Parallel Computing with the Pi-calculus

Frédéric Peschanski

Parallel Computing with the Pi-calculus

Frédéric Peschanski
Université Pierre & Marie Curie UPMC - LIP6 - Paris, France
first.last@lip6.fr

Abstract

To tackle the multi-core programming challenge, we investigate the design and implementation of concurrency-oriented programming languages. Our approach mimics the evolution from lambda-calculi to functional programming languages, but with the pi-calculus as a starting point. To fill the gap between the abstract calculus and its implementations, we introduce the pi-threads: an intermediate language and its abstract machine.The stackless architecture of the abstract machine makes the underlying algorithms both simple and naturally concurrent. The scheduling, for instance, can be operated in a completely decentralized way. Another remarkable feature of the abstract machine is its garbage collector. We adopt a reference counting scheme that can be characterized formally using only two semantic rules. Moreover, it provides original solutions to the usual shortcomings of reference counting: the overhead caused by the maintenance of the reference counts - we only track global references - and the complex issue of collecting cyclic structures - reinterpreted as the (in our case, much simpler) problem of detecting partial terminations.

1. Introduction

The democratization of *multi-core* technologies has a profound impact on the design of programming languages [18]. There is the *pragmatic* way of adding extra layers of abstractions and associated mechanisms to existing languages. We favor the complementary way of establishing the language principles on minimal and solid theoretical foundations. Indeed, our approach mimics the evolution from abstract λ-calculi to functional programming languages, but taking the π-calculus [9, 17] as a starting point. The choice of the π-calculus is mainly driven by the following considerations: (1) it is a minimal language with concurrency at its core, (2) it is very expressive in that many computational structures (functions, objects, etc.) can be encoded in a concise way and (3) it benefits from a large body of theoretical works. However, despite the success of the theory, there is a surprising lack of practical tools both for software engineering (modeling, verification, etc.) and programming based on the π-calculus.

In this paper we introduce the π-threads, an intermediate language and associated abstract machine to fill the gap between the abstract calculus and the implementations. This applied calculus is realized as a virtual machine named the *Parallel Commitment Machine* (PCM). In this paper we focus on the abstract machine but we also present more informally the basic algorithms driving the PCM. One remarkable characteristic of these machines is their *stackless architecture*. The stack is a support for sequential computations, and it is always troublesome to deal with it in a concurrent setting. The common possibilities are to either share or distribute the stack. Our approach is radical: we simply remove the (need for a) stack ! Indeed, a stack is only required for a language involving nested scopes. And the unbounded size of the stack comes from non-tail recursive calls. Our intermediate language - and indeed the π-calculus itself - has none of these. Of course, we do not say we avoid completely the stack, only we have complete freedom about how it may be employed (cf. *chain reactions* in [13]). The gain is important: we can design very simple - and naturally parallel - scheduling and resource management algorithms. The scheduler, for instance, operates in a completely decentralized way. This is thanks to the use of *explicit commitments* describing why the waiting processes are waiting, and how they can be awaken. Another remarkable feature of the abstract machine is its garbage collector. We adopt a reference counting scheme that can be characterized formally using only two semantic rules. Moreover, it provides original solutions to the usual shortcomings of reference counting: the overhead caused by the maintenance of the reference counts - we only track global references - and the complex issue of collecting cyclic structures - reinterpreted as the (in our case, much simpler) problem of detecting partial terminations.

The outline of the paper is as follows. In Section 2 we introduce the π-calculus variant that we use as an intermediate language in our compilation toolchain. In Section 3 we describe the π-threads abstract machine and its operational semantics. Some critical properties of the semantics are also discussed. By lack of space, we only provide proof outlines. An overview of the PCM virtual machine is proposed in Section 4. In particular, we describe its overall architecture, the principles of the scheduling algorithm and its garbage collector. The related works are discussed in Section 5.

2. The π-calculus as an intermediate language

Definition		**def** $D(x_1,\ldots,x_n) = P$		
Process	P	::=	**end**	Termination
			$\mid \sum_i [g_i]\alpha_i, P_i$	Guarded choice
			$\mid D(v_1,\ldots,v_n)$	Call
Prefix	α	::=	**tau**	Silent
			$\mid c!v$	Output
			$\mid c?(x)$	Input
			$\mid$ **new**(c)	Channel creation
			$\mid$ **spawn**$\{P\}$	Thread creation

Table 1. Syntax of the π-threads language

The syntax of the π-threads calculus is summarized in Table 1. It corresponds to an applied variant of the π-calculus, biased towards efficient operationalization. The atomic actions of the languages are

the creation of communication channel c as **new**(c), the output c!v of value v on channel c and the input c?(x) on channel c of a value bound to variable x. We adopt a CSP-like syntax because of its elegant textual representation. Note, however, the extra parentheses around variables to emphasis their use as *binders*. The tau action is silent, like a no-operation instruction. The creation of a new process is also an atomic action **spawn**$\{P\}$ where P is the process expression to run in parallel.

The construct $[g]\ \alpha.P + [h]\ \beta.Q$ is the non-deterministic choice between two execution branches, with boolean guards g on the left, or h on the right. Each possible branch of execution must begin with an atomic action, followed by a continuation (a process expression). The informal semantics - slightly different from the original π-calculus - is as follows. First, the left branch is tried and for this the guard g is evaluated[1]. If it evaluates to true then the atomic action α is tried. If the latter is a **tau**, a **new** or a **spawn**, then it is executed right away and the whole process continues as P. If it is an output c!v then the action is only executable if there is a process running in parallel, itself ready for receiving on channel c, e.g. c?(x).R. In this case the choice process will continue as P and the parallel process will continue as R but with x bound to the evaluation of the sent value, which we will note $\bar{v}$. If either the guard g evaluates to false or the action α is not executable, then the second branch is tried instead following the same principles. If both branches are disabled - i.e. not executable - then the whole choice is blocking until one of the branch become executable. The choice operator is generalized to an arbitrary (albeit finite) number of branches, as $\sum_i [g_i].P_i$. As a convenience we will write $[g]\alpha.P$ a choice with only one branch, and also often abbreviate [true] $\alpha.P$ as $\alpha.P$.

In the π-calculus there is no termination construct *per se*, however we need the **end** terminator to take care of some clean-up in the operational semantics. Note that in many examples we omit the terminator and implicitly assume its presence.

Finally we provide an explicit construct for parametric definitions and (possibly recursive) *terminal calls*. For example, the famous *Fibonacci* function can be expressed as follows:

```
def Fib(n m p : int, r : chan<int>) =
    [n=0] r!m,end + [true] tau,Fib(n-1,m+p,m,r)
```

The Fib definition is parameterized by three integer and a channel r conveying the return value as an integer. Note that we added extra type annotations in the example, which helps the understanding. The type system, quite standard, will not be detailed for the sake of concision. As explained previously, the choice is *locally ordered* in that the leftmost branch will be tested before the right one. This makes very easy the encoding of alternatives :

$$\left[\begin{array}{l} \textbf{if } C \textbf{ then } P \textbf{ else } Q \stackrel{\text{def}}{=} [C]\textbf{tau},P + [\text{true}]\ \textbf{tau},Q \\ \textbf{if } C \textbf{ then } P \stackrel{\text{def}}{=} [C]\textbf{tau},P \end{array}\right.$$

Thanks to this derived syntax, we can give a more classical definition of *Fibonacci* :

```
def Fib(n m p : int, r : chan<int>) =
    if n=0 then r!m else Fib(n-1,m+p,m,r)
```

[1] In this paper, we do not detail the syntax and formal semantics for boolean expressions used in guards, and more generally for values of basic datatypes: integers, etc. The reason is that this is an orthogonal issue and there is nothing significantly remarkable at that level.

