Principes et Pratiques de la Programmation Concurrente en Pi-calcul

Frédéric Peschanski

UPMC Paris Universitas - LIP6
4, Place Jussieu, 75252 Paris cedex 05, France
prenom.nom@lip6.fr

RÉSUMÉ. *Nous étudions la problématique du mélange de paradigmes de programmation variés plongés dans un environnement concurrent. Dans cette perspective, nous poursuivons un cheminement similaire à celui qui conduit du λ-calcul aux langages fonctionnels, mais en prenant comme point de départ le π-calcul. Nous proposons la machine abstraite des π-threads dont la principale originalité est l'absence de pile d'exécution. Cette caractéristique permet de nous reposer dans nos implémentations sur des algorithmes simples et naturellement concurrents, notamment pour ce qui concerne l'ordonnancement et le ramasse-miettes. Pour ce dernier, nous proposons un algorithme permettant de récupérer très simplement les cycles de processus en interblocage et autres structures bloquantes.*

ABSTRACT. *We study the problem of mixing various programming paradigms within a concurrent environment. To that end, we mimic the elaboration of functional languages from λ-calculi, but with the π-calculus as a starting point. We introduce the π-threads, an abstract machine with a stackless architecture. This leads to simple and naturally concurrent algorithms, especially for the scheduler and the garbage collector. We propose a very simple GC algorithm that is able to collect cycles of interblocking processes as well as other blocking structures.*

MOTS-CLÉS : *Concurrence, Pi-calcul, Architecture sans Pile, Rammasse-miettes*
KEYWORDS: *Concurrency, Pi-calculus, Stackless architecture, Garbage collection*

1. Introduction

Nous nous intéressons dans ce travail à la problématique du mélange des paradigmes de programmation, un aspect central des langages applicatifs. La question que l'on se pose en particulier est l'intégration de paradigmes variés au sein d'un environnement *concurrent*. L'avénement des technologies multi-cœurs a remis cette ancienne problématique sur le devant de la scène. La métaphore fonctionnelle représente notre principale source d'inspiration. Cependant, de notre point de vue, cette métaphore même «agrémentée» (ex. : programmation monadique) n'offre pas une caractérisation naturelle de la concurrence, au contraire de la métaphore des *processus communicants*. De fait, nous pensons que les calculs de processus - en particulier le π-calcul [11, 19] - sont plus aptes à «mélanger» les paradigmes dans un environnement concurrent. Ainsi même les fonctions possèdent une «jolie» interprétation en termes de processus[1].

Dans ce travail, notre cheminement est similaire à celui qui conduit du λ-calcul aux langages fonctionnels. La différence fondamentale est que nous prenons le π-calcul comme point de départ. Nous introduisons dans cet article le calcul des π-threads qui représente notre «équivalent» d'une *machine abstraite*. Cette machine est présentée à la manière des algèbres de processus pour une plus grande proximité avec le π-calcul. La caractéristique la plus remarquable de cette machine est l'absence de pile d'exécution du fait de la nature terminale des appels récursifs et la gestion «à plat» des environnements lexicaux. Cette caractéristique permet de nous reposer dans nos implémentations sur des algorithmes simples et naturellement concurrents, notamment pour ce qui concerne l'ordonnancement et le ramasse-miettes.

La machine virtuelle PCM (*Process Commitment Machine*) réalise de façon concrète le calcul des π-threads. Elle se distingue elle aussi par son absence de structure de pile. Dans cet article, nous discutons principalement du calcul abstrait mais les caractéristiques fondamentales de la machine PCM sont également présentées. Notre modèle de calcul est de fait assez fidèle au π-calcul «classique», ce qui nous permet de

1. Une caractéristique intéressante de l'encodage de fonctions à base de processus - de plus bas niveau - est que la stratégie d'évaluation est encodable au même niveau d'abstraction [19, 4].

nous reposer sur un corpus théorique important mais aussi de connecter nos implémentation à des outils de modélisation et de vérification, notamment [14]. En pratique, un «véritable» langage de programmation doit avant tout être supporté par des implémentations aussi efficaces que possible, en tout cas d'un point de vue algorithmique. En comparaison de notre précédente publication sur ce travail [15], les performances actuelles de nos prototypes sont nettement améliorées. Pour cela, nous avons complètement revu la sémantique opérationnelle ainsi que l'algorithme d'ordonnancement qui est désormais en temps «quasi-constant». Nous avons également modifié le ramasse-miettes avec l'introduction d'un collecteur de seconde génération pour les structures bloquantes. L'originalité de ce collecteur basé sur le comptage de références est que, contrairement aux ramasse-miettes classiques [12], il ne détecte pas les cycles mais les structures plus générales de processus «indéfiniment endormis».

Le plan de l'article est le suivant. En Section 2 nous introduisons une variante du π-calcul conçue pour servir de prototype de langage de programmation. En Section 3 nous entrons dans le vif du sujet en présentant la sémantique opérationnelle du calcul des π-threads. Certaines propriétés fondamentales de cette sémantique sont également discutées. Par manque de place nos schémas de preuve restent largement informels. Un aperçu de la machine PCM est proposé en Section 4. Nous insistons sur son architecture, son ordonnanceur et son ramasse-miettes. Un panorama de langages et d'implémentations connexes est proposé en Section 5.

2. Introduction à la programmation «en π-calcul»

La Table 1 propose la syntaxe abstraite du calcul des π-threads qui représente notre abstraction de programmation. Avant de nous intéresser à la programmation concurrente, et pour appuyer notre discours «multi-paradigmes», étudions l'encodage des fonctions. Dans un environnement où tout est processus ou canal (comme pour notre compilateur natif), les paradigmes connus doivent être encodables efficacement. Pour les récursions primitives, le constructeur d'appel en position terminale

Definition		**def** $D(x_1,\ldots,x_n) = P$		Définition
Process	P	: :=	**end**	Terminaison
			$\mid \sum_i [g_i]\alpha_i, P_i$	Choix gardés
			$\mid D(v_1,\ldots,v_n)$	Appel
Prefix	α	: :=	**tau**	Pas interne
			$\mid c\,!v$	Emission
			$\mid c\,?(x)$	Réception
			$\mid$ **new**(c)	Création de canal
			$\mid$ **spawn**$\{P\}$	Création de thread

Tableau 1 – Syntaxe du calcul des π-threads

permet un encodage direct. Ainsi pour la célèbre *Fibonacci* on pourra écrire :

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

L'opérateur + correspond au choix non-déterministe entre plusieurs branches de calcul. Cet opérateur, central dans le π-calcul, «joue» un coup en avance, c'est-à-dire qu'il ne s'engage dans une branche que si la première action de cette branche est effectivement réalisable. Cela permet d'encoder des structures complexes de synchronisation avec un maintien d'état minimal, au contraire des approches transactionnelles [7]. Les branches sont également gardées par une expression booléenne (true par défaut). Dans la sémantique que nous proposons, il s'agit de plus d'un choix ordonné c'est-à-dire que la branche de gauche sera testée avant la branche de droite. De ce fait, les alternatives s'encodent avec + très simplement : **if** C **then** P **else** $Q \stackrel{\text{def}}{=}$ [C] **tau**,P + [true] **tau**,Q

Cet encodage nous permet de proposer une définition plus classique de *Fibonacci* :

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

La principale différence par rapport à un langage fonctionnel est qu'il n'existe pas de notion de valeur de retour pour une définition de processus. Pour simuler ce retour nous utilisons un canal dédié[2] : r dans l'exemple ci-dessus.

Dans les exemples qui suivent nous prenons quelques libertés concernant la syntaxe, notamment en manipulant explicitement des expressions de types basiques. Le **end** terminal est la plupart du temps omis et certains autres raccourcis syntaxiques seront expliqués par la suite. Nous typons explicitement les paramètres des définitions mais considérons comme synthétisées les autres informations de type. Notre système de type est tout a fait classique et ne sera donc pas discuté dans cet article.

Comme les appels récursifs sont forcément terminaux, il faut encoder les appels non-terminaux avec des processus et des canaux. Ainsi pour l'autre célèbre fonction d'*Ackermann* on pourra écrire :

```
def Ack(n p : int, r : chan<int>) =
   [n=0] r !(p+1) + [p=0] Ack(n-1,1,r)
   + new(r1),spawn{Ack(n,p-1,r1)}, r1 ?(pp),Ack(n-1,pp,r)
```

Dans la troisième branche, qui correspond au cas «doublement» récursif de la célèbre fonction, l'appel en argument est lancé en parallèle avec un processus qui attend que cet argument soit calculé avant de passer à la continuation du calcul[3]. On peut s'interroger à juste titre sur l'efficacité d'un tel encodage. Ce point est largement discuté dans [15] mais l'idée est que dans le cas des encodages fonctionnels, les canaux mis en jeu sont dits *linéaires* [19] ou à usage unique. C'est le cas par exemple pour les canaux r et r1 ci-dessus. Il n'ont pas à être considérés par le ramasse-miettes puisque l'on peut les récupérer lors de leur première et unique utilisation. En fait, un simple mot mémoire suffit à les implémenter.

---

2. Les structures relationelles sont encodables en utilisant des canaux de retour multiples.
3. Une continuation de calcul fonctionnel possède la structure d'un processus en attente sur un canal qui marque la fin de l'étape précédente du calcul. Avec le π-calcul, on peut imaginer de nombreuses variations : continuations multiples (avec plusieurs canaux), continuations non-déterministes (avec des choix), communicantes, etc.

Au delà des fonctions, de nombreux autres paradigmes sont assez naturellement exprimables dans ce langage. L'article [15] et l'implémentation associée proposent un certain nombre d'exemples, notamment dans le domaine des langages de flots (*dataflows*). Nous pouvons en guise d'illustration proposer une variante de *Fibonacci* en flots :

```
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)
```

Ici la définition FFib crée un flot d'entiers sur le canal r qui retourne au fur et à mesure les termes de *Fibonacci* vue comme une suite : 1 1 2 3 5 8, etc.

L'encodage succinct des fonctions ou de tout autre structure de calcul «classique» est important, mais c'est surtout les *structures concurrentes* pour la programmation parallèle qui nous intéressent. En guise d'exemple, considérons l'implémentation d'un *pattern* simple de sections critiques en exclusion mutuelle. La structure primitive de synchronisation est le *canal de communication*. Un simple canal permet de simuler un verrou du fait de la sémantique de communication synchrone mise en jeu dans le π-calcul. Pour acquérir le «verrou» on peut lire sur le canal, ce qui bloquera tant qu'aucun émetteur n'aura écrit dessus. Et pour le relâcher c'est tout aussi simple, il suffit de réaliser une émission. On peut donc donner une structure canonique pour les sections critiques :

```
def SectCrit(lock : chan<>) = lock ?,/* Section Critique */,lock !
```

Le type chan<> représente les canaux utilisés pour de la synchronisation pure, sans passage de valeur. Bien sûr, la section critique est exécutée en exclusion mutuelle uniquement si le canal lock est utilisé de façon correcte. Par exemple, si un processus relâche deux fois le verrou, deux sections critiques pourront être exécutées en parallèle, ce qui invalidera la propriété d'exclusion mutuelle. Une façon de rendre le protocole plus robuste, outre une discipline de type (cf. [9]), est d'encapsuler notre verrou dans un processus, par exemple de la façon suivante :

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

La définition pour les sections critiques est adaptée en conséquence :

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

Lorsque le verrou est acquis (synchronisation sur le canal lock), un canal release de relâchement est reçu par le client. Ce canal est *privé* entre le processus verrou et son client. Le phénomène mis en jeu ici est le *passage de canal* - trait caractéristique du π-calcul - qui permet de communiquer (la référence d') un canal via un autre canal. Le type du canal lock est emblématique de ce phénomène : chan<chan<>>. La robustesse de notre protocole repose désormais uniquement sur le code de la section critique (par exemple, il faut faire attention si l'on communique le canal privé release).

Pour terminer notre aperçu, illustrons le fait qu'il est très simple d'encoder des structures d'*objets actifs* en utilisant le choix + comme sélecteur de méthode. Considérons un autre *pattern* de programmation concurrente : la gestion d'un *pool* de tâches à exécuter. Le cahier des charges est simple, il s'agit de permettre à un nombre maximal de tâches de s'exécuter parallèlement. Si le *pool* de tâches est plein alors les autres tâches sont simplement mises en attente. C'est une variante «déguisée» du fameux sémaphore :

```
def TaskPool(remain max : int, enter leave : chan<int>) =
  leave ?(id),TaskPool(remain+1,max,enter,leave)
  + [remain>0] enter ?(id),TaskPool(remain-1,max,enter,leave)
```

Ici, le sélecteur de méthode utilise deux canaux distincts : le nom de canal correspond donc au nom de la méthode[4]. Pour les tâches, le mode d'emploi est simple : il suffit d'attendre l'accès au *pool* et de prévenir ce dernier lorsque la tâche est terminée. La structure générale d'une tâche est donc la suivante :

4. Avec des structures comme les types somme, il est également possible de construire des sélecteurs de méthodes avec les gardes booléens, ce qui permet d'économiser les canaux.

```
def Task(id : int,enter leave : chan<int>) =
  enter !id,/* Tâche à exécuter */,leave !id
```

On remarque ici que les tâches doivent se conformer au «mode d'emploi» du TaskPool, et par exemple ne pas entrer ou le quitter plusieurs fois de suite. Une version plus sûre du protocole peut être obtenue en encapsulant le canal de relâchement leave, à la manière de l'encapsulation des verrous vue précédemment.

Ceci clôt notre petit tour d'horizon, qui nous l'espérons illustre le pouvoir expressif du langage utilisé, notamment pour mélanger les paradigmes. Concernant la concurrence, des exemples plus complets sont proposés dans un tutoriel séparé [13].

Par rapport au π-calcul «traditionnel», il est important de remarquer que nous proposons une interprétation plus «réaliste» de certains constructeurs. Ainsi il n'y a pas de constructeur parallèle primitif mais un préfixe spawn pour la création dynamique de processus. Les *match* et *mismatch* sont remplacés et généralisés par les gardes explicites dans les branches des choix. Pour ce dernier, rappelons le fait que l'on exploite l'ordre des branches. Cette interprétation est naturelle du point de vue du programmeur et compatible avec l'interprétation «purement» non-déterministe du π-calcul. Remarquons qu'il ne s'agit pas d'un *choix prioritaire* au sens de [16], ce qui sera discuté dans la section suivante.

3. Le calcul des π-threads

Nous présentons dans cette section la sémantique opérationnelle du calcul des π-threads, notre variante appliquée du π-calcul. Ce calcul possède de nombreux attributs d'une machine abstraite, mais présentée à la manière des algèbre de processus pour une plus grande proximité avec le π-calcul sous-jacent[5]. La syntaxe des processus est celle de la Table 1 à laquelle on ajoute un préfixe d'action **wait** à usage interne.

Les termes manipulés dans le calcul sont de nature contextuelle, de la forme suivante :

5. Par faute de place, nous ne discutons pas dans cet article des liens étroits qui relient les π-threads au π-calcul. Cet aspect fera sans doute l'objet d'une communication future mais remarquons la proximité des constructions syntaxiques des deux langages.

$\Delta \vdash \Pi_1 \parallel \ldots \parallel \Pi_n$ où chaque Π_i est un π-thread de la forme $[\Gamma_i; \delta_i] : P_i$. De façon un peu plus concise on notera : $\Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i$

Le composant Δ représente l'*environnement global* des π-threads. Il s'agit d'un ensemble d'identités permettant de désigner de façon unique les canaux et threads. Pour chaque thread d'identité $\hat{p}_i$ correspond une expression de processus[6] P_i. Ce dernier évolue dans un contexte local constitué d'un ensemble d'engagements (*commitments*) Γ_i ainsi que d'un environnement lexical δ_i qui associe comme de coutume variables et valeurs. Par exemple, pour tout thread on a $\delta_i(\mathsf{pid}) = \hat{p}_i$ (i.e. la variable pid est associée à l'identificateur du processus courant). Notons que cet environnement est «plat» : il ne peut y avoir qu'une seule liaison par nom de variable.

La sémantique proposée est essentiellement basée sur l'annonce explicite par les processus de leurs engagements concernant leurs interactions potentielles : le fameux «coup d'avance» permettant d'implémenter l'opérateur de choix. L'ensemble Γ_i contient donc un ensemble d'engagements, qui peuvent être :

– soit des engagements d'émission (*output commitments*) notés $\hat{c} \Leftarrow v : Q$. Ici il s'agit de l'engagement fait par le processus d'émettre la valeur v sur le canal d'identité $\hat{c}$. La continuation de l'engagement est Q.

– soit des engagements de réception (*input commitments*) notés $\hat{c} \Rightarrow x : Q$. Ici on reçoit sur $\hat{c}$ et la valeur reçue est associée à la variable x dans la continuation Q.

Les règles de base de la sémantique opérationnelle du calcul des π-threads sont décrites en Table 2. Les règles (par_L) et (par_R) implémentent de façon classique la sémantique d'entrelacement qui «simule» le parallélisme. La règle $(step)$ explique l'interprétation du **tau** comme

6. Pour être précis, un processus est une expression syntaxique selon les règles de la Table 1. Un (π-)*thread* est l'adjonction d'une telle expression de processus à un environnement local et un ensemble d'engagements. Il s'agit donc d'un processus en cours d'exécution. Mais cette distinction n'est pas si fondamentale et nous utiliserons donc les deux termes de façon quasi-interchangeable dans le reste de l'article.

$$\frac{\Delta \vdash A \to \Delta' \vdash A'}{\Delta \vdash A \parallel B \to \Delta' \vdash A' \parallel B}\ (par_L)$$

$$\frac{\Delta \vdash B \to \Delta' \vdash B'}{\Delta \vdash A \parallel B \to \Delta' \vdash A \parallel B'}\ (par_R)$$

$$\frac{\overline{g} = \text{true}}{\Delta \vdash [\Gamma;\delta] : [g]\textbf{tau}, P + \sum_i Q_i \to \Delta \vdash [\emptyset,\delta] : P}\ (step)$$

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

$$\frac{\overline{g} = \text{true}}{\begin{array}{l}\Delta \vdash [\Gamma;\delta] : [g]\textbf{spawn}\{P\}, Q + \sum_i R_i \\ \to \Delta, \hat{p} \vdash [\emptyset;\delta, \mathsf{pid} \triangleright \hat{p}] : P \parallel [\emptyset;\delta] : Q\end{array}}\ (spawn)$$

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

$$\frac{\textbf{def}\ D(x_1,\ldots,x_n) = P}{\begin{array}{l}\Delta \vdash [\Gamma;\delta] : D(v_1,\ldots,v_n) \\ \to \Delta \vdash [\emptyset; \mathsf{pid} \triangleright \delta(\mathsf{pid}), x_1 \triangleright v_1, \ldots, x_n \triangleright v_n] : P\end{array}}\ (call)$$

$$\frac{}{\Delta, \hat{p} \vdash [\Gamma;\delta, \mathsf{pid} \triangleright \hat{p}] : \textbf{end} \to \Delta \vdash \emptyset}\ (inert)$$

Tableau 2 – Sémantique : règles de base

première branche d'un choix[7]. Un **tau** n'est jamais bloquant et la continuation P est donc exécutée. On remarque que les engagements du processus sont effacés lorsqu'une continuation est démarrée. Le garde g du **tau** doit être évalué à true pour que la branche soit empruntée. Dans cet article nous ne discutons pas de l'évaluateur pour les expressions des types de base (booléens, entiers, etc.). On note simplement l'évaluation d'une expression e par $\overline{e}$. La règle (new) explique la création d'un canal de communication par le préfixe **new**. Comme précédemment, cette action n'est pas bloquante. La conséquence est la création d'un nouvel identificateur global $\hat{c}$ que l'on associe à la variable formelle c dans l'environnement local. Pour la création de processus, avec **spawn**, un nouveau π-thread fils est lancé en parallèle de son père. L'environnement du père est hérité dans le fils, à part bien sûr l'identifiant pid du fils qui doit être créé pour l'occasion. La règle $(next)$ permet de «sauter» une branche gardée par false. Cette règle montre que les choix s'analysent de la gauche vers la droite, ce qui influence la sémantique comme nous le verrons en discutant de la communication. L'appel de définition en position terminale est caractérisé par la règle $(call)$. Il est possible d'encoder les définitions et appels directement avec des processus et des canaux, mais c'est peu efficace (il faut créer artificiellement des processus) et on perd surtout d'intéressantes propriétés relatives au ramassage des miettes. On remarque en effet que la règle opère un important «nettoyage» de l'environnement lexical, puisque seules les variables de l'appel sont conservées (en plus du pid). La règle $(inert)$ explique très simplement le **end** qui termine un processus. Cette fois-ci encore l'impact est important sur le ramasse-miettes puisque l'environnement lexical est «détruit».

La Table 3 propose les règles qui implémentent la communication et la synchronisation dans les π-threads. Ces règles forment un peu le cœur du calcul. Pour l'émission, on distingue deux cas. Dans le premier cas, avec la règle $(send)$, le garde g devant le préfixe d'émission c !v s'évalue en true. On peut s'interroger sur le fait de pré-calculer la valeur du garde plutôt que de l'enregistrer pour une évaluation ultérieure. Au-

7. La plupart des règles de sémantique sont expliquées dans le contexte d'un choix. Le cas particulier d'un préfixe isolé peut être encodé par un choix avec une seule branche d'exécution. Le choix à 0 branche se résume à **end** (0 dans le π-calcul).

$$\frac{\overline{g} = \text{true}}{\begin{array}{l}\Delta \vdash [\Gamma;\delta] : [g]c\,!v,P + \sum_i Q_i \parallel [\Gamma', \hat{c}{\Rrightarrow}x : R;\delta'] : S \\ \rightarrow \Delta \vdash [\emptyset;\delta] : P \parallel [\emptyset;\delta', x{\triangleright}\overline{v}] : R\end{array}}\ (send)$$

$$\frac{\nexists \hat{c}{\Rrightarrow}x : S \in \bigcup_j \Gamma_j}{\begin{array}{l}\Delta \vdash [\Gamma;\delta] : [g]c\,!v,P + \sum_i Q_i \parallel \prod_j [\Gamma_j;\delta_j] : R_j \\ \rightarrow \Delta \vdash [\Gamma, \hat{c}{\Lleftarrow}v : P;\delta] : \sum_i Q_i + \textsf{wait} \parallel \prod_j [\Gamma_j;\delta_j] : R_j\end{array}}\ (out)$$

$$\frac{\overline{g} = \text{true}}{\begin{array}{l}\Delta \vdash [\Gamma;\delta] : [g]c\,?(x),P + \sum_i Q_i \parallel [\Gamma', \hat{c}{\Lleftarrow}v : R;\delta'] : S \\ \rightarrow \Delta \vdash [\emptyset;\delta, x \triangleright \overline{v}] : P \parallel [\emptyset;\delta'] : R\end{array}}\ (recv)$$

$$\frac{\nexists \hat{c}{\Lleftarrow}v : S \in \bigcup_j \Gamma_j}{\begin{array}{l}\Delta \vdash [\Gamma;\delta] : [g]c\,?(x),P + \sum_i Q_i \parallel \prod_j [\Gamma_j;\delta_j] : R_j \\ \rightarrow \Delta \vdash [\Gamma, \hat{c}{\Rrightarrow}x : P;\delta] : \sum_i Q_i + \textsf{wait} \parallel \prod_j [\Gamma_j;\delta_j] : R_j\end{array}}\ (in)$$

Tableau 3 – Sémantique : communication

trement dit, l'interprétation choisie est statique plutôt que dynamique. En fait, il faut remarquer dans la sémantique que l'environnement local n'est jamais modifié *avant* d'emprunter effectivement une branche d'un choix. Parmi les préfixes, seules les réceptions et les créations de canaux ou de processus permettent de modifier l'environnement local, mais uniquement une fois qu'elles ont eu lieu.

En plus de la validation (statique) du garde, il est nécessaire pour pouvoir réaliser l'émission de «trouver» un processus possédant un engagement de lecture depuis le canal identifié par $\hat{c}$. La conjonction de ces deux conditions entraîne la synchronisation entre les deux processus. L'émetteur poursuit dans la continuation P et le récepteur reprend son exécution par la continuation R enregistrée dans l'engagement. La variable de réception x est liée à la valeur reçue dans cette continuation.

On remarque ici que cette valeur est calculée de façon paresseuse, au moment de la synchronisation. S'il n'y a pas d'engagement de partenaire pour le canal concerné, alors c'est la règle (out) qui s'applique. Dans ce cas, le processus émetteur enregistre un engagement d'émission et la branche suivante est testée. L'instruction **wait** injectée marque le fait que la branche n'est pas supprimée mais en attente éventuelle de réveil. Notons que la règle (out) à une portée globale et ne peut donc être utilisée conjointement avec (par_L) et (par_R). Il est en effet nécessaire de connaître tous les engagements effectués sur le canal concerné. Pour la réception les cas $(recv)$ et (in) sont exactement symétriques.

Pour bien comprendre le fonctionnement de ces règles, étudions quelques propriétés importantes.

Proposition 3.1 *Dans un agent* $\Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i$ *il ne peut exister de canal* $\hat{c} \in \Delta$ *tel que :*

$\exists j, k (j \neq k)$ *avec* $\hat{c} \Leftarrow v_j : Q_j \in \Gamma_j$ *et* $\hat{c} \Rightarrow x_k : Q_k \in \Gamma_k$

Informellement, cette proposition exprime qu'à un moment donné, il ne peut exister pour un canal donné au sein d'un agent que des engagements par des threads distincts d'une seule *polarité* (émission ou réception mais pas les deux). Vu d'une autre façon, la règle $(send)$ fonctionne exclusivement avec (in) et (out) avec $(recv)$.

Concernant l'articulation entre le choix et la communication, étudions le terme suivant (suggéré par un relecteur anonyme) :

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

Dans le π-calcul classique, si l'on suppose les canaux a et b restreints alors il existe deux branches d'exécution possibles : la synchronisation sur a ou b. Avec un choix *globalement* prioritaire (en privilégiant globalement la branche de gauche), au sens de [16], on serait ici dans une situation de blocage. Dans notre interprétation, la branche de gauche du choix est *localement* prioritaire. Ainsi les différentes possibilités d'exécution sont les suivantes :

– Le processus de gauche enregistre $\hat{a} \Leftarrow v : P$ puis $\hat{b} \Leftarrow w : Q$ via la règle (out). Le processus de droite teste sa première branche et trouve

un partenaire donc $(recv)$ permet la synchronisation sur b. Le cas symétrique (enregistrements à droite) entraîne la synchronisation sur a.

– Le processus de gauche enregistre $\hat{a}\Leftarrow v : P$ puis le processus de droite enregistre $\hat{b}\Rightarrow x : R$. Deux cas sont possibles à la prochaine étape : soit on infère $(recv)$ pour le processus de droite (synchronisation sur a), soit on infère $(send)$ pour le processus de gauche (synchronisation sur b). Il reste les deux cas symétrique.

En comptabilisant toutes les possibilités d'exécution de ce choix, on voit que l'on respecte au niveau global le caractère non-déterministe du choix du π-calcul. La principale différence est que les réductions sont ici de plus bas niveau : plusieurs réductions dans les π-threads correspondent à une réduction atomique dans le π-calcul classique. Ce n'est bien sûr pas étonnant pour une machine abstraite.

Le cas du **tau** est plus déterministe puisque le choix est ordonné. Ainsi, dans le terme [true] **tau**,P+[true] **tau**,Q c'est forcément la branche de gauche qui sera empruntée. Le π-calcul classique fournit une interprétation plus abstraite, en considérant qu'il s'agit d'un choix non-déterministe interne. Pour répondre à un soucis d'efficacité, on détermine ce cas et on ne couvre donc qu'une partie des choix possibles. Il est normal que l'implémentation soit parfois plus déterministe que l'abstraction, le cas inverse serait bien sûr problématique.

Les règles de la Table 4 résument de façon concise la sémantique du ramasse-miettes pour le calcul des π-threads. Ce dernier est basé sur le principe du *comptage de références* des canaux de communication. Nous renvoyons à [15] pour une discussion plus approfondie sur les motivations concernant ce choix mais pour résumer, disons que les algorithmes associés se parallélisent de façon très simple, ce qui n'est pas le cas des algorithmes basés sur le traçage. Le prédicat knows permet de tester si un canal est connu ou non d'un processus en inspectant son environnement local. Ceci permet de calculer, dans un ensemble de processus en parallèle, le nombre $\mathrm{globalrc}$ de ces processus qui connaissent un canal donné.

Il y a ensuite deux cas de figures. Tout d'abord, un canal qui n'est connu par aucun thread (i.e. tous les knows pour ce canal sont faux) possède un compteur global $\mathrm{globalrc}$ également à 0. Dans ce cas, le

$$\mathrm{knows}(\hat{c}, [\Gamma; \delta] : P) \stackrel{\text{def}}{=} \exists x \in \mathrm{dom}(\delta),\ \delta(x) = \hat{c}$$

$$\mathrm{globalrc}(\hat{c}, \Delta \vdash \prod_i [\Gamma_i; \delta_i] : P_i) \stackrel{\text{def}}{=} \sum_i \begin{cases} 1 \text{ if } \mathrm{knows}(\hat{c}, [\Gamma_i; \delta_i] : P_i) \\ 0 \text{ otherwise} \end{cases}$$

$$\frac{\mathrm{globalrc}(\hat{c}, A) = 0}{\Delta, \hat{c} \vdash A \rightarrow \Delta \vdash A}\ (reclaim)$$

$$\frac{\forall \hat{c} \in \bigcup_i \Gamma_i,\ \mathrm{globalrc}(\hat{c}, \prod_j [\Gamma_j; \delta_j] : Q_j) = 0}{\begin{array}{l}\Delta \cup \bigcup_i \hat{p}_i \vdash \prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait} \parallel \prod_j [\Gamma_j; \delta_j] : Q_j \\ \rightarrow \Delta \vdash \prod_j [\Gamma_j; \delta_j] : Q_j\end{array}}\ (stuck)$$

Tableau 4 – Sémantique : ramasse-miettes

canal peut-être bien sûr «ramassé», ce qui est caractérisé par la règle $(reclaim)$ de la sémantique. Bien sûr, cela ne suffit pas à récupérer tous les processus bloqués, notamment à cause des cycles éventuels. Dans [15] une extension - les canaux *transients* - est proposée pour répondre à cette limite. Elle permet l'encodage de structures de graphes par exemple, mais dans le cas général on peut tout de même se retrouver dans des situations d'interblocage (contrôle cyclique).

La règle $(stuck)$ de la Table 4 fournit une solution générale au problème. De façon (peut-être) étonnante, elle ne discute pas explicitement de cycle mais plutôt de «clique» de processus indéfiniment bloqués. L'idée est vraiment simple : s'il existe dans le contexte global une clique de processus en attente sur des canaux uniquement référencés par cette même clique, alors toute la clique est indéfiniment bloquée et peut donc être ramassée dans son ensemble. Pour discuter un peu plus formellement des propriétés de cette règle, regardons tout d'abord si effectivement elle permet de ramasser les cycles de processus en interblocage.

La structure générale d'un cycle minimal de taille $n > 2$ est représentée sur la Figure 1. Le processus P_0 est en attente de réception sur le

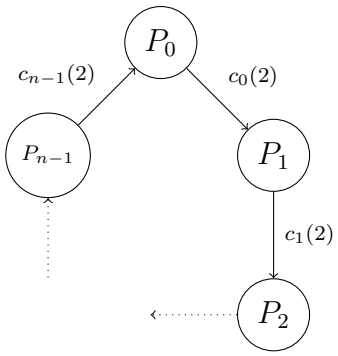

Figure 1 – Processus cycliques en attente

canal c_{n-1}. Si le processus est débloqué alors il effectuera une émission sur le canal c_0 qui bloque le processus P_1. De la même façon le processus P_i ($0 \leq i < n$) est en attente sur le canal c_{i-1} (*modulo* n la taille du cycle) puis, s'il est débloqué, écrit sur le canal c_i. Nous indiquons sur la figure le nombre de références globales de chaque canal mis en jeu dans ce cycle, c'est-à-dire $\mathrm{globalrc}(\hat{c}_i, \Pi_i[\Gamma_i; \delta i] : P_i)$. Ce compte est toujours 2 pour que le cycle soit minimal. Cela signifie que chaque canal c_i est connu uniquement de son émetteur P_i et de son récepteur P_{i+1}.

En terme d'engagements, un tel cycle de processus correspond à l'encodage suivant :

$$
\begin{array}{l}
[\hat{c}_{n-1} \Rrightarrow x_0 : Q_0; \delta_0] : \sum \textbf{wait} \\
\parallel [\hat{c}_0 \Rrightarrow x_1 : Q_1; \delta_1] : \sum \textbf{wait} \\
\parallel [\hat{c}_1 \Rrightarrow x_2 : Q_2; \delta_2] : \sum \textbf{wait} \\
\parallel \ldots \\
\parallel [\hat{c}_{n-2} \Rrightarrow x_{n-1} : Q_{n-1}; \delta_{n-1}] : \sum \textbf{wait}
\end{array}
$$

Avec pour chaque i ($1 \leq i < n$), $\mathrm{knows}(\hat{c}_{i-1}, [\Gamma_i; \delta_i] : P_i) = \mathrm{true}$, $\mathrm{knows}(\hat{c}_i, [\Gamma_i; \delta_i] : P_i) = \mathrm{true}$ avec ainsi un compte total $\mathrm{globalrc}$ valant 2 pour garantir la minimalité du cycle.

Proposition 3.2 *La clique formée par les processus* $\{P_0, \ldots, P_{n-1}\}$ *est récupérable par la règle* $(stuck)$

La preuve est en fait directe. Il s'agit simplement d'observer qu'aucun processus extérieur à la clique ne peut connaître un des canaux $\hat{c}_i$ puisque le compte global de références vaut 2. Ce cycle minimal est donc un cas particulier des cliques de processus caractérisées par la règle $(stuck)$.

En complément, il est important de vérifier que la règle $(stuck)$ n'admet pas de *faux positifs* : c'est-à-dire le ramassage de processus qui pourraient éventuellement être réveillés.

Proposition 3.3 *Tout processus P_i d'une clique $\{P_0, \ldots, P_{n-1}\}$ ne peut être réveillé*

Le schéma de la preuve procède par l'absurde. Supposons qu'il existe un processus Q différents des $P_0, \ldots, P_{n-1}$ permettant de réveiller un des processus P_i. Deux cas sont à considérer selon que P_i ait déposé un engagement d'émission $\hat{c} \Leftarrow v : R$ ou de réception. Dans le premier cas, le réveil de P_i est conditionné par l'existence d'une processus Q de la forme [h] c !v,$Q_1 + \sum_{i=2}^{n} Q_i$. Le réveil se produit potentiellement par la règle $(send)$ de la Table 3. Mais cela suppose que $\mathrm{globalrc}(\hat{c}, P_i \parallel Q) \geq 1$ ce qui contredit l'hypothèse de la règle $(stuck)$. Le cas pour l'engagement de réception est parfaitement symétrique, le réveil se produisant via $(recv)$.

4. La machine virtuelle PCM

Dans cette section nous proposons un aperçu de l'architecture de la machine virtuelle PCM qui implémente le modèle de calcul des π-threads décrit dans les sections précédentes. Par manque de place, nous nous concentrons dans cette section sur la description en pseudo-code des algorithmes principaux mis en œuvre dans la machine.

4.1. *Architecture*

L'architecture de la machine PCM est très simple et se résume à une structure de données d'*agent* dont le pseudo-code est décrit à gauche de la Table 5. Un agent possède un ensemble chans de (références

```
record Agent {
  chans : Set[Channel]
  current : Thread
  ready : Set[Thread]
  wait : Set[Thread]
}
```

```
record Channel {
  globalrc : Int
  waitrc : Int
  incommits : Set[Commit]
  outcommits : Set[Commit]
}
```

```
record Thread {
  env : Array[Value]
  registers : Array[Value]
  knows : Set[Channel]
  commits : Set[Commit]
  pc : Int
}
```

Tableau 5 – Machine PCM : structures de données principales

de) canaux exploitables par les threads. Le thread courant - en cours d'exécution - est référencé par le champ current. Les champs ready et wait maintiennent la liste des threads, organisée de façon adéquate pour l'algorithme d'ordonnancement utilisé (cf. ci-dessous). Retenons que l'ensemble $\{\mathsf{current}\} \cup \mathsf{ready} \cup \mathsf{wait}$ permet de retrouver l'ensemble des threads gérés par l'agent. Il n'y a aucune autre information partagée par l'ensemble des threads, nous visons ainsi une décentralisation maximale du contrôle.

La structure d'un canal de communication est décrite au milieu de la Table 5. Le champ globalrc contient pour chaque canal la valeur de la fonction $\mathrm{globalrc}$ de la Table 4, c'est-à-dire le nombre de threads ayant *au moins* une référence à ce canal. Pour le ramasse-miettes (cf. ci-dessous), il est également nécessaire de maintenir le nombre de processus ayant des engagements (et donc en attente) sur ce canal. C'est le rôle du champ waitrc. Les champs incommits et outcommits référencent les engagements faits par les processus pour ce canal. On retrouve ces engagements par thread (i.e. l'ensemble Γ), cette redondance d'information est nécessaire pour des raisons d'efficacité.

Les threads sont structurés comme indiqué à droite de la Table 5. On remarque ici aussi l'absence de pile d'exécution tant au niveau de

l'agent global que des threads individuels. En revanche, chaque thread possède un environnement local env ; il s'agit d'un tableau dont la taille est calculée lors de la compilation. Les variables sont les indices du tableau et référencent les valeurs. On voit ici que la structure de l'environnement lexical n'est effectivement pas arborescente. Chaque thread possède également son propre jeu de registres. Le champ knows enregistre les canaux connus par le processus (ce qui permet de ne pas avoir à scruter l'environnement local). Cