provide syntactic conventions which are formally part of the so-called structural equivalence (see table 2).

It is not the purpose of the paper to describe in details each of the proposed language constructs. These are thoroughly investigated and commented in the abundant literature on process algebras in general and the Pi-calculus in particular (the proposed syntax is mostly inspired by [14]). Let us anyway illustrate the main language features (channel and process creation, communication and name-passing) on an example program, as follows:

[1] A symbol is an identifier string prefixed by a colon, e.g. :symbol.

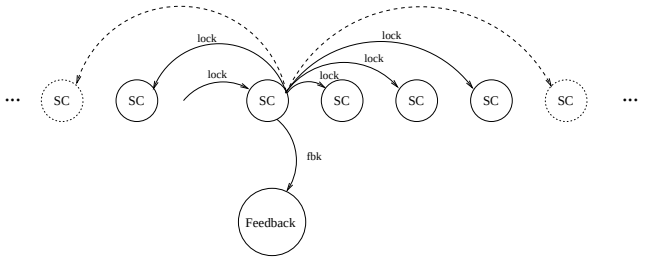

Figure 1. Example of concurrent program with channel passing

```
def CS(n,lock) = lock?(x),/*Crit.*/x!(n)/*Sect.*/,lock!(x);

def Feedback(fbk) = fbk?(n),#println("Lock taken by "+n),
                    Feedback(fbk);

def Launch(n,max,lock) =
    if n<max then [ Launch(n+1,max,lock) || CS(n,lock)];

tnew(lock,fbk),
   [ Launch(0,10000,lock) || Feedback(fbk) || lock!(fbk) ]
```

This program is composed of three parametrized definitions named CS, Feedback and Launch, followed by the entry-point of the program. It starts with the creation of two fresh channels lock and fbk using the tnew prefix. Three processes are then executed concurrently: the first one calls the Launch definition, the second one calls Feedback and waits on channel fbk. The last process is a single output of channel fbk through channel lock, which is written lock!(fbk). In this expression, we call fbk the *object* and *lock* the *subject* of the output prefix. The distinctive case of *name passing* is when a channel is used as object in an input or output prefix. Launch is a recursive definition that starts max-n processes, each of them running the CS definition. This definition is parametrized by an identity (an integer n) and a reference to the channel lock. According to the Pi-calculus semantics, exactly one of the CS processes will synchronize with the single output lock!(fbk). At this point of execution, the structure of the system is as depicted in figure 1. Note that an arrow does not represent a channel per se but more precisely a reference to a channel. The direction implies an intention to input or output on the channel (hence, multiple or bidirectional arrows may occur).

The synchronized CS process receives through lock a reference to the channel fbk and then sends its own identifier on the same channel, a typical example of *name passing*. This triggers the Feedback process which prints some information on the console. Finally, the activated CS process forwards the fbk channel on lock and then dies after synchronizing. The synchronization activates another CS process, and so on, until all CS processes have been activated. The expected output of the program is thus a series of 10000 feedback messages, each one corresponding to the atomic activation of the critical section enclosed between the input and output prefixes on the lock channel in the CS definition.

2.2 Operational semantics

Following the long-tradition of programming languages inspired by theory such as functional or logic programming dialects, our aim is to derive our runtime environment from a solid and sound theory. For this purpose, the *operational semantics* of the proposed

language play a prominent role. A non-exhaustive list of uses for a formal operational semantics includes:

- giving a precise meaning to each language construct,
- serving as non-ambiguous specifications for implementors,
- deriving higher-level constructs through *encodings*,
- enabling software verification of program properties.

Since we present the runtime system in the paper, we mostly exploit the first two items but longer-term objectives emphasize the last ones. For the core part of the cube-calculus, the operational semantic rules are given in table 3. These rely on a set of semantic functions presented in table 4. The presentation is rather standard, following a *structured operational semantics* (SOS) style. In contrast to standard semantics for the Pi-calculus, we do not present *compositional semantics* for open terms. It is fair to only consider closed terms for the specification of a runtime environment. Hence, a reduction semantics is proposed. Moreover, following the applied lambda-calculi tradition, an explicit environment is modeled, rather than a more abstract substitution-based scheme. As a matter of fact, two levels of environments are proposed. This is reflected by the syntax of the terms manipulated in the semantics, which is as follows:

$$\text{Agent } \mathcal{A}, \ldots \quad ::= [\Gamma]_n \vdash \gamma_1 : P_1 \parallel \cdots \parallel \gamma_m : P_m$$

An *agent* $\mathcal{A}$ defines a *global environment* $[\Gamma]_n$ as well as a set of *processes* P_i in parallel. At the level of terms, we assume the commutativity of the parallel operator[2]. Each P_i is running within its own *local environment* γ_i. The expression $\gamma : P$ is *well-formed* only if all the free and bound variables in P are defined in γ. By default, variables in γ are initialized to the undefined value $\perp$.

In the presented semantics, we only consider the execution of a single agent. However this denotes of course a potentially infinite number of concurrent processes[3]. Note that distributed extensions for the cube-calculus, allow multiple agents, are presented in [13]. The contents of the global environments is first composed of an index n used as a fresh identifier generator. The second component is a relation between *channel identifiers* and information about these channels, mostly related to resource management and reactive semantics (see sections 3 and 4).

As for the syntax, our purpose is not to investigate all the details of the semantic rules for all the proposed language constructs, most of them being close to the standard Pi-calculus versions (see [5], chapter 8 for example). Let us still describe the (Sum) rule of table 3. It states that if an agent with environment $[\Gamma]_n$ running a process P in local environment γ reduces (or executes in one-step) to an agent with environment $[\Gamma']_{n'}$ running P' in local environment γ', then the same agent running $P+Q$ *may* also reduce to the agent running P' conserving the environments. However, we know from the structural congruence that the $+$ operator is commutative, so, by the rule $(Struct)$ of the operational semantics, it *may* also be the case that $P+Q$ (or in fact $Q+P$ through $(Struct)$) reduces to Q' in some $[\Gamma'']_{n''}$ and γ''. This shows the non-deterministic nature of $+$. The (Par) rule works similarly for independent (non-communicating) parallel processes, but of course both sides of the parallel remain active whereas the $+$ operator makes a choice between the two branches of execution.

One of the most important rules is the communication rule (Com) that associates an output to an input prefix in two distinct concurrent processes and performs value-passing between them. Let us explain this (Com) rule in full details. First, the hypotheses are as follows. The reduction rule considers an agent with global environment $[\Gamma]_n$. Around a parallel, a left process in local environment γ performs a blocking output $c!(\widetilde{v})$ on channel c with values $\widetilde{v}$ (this is a common abbreviation for polyadic values), followed by a continuation P (alternatively Q if the communication is not enabled). The right process in local environment δ is waiting for an input on channel c. The enabling conditions for the communication to take place are as follows. First, the variables c in the left process and d in the right process must reference the same channel. For this to hold, $\gamma(c)$ and $\delta(d)$ must be equal. Moreover, the channel must have no owner which means that we are not meddling with reactive semantics (see section 4). The result of the reduction is the activation of the continuation P for the left process. In the right process, the received values (i.e. all $[\![v_i]\!]_\gamma$ that are the evaluations of the value expressions v_i in environment γ) are bound to variables x_i's in the local environment δ. Because of channel-passing, the global environment may also be affected through communication. This is described by the *updates* function of table 4. The precise meaning of this function shall be presented in the next section.

3. Resource management

From the preceding discussion, one may ask how the proposed semantics may be operated in practice. In fact, the structure of the implementation for the core constructs of the language is very simple. What is needed are a few data structures for channels and processes as well as an interpreter for the basic prefixes of the language. The parallel and sum operators of course require a proper scheduler structure and algorithm[4]. From an implementation point of view, this denotes a few thousand lines of C but nothing notably new from a scientific perspective. In this section and the following one, we shall describe innovative features of the implementation that are specific to our approach.

3.1 Resources creation and freshness

In cube-calculus expressions, only two kinds of resource may be created: channels and processes. The rules $(tRes)$ and $(sRes)$ (factorized in table 3 as rules (θRes) with $\theta \in \{t, s\}$) explain how to create channels. They show that $\theta\mathsf{new}(\widetilde{c_i})$ prefixes are always executable (or reducible). The result is to bind a set of variables $\widetilde{c_i}$ to a set of corresponding fresh channel identifiers $\hat{c_i}$ (fresh in $dom(\Gamma)$). The newly created channels have a reference count of 1 and no owner. For the $(tRes)$ rule the channels are said to be *transient* and we note $1^t_\emptyset$ and for the $(sRes)$ rule they are *static channels* and we note $1^s_\emptyset$. The difference relates to the collection of cycles we discuss later on. The functions *ckind*, *cown*, *cpartner* and *cref* of table 4 are used to access the different pieces of information attached to the channel identifier in the global environment.

The creation of new processes is defined by the rule $(Fork)$ of table 3, which is a simple rewrite of the parallel operator. Note that the rule atomically creates a set of processes, so that the order of creation is not fixed by the semantics. One resulting process, the one on the right, corresponds to the continuation of the spawning process. The other processes are dynamically created and inherit the same local environment γ from their parent. The only difference is that a variable id is bound to a fresh value corresponding to the process identifier in each local environment. The freshness property relies on the following proposition:

[2] The $(Fork)$ rule of table 3 distinguishes the rightmost process so we do not assume the commutativity of the parallel operator in the structural congruence.

[3] In the implementation, the maximum number of concurrent processes is bound by the available memory.

[4] Our objective is to support a fine-grained concurrency model. We thus favor VM-level threading in the implementation. However, we plan to take advantage of system-level threads at a higher level of granularity.

$$\frac{[\Gamma]_n \vdash \gamma : P \longrightarrow [\Gamma']_{n'} \vdash \gamma' : P'}{[\Gamma]_n \vdash \gamma : P + Q \longrightarrow [\Gamma']_{n'} \vdash \gamma' : P'} \ (Sum) \qquad \frac{[\Gamma]_n \vdash \gamma : P \longrightarrow [\Gamma']_{n'} \vdash \gamma' : P'}{[\Gamma]_n \vdash \gamma : P \parallel \delta : Q \longrightarrow [\Gamma']_{n'} \vdash \gamma' : P' \parallel \delta : Q} \ (Par)$$

$$\frac{\mathcal{B} \equiv \mathcal{A} \quad [\Gamma]_n \vdash \mathcal{A} \longrightarrow [\Gamma']_{n'} \vdash \mathcal{A}' \quad \mathcal{B}' \equiv \mathcal{A}'}{[\Gamma]_n \vdash \mathcal{B} \longrightarrow [\Gamma']_{n'} \vdash \mathcal{B}'} \ (Struct) \qquad \frac{v_i \in dom(\gamma)}{[\Gamma]_n \vdash \gamma : \#\mathbf{prim}(\widetilde{v_i}), P \longrightarrow [\Gamma]_n \vdash \gamma : P} \ (Prim)$$

$$\frac{[\![v]\!]_\gamma = \mathbf{true}}{[\Gamma]_n \vdash \gamma : \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow [\Gamma]_n \vdash \gamma : P} \ (Then) \qquad \frac{[\![v]\!]_\gamma = \mathbf{false}}{[\Gamma]_n \vdash \gamma : \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow [\Gamma]_n \vdash \gamma : Q} \ (Else)$$

$$\frac{[\![v]\!]_\gamma = [\![v_i]\!]_\gamma}{[\Gamma]_n \vdash \gamma : \mathbf{case}(v)\{v_1 \Rightarrow P_1 \mid \ldots \mid v_n \Rightarrow P_n\} \longrightarrow [\Gamma]_n \vdash \gamma : P_i} \ (Switch)$$

$$\frac{\bigcup_i \{\hat{c_i}\} \cap dom(\Gamma) = \emptyset}{[\Gamma]_n \vdash \gamma : \theta\mathbf{new}(\widetilde{c_i}), P \longrightarrow [\mathit{allocs}(\Gamma, \bigcup_i \{\gamma(c_i)\}, \bigcup_i \{\hat{c_i}^\theta\})]_n \vdash \gamma \uplus \bigcup_i \{c_i \triangleright \hat{c_i}\} : P} \ (\theta Res), \quad \theta \in \{t, s\}$$

$$\frac{l_1 = n \quad l_2 = l_1 + 1 \quad \ldots \quad l_m = l_1 + m - 1}{[\Gamma]_n \vdash \gamma : P_1 \parallel \cdots \parallel P_m \parallel Q \longrightarrow [\mathit{inherit}(\Gamma, \gamma, m)]_{n+m} \vdash \gamma \uplus \{id \triangleright l_1\} : P_1 \parallel \cdots \parallel \gamma \uplus \{id \triangleright l_m\} : P_m \parallel \gamma : Q} \ (Fork)$$

$$\frac{\gamma(c) = \delta(d) \quad \mathit{cown}(\Gamma(\gamma(c))) = \emptyset}{[\Gamma]_n \vdash \gamma : c!(\widetilde{v_i}), P + Q \parallel \delta : d?(\widetilde{x_i}), R + S \longrightarrow [\mathit{updates}(\Gamma, \bigcup_i \{\delta(x_i)\}, \bigcup_i \{[\![v_i]\!]_\gamma\})]_n \vdash \gamma : P \parallel \delta \uplus \bigcup_i \{x_i \triangleright [\![v_i]\!]_\gamma\} : R} \ (Com)$$

$$\frac{\mathbf{def}\ A(\widetilde{x_i}) = P;}{[\Gamma]_n \vdash \gamma : A(\widetilde{v_i}) \longrightarrow [\mathit{updates}(\Gamma, img(\gamma), \bigcup_i \{[\![v_i]\!]_\gamma\})]_n \vdash \bigcup_i \{x_i \triangleright [\![v_i]\!]_\gamma\} : P} \ (Call)$$

$$\frac{}{[\Gamma]_n \vdash \gamma : \mathbf{let}(\widetilde{x_i} = \widetilde{v_i}), P \longrightarrow [\mathit{updates}(\Gamma, \bigcup_i \{\gamma(x_i)\}, \bigcup_i \{[\![v_i]\!]_\gamma\})]_n \vdash \gamma \uplus \bigcup_i \{x_i \triangleright [\![v_i]\!]_\gamma\} : P} \ (Lex)$$

Table 3. The semantics of the core cube-calculus

reactive owner of channel	$\mathit{cown}(k^\theta_\omega) \triangleq \omega$
channel kind (transient,static)	$\mathit{ckind}(k^\theta_\omega) \triangleq \theta$
reference count of channel	$\mathit{cref}(k^\theta_\omega) \triangleq k$
reference count of variable	$\mathit{nbrefs}(\gamma, \hat{c}) \triangleq card(\{d \mid d \in dom(\gamma) \cap \gamma(d) = \hat{c}\})$
environment inheritance	$\mathit{inherit}(\Gamma, \gamma, n) \triangleq \Gamma \uplus \{\hat{c} \triangleright (k + n \times \mathit{nbrefs}(\gamma, \hat{c}))^\theta_\omega \mid \hat{c} \in dom(\Gamma) \cap img(\gamma) \ \wedge \ \Gamma(\hat{c}) = k^\theta_\omega\}$
global environment extension	$\mathit{refs}(\Gamma, \widetilde{C}) \triangleq \Gamma \uplus \{\hat{c} \triangleright (k+1)^\theta_\omega \mid \hat{c} \in \widetilde{C} \cap dom(\Gamma) \ \wedge \ \Gamma(\hat{c}) = k^\theta_\omega\}$
global environment contraction	$\mathit{unrefs}(\Gamma, \widetilde{C}) \triangleq \Gamma \uplus \{\hat{c} \triangleright (k-1)^\theta_\omega \mid \hat{c} \in \widetilde{C} \cap dom(\Gamma) \ \wedge \ \Gamma(\hat{c}) = k^\theta_\omega\}$
environment allocation	$\mathit{allocs}(\Gamma, \widetilde{C}, \widetilde{D^\Theta}) \triangleq \mathit{unrefs}(\Gamma, \widetilde{C}) \uplus \{\hat{d} \triangleright 1^\theta_\emptyset \mid \hat{d}^\theta \in \widetilde{D^\Theta}\}$
environment update	$\mathit{updates}(\Gamma, \widetilde{C}, \widetilde{D}) \triangleq \mathit{refs}(\mathit{unrefs}(\Gamma, \widetilde{C}), \widetilde{D})$

Table 4. Semantic functions

PROPOSITION 1. *Let $\mathcal{A} \triangleq [\Gamma]_n \vdash \gamma_1 : P_1 \parallel \cdots \parallel \gamma_p : P_p$ an agent such that $\forall 1 \leq i < j \leq p$ we have $\gamma_i(id) \neq \gamma_j(id)$ and $\gamma_i(id), \gamma_j(id) < n$. If $\mathcal{A} \longrightarrow \mathcal{A}'$ with $\mathcal{A}' \triangleq [\Gamma]_{n'} \vdash \gamma'_1 : P'_1 \parallel \cdots \parallel \gamma'_{p'} : P'_{p'}$ then $\forall 1 \leq i < j \leq p'$ we have $\gamma'_i(id) \neq \gamma'_j(id)$ and $\gamma'_i(id), \gamma'_j(id) < n'$.*

The proof is by a simple structural induction on the semantics.The only rule to consider effectively is the one that changes the index n in the global environment, that is, the $(Fork)$ rule. We see that n is updated when the rule triggers as $n' = n + m$ where m is the number of forked processes. Each of these new processes, say $\gamma_k : P_k$ with $n \leq k < n+m$, is given a fresh and unique identifier in $[n, n + m - 1]$, which is enough to conclude the proof. □

The freshness property is important for the reactive semantics, as explained in section 4.

The final step in the $(Fork)$ rule is to update the global environment. The newly created processes inherit the local environment of their parent. The formal definition is in the *inherit* function of table 4.

3.2 Stackless recursion

In most if not all programming languages, recursive calls as well as the implied control stack occupy an important place. In the cube-calculus, recursion represents in fact the only means by which repetitive or infinite computations may be expressed. The semantics of the call operator are given by the rule $(Call)$ of table 3. The first thing it does is to bind all the parameter variables $x_1, \ldots, x_n$ to the argument values in an empty local environment. We can see in the semantic rules that local environments adopt a flat structure, i.e. they may not be nested. Actually, in all the rules where the local environments in the right hand side must be modified, such as the (Com) rule, the overriding operator $\uplus$ is employed. In $\delta \uplus \bigcup_i \{x_i \triangleright [\![v_i]\!]_\gamma\}$, all the bindings for the x_i's in δ are replaced by new ones. This flat structure for local environments is the first

reason why a stack is not required. The second reason for the stackless architecture comes from the fact that the syntax enforces *tail recursion*. For general, non-tail recursion, consider the canonical example of the ackermann function, that we encode as follows:

```
def Ack(n,p,r) = if n = 0 then r!(p+1) else
      if p = 0 then Ack(n-1,1,r) else
         tnew(r1), [ Ack(n,p-1,r1) || r1?(pp),Ack(n-1,pp,r) ];
```

Here, the stack is built *on demand* using channels and processes. The main drawback is that we do not exploit the processor stack, and as such native performance is out of reach for general recursion (at least on current register/stack-based processors). Moreover, memory consumption increases by at least one channel (a minimum of one memory word for the reference count and control flags) for every stack frame to "simulate". The main advantage is that processes become really lightweight without nested stack frames attached to them. We may even compute an upper bound for the memory needed by a process at compile-time[5]. Also, this removes the burden of stack analysis in order to freeze the VM state (or parts of the VM state). Another major argument is that this simplifies greatly the garbage collection issue, as discussed in the next section. A minor point is that having no stack also means having no stack limit[6].

3.3 Garbage collection

The CubeVM environment uses a reference counting scheme for garbage collection. An important design choice is that reference counts are placed on channels and not on processes. It is a little bit like counting occurrences of method calls instead of object references in an object-oriented setting. All the collection is handled by the processes themselves at key points of their executions. In the semantics, the garbage collection points are those involving the functions *refs*, *unrefs* and *updates* of table 4. These are spread in the rules (Com), $(Call)$ and (Lex) of table 3. To illustrate the meaning of these functions, consider the modification of the global environment Γ written as *updates*$(\Gamma, \{\gamma(x)\}, \{\hat{c}\})$. The variable x is in the local environment γ and $\hat{c}$ is a channel identifier. Since the local environment is flat, the reference count for the previous bindings for x, i.e. *cref*$(\Gamma(\gamma(x)))$, must be decremented. This is expressed as *unrefs*$(\Gamma, \{\gamma(x)\})$. Complementarily, we must increment the reference count for the new value $\hat{c}$, which we write *refs*$(\Gamma, \{\hat{c}\})$. These functions are generalized in table 4 to handle polyadic values.

In table 5 we define extra operational rules to support the collection scheme. The small-step semantics presentation demonstrates, in fact, that the collection is performed locally, without any external algorithm. First, the rules $(Collect)$ and $(Finish)$ explain how to reclaim channels of count zero and also how to reclaim the resources of a terminating process. The rule $(DeadEnd)$ deals with the collection of waiting and isolated processes. We may take advantage of the minimalism of the syntax to extract the general syntactic form of a *waiting process* as follows:

$$\gamma : \sum_i c_i!(\widetilde{v_i}), P_i + \sum_j c_j?(\widetilde{x_j}), Q_j \qquad i + j \geq 1$$

A waiting process is thus a sum of non-deterministic branches, each branch being guarded by either an input or output prefix. In any other case, it is easy to prove from the semantics that the process may advance its execution. A waiting process is also said to be *isolated* if all the channels the process is waiting on are only known to the process itself. This is the case for a channel $\hat{c}$ with a reference count of 1. But it is also possible to have multiple references to $\hat{c}$ in the local environment γ. For example, suppose we write **let**(d=c) where c is a channel of identifier $\hat{c}$. Then, we have a reference count of 2 for $\hat{c}$ in the global environment, but both references are within the local environment γ, respectively $\gamma(c)$ and $\gamma(d)$. So, we need to compare the global reference count of a channel (each c_i and c_j in rule $(DeadEnd)$) with the set of local references for the same channel. Given a channel $\hat{c}$, then its global reference count is *cref*$(\Gamma(\hat{c}))$ and its local count in a process $\gamma : P$ is *nbrefs*$(\gamma, \hat{c})$ (see table 4). When these two counts are identical, the channel is only known to the process P. Since a process can not directly communicate with itself in the Pi-calculus, any attempt to input or output on such a channel is blocking in a permanent way. Such a communication is said to be *disabled*. A waiting process that has all its alternative branches disabled is then *isolated* and may be reclaimed because none of its execution branches may be ever enabled again.

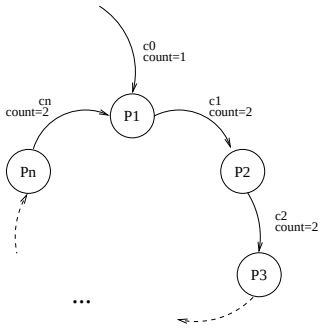

Figure 2. Example of a cyclic system

Finally, the rules $(DeadIn)$ and $(DeadOut)$ of table 5 deal with the collection of cycles. The problem of cycle collection is twofold. Firstly, cycles must be detected properly. And secondly, in order to reclaim concurrent processes, we must discriminate passive cycles from active ones. Of course, the objective is to remove only the passive cycles. The beauty of the proposed approach is that both requirements are handled simultaneously, in a very simple way. To illustrate the solution, suppose a cyclic structure of processes as represented in figure 2.

On each channel we indicate the name of the channel as well as its reference count. We can see that the process P_1 references, as input, a channel c_0 with reference count 1. A reference count of 1 means that only one process knows the channel. The $(DeadIn)$ rule tells that a process attempting to communicate on a channel with reference count of 1 should be reclaimed, the communication being permanently disabled. In the example, this implies the collection of process P_1. Of course, the collection is also triggered if the number of global references equals the number of local references, as for the $(DeadEnd)$ rule. After the collection of P_1, the reference count for the channel c_1 will decrease by one, which triggers P_2's collection, and so on until eventually P_n disappears and the whole cycle is collected. The $(DeadOut)$ rule deals similarly with outputs on uniquely owned channels. As such, P_n may be reclaimed before P_2 if it tries to output on channel c_n. This is an almost optimally concurrent GC scheme since all the detection and collection are performed locally, without the need for any external logic.

[5] The memory needed by a process is statically bound by the size of the local environment needed to call the definition, reachable by the process, with the largest lexical environment. Note that this is not the memory needed for the channels created by a process, nor the number of created channels and processes, which are dynamic properties.

[6] Stack requirements may be high: ackerman(4,1) will "stack overflow" most implementations, not ours.

$$\frac{}{[\Gamma, \hat{c} \triangleright 0^{\theta}_{\omega}]_n \vdash \gamma : P \longrightarrow [\Gamma]_n \vdash \gamma : P} \ (Collect) \qquad \frac{\gamma \neq \emptyset}{[\Gamma]_n \vdash \gamma : 0 \longrightarrow [\mathit{unrefs}(\Gamma, img(\gamma))]_n \vdash \emptyset : 0} \ (Finish)$$

$$\frac{\mathit{cref}(\Gamma(\gamma(c_i))) = \mathit{nbrefs}(\gamma, \gamma(c_i)) \quad \mathit{cref}(\Gamma(\gamma(c_j))) = \mathit{nbrefs}(\gamma, \gamma(c_j))}{[\Gamma]_n \vdash \gamma : \sum_i c_i!(\widetilde{v_i}), P_i + \sum_j c_j?(\widetilde{x_j}), Q_j \longrightarrow [\mathit{unrefs}(\Gamma, img(\gamma))]_n \vdash \emptyset : 0} \ (DeadEnd)$$

$$\frac{\mathit{ckind}(\Gamma(\gamma(c))) = t \quad \mathit{cref}(\Gamma(\gamma(c))) = \mathit{nbrefs}(\gamma, \gamma(c))}{[\Gamma]_n \vdash \gamma : c!(\widetilde{v}), P + Q \longrightarrow [\mathit{unrefs}(\Gamma, img(\gamma))]_n \vdash \emptyset : 0} \ (DeadOut)$$

$$\frac{\mathit{ckind}(\Gamma(\gamma(c))) = t \quad \mathit{cref}(\Gamma(\gamma(c))) = \mathit{nbrefs}(\gamma, \gamma(c))}{[\Gamma]_n \vdash \gamma : c?(\widetilde{x}), P + Q \longrightarrow [\mathit{unrefs}(\Gamma, img(\gamma))]_n \vdash \emptyset : 0} \ (DeadIn)$$

Table 5. Semantics for Garbage Collection

We define now the notion of *disabled* processes, whose general form, in global environment Γ, is as follows:

$$\gamma : \sum_{i \in I} \alpha_i, Q_i \text{ such as } \exists k \in I, \begin{cases} \alpha_k = c!(\widetilde{v}) \vee \alpha_k = c?(\widetilde{x}) \\ \mathit{cref}(\Gamma(\gamma(c))) = \mathit{nbrefs}(\gamma, \gamma(c)) \end{cases}$$

Intuitively, a disabled process is trying to perform at least one disabled communication. Such a process is a good candidate for garbage collection since parts of its behavior are, from now on, permanently inactive. Because of name-passing, this partial condition may be too restrictive. In rare occasions, processes need to *conserve* a channel reference for future use. As an illustration, imagine an object with an interface that no one knows except the object itself. Still, external objects may like to *ask* for this interface at runtime, which is exactly what the IUnknown interface of COM components is about [16]. Consider the following example:

```
def Component(iunknown,self) =
      iunknown?(x),x!(self),Component(iunknown,self)
  +  self?(message),  // ... etc ...
```

In order to interact with this component, we must know the self channel. If no one knows this channel except the component, then the GC will reclaim it. But if we look at the code, it is possible to use the iunknown channel to get back a reference to self from the outside. In order to deal with this kind of situation, we have to flag the self channel as *static* which means that it will not trigger the cycle detection and collection algorithm. The prefix snew is used to create channels flagged as static.

The most important aspect of garbage collection is the safety property of the proposed algorithm. We have to check that it applies exclusively on those process terms that we identified as candidates for collection. Formally, we define a collection as *safe* when it satisfies the following property:

PROPOSITION 2. *Let P be a process such that $\Gamma \vdash \gamma : P \longrightarrow \Gamma' \vdash \emptyset : 0$ then P is either isolated or disabled on transient channels, or it is the inert process 0. Moreover, $\Gamma' = \mathit{unrefs}(\Gamma, img(\gamma))$.*

The only rules that match the $\emptyset : 0$ in the right-hand side of the conclusion are $(Finish)$ for inert processes, $(DeadEnd)$ for isolated ones, and $(DeadIn)$ and $(DeadOut)$ for disabled processes (but only for transient channels). We can see that all rules individually respect proposition 2, which thus applies on the whole semantics by structural induction. □

4. Reactive semantics

In the Pi-calculus, the role of the parallel operator is twofold. First, it denotes the concurrent execution of the two processes it separates, which results in *non-sequential* semantics. Second, it is a *composition* operator because according to the rule (Com) of table 3, processes may only communicate around a parallel operator.

4.1 Sequential behaviors

In order to allow explicit sequential computations, we introduce the notions of *reactive channels* and *reactive processes*. A reactive channel is a "normal" (active) channel flagged with an *owner process*. The react(c) prefix is used to flag channel c as reactive. After reducing the prefix, the process becomes the owner of the channel and switches from an active mode to a reactive mode of execution. It will only be activated if another process (reactive or active) outputs on channel c. This activation is notably unidirectional, from output to input and not the converse.

Table 6 presents the rules that drive the execution of reactive processes. The switch from the active to the reactive mode of execution for a process, as discussed previously, is handled by rule $(RMode)$. We can see that owning a channel $\hat{c}$ puts a decoration $\gamma(id)$ (which is the identifier of the owner process) in the global environment. The rule $(RCom)$ describes a communication on a reactive channel. The left-hand process outputs a set of values on channel c. To trigger the rule, this channel must be owned by the right-hand process (obviously in reactive mode). If these conditions are not satisfied, then the active rule (Com) applies. The consequences of both the (Com) and $(RCom)$ rules are very similar. A fundamental difference is that in the case of $(RCom)$ there is no choice for the receiver process. We formalize a stronger property that states the uniqueness of the reduction of a given output prefix through $(RCom)$. For this we need to restrict the form of the receiver processes. In a process $\gamma : P$, an *ambiguous sum* on $\gamma(c)$ is a sum expression $c?(\widetilde{x}) + \sum_i \alpha_i.Q_i$ in which there exists $\alpha_j \triangleq d?(\widetilde{y})$ such that $\gamma(c) = \gamma(d)$. An ambiguous sum is thus a sum guarded by at least two input prefixes on the same channel. Now we may state the reduction uniqueness property as follows:

PROPOSITION 3. *Let $\mathcal{A}$ be an agent such that $\mathcal{A} \triangleq [\Gamma]_n \vdash \cdots \parallel \gamma : c!(\widetilde{v_i}), P + Q \parallel \cdots$ and without any ambiguous sum on $\gamma(c)$. Consider also $\mathcal{A}'$ and $\mathcal{A}''$ two agents such that $\mathcal{A}' \triangleq [\Gamma']_{n'} \vdash \cdots \parallel \gamma' : P \parallel \cdots$ and $\mathcal{A}'' \triangleq [\Gamma'']_{n''} \vdash \cdots \parallel \gamma'' : P \parallel \cdots$. If $\mathit{cown}(\Gamma(\gamma(c))) \neq \emptyset$ and $\mathcal{A} \longrightarrow \mathcal{A}'$ as well as $\mathcal{A} \longrightarrow \mathcal{A}''$, then we have $\mathcal{A}' = \mathcal{A}''$*

To consume an output prefix, the only rule enabled for the reduction $\mathcal{A} \longrightarrow \mathcal{A}'$ is $(RCom)$. The (Com) rule cannot be triggered because channel c has a non-empty owner. First, by the freshness guarantee of the (θRes) rules, we know that a channel

$$\frac{\hat{c_i} = \gamma(c_i) \quad \Gamma(\hat{c_i}) = k^{\theta}_{\omega}}{[\Gamma]_n \vdash \gamma : \mathbf{react}(\widetilde{c_i}), P \longrightarrow [\Gamma, \hat{c_i} \triangleright k^{\theta}_{\gamma(id)}]_n \vdash \gamma : P} \ (RMode)$$

$$\frac{\mathit{cown}(\Gamma(\gamma(c))) = \delta(id) \quad \gamma(c) = \delta(d) \quad \Gamma(\hat{c}) = k^{\theta}_{\omega}}{\begin{array}{l}[\Gamma]_n \vdash \gamma : c!(\widetilde{v_i}), P + Q \parallel \delta : d?(\widetilde{x_i}), R + S \\ \longrightarrow [\mathit{updates}(\Gamma, \bigcup_i \{\delta(x_i)\}, \bigcup_i \{[\![v_i]\!]_\gamma\}), \gamma(c) \triangleright k^{\theta}_{\omega}]_n \vdash \gamma : P \parallel \delta \uplus \bigcup_i \{x_i \triangleright [\![v_i]\!]_\gamma\} : R\end{array}} \ (RCom)$$

Table 6. Rules for reactive semantics

identifier (here $\gamma(c)$) is unique in the global environment. This explains that $\gamma(c)$ maps to at most one reference to an owner process. Moreover, we know that the channel is reactive, which means that the owner reference is not empty. Now, in the $(RCom)$ rule, the process $\mathcal{P}_1$ in $\mathcal{A}$ that can interact with $\gamma : c!(\widetilde{v_i}), P + Q$ is of the form $\delta : d?(\widetilde{x_i}), R + S$ such as $\delta(id) = \mathit{cown}(\Gamma(\gamma(c)))$. Suppose these two processes interact along the reduction $\mathcal{A} \longrightarrow \mathcal{A}'$. If we consider a second reduction $\mathcal{A} \longrightarrow \mathcal{A}''$ with $\mathcal{A}' \neq \mathcal{A}''$, then there must be another process $\mathcal{P}_2$ in $\mathcal{A}$ different from $\mathcal{P}_1$ but of a similar form $\xi : e?(\widetilde{y_i}), T + U$ such that $\xi(id) = \mathit{cown}(\Gamma(\gamma(c)))$. From proposition 1, we know that if $\delta(id) = \xi(id)$, then $\mathcal{P}_1 = \mathcal{P}_2$. Thus, $\mathcal{P}_1$ is the unique process that inputs on $\gamma(c)$ through $(RCom)$. Since it does not contain any ambiguous sum on $\gamma(c)$, the input prefix involved is unique. And from this we can conclude that a unique reduction can be inferred in the semantics so that $\mathcal{A}' = \mathcal{A}''$. □

The conclusion is that for a given output prefix on a reactive channel, under the assumption that there is no ambiguous sum on that channel, then the $(RCom)$ rule can be triggered in a deterministic way. This means that it can be implemented by a sequential algorithm.

4.2 Chain reactions

We can not decide, in general, whether a given program is sequential or not. We define in this section the general form of *chain-reactions* that are processes of which we can prove the sequential behavior statically. For this, we first give a few definitions.

A *free input* is an occurrence of $c?(\widetilde{x_i})$ in an expression P where c is not bound by either a new or another input prefix in P up to structural congruence and aliasing[7].

Let $\mathbf{new}(\widetilde{c_i}, \widetilde{c_j}), Q$ an expression with $Q \equiv \cdots \parallel P \parallel \cdots$. We say that P is in *chain-reaction form* on the set of channels $\widetilde{c_i}$ if all the input prefixes in P are free and exclusively on the c_i's. Moreover, P must not contain any ambiguous sum and every input on c_i in Q must be in P. Finally, if there is a process in P written $\mathbf{new}(\widetilde{b_i}), R$ with $R \equiv R_1 \parallel \cdots \parallel R_n$, then all the R_i's $(i < n)$ are also chain-reactions for disjoint subsets of $\widetilde{b_i}$. The subprocess R_n is the continuation of P. It may still be in chain-reaction form for a disjoint subset of the b_i's but must also preserve the chain-reaction form on the c_i's.

To comment on this definition, notice first that the channels $\widetilde{c_i}$ are only known to the subprocesses of Q. Moreover, P is the only subprocess performing (non-ambiguous) inputs on these channels. Suppose now an agent $\mathcal{A}$ containing Q. If $\mathcal{A}$ reduces to an agent of the form $[\Gamma]_n \vdash \ldots \parallel \gamma : c!(\widetilde{v_i}), R + S \parallel \ldots$ with $c \in \widetilde{c_i}$ up to aliasing, then only P can perform the corresponding input. This says, in fact, that in the case of processes in chain-reaction forms, any of the two communication rules (Com) (for active channels) or $(RCom)$ (for reactive ones) may trigger indifferently, and in a deterministic way.

[7] In the syntax, the only way to create an alias y for a variable x in an expression P up to structural congruence is to write $\mathbf{let}(y = x), P$.

Another way to explain this is to say that a given program, slightly modified to trigger reactive semantics instead of active ones, may be operated sequentially, without the need of the active semantics (and underlying scheduler). In order to formalize this property and prove it correct, we define inductively on the syntax the transformation *unchain* consisting in removing all the react prefixes such that $\mathit{unchain}(\mathbf{react}(\widetilde{c_i}), P) \triangleq P$. This transformation naturally extends to whole agents, provided that all channel owners are removed from the global environment. Then, we may state the following proposition:

PROPOSITION 4. *Let* $\mathbf{react}(\widetilde{c_i}), P_j$ *the complete set of the processes prefixed by react in a given agent* $\mathcal{A}$*. If all the* P_j*'s are chain-reactions and if there is an agent* $\mathcal{A}'$ *such that* $\mathcal{A} \longrightarrow^* \mathcal{A}'$*, then* $\mathit{unchain}(\mathcal{A}) \longrightarrow^* \mathit{unchain}(\mathcal{A}')$*. Reciprocally, if* $\mathit{unchain}(\mathcal{A}) \longrightarrow^* \mathcal{A}''$ *then there exists* $\mathcal{A}'$ *such as* $\mathit{unchain}(\mathcal{A}') = \mathcal{A}''$ *and* $\mathcal{A} \longrightarrow^* \mathcal{A}'$*.*

In chain-reactions, all distinct input prefixes are performed on distinct channels. This may be easily deduced from the definition. As such, chain-reactions enjoy a receiver uniqueness property, not only in the case of reactive channels. As a consequence, proposition 3 still applies even for a channel $\hat{c}$ with $\mathit{cown}(\hat{c}) = \emptyset$. This means that for a given output prefix, if the (Com) rule is triggered, then it may only be so in a deterministic way, as for the $(RCom)$ rule for reactive channels. The proof is similar to the one for proposition 3, so we omit it here. Then, it is easy to show that the property applies globally by structural induction on the semantic rules. Indeed, all the inference rules of the semantics are the same for both the active and the reactive cases. □

4.3 Examples

In this section we discuss the expressivity of the subset of the cube-calculus corresponding to processes in chain-reaction forms. We show the reactive encoding of example programs involving various programming paradigms. We also illustrate the performance gains one might expect by applying the reactive semantics of the CubeVM in practice. Table 7 compares the execution times for both active and reactive semantics on the benchmark programs. The acceleration is also indicated. On table 8, we compare with the python interpreter (in version 2.3), which is one of the most successful interpreted technologies around. All the programs are available on line (at [2]). The test machine is an Intel P4-2Ghz powered computer with 512Mb RAM and 40Gb Hard drive. It is running the Linux operating system with kernel 2.6.8. We begin with the critical section example of section 2, testing with 10000 processes. We make the fbk channel reactive in the reactive variant, which shows that the reactive semantics are not incompatible with channel passing. It also shows that the reactive version is more that 10 times faster compared to the active one. It is of course not surprising that most of the time is spent in the scheduler for the non-sequential variant of the example. A similar program is tested in the Python environment, but the obtained figures are not really relevant since

program	active (s)	reactive (s)	ratio
sc(10000)	14.17	0.98	≈ 14.5
ack(3,7)	2.29	1.15	≈ 2.0
fib(27)	2.10	0.55	≈ 3.8
tak(96,48,32)	1.05	0.69	≈ 1.5
sieve(10000)	3.51	0.32	≈ 11.0
objinst(1500000)	5.15	5.11	≈ 1.0
threads-flow(3000)	3.41	0.25	≈ 13.6

Table 7. Reactive vs. active semantics

program	active (s)	reactive (s)	python (s)
sc(250)	0.05	0.01	0.29
ack(3,7)	2.29	1.15	1.29
fib(27)	2.10	0.55	0.51
tak(96,48,32)	1.05	0.69	0.37
objinst(1500000)	5.15	5.11	16.50
threads-flow(250)	0.18	0.03	0.14

Table 8. Comparing with Python [3]

the interpreter seems to support only a few hundred threads at least on the Linux platform.

4.3.1 Functional computations

Functional programming relies on a deterministic computational model that offers a nice expressivity benchmark for processes in chain-reaction forms. In order to illustrate the reactive encoding of functional systems, we propose to rewrite the ackermann example of section 3, which can clearly be executed sequentially since it is deterministic. The reactive encoding is as follows:

```
def Ack(n,p,r) = if n = 0 then r!(p+1) else
      if p = 0 then Ack(n-1,1,r) else
         tnew(r1), [ Ack(n,p-1,r1) ||
                     react(r1),r1?(pp),Ack(n-1,pp,r) ];
```

It is very easy to show that the ackermann definition is in chain-reaction form, so that removing the react prefixes would not affect the semantics. Of course, we expect better performance for the reactive version. Table 7 shows the relative performance of the active vs. reactive semantics for the ackermann example. The reactive variant is approximately twice faster. Along the same lines, other functions may be encoded and tested. We show the results for the non-tail recursive encoding of the well-known Fibonacci and tak functions (see [2] for the encodings of these). We only consider non-tail recursions so that we do not rely on tail-call elimination that is not always supported[8]. Moreover, this gives an idea of the relative cost of simulating stack frames in the CubeVM. The tak function has some potential for concurrent evaluation which is well illustrated by the fact that the gains obtained by the reactive variant are not so high. If compared to the python interpreter execution times, the results of table 8 shows that the CubeVM, at least in reactive semantics, provides similar performance. The ack example is faster in cube but the tak example performs better in Python.

4.3.2 Dataflow systems

We next illustrate the expressivity of the language by exhibiting the encoding of a sequential *dataflow* system. We give below a simple (and not fully optimized) version of Eratosthenes's sieve. The idea is to implement a generator process feeding the numbers to be sieved to a chain of filtering processes. Each one of these filters is associated to a given prime number. When a filter receives a candidate number, it filters out multiples of the prime numbers it is associated to. The other candidate number are forwarded to the next filter in the chain. At the end of the chain, a new prime number is discovered. As a result, a corresponding sieving process is added to the chain, until the specified number of prime numbers to discover is reached. This algorithm is encoded in the cube-calculus, in an almost literal way, as follows:

```
def GenInt2(i,n,out) = if i<n then out!(i),GenInt2(i+2,n,out);

def PrimeFilter(in,n,out) =
    in?(x),if (x % n) = 0 then PrimeFilter(in,n,out)
    else out!x,PrimeFilter(in,n,out);

def PrimeSink(in) =
    in?(x),#print(x),#println(" is prime"),tnew(out),
    [ react(out),PrimeSink(out) || PrimeFilter(in,x,out) ];

tnew(gen,out),
    [ GenInt2(3,1000,gen) || react(gen),PrimeFilter(gen,2,out) ||
      react(out),PrimeSink(out) ]
```

It is arguably an intuitive encoding of the algorithm. This example is yet again in chain-reaction form and it can be made more than ten times faster with the reactive optimization. No comparison is made with python which is clearly not a language for dataflow systems.

4.3.3 Object-based computations

Object-orientation is probably the most important programming style today. Most object-oriented programming languages allow the definition of classes encapsulating data (slots) and behavior (methods). Another important feature is the so-called *data hiding*. Of course, object models invariably support some form of *inheritance*. Below is a very simple example in the Python programming language [3] illustrating all these fundamental features[9]:

```
class Cell:
    def __init__(self,v):
        self.value=v
    def set(self,v):
        self.value:=v
    def get(self):
        return self.value

class CellPrint(Cell):
    def __init__(self,v):
        Cell.__init__(self,v)
    def print(self):
        print self.get()

c = CellPrint(10)
print c.get()
c.set(12)
c.print()
```

In the main program, the variable c references a CellPrint object containing value 10. The get method is called, which is defined in the Cell superclass. The value 12 is then stored and the print method of the subclass is invoked. We expect the output "1012" on the console.

The problem with the cube-calculus, as for most variants of the Pi-calculus, is that it does not provide any basic notion of *objects*, *classes*, *methods* and *slots* or *inheritance*. Everything must

[8] The python interpreter, in version 2.3, does not seem to eliminate tail-calls.

[9] We do not discuss the encoding of method bodies, which could use imperative, functional or dataflow programming styles.

be encoded using processes and channels. Related work detail various encodings of objects in Pi-calculus or other process algebras [10, 17]. An object is generally associated to a recursive process. Each method can be associated to a channel, or we may dispatch on a unique channel to optimize space. Moreover, most object-based interactions are sequential. We should thus try to rely on the reactive semantics. Consider the following encoding:

```
def Cell(self,value) =
    self?(message,ret),case(message) {
      (:set,v) ⇒ Cell(self,v)
    | (:get) ⇒ ret!(value),Cell(self,value) };

def MakeCell(self,value) = Cell(self,value);
```

The Cell class is encoded by two definitions. The first one, named Cell like the class name, is the encoding of the instance behavior. The second one is named MakeCell, it encodes the constructor. For this simple example, both definitions match. The first parameter is a channel named self. There is also one parameter for each slot entry (here, only one parameter for the value slot). The parameters are only known to the body of the definition, implementing *data hiding* as required. Each Cell process is waiting on self for a message as well as a private reply channel named ret. The message is a tuple containing the name of the method, represented by a symbol prefixed by a colon (internalized as a unique integer), and the optional parameters. It is dispatched using the case construct which pattern-matches on tuples. If the method is :set, then we recursively call the Cell definition with updated value parameter. We see here that the static behavior of objects is obtained through recursion (as in [9]). The :get method sends on the ret channel the current value of the slot and then loops. Inheritance is obtained through reactive composition and delegation as follows:

```
def CellPrint(self,super) =
   self?(message,ret),
     case(message) {
       (:print) ⇒ tnew(sc),react(sc),super!(:get,sc),sc?(v),
                  #print(v),CellPrint(self,super)
     | _ ⇒ super!(message,ret),CellPrint(self,super) };

def MakeCellPrint(self,value) =
  tnew(super), [ react(super),MakeCell(super,value) ||
                 CellPrint(self,super) ];
```

The MakeCellPrint constructor takes a value parameter which is used to start a (reactive) Cell process identified as super. Another reactive process executing the CellPrint definition is also started. Two links must be provided: the self channel for the subclass part and the super channel for the superclass. Before explaining the dispatch mechanism, let us encode the main program as follows:

```
tnew(c), [ react(c),MakeCellPrint(c,10) ||
  tnew(rc),react(rc),c!((:get),rc),rc?(v),
  #print(v),c!((:set,12),rc),c!((:print),rc) ]
```

The channel c identifies the object we create. The CellPrint constructor is started in parallel with the main program. In order to call method get, we send on c the (:get) dispatch message, and also the return channel that we name rc. This message activates the CellPrint process which is reacting on c (also known as self in the definition of CellPrint). The message is dispatched to the default case _, which is delegated to the Cell process reacting on the super channel (known as self within the Cell definition). A second dispatch leads to the output of the slot value on ret which is in fact rc. Thus, the answer goes directly back to the first caller. This encoding is roughly of the same size if compared to the pseudo object-oriented code above. For developers exercised to name-passing programming, it is also almost as readable.

In table 7, we compare the active and reactive encodings for examples of object-oriented programs coming from (an old version) of the computer language shootout [1]. We can see that applying the reactive semantics on the objinst example does not accelerate the computation in a noticeable way. The reason is that this example performs many method calls on very few objects, which is not a problem for the scheduler. The threads-flow example (also called process in the shootout), on the contrary, uses many objects and many threads, which gives a big acceleration in the case of reactive semantics. The comparison with python, which we should take very carefully, shows that the CubeVM may be, at least at first sight, a very fast interpreter for object-oriented interactions. More experiments should be conducted to really state on that matter.

Process $P, Q, \ldots ::=$	
(inert)	0
(sum)	$\mid P + Q$
(parallel)	$\mid P \mid Q$
(prefix)	$\mid \alpha.P$
(restriction)	$\mid (\nu a)P$
(match)	$\mid [x = y]\ P$
(mismatch)	$\mid [x \neq y]\ P$
(replication)	$\mid *P$
Prefix $\alpha, \ldots$ ::=	
(silent)	τ
(input)	$\mid c(x_1, \ldots, x_n)$
(output)	$\mid \bar{c}\langle v_1, \ldots, v_n \rangle$

Table 9. Syntax of the polyadic Pi-calculus

5. Encoding the polyadic Pi-calculus

In this section, we discuss the relationship between the Pi-calculus and the applied variant we propose.

In table 9 we recapitulate the syntax of the polyadic Pi-calculus as presented in [10]. Most of the constructs are similar if not identical to the cube-calculus ones. The *replication* operator is a notable exception. In fact, it is well known that recursive definitions as implemented by the cube-calculus may be encoded using replication and communication. But the converse is also possible. Consider the following Pi-calculus process:

$$(\nu gen, c) * (gen(out).(\nu id).\overline{out}\langle id\rangle) \mid \overline{gen}\langle c\rangle.c(x).P$$

In this system, the process on the left is a generator of unique identifier. The one on the right of the parallel operator (noted | in the Pi-calculus) asks for such an identifier. In the Pi-calculus semantics, this system is structurally equivalent to the following one:

$$\begin{array}{rl} \equiv & (\nu gen, c) * (gen(\underline{out}).(\nu id)\overline{out}\langle id\rangle) \\ & \mid (\nu id)(gen(out).\overline{out}\langle id\rangle) \\ & \mid \overline{gen}\langle c\rangle.c(x).P) \text{ with } id \text{ fresh in } P \end{array}$$

The replication $*P$ is structurally congruent to $*P \mid P$ and the scope of restrictions can be extruded. The system then reduces as follows:

$$\begin{array}{rl} \longrightarrow & (\nu gen, c) * (gen(out).(\nu id)\overline{out}\langle id\rangle) \\ & \mid (\nu id)\ (\overline{c}\langle id\rangle \mid c(x).P) \\ \longrightarrow & (\nu gen, c) * (gen(out).(\nu id)\overline{out}\langle id\rangle) \\ & \mid (\nu id)\ (0 \mid P\{id/x\}) \end{array}$$

$\{\!|0|\!\} \triangleq 0$
$\{\!|P + Q|\!\} \triangleq \{\!|P|\!\} + \{\!|Q|\!\}$
$\{\!|P \mid Q|\!\} \triangleq \{\!|P|\!\} \parallel \{\!|Q|\!\}$
$\{\!|\alpha.P|\!\} \triangleq \{\!|\alpha|\!\}, \{\!|P|\!\}$
$\{\!|(\nu a)P|\!\} \triangleq \mathsf{snew}(a), \{\!|P|\!\}$
$\{\!|[x = y]\ P|\!\} \triangleq \mathsf{if}\ x = y\ \mathsf{then}\ \{\!|\ P\ |\!\}$
$\{\!|[x \neq y]\ P|\!\} \triangleq \mathsf{if}\ x \neq y\ \mathsf{then}\ \{\!|\ P\ |\!\}$
$\{\!|*P|\!\} \triangleq A_{*P}(x_1, \ldots, x_n)$ where $\{x_i\}$ are the free names of P

$\{\!|\tau|\!\} \triangleq \#\mathsf{noop}()$: a silent primitive
$\{\!|c(x_1, \ldots, x_n)|\!\} \triangleq c?(x_1, \ldots, x_n)$
$\{\!|\bar{c}\langle x_1, \ldots, x_n\rangle|\!\} \triangleq c!(x_1, \ldots, x_n)$

$\{\!|0|\!\}_{\mathsf{def}} \triangleq \emptyset$
$\{\!|P + Q|\!\}_{\mathsf{def}} \triangleq \{\!|P|\!\}_{\mathsf{def}}; \{\!|Q|\!\}_{\mathsf{def}}$
$\{\!|P \mid Q|\!\}_{\mathsf{def}} \triangleq \{\!|P|\!\}_{\mathsf{def}}; \{\!|Q|\!\}_{\mathsf{def}}$
$\{\!|\alpha.P|\!\}_{\mathsf{def}} \triangleq \{\!|P|\!\}_{\mathsf{def}}$
$\{\!|(\nu a)P|\!\}_{\mathsf{def}} \triangleq \{\!|P|\!\}_{\mathsf{def}}$
$\{\!|[x = y]\ P|\!\}_{\mathsf{def}} \triangleq \{\!|P|\!\}_{\mathsf{def}}$
$\{\!|[x \neq y]\ P|\!\}_{\mathsf{def}} \triangleq \{\!|P|\!\}_{\mathsf{def}}$
$\{\!|*P|\!\}_{\mathsf{def}} \triangleq \{\!|P|\!\}_{\mathsf{def}};$
$\mathsf{def}\ A_{*P}(x_1, \ldots, x_n) = A_{*P}(x_1, \ldots, x_n) \parallel \{\!|P|\!\};$

Table 10. Encoding of the polyadic Pi-calculus

Translated into the cube-calculus, we have to encode the replication as both a definition and a call to that definition in the translated expression. For example we can write:

```
def UniqueName(gen) = gen?(out),snew(id),out!(id)
                      || UniqueName(gen);

snew(gen,c),[ UniqueName(gen) || gen!(c),c?(x),{|P|} ]
```

More generally, the encoding of a Pi-calculus process P is divided into a set of definitions associated to replicated processes, written $\{\!|P|\!\}_{\mathsf{def}}$, and a cube-calculus process expression $\{\!|P|\!\}$. These are defined inductively on the syntax of the Pi-calculus, as shown in table 10. For completeness, we should then exhibit the fact that the reductions in the Pi-calculus are matched by corresponding reductions in the cube-calculus translation and vice-versa. But the only non-trivial part of the translation relates to replication vs. recursion, which has been thoroughly investigated in the literature (e.g. [17]). All the other Pi-calculus constructs are translated in an almost literal way.

6. Related work

The CubeVM adopts a fully interpreted architecture. In comparison, most of the implementations for variants of the Pi-calculus employ more "traditional" compilation techniques. For instance, the Pict [14] and Nomadic pict [19] implementations generate C code via an abstract machine representation. The Jocaml language [6] is designed as an extension of the Ocaml programming systems. It supports both bytecode and native compilation. Other implementations (e.g. [9]) rely on the Java Virtual Machine. The extra complexity of compiling to languages foreign to process algebras (stack, or stack-register based machine languages) seem to be motivated by the expectation of increased performance. We conduct a benchmark experiment on-line [2] but it is yet too early to draw any conclusion about the results we obtained so far. However, for most of the examples that we tried, most notably the ones involving fine-grained concurrency and name-passing features, the CubeVM implementation turned out to be quite fast for an interpreter (except for general and deeply recursive functions, as we expected). Of course, more realistic experiments should be considered to confirm this observation. We see, though, two basic factors to explain the fact that the Pi-calculus is particularly adapted for interpreted technologies. First, it is a highly-concurrent language, which makes almost unavoidable the implementation of a dedicated scheduling algorithm. Embedding the scheduler in a compiled program or in a generic interpreter should not make much difference. The second reason is that actual operating systems and hardware rely on machine languages that are alien to the Pi-calculus concepts and semantics. As such, most of the high-level optimizations (such as the reactive semantics we propose in this paper) must be found and applied on the calculus itself at compile-time. In consequence, we think (and envisage as a future work) that compilation to native code should be performed just-in-time.

In term of expressivity at the language level, there is a difference between the variant of the Pi-calculus that we implement compared to most of the related work. For instance, the cube-calculus we propose is closer to the original Pi-calculus. It allows output prefixing (which is disallowed in [6, 14, 19, 9]) and does not restrict the passing of name capabilities. However, unlike its competitors, the cube-calculus is a bytecode language and is as such quite low-level.

Most runtime systems for dynamic programming languages support some form of garbage collection. Our requirement is to support an efficient and highly concurrent GC. Tracing algorithms (mark and sweep, semispace-copying and mark-and-compact) are the most widely spread. To our knowledge, the first implementation to deal with cycles in a reference counting GC is Rob Pike's newsqueak [15]. The simple idea is to avoid cycles by enforcing value-passing semantics. But it is also a source of inefficiency because values must be copied each time they are updated. More recent works show that reference counting collection with cycle detection may also be implemented both concurrently and efficiently in an object-oriented setting [4]. This algorithm still relies on the buffering of potential roots and a somewhat non-trivial three-color algorithm. Additionally, it deals only with objects, which are passive data structures. In this paper, we show that the GC algorithm of the CubeVM is so simple that it can be fully characterized in the structured operational semantics of the cube-calculus. It is then easy to implement the GC and, even more importantly, reason about it. Moreover, the CubeVM GC reclaims processes and not just passive objects. A similar feature is proposed in the framework of actor systems [20]. The problem with actors is that only the senders of messages know the receivers. This means that actor references must be inverted in order to detect deadlocks on the receivers' side. As explained in [20], the inversion of references consumes time and resources. And then standard collection tracing schemes must be employed, which rely on an explicit reachability graph. Comparatively, in the CubeVM, channel references (and count) are known by both the sender and receiver processes on these channels, so no costly reference inversion or construction of reachability graphs are necessary.

There are multiple ways to interpret the word "stackless". In *Stackless python* [18] and other common interpreters, it means that the C (and thus processor) stack is not exploited so that stack frames are explicitly constructed and manipulated. The specifications for the *Scheme* programming languages enforce *tail-call elimination* [11], which is another way to bypass the stack when possible. The case for the CubeVM is different. As we show in this paper, the cube-calculus has recursion but does not require any stack support at all. When an explicit stack is absolutely needed, for example in the case of non-primitive recursive functions, channels and processes can be used to simulate the stack frames.

Many Pi-calculus encodings of functional, dataflow or object-based systems do not need the fully concurrent and non-deterministic nature of its operational semantics. In Jocaml [6], the syntax of the language is restricted so that the receiver for a given channel is known at compilation time. But the receiver capability may not be sent, which somewhat cuts the Pi-calculus heritage in half. With this limitation (that we find important) in mind, this allows to increase the performance of the scheduling code because a sender process does not have to "look for" a corresponding receiver. In the CubeVM, we propose the reactive semantics for almost the same purpose. The main difference is that they do not restric the language in term of expressive power. In both active and reactive semantics, the input capability may be passed between processes, as in the pure Pi-calculus.

7. Conclusion and future work

The CubeVM we describe in this paper is an interpreter for a bytecode language that is very close to the Pi-calculus. The main motivation behind this work is to study in practice the expressivity and usefulness of programming languages inspired by the theories of concurrency and mobility. The Pi-calculus and variants, that are still being studied from a theoretical perspective, represent in our opinion a whole new world to discover.

But beyond the experience, our objective is not to develop yet another toy prototype, inefficient and/or buggy, which is in fact rather easy to embed in an existing language (but few languages support the large number of threads needed by even simple Pi-calculus programs). Moreover, we show in this paper that the resource management scheme involved in name passing calculi is very different from more traditional bytecode languages for stack- or stack/register-based virtual machines. New problems appear such as the need for dedicated and efficient scheduling algorithms. The reactive part of the semantics shows that, under some assumptions (namely the fact that the process definition is in chain-reaction form), the scheduler may be bypassed. But old problems also disappear, most notably the need to exploit and reason about control stacks. This mainly comes from the fact that the local environment of processes is flat, and that all recursive calls are in tail position. We think that most of the simplicity and efficiency of the resource management scheme implemented by the CubeVM come from this stackless architecture. Our vision is that the stack represents, in terms of control, what separates the most *executions* (or running programs) from (non-running, static) programs. Our long-term goal is that of ubiquitous *mobility* for which dealing with executions and moveable execution environments are most significant, if compared to manipulating (source-code or binary) programs.

At the language level, we discuss various process forms in this paper: waiting or isolated processes, chain-reaction forms and so on. A natural follow-up would be to investigate type systems to detect these forms of processes, most notably chain-reactions, statically and automatically. For instance, the react prefix we present in the paper, and which triggers the (fast) reactive semantics could be inserted automatically at compile-time. We developed, separately, an even stricter form to characterize fully deterministic (and not just sequential) systems. These forms and their detection through static typing could serve as a basis for an efficient JIT extension.

In a complementary work we develop an efficient implementation of the distributed extensions for the cube-calculus. These rely on a semantic model called the *Interaction Spaces* that we discuss precisely in [13].

References

[1] The computer language shootout. `http://shootout.alioth.debian.org/`.

[2] The CubeVM project. `http://www-poleia.lip6.fr/~pesch/cube`.

[3] The Python programming language. `http://www.python.org`.

[4] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *ECOOP*, volume 2072 of *LNCS*, pages 207–235. Springer, 2001.

[5] J. Bergstra, A. Ponse, and S. Smolka, editors. *Handbook of process algebra*. Elsevier, 2001.

[6] L. M. Cédric Fournet, Cosimo Laneve and D. Rémy. Implicit typing à la ML for the join-calculus. In *Proc. of the 1997 8th International Conference on Concurrency Theory*. Springer-Verlag, 1997.

[7] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[8] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[9] L. Lopes, F. Silva, and V. T. Vasconcelos. A virtual machine for the TyCO process calculus. In *PPDP'99*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag, Sept. 1999.

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

[11] I. N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, J. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[12] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference, 1996.

[13] F. Peschanski. Mobile agents in interaction spaces. In *Foundations of Coordination Languages and Software Architectures*, ENTCS. Elsevier, Aug. 2005.

[14] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[15] R. Pike. The implementation of newsqueak. *Software - Practice and Experience*, 20(7):649–659, 1990.

[16] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[17] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[18] C. Tismer. Continuations and stackless python or "how to change a paradigm of an existing program". In *Proceedings of the 8th International Python Conference*, 2000.

[19] A. Unypoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *POPL 2001. ACM Sigplan Notices*, volume 36, March 2001.

[20] A. Vardhan and G. Agha. Using passive object garbage collection algorithms for garbage collection of active objects. In *ISMM'02: Proceedings of the 3rd international symposium on Memory management*, pages 106–113, New York, NY, USA, 2002. ACM Press.

[21] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.