The syntax enforces all the calls to be in tail position[2]. Non-tail calls must be encoded using channels and processes. We illustrate the principles with the other famous *Ackermann* function :

```
    def Ack(n p : int, r : chan<int>) =
if n=0 then r!(p+1) else if p=0 then Ack(n-1,1,r)
else new(r1:chan<int>), [ Ack(n,p-1,r1) || r1?(pp),Ack(n-1,pp,r) ]
```

The third branch of the choice corresponds to the "doubly" recursive case of the famous function. The nested call is spawned in parallel with the process that waits for the result of the inner computation, which enables the continuation of the computations[3]. We use here a composition operator || that can be easily derived from **spawn** and prefixing. It might seem quite inefficient to encode such a deterministic computation using processes and channels. In [13] we explain how to compile such forms - named *chain reactions* - so that they can exploit the processor stack.

Beyond functions, there are many programming abstractions that can be encoded concisely using the π-calculus. An example is the domain of *dataflow* systems. A dataflow for computing *Fibonacci* numbers can be written as follows:

```
def FFib(r : chan<int>) = r!1,r!1,FFib'(1,1,r)
def FFib'(n m : int, r :chan<int>) = r!(n+m),FFib'(m,n+m,r)
```

Here the definition FFib generates a flow of integers on the channel r, which incrementally returns the terms of the *Fibonacci* suite: 1 1 2 3 5 8, etc. Another classical example is the (concurrent) computation of prime numbers:

```
def Gen(i n p : int, gen : chan<int>) =
  if i<n then gen!i,Gen(i+p,n,p,gen)

def PrimeFilter(n : int, in out : chan<int>) =
  in?(x), if (x % n) = 0 then PrimeFilter(n,in,out)
          else out!x,PrimeFilter(n,in,out)

def EndFilter(in primes : chan<int>) =
  in?(x), primes!x, new(out:chan<int>),
  [ EndFilter(out,primes) || PrimeFilter(in,x,out) ]

def Primes(primes : chan<int>, nb : int) =
  new(gen:chan<int>),
  [ gen!2,Gen(3,nb,2,gen) || EndFilter(gen,primes) ]
```

In dataflows, it is very easy to obtain modular systems, here a stream of even integers is produced (definition Gen), which is connected to a chain of PrimeFilter processes, terminated by an EndFilter process. The is one filter for each prime number discovered so far. Any odd integer going through the whole chain of filters is thus a prime number. Actually we could stop at the filter for the smallest prime greater than the square root of the candidate, but the structure would be slightly more complex. The interest of such an architecture is that it is indeed highly concurrent: multiple candidates can be tested in parallel, forming a pipeline along the chain of filters.

More classical *patterns* for concurrency can be of course encoded in the proposed language. As an example consider the implementation of *critical sections* running in *mutual exclusion* (e.g. to ensure the exclusive access to a given channel). The primitive mean of synchronization in the π-calculus is the *communication channel*. The synchronous semantics of communications allow to encode concurrent locking very easily. A channel is basically a

[2] Allowing only non-tail calls in the language might appear as quite a restriction. It is important to remind, though, that this constraint only applies at the level of the intermediate language. It is easy to offer non-tail calls as a derived construct.

[3] The structure of a functional continuation corresponds to a process waiting on a private channel that marks the end of the previous step of the computation.

lock. In order to "acquire" the lock, one must read on the channel, which will block until someone writes on it (or *vice-versa*). Such a write operation allows to"release" the lock. Following these principles we can provide a canonical structure for critical sections:

```
def SectCrit(lock : chan<>) = lock?,/* Crit. Sect. */,lock!
```

The type chan<> is used for "pure" synchronization as in CCS, the ancestor of π. Note that the critical section is executed in mutual exclusion only if the lock channel is used correctly. For example, if a process releases the lock twice, then two critical sections could be executed in parallel. A more robust version of the protocol can be obtained by encapsulating the lock in a process, for example as follows:

```
def Lock(lock : chan<chan<>>) =
    new(release),lock!release,release?,Lock(lock)
```

The definition for critical sections is modified accordingly:

```
def SafeSectCrit(lock : chan<chan<>>) =
    lock?(release),/* Critical Section */,release!
```

When the lock is acquired (synchronization on the lock channel), a private channel release is received by the client. The communication of private channels is the most distinctive feature of the π-calculus. The type of the channel lock is emblematic of this feature: chan<chan<>>. The protocol is more robust because there is less sharing, but one still has to be careful not to communicate the private channel, which can be enforced by a type discipline [17].

It is also very natural to encode *active objects* using the + operator as a method selector. We illustrate this with another classical concurrent programming pattern. The principle is to allow a bounded number of tasks to run in parallel. If the pool is full, then the other tasks must wait. The definition is as follows:

```
def TaskPool(nb : int, enter :chan<chan<>>, leave : chan<>) =
    leave?,TaskPool(nb+1,enter,leave)
    + [nb>0] enter?(release),spawn{release?,leave!},
             TaskPool(nb-1,enter,leave) ]
```

The idea of the construction is to use distinct channels to discriminate the methods to select. Here we implement two methods: the enter method to gain access to the task pool, and the complementary leave method. The choice is locally ordered, and we exploit this to put a priority on leaving the task pool. In order to enter the pool, one must obtain a permit between 0 and nb. A permit is given back to the task pool using a private channel release provided by the tasks, which must also follow the acquire/release protocol:

```
def Task(enter : chan<>) =
    new(rel:chan<>),enter!(rel), /* task behavior */, rel!
```

Once again we exploit the distinguishing feature of the π-calculus: passing (private) channel references among processes.

As a summary, the proposed language - an applied variant of the π-calculus - allows consise encodings of various computational structures: functions, (active) objects, dataflows and concurrency patterns. Of course, this is still quite a low-level language and we only motivate its use as an intermediate language in a compiler toolchain.

3. The π-threads abstract machine

We introduce in this section the abstract machine that provides the operational semantics of the π-threads calculus. We adopt a process calculus presentation of the abstract machine to remain close the π-calculus. The syntax of the process expressions is the one of Table 1, with the extension of a **wait** prefix for internal usage.

The terms manipulated in the semantics are parallel processes – the π-threads – put in the context of an agent (i.e. a scheduler). The latter has the following form:

$$\Delta \vdash \Pi_1 \parallel \ldots \parallel \Pi_n$$

where each Π_i is a π-thread of the form $[\Gamma_i; \delta_i] : P_i$. A little bit more concisely, we will denote an agent as follows:

$$\Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i$$

The Δ component represents the *global environment* of the π-threads. This is the set of unique channel identities, the only globally shared information. Each π-thread is associated to a corresponding process expression[4] P_i. The latter evolves in a local context composed of a set of *commitments* Γ_i and a lexical environment δ_i, which as usual binds injectively variables to values. Note that as announced there is no nesting allowed: there can be at most one binding for a given variable[5].

A fundamental characteristic of the semantics is the explicit advertising by the processes of their commitments regarding their potential interactions. This is how we can "play one move in advance" to implement the choice operator. Each element of the commitment set Γ_i is either:

- an *output commitment* $\hat{c} \Leftarrow v : Q$ of the process to emit the value v on the channel identified as $\hat{c}$ with the continuation Q, or
- an *input commitment* $\hat{c} \Rightarrow x : Q$ to receive on channel $\hat{c}$ with the received value bound to variable x in the continuation Q.

It is possible to compute static bounds for the sizes of the local environment and the commitment set of a π-thread using the functions defined in Table 4. The function esize computes the (tight) bound for the size of the local environment required to call a given definition. The principle, roughly, is to count the number of free variables in the body of the definitions. For the choice operator we take the maximal size of the different branches. The size of its commitment set can be obtained in a similar (but simpler) way using the function csize. Informally, the number of commitments is bound by the number of input/output prefixes of the largest choice.

PROPOSITION 1. *The pre-allocated sizes of the local environment and the commitment set can be computed statically*

This is thanks to the purely syntax-driven definition of the functions esize and csize. Practically, we calculate the bounds at compile-time and attach them (in the generated code) to the definitions of the compiled program. When spawning a new π-thread, we can preallocate its environment and commitment set using the bounds calculated for the definitions it references.

3.1 Basic rules

The basic rules for the operational semantics of the calculus are summarized in Table 2. Unlike the traditional *structured operational semantics* (SOS) presentation, all the rules take place in the global context of a whole agent. The first rule (par) is a purely formal artifact: it simulates in the semantics the independence of concurrent processes. In the rule A and B represent an arbitrary

[4] Precisely, a *process* is a well-formed expression according to the rules of Table 1. A *π-thread* is the adjunct of such a process expression to a local environment and a set of commitments. It is thus a process in its running environment. However, this distinction is not very important so in the remaining of the paper we use interchangeably the terms π-thread, thread and process to roughly express the same thing.

[5] The lack of nesting in environments makes the π-calculus, in a way, simpler than its main inspirator the λ-calculus.

$$\frac{\Delta \vdash A \parallel B \to \Delta' \vdash A' \parallel B}{\Delta \vdash B \parallel A \to \Delta' \vdash B \parallel A'}\ (par) \qquad \frac{\overline{g} = \text{true}}{\Delta \vdash [\Gamma;\delta] : [g]\mathsf{tau}.P + \sum_i Q_i \parallel B \to \Delta \vdash [\emptyset,\delta] : P \parallel B}\ (step)$$

$$\frac{\overline{g} = \text{true}}{\Delta \vdash [\Gamma;\delta] : [g]\mathsf{new}(c).P + \sum_i Q_i \parallel B \to \Delta, \hat{c} \vdash [\emptyset;\delta, c \triangleright \hat{c}] : P \parallel B}\ (new)$$

$$\frac{\overline{g} = \text{true}}{\Delta \vdash [\Gamma;\delta] : [g]\mathsf{spawn}\{P\}.Q + \sum_i R_i \parallel B \to \Delta \vdash [\emptyset;\delta] : P \parallel [\emptyset;\delta] : Q \parallel B}\ (spawn)$$

$$\frac{\overline{g} = \text{false} \quad \Delta \vdash [\Gamma;\delta] : \sum_i Q_i \parallel B \to \Delta' \vdash [\Gamma';\delta'] : R \parallel B'}{\Delta \vdash [\Gamma;\delta] : [g]P + \sum_i Q_i \parallel B \to \Delta' \vdash [\Gamma';\delta'] : R \parallel B'}\ (next)$$

$$\frac{\mathsf{def}\ D(x_1,\ldots,x_n) = P}{\Delta \vdash [\Gamma;\delta] : D(v_1,\ldots,v_n) \parallel B \to \Delta \vdash [\emptyset; x_1 \triangleright \overline{v_1}, \ldots, x_n \triangleright \overline{v_n}] : P \parallel B}\ (call) \qquad \frac{}{\Delta \vdash [\Gamma;\delta] : \mathsf{end} \to \Delta \vdash \emptyset}\ (inert)$$

Table 2. Operational semantics: basic rules

$$\frac{\overline{g} = \text{true} \quad \delta(c) = \hat{c}}{\Delta \vdash [\Gamma;\delta] : [g]c!v.P + \sum_i Q_i \parallel [\Gamma', \hat{c} \Rrightarrow x\colon R; \delta'] : S \parallel B \to \Delta \vdash [\emptyset;\delta] : P \parallel [\emptyset; \delta', x \triangleright \overline{v}] : R \parallel B}\ (send)$$

$$\frac{\overline{g} = \text{true} \quad \delta(c) = \hat{c} \quad \hat{c} \notin \bigcup_j \Gamma_j}{\Delta \vdash [\Gamma;\delta] : [g]c!v.P + \sum_i Q_i \parallel \prod_j [\Gamma_j;\delta_j] : R_j \to \Delta \vdash [\Gamma, \hat{c} \Lleftarrow v\colon P; \delta] : \sum_i Q_i + \mathsf{wait} \parallel \prod_j [\Gamma_j;\delta_j] : R_j}\ (out)$$

$$\frac{\overline{g} = \text{true} \quad \delta(c) = \hat{c}}{\Delta \vdash [\Gamma;\delta] : [g]c?(x).P + \sum_i Q_i \parallel [\Gamma', \hat{c} \Lleftarrow v\colon R; \delta'] : S \parallel B \to \Delta \vdash [\emptyset; \delta, x \triangleright \overline{v}] : P \parallel [\emptyset;\delta'] : R \parallel B}\ (recv)$$

$$\frac{\overline{g} = \text{true} \quad \delta(c) = \hat{c} \quad \hat{c} \notin \bigcup_j \Gamma_j}{\Delta \vdash [\Gamma;\delta] : [g]c?(x).P + \sum_i Q_i \parallel \prod_j [\Gamma_j;\delta_j] : R_j \to \Delta \vdash [\Gamma, \hat{c} \Rrightarrow x\colon P; \delta] : \sum_i Q_i + \mathsf{wait} \parallel \prod_j [\Gamma_j;\delta_j] : R_j}\ (in)$$

Table 3. Operational semantics: rules for communication

$$\begin{cases} \text{esize}(\mathsf{def}\ D(x_1,\ldots,x_n) = P) \stackrel{\text{def}}{=} \text{esize}^n_{\{x_1,\ldots,x_n\}}(P) \\ \text{esize}^n_V(\mathsf{end}) \stackrel{\text{def}}{=} n \\ \text{esize}^n_V(\sum_i [g_i]\alpha_i, P_i) \stackrel{\text{def}}{=} \max_i\{\text{esize}^n_V(\alpha_i, P_i)\} \\ \text{esize}^n_V(E(v_1,\ldots,v_m) \stackrel{\text{def}}{=} \max(n, \text{esize}(\mathsf{def}\ E(y_1,\ldots,y_m) = Q)) \end{cases} \qquad \begin{cases} \text{esize}^n_V(\mathsf{c?(x)}.P) \stackrel{\text{def}}{=} \begin{cases} \text{esize}^{n+1}_{V\cup\{x\}}(P) \text{ if } x \notin V \\ \text{esize}^n_V(P) \text{ otherwise} \end{cases} \\ \text{esize}^n_V(\mathsf{new(c)}.P) \stackrel{\text{def}}{=} \begin{cases} \text{esize}^{n+1}_{V\cup\{c\}}(P) \text{ if } c \notin V \\ \text{esize}^n_V(P) \text{ otherwise} \end{cases} \\ \text{esize}^n_V(\alpha, P) \stackrel{\text{def}}{=} \text{esize}^n_V(P) \text{ otherwise} \end{cases}$$

$$\begin{cases} \text{csize}(\mathsf{def}\ D(x_1,\ldots,x_n) = P) \stackrel{\text{def}}{=} \text{csize}(P) \\ \text{csize}(\mathsf{end}) \stackrel{\text{def}}{=} 0 \\ \text{csize}(\sum_i [g_i]\alpha_i, P_i) \stackrel{\text{def}}{=} \max(\sum_i \text{csize}(\alpha_i), \max_i\{\text{csize}(P_i)\}) \\ \text{csize}(E(v_1,\ldots,v_m) \stackrel{\text{def}}{=} \text{csize}(\mathsf{def}\ E(y_1,\ldots,y_m) = Q)) \end{cases} \qquad \begin{cases} \text{csize}(\mathsf{c?(x)}) \stackrel{\text{def}}{=} 1 \\ \text{csize}(\mathsf{c!v},) \stackrel{\text{def}}{=} 1 \\ \text{csize}(\alpha) \stackrel{\text{def}}{=} 0 \text{ otherwise} \end{cases}$$

Table 4. Compile-time bounds

number of π-threads. Thanks to (par), it is possible to define all the other rules assuming the redexes to be located at the leftmost part of the terms. The rule $(step)$ gives an interpretation for the **tau** action as the first branch of a choice. The only condition to start executing the continuation P is that the associated guard must evaluate to true. Because it is an orthogonal issue, we do not discuss the evaluator for expressions of the basic data-types (booleans, integers, etc.). We simply denote $\overline{e}$ the evaluation of an expression e. If the guard is enabled, the **tau** itself is a non-blocking operation so the continuation is simply started. Since the thread is actually running, it is important to ensure it makes no commitment so the corresponding set Γ is emptied. Similarly to $(step)$, many of the semantic rules are explained in the context of a choice. The particular case of an isolated prefix can be encoded by a choice with a unique branch. The empty choice corresponds to **end** (0 in the π-calculus).

The rule (new) explains the creation of a new communication channel. Its effect is to create a new global identifier $\hat{c}$ which is bound to the variable c in the local environment δ. The creation of a new thread is interpreted by the rule $(spawn)$. The behavior is standard: a new child thread is started in parallel and inherits the local environment from its parent. The rule $(next)$ allows to skip a branch protected by a guard evaluating to false. As explained

previously, the proposed semantics implement a local notion of priority, which is reflected by this rule.

The calling conventions are implemented by the rule $(call)$ of the semantics. Since call sites can only be in terminal positions, the principle is to simply "jump" to the definition and overwrite the local environment with the arguments of the call (here we use a strict evaluation but a lazier variant could be proposed). As illustrated in e.g. [17], it is possible to encode the definitions and calls using the other constructs of the π-calculus, in particular the communication primitives and the parallel operator. However, the primitive tail-calls can be implemented directly in a much more efficient way. Moreover, the calls remain within the boundaries of a single thread. Last but not least, the $(call)$ rule is also an important GC barrier because all the bindings except the ones captured by the call arguments are removed. The rule $(inert)$ explains the **end** action corresponding to the normal termination of a thread. It is also an important GC barrier.

3.2 Communication

The communication and synchronization primitives are most critical in the calculus. Their semantics are described by the rules of Table 3. For the emission, we distinguish two cases. In the first one, corresponding to the $(send)$ rule, the emission/synchronization can be performed immediately. There are two conditions for this. First, the guard protecting the branch of execution must evaluate to true. Moreover, there must exist an input commitment made by another process (S in the rule) on the same channel. The local environment in the receiver is updated so that the input variable is bound to the received value. The notation $\delta_i, x \triangleright \overline{v}$ must be interpreted as an update: the variable x is bound to the value $\overline{v}$ and any previous binding is simply discarded. We remind that there is no nesting allowed, and thus no stack implied.

The (out) rule explains the case when there is no thread available with a matching input commitment. Consequently, a new output commitment is recorded and the next branch of the choice is promoted to a redex position. The injection of the **wait** action ensures that if all the branches are disabled, then the thread is ultimately suspended (a situation denoted $\sum$ **wait**). If the guard evaluates to false, then the $(next)$ rule is employed. Here we evaluate the guards before resolving the choice. An alternative would be to record the symbolic expression of the guards so that they may be lazily evaluated. However, we remark that in the proposed semantics the lexical environment can never be updated *before* a branch is activated in a choice. Put in other terms, it is useless to postpone the evaluation of the guards because they cannot change until the choice is resolved. The case for reception is characterized by the rules $(recv)$ and (in). In the proposed semantics, the communicated values are evaluated in a lazy way. This can be seen in the $(recv)$ rule: the value v is recorded as a symbolic expression in the output commitment, and it is only evaluated at the time of synchronization. In most situations , when there is more than one way to synchronize, this "evaluate-by-need" scheme is most efficient.

The interactions between the communication rules (and also $(next)$) are clearly not trivial, mainly because they underlie a notion of (local) transaction. We may study a few important properties of the protocol.

PROPOSITION 2. *In an agent* $\Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i,\ \nexists \hat{c} \in \Delta$ *s.t.* $\exists j, k (j \neq k)$ *with* $\hat{c} \Leftarrow v_j : Q_j \in \Gamma_j$ *and* $\hat{c} \Rightarrow x_k : Q_k \in \Gamma_k$

Informally, this states that all the commitments made to a channel in a given global state are of a single *polarity*, i.e. there are either output or input commitment(s) but not both at the same time. Put in other terms, the rule $(send)$ works exclusively with (in), and (out) with $(recv)$.

To study the interaction between the choice construct and the communication primitives, consider the following term:

[true] a!v,P+ [true] b!w,Q || [true] b?(x),R+ [true] a?(y),S

In the "classical" π-calculus (the reduction semantics), there are exactly two possible executions of the previous term: synchronizing either on a or b. With a *global prioritized choice* (as e.g. [14]), the situation is generally interpreted as a *deadlock*. In the interpretation we propose, the left branch of the choice is only *locally* prioritized, which means we have the following possible outcomes:

- The left-hand process first records the commitment $\hat{a} \Leftarrow v : P$ and then $\hat{b} \Leftarrow w : Q$ using the rule (out) of the semantics. It next switches to a waiting mode. The right-hand process then tries its leftmost branch and finds a partner so the synchronization on b is allowed through the rule $(recv)$. Since the guard does not evaluate to false, the $(next)$ rule cannot apply and there is no other possibility. The symmetric case, i.e. the right-hand process records its two commitments in a row, leads to the synchronization on a.
- The left-hand process records its first commitment $\hat{a} \Leftarrow v : P$ but a context switch gives the token to the right-hand process. The latter records its first commitment $\hat{b} \Rightarrow x : R$ with two possibilities in the next step: either we infer through $(recv)$ a synchronization on a for the right-hand process, or a synchronization on b initiated using $(send)$ by the left-hand process. We must also account for the remaining two symmetric cases.

By summarizing all the possibilities of executions for the example, we can see that the non-deterministic nature of the construct is preserved at the global level, even if there is some priority involved locally. Of course, the reductions involved operate at a lower-level of abstraction if we compare to the π-calculus, which is not a surprise since it is an abstract machine for the calculus.

When there is no synchronization involved, for example in the case of the **tau** prefix, the proposed semantics are more deterministic because of the local priority of the left branch of the choice. Consider as an illustration the following term:

[true] **tau**,P+[true] **tau**,Q

Here, there is no alternative but to take the leftmost branch, which would be only one of the two possibilities in the π-calculus. A purely internal choice is only interesting at the level of specifications because it is a way to abstract away from the implementations. The local priority scheme is much easier to understand from a programming perspective and its implementation is more natural and efficient. Strictly speaking, the π-threads represent more a *refinement* than a variant of the π-calculus semantics.

3.3 Garbage collection

Our work on garbage collection principles for the π-calculus started with [13]. The major step we recently achieved was to down-size the formal model of the GC to only two semantic rules, as defined in Table 5. We use a principle of *reference counting* which is best known for its easy parallelization [10] but also its non-trivial handling of cyclic collections and the usually higher consumption of resources (CPU and memory) if compared to tracing algorithms [4], mostly because of the need to maintain the reference counts.

Central to our model is the knows predicate of Table 5, which tells if a given process owns *at least* one reference to the specified channel. This allows to calculate the number globalrc of processes who actually know about the channel c. This global reference count is the only information we retain in the GC model and we do not track the total number of references to the channels (induced by aliasing in the local environments).

$$\text{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \text{dom}(\delta),\ \delta(x) = \hat{c} \qquad \text{globalrc}(\hat{c}, \Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i) \stackrel{\text{def}}{=} \sum_i \begin{cases} 1 \text{ if knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) \\ 0 \text{ otherwise} \end{cases}$$

$$\frac{\text{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \to \Delta \vdash A}\ (reclaim) \qquad \frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \text{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\Delta \vdash \prod_i [\Gamma_i; \delta_i] : \sum \textbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \to \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j}\ (stuck)$$

Table 5. Operational semantics: rules for garbage collection

The first GC rule $(reclaim)$ is self-speaking: if a given channel is not referenced by any process - a situation occurring when the globalrc for the channel is zero - then it can be safely reclaimed.

The second rule $(stuck)$ provides a general solution for the garbage collection of threads. The correct understanding of the rule requires to change the way we generally consider the garbage collection problem. The reason is that the resources we manipulate are not memory cells, pointers, etc. but only channels and processes. From this point of view, the GC problem can be reinterpreted as a *partial termination detection* issue, which is exactly what is performed by the $(stuck)$ rule. The idea really is simple: if in the global context there exists a *clique* of processes, all of them waiting for external commitments on channels only referenced by the clique itself, then the whole structure (including all its processes) is indefinitely blocking. In such a situation, the best thing to do is to reclaim the clique as a whole.

We now discuss more formally about the soundness and completeness of the proposed garbage collection scheme. This commonly relates to the notion of *reachability*. The garbage collector semantics is sound if it does not reclaim any reachable memory cell. Complementarily, it is complete if it actually reclaim all unreachable cells. In the case of the π-graphs, there are no memory cells *per se* but only channels and processes. An alternative notion of reachability must be proposed, which is as follows:

DEFINITION 1. *Let an agent $\mathcal{A} \stackrel{\text{def}}{=} \Delta \vdash \Pi_{i=1}^{n} [\Gamma_i; \delta_i] : P_i$.*
*A process $[\Gamma_j; \delta_j] : P_j (1 \leq j \leq n)$ in $\mathcal{A}$ is said **enabled** iff:*

- *it is active (i.e. $P_j \neq \sum \textbf{wait}$), or*
- *it is waiting ($P_j = \sum \textbf{wait}$) and there is a channel $\hat{c} \in \Gamma_j$ and an enabled process $[\Gamma_k; \delta_k] : P_k (1 \leq k \neq j \leq n)$ in $\mathcal{A}$ such that $\text{knows}(\hat{c}, [\Gamma_k; \delta_k] : P_k) = \text{true}$*

*A process is said **disabled** iff it is not enabled*

In this definition, the question of reachability translates to a notion related to process scheduling. A process is said enabled if either it is actually runnable (running or ready for scheduling) or if it is waiting but may become ready for scheduling in the future. In the proposed framework, this can only happen if at least one other process - itself enabled - knows about at least one of the channels that can be used to awake the waiting process. If none of these conditions is satisfied, then there is no way the process could be ever awaken, and in this case we say it is disabled.

We see that the definition has an inductive flavor: the chain of disabled processes eventually boils down to an active process, which we formalize as follows:

LEMMA 1. *If a process $[\Gamma_1; \delta_1] : P_1$ is enabled, then there exist a collection of processes $[\Gamma_2; \delta_2] : P_2, \ldots, [\Gamma_n; \delta_n] : P_n$ and channels $\hat{c_1}, \ldots, \hat{c_n}$ ($n \geq 2$) such that $\forall i,\ 1 \leq i < n,\ \hat{c_i} \in \Gamma_i \wedge \text{knows}(\hat{c_i}, [\Gamma_{i+1}; \delta_{i+1}] : P_{i+1}) = \text{true}$, and $P_n \neq \sum \textbf{wait}$*

An important hypothesis for the correct behavior of the GC semantics is to restrict the discussion to executions involving a finite number of processes within an agent. Under this assumption, the previous lemma trivially follows from Definition 1 by a simple inductive argument.

From this we can develop a soundness argument, by showing that the garbage collection semantics does not involve false positives.

PROPOSITION 3. *The $(stuck)$ rule does not reclaim any enabled processes*

First, we can see that the rule cannot reclaim active processes because the reclaimed processes are waiting (i.e. in state $\sum \textbf{wait}$). For the rest we proceed *ab absurdo*. Suppose there exists an (inactive) enabled process $[\Gamma_1; \delta_1] : P_1$ that is a member of the processes $\Pi_i [\Gamma_i; \delta_i] : \sum \textbf{wait}$ in the left-hand side of the reduction defined by the conclusion of the $(stuck)$ rule. By hypothesis of the rule $\forall \hat{c} \in \Gamma_1, \text{globalrc}(\hat{c}, \Pi_j [\Gamma_j; \delta_j] : Q_j) = 0$, i.e. all the processes knowing all the channels blocking P_1 are also to be reclaimed. Moreover, the whole chain $P_1, \ldots, P_n$ ($n \leq 2$) implied by Lemma 1 should be reclaimed also because if any of the channels $\hat{c_1}, \ldots, \hat{c_n}$ linking the $P_i's$ is known to the Q_j's, then the hypothesis of the $(stuck)$ rule would be contradicted. But this in turn contradicts in Lemma 1 the fact that $P_n \neq \sum \textbf{wait}$. In consequence the process P_1 cannot be reclaimed, which concludes the proof.

For the completeness part, we may show that it is possible to reclaim all disabled processes as a once.

PROPOSITION 4. *Given an agent $\mathcal{A} \stackrel{\text{def}}{=} \Delta \vdash \Pi_i [\Gamma_i; \delta_i] : P_i \parallel \Pi_j [\Gamma_i; \delta_i] : Q_j$ such that all the P_i's are disabled and all the $Q'_j s$ are enabled. Then we can infer $\mathcal{A} \to \Delta' \vdash \Pi_j [\Gamma_j; \delta_j] : Q_j$ by rule $(stuck)$*

Since the P_i's are all disabled and all the Q_j's enabled, by Definition 1 it must be the case that $\forall \hat{c} \in \bigcup_i \Gamma_i,\ \text{knows}(\hat{c}, [\Gamma_j; \delta_j] : Q_j) = \text{false}$ (i.e. no enabled process may "reach" a disabled one). In consequence, $\text{globalrc}((\hat{c}, \Pi_j [\Gamma_j; \delta_j] : Q_j) = 0$ which is exactly the hypothesis of the $(stuck)$ rule. This concludes the proof.

Note that we do not discuss the collection of "unreachable" cycles *per se*, since the $(stuck)$ rule is able to reclaim *any* structure of disabled processes. While as demonstrated it is *possible* to reclaim all disabled processes at once, it is not mandatory to do so and the rule $(stuck)$ allows a more incremental way of reclaiming processes. The following proposition will prove most useful when we detail the garbage collection algorithms in Section 4.3.

PROPOSITION 5. *Any disabled process can be reclaimed by the $(stuck)$ rule*

Consider an arbitrary disabled process $[\Gamma_1; \delta_1] : P_1$. We must show that there is a collection of (disabled) processes that can be reclaimed together with P_1 through the rule $(stuck)$. By Definition 1 we know that P_1 is in a waiting state $\sum \textbf{wait}$ and for any $\hat{c} \in \Gamma_i$ and any process $[\Gamma_i; \delta_i] : P_i$, $\text{knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) = true$ implies P_i is also disabled. Under the finite agent assumption, the transitive closure of such interlinked disabled process is finite. By taking the complement of this set of disabled process as the $Q'_j s$ in the conclusion of $(stuck)$, we satisfy the hypothesis of the rule and thus

```
record Agent {
  chans : Set[Channel]
  run : Queue[PiThread]
  ready : Queue[PiThread]
  wait : Queue[PiThread]
  old: Queue[PiThread]
  date: Int
}

record Channel {
  taken: Lock
  globalrc, waitrc : Int
  commits : Set[Commit]
}

record PiThread {
  env : Array[Value]
  knows : Set[Channel]
  state: { R, Y, W, O }
  commits : Array[Commit]
  clock, date : Int
}

record Commit {
  kind : { IN, OUT }
  thread : PiThread
  chan : Channel
  val: Value
  clock : Int
}
```

Table 6. PCM : main data structures

the whole clique of disabled processes can be reclaimed, together with P_1, which concludes the proof.

4. The Parallel Commitment Machine

We give in this Section an overview of the Parallel Commitment Machine (PCM), which is a parallel implementation of the π-threads calculus. The challenge is to decentralize the control of the abstract machine. In fact, there are only a few places where some centralization is implied by the operational semantics. The first place is in the communication rules, when a process must look for a partner in order to synchronize. In the worst case it has to ask every other processes in the agent. To decentralize this knowledge, a simple but effective idea is to "inverse" the commitment relation and record in each channel the commitments made by processes on it. Using this information, the lookup phase may now be performed in constant time because it suffices to take an arbitrary commitment made on the channel to find a partner. There is a drawback, though, because when a process is ready for execution, it has to remove all its commitments and this knowledge may be spread over several channels. A very effective work-around is to employ a simple lazy invalidation scheme based on logical clocks. The $(stuck)$ garbage collection rule also involves a non-local knowledge, but there is a relatively simple way of reconstructing this knowledge in an incremental (and parallel) way. For concurrency control, the general principle is extremely simple: the channels must be accessed in a mutually exclusive way by processes, and that is almost all about it. We now give a few details about how these principles can be actually implemented.

4.1 Architecture

The architecture of the PCM can be presented with a few data structures and associated algorithms. The only global-level entity is the *agent*, as described in the top-leftmost pseudo-code of Table 6. An agent manages a set of channels (field chans) accessed in parallel by a set of threads. The main goal of the agent is to support the fast scheduling of the threads. The threads are distributed in four different scheduler queues, depending on their running state. The actually running threads are placed in the run queue. The size of this queue corresponds to the number of "physical" threads available in the underlying hardware. In a micro-threading situation it is a singleton. The ready queue contains the threads that are next planned for execution. In contrast, the threads in the wait queue cannot be executed right away, i.e. they have deposited some commitments. Finally, the old queue contains waiting threads of the second generation. The field date allows to decide whether a waiting thread is in the first or the second generation. These relate to the garbage collection algorithm presented below.

A communication channel, as described by the Channel record, operates similarly to a lock for mutual exclusion. The underlying implementation should employ efficient machine-level atomic conditional updates (e.g compare-and-swap operations) because the threads only access to the channels for a very short period of time. The field globalrc is the global reference count for the considered channel, as described in Table 5. The GC also requires to maintain the number waitrc of the waiting processes having at least a reference to the channel. A useful invariant to remember is waitrc $\leq$ globalrc. Finally, the commitments made on the channel are recorded in the commits field. This information is already available in the threads but as explained before we need this redundancy for optimal scheduling.

The implementation of a thread corresponds to the PiThread structure of Table 6. Each thread has a local environment, a set of registers and a special knows set that implements locally the knows predicate of the GC semantics (cf. Table 5). As expected from the formal semantics, there is no stack structure involved, and the size of all the components except knows can be computed at compilation time (cf. Proposition 1). The commitments made by the thread are recorded in the set commits. The clock field is a logical clock for lazy commitment invalidation. When a thread makes a commitment, the value of the clock is saved in the Commit structure.. A commitment is only valid if its clock value matches the one of the committing thread. It is thus only required to increment the clock to invalidate all the previous commitments made by the thread when it is awaken. Such invalid commitments can be reclaimed in a lazy way when a thread gets access to the corresponding channel structure, in general in the scheduling code. The date counter marks the time when the thread entered the wait queue of the scheduler. After some time, the threads with an old date will enter the old queue for potential garbage collection (see below). Finally, the next instruction to execute is pointed by the program counter pc.

4.2 Scheduler

The scheduler of the PCM is most remarkable by its decentralized control. The main idea is to select the π-threads from the ready queue and give them access to the CPU resources by putting them in the run queue. Of course, not all threads are always ready to run. A simple but useful invariant to remember is that a thread $[\Gamma;\delta] : P$ can only be in the ready or run queue if $\Gamma = \emptyset$. Complementarily, it can only be in the wait or old queue if $\Gamma \neq \emptyset$, which means it actually has some valid commitment deposited. Now, the decision to switch from a ready state to a wait state (and conversely) can be taken in a purely local way:

- if a (running) thread tries to perform an emission or a reception on a given channel ch, it has to find a (waiting) partner with some matching commitment. This information can be obtained almost instantaneously by inspecting the commitments made on the channel ch by fetching an arbitrary valid commitment. If there is no such valid partner then the thread deposits its commitments and goes into the wait queue. Because of the lazy invalidation of commitments, the lookup is not strictly speaking performed in constant time. It may be necessary to remove a set of invalid commitments before actually finding a match. However, these extra steps correspond to the (delayed) removal of the commitments, there is no waste here.
- In the case of a non-deterministic choice, we proceed as follows. First, during a polling phase, the branches of the choice are tested in a left-right ordering. If the guard of the current branch evaluates to false then the branch is simply skipped. If the guard evaluate to true and the guarded action is executable,

then the branch is selected immediately. If the action is not executable, then we prepare the commitments to deposit as part of a (local) transaction. If a branch is executed then the transaction is simply aborted but if all the branches are blocking, the transaction is committed (which means the commitments it contains are effectively recorded) and the thread is put in the wait queue. Note that the choice must appear as atomic, and thus the channels must be locked during the initial polling phase. In our implementations we adopt a simple pessimistic locking approach because the transactions are very lightweight: there are in general no more than a dozen branches to test, and the locks must be only retained during the polling phase. Of course, a more optimistic approach could be also experimented.

In conclusion, the main property of the scheduling algorithm is that its logic is distributed over the π-threads themselves, and there is no need for any centralized intervention.

4.3 Garbage collector

The implementation of an efficient garbage collector is probably one of the most critical and complex issue when developing a virtual machine [4]. As suggested by the rules $(reclaim)$ and $(stuck)$ of the semantics (cf. Table 5), the principles underlying the garbage collector of the PCM are remarkably simple. For the $(reclaim)$ rule, a π-thread can safely reclaim a channel it encounters if the latter has a global reference count of 0.

In a similar way, a thread can check if an emission or a reception is possible by first testing the value of the field globalrc for the concerned channel. If this count is 1 then it means the channel is only referenced by the thread trying to emit or receive. Put in other terms, there is no way this emission or reception could be performed because there cannot be any partner. The same situation occurs in a choice where all the involved channels are only known to the current process. In both situations the thread can be safely reclaimed. We exploit here the following derived rule :

$$\frac{\forall \hat{c} \in \Gamma,\ \mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{p} \vdash [\Gamma; \delta, \mathsf{pid} \triangleright \hat{p}] : \sum \mathbf{wait} \parallel A \to \Delta \vdash A}\ (stuck_1)$$

The interest of this derivative of $(stuck)$ is that it can be decided in a purely local manner. Of course, it does not cover the general case of e.g. cyclic structures. In most garbage collectors based on reference counting, non-trivial algorithms for cycle detection in graphs are employed [10]. As explained previously, we do not need in the case of π-threads to actually detect any cyclic structure, but only a partial termination. In the context of the PCM, where every resources are either threads or channels, the latter problem enjoys a (much) simpler solution than the former !

```
procedure wait(a:Agent, th:PiThread) {
  for(ch:Channel ∈ th.knows) {
    ch.waitrc := ch.waitrc + 1
  }
  a.wait := a.wait ∪ { th }
  th.date = a.date ; a.date = a.date + 1 ;
}

procedure awake(a:Agent, th:PiThread) {
  for(ch:Channel ∈ th.knows) {
    ch.waitrc := ch.waitrc - 1
  }
  a.wait := a.wait \ { th }
}
```

Table 7. Algorithms : waiting and awaking

From an algorithmic point of view, the principle of rule $(stuck)$ is to determinate a clique of waiting threads verifying a global property on the reference count for the channels on which the threads are waiting. The hypothesis of the rule requires to count the total number of threads outside the clique, i.e. the threads that are *not* waiting on any of these channels. We must obtain a count of 0 which means no thread outside the clique may ever wake any of the threads *within* the clique. A way to simply separate the outside from the inside of such a clique is to maintain a count of the (global) references to the channels from waiting threads. This is the role of the waitrc field of the Channel structure (cf. Table 6). When a thread is put of removed from the wait (or old) queue, it is required to refresh this value, as explained in Table 7.

```
procedure gc2(a:Agent,th:PiThread) {
  var clique:Set[PiThread] := ∅
  var candidates:Set[PiThread] := { th }
  do {
    var candidate:PiThread := choose(candidates)
    for(ch:Channel ∈ chans(candidate.commits)) {
      if ch.waitrc = ch.globalrc then {
        candidates := candidates
                      ∪ procs(ch.inCommits∪ch.outCommits)
                      \ clique
      } else return
    }
    clique := clique ∪ { candidate }
  } while(candidates≠ ∅)
  for stuck:PiThread ∈ clique {
    reclaim(stuck)
  }
}
```

Table 8. Algorithms : second generation collector

PROPOSITION 6. *For any channel ch, if ch.waitrc=ch.globalrc then all the threads knowing the channel are waiting.*

Despite being obvious, this proposition is at the heart of our GC algorithm for second generation (i.e. old) garbage, whose pseudo-code is provided in Table 8. The objective is to construct, incrementally and concurrently, a clique of threads of the form $\prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait}$ which is isolated in terms of commitments on channels. First, we have to pick up a candidate among the waiting threads. A good heuristic is to select a candidate who started waiting "a long time ago", which can be obtained by inspecting the date field. Past a given date (one of the very few parameters of our algorithm), a waiting thread is put in the old queue of the scheduler. The gc2 algorithm picks up its initial candidate among such old threads. In the algorithm, the candidate corresponds to the variable th. We first construct an empty clique and a set of candidates, initially a singleton containing the candidate thread th. In the main loop of the algorithm, the principle is to try to empty the set of candidates to populate the clique. For this we first choose an arbitrary thread within candidates. We then analyze the set of channels that are referenced by the commitments of this thread. If one of these channels is known by a ready or running thread, then we know at least one thread outside the clique would be able to awake at least one of the threads within the clique, so we abandon this instance of the algorithm. Now, if the equation ch.waitrc=ch.globalrc holds then we know that all the threads having some reference to ch are waiting, they are thus also new candidates to be added to the clique we try to construct. We add these threads to the set of candidates if they are not already present in the clique. If at some point we manage to empty candidates then it means we constructed a clique that is valid with respect to the hypothesis of the $(stuck)$ rule. In consequence, all the threads within the clique are indefinitely blocking and can be all reclaimed at once.

It is easy to see the gc2 algorithm is terminating because at worst we must examine as candidates all the waiting threads in the agent (note that the detection of a global termination is more efficiently detected when both the ready and run queues are empty). For the correctness wrt. $(stuck)$, we have not yet elaborated the formal proof but we started recently its verification in a proof assistant. At the time of writing this paper the formalization remains unfinished.

4.4 Implementations

We developed four different implementation of the π-calculus semantics, three of which are directly based on the π-thread calculus as presented in this paper. The CubeVM [13], our first implementation, is an interpreter based on an older version of the semantics. It is still a useful pedagogical for students/learners to experiment with the π-calculus constructs and semantics. The LuaPi implementation is a small library based on the coroutine mechanisms of the Lua scripting language. More than just a toy, this implementation shows that the π-threads model can be implemented in a very concise way with minimal runtime support. It is also used as a pedagogical tool and it is available online with a companion tutorial [11]. For the parallel variant, we developed a library above the multi-threading facilities of the Java programming language. Most of the parallel algorithms of the PCM where developed in the context of this library, which is also available online[6]. The implementation uses modern concurrency control features and lock-free algorithms, especially the lazy invalidation scheme explained previously. A termination detection algorithm is also provided, based on the gc2 algorithm presented in this paper. Finally, we are working on a complete toolchain that targets the PCM directly. Although it is at a very early stage of development, preliminary benchmarks reveal promising performance results. More informations about these various implementations and benchmarks can be found on the web site for the project[7].

5. Related work

There exists a large variety of language abstractions for parallel programming. In our work, we are mostly interested by programming languages and implementations based on the metaphor of communicating processes. Our main object of study is the π-calculus [9, 17], of which there are very few implementations. The most famous one is the Pict programming language, which is an implementation of an asynchronous variant of the π-calculus without a choice operator [15]. The *closures* used for efficient scheduling in Pict are quite similar to our commitments. One important difference is that in Pict a given process can deposit at most one closure, whereas we allow simultaneous commitments to support the choice operator. To address the problem of efficiently removing commitments, we propose a lazy invalidation scheme based on logical clocks. Because of its asynchronous nature, the closures are always input-guarded in Pict, whereas we allow a mix of input and output guards. Still for efficiency reasons, we also propose a lazy evaluation of output guards.

The Occam-π language [20] is an extension of the Occam language, itself based on the CSP formalism. We borrow some features of the Occam-π syntax but the latter is a much richer language. We favor an intermediate view of the process-oriented language, which may "disappear" behind higher-level derived constructs. It seems that Occam-π, as plain Occam, only allow input guards in choice expressions - probably for efficiency concerns - whereas we allow a mix of input and output guards. We did not find, however, a description for an intermediate language with formal operational semantics which would allow more thorough comparisons. At the implementation level, it would be interesting to compare the scheduling and garbage collection principles of Occam-π implementations with ours. But at the current stage of our implementations this study seems premature.

Our own CubeVM [13] is a reasonably efficient interpreter for an applied variant of the π-calculus. The choice operator implemented by the VM is purely non-deterministic but synchronization requires a linear search of partners in the scheduling algorithms. In comparison, our new operational semantics based on explicit commitments yield a much more efficient scheduling algorithm. The micro-benchmarks available online show that the impact of this change is huge in term of performances. Another important innovation is the new semantics and algorithms for second generation garbage collection of threads. The efficient collection of cycles in algorithms based on reference counting is, still today, an active area of research [10]. The originality of our approach is to reinterpret the problem as the detection of a partial termination of concurrent processes. The cyclic configurations are a special case of this more general problem, and the solution is indeed simpler than the explicit detection of cycles.

The extension *Concurrent ML* (CML) for the SML/NJ compiler of Standard ML supports a programming style very close to the π-calculus [16]. In fact the combinators for synchronization events supported by CML makes it even more expressive: it is for example possible to encode dynamic choices with the possibility to insert/remove execution branches at runtime. However, it is not obvious how we may generalize the guarded choices to more complex event combinators while maintaining the current balance between the simplicity of the operational semantics and the efficiency of its implementation(s) (especially for the scheduling and garbage collection). At the compiler level, relying on arbitrarily complex event combinators would make harder the static analysis of the language (e.g. the calculation of static bounds cf. Table 4). We plan, however, to introduce less intrusive extensions, in particular multicast or asynchronous channels. As a library, CML can exploit the features of the SML/NJ compiler : higher-order functions, lightweight continuations, garbage collector, etc. In our approach, we try to rely on a minimal runtime support (basic datatypes and operations, low-level operating system interface). Finally, an important reason to stick with the π-calculus is to establish a formal connection with the abstract calculus, and in the longer term to connect the implementations with modeling/verification tools (especially [12]).

Recently, *Google* released the Go programming language [6] with native support for concurrent processes (called *goroutines*) and synchronous (as well as asynchronous) channels. At the implementation level, an interesting characteristic of Go is the support of a mix of micro and macro-threading, based on a notion of *stack segmentation*. The principle is simple : the "real" stack is divided in various zones, each of which is associated to a given thread. If the allocated zone is not sufficient (e.g. because of too many nested calls), the extra memory can be allocated at runtime. The π-threads have no stack but only a local environment whose size is computed at compilation time. The segmentation of the stack can be used to store the local environments for the actually running threads, with the big difference of a compiler support. There are only (very) informal specifications available for the Go language so it is difficult to compare the approaches in more details. Finally, it seems that the GC framework for Go is still at a very early stage of development, maybe our GC scheme could be of some interest ?

The *monadic functional programming* metaphor offers an elegant way to mix pure functional code with imperative traits: side effects, input/output and parallelism. The *M-vars* of Concurrent Haskell [8] roughly correspond to a π-calculus without a choice operator. A powerful transactional choice is proposed with the

[6] http://code.google.com/p/javapi/

[7] http://lip6.fr/Frederic.Peschanski/pithreads

STM [7]. Unlike the π-calculus choice, which plays "one move in advance", the transactional choice can play many moves before an actual branch of execution is committed. The proximity with the database transactions makes the metaphor appealing, but the underlying formal semantics and implementation support are also much more involved. It appears to us that "one move in advance" is already a great deal of expressive power.

The *join patterns* proposed in the JoCaml programming language [5] are quite interesting from a π-calculus point of view because they offer an orthogonal way of composing behaviors. The non-deterministic choice plays the role of a «OR» combinator whereas the joins play the role of «AND». The two constructs can thus be combined in a natural way. In the LuaPi implementation we added the join patterns natively, because it is easy to do so in a micro-threading environment. It also supports join patterns with both input and output guards whereas only input guards are supported in JoCaml. Another approach is to encode the joins using the choice operator and recursion, which is more amenable to a truly parallel implementation. In this case, however, only joins with input guards can be supported. Consider for example the join pattern {c?(x) & d?(y)}P. In order to execute the continuation P, the process must be able to *simultaneously* read on the channels c and d. This behavior can be encoded as follows:

```
def J = c?(x), [ d?(y),P + c!x,J]
```

The order of the clauses in the choice is important, here we make the left-branch prioritized so that we put a priority on resolving the join pattern, and only if it is not possible we offer to "give back" the value we got though the channel c.

Erlang [3] is quite famous for its concurrency model based on the *actor* model of computation [1]. With actors the processes interact in an asynchronous way using message passing but without any notion of communication channel. In consequence, the processes must know about each others to interact, i.e. they share their identity. In the π-calculus there is less implicit coupling between the processes because they are anonymous, and the only shared information are the channels they use to communicate. Moreover, the garbage collection problem is more complex in the case of actors because it is required to maintain the "inverse" references [19]. We favor in this work the synchronous interpretation of the π-calculus but there are natural extensions of the present work to asynchronous semantics. The first possibility would be to adopt the asynchronous-π approach [2] by disallowing output prefixes. The output commitments (without continuations) would then correspond to buffer cells. A second approach, more operational, would be to introduce buffered channels.

6. Conclusion

We advocate in this work and paper the use of an applied variant of the π-calculus – the π-threads – as an intermediate language in a compiler toolchain for a parallel programming environment. We believe the proposed operational semantics to be simple and "natural", especially if we consider the fact that they to not characterize the language itself but its (abstract) implementation. This includes important properties related to thread scheduling and garbage collection. A striking feature of the PCM machine is its stackless architecture which, definitely, is a major advantage when it comes to parallelism. We see the explicit commitment model as probably the most natural interpretation of the "one move in advance" kind of non-determinism promoted by the π-calculus and other similar process algebras. The direct consequence is the decentralized scheduling algorithm. Last but not least, the GC algorithms proposed in the PCM provide a simple and general solution to the issue of garbage collection for parallel processes.

Our current research mostly focuses on the intermediate and backend layers of the compiler toolchain. We investigate the type-directed static analysis of the intermediate language and in particular the compile-time detection of *chain reactions* for *automatic stack injection* (cf. [13] for a *manual* approach). Given the proximity of the calculus and the virtual machine, we recently initiated a project for a *certified* implementation of the PCM using a proof assistant. The longer term objective would be the design of a certified compilation toolchain for the π-calculus.

References

[1] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, Jan. 1997.

[2] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. In *Concur'96*, volume 1119 of *LNCS*. Springer, 2001.

[3] J. Armstrong. A history of erlang. In *HOPL*, pages 1–26, 2007.

[4] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM '04*, pages 37–48. ACM, 2004.

[5] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Concur'96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.

[6] Google. *The Go Programming Language Specification*. `http://golang.org/doc/go_spec.html`, 2009.

[7] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[8] S. L. P. Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL'96*, pages 295–308, 1996.

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

[10] H. Paz, D. F. Bacon, E. K. Kolodner, E. Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[11] F. Peschanski. (lua)pi-threads tutorial. Technical report, UPMC Paris Universitas – LIP6, `http://luaforge.net/docman/view.php/505/5768/LuaPiTut.pdf`, 2008.

[12] F. Peschanski and J.-A. Bialkiewicz. Modelling and verifying mobile systems using pi-graphs. In *Sofsem'09*, volume LNCS 5404, pages 437–442. Springer, 2009.

[13] F. Peschanski and S. Hym. A stackless runtime environment for a pi-calculus. In *VEE '06*, pages 57–67. ACM Press, 2006.

[14] I. Phillips. Ccs with priority guards. In *CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2001.

[15] B. C. Pierce and D. N. Turner. Pict: a programming language based on the pi-calculus. In *Proof, Language, and Interaction*, pages 455–494. The MIT Press, 2000.

[16] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

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

[18] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, March 2005.

[19] A. Vardhan and G. Agha. Using passive object garbage collection algorithms for garbage collection of active objects. In *ISMM'02*, pages 106–113. ACM Press, 2002.

[20] P. H. Welch and F. R. M. Barnes. Communicating mobile processes. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*. Springer, 2005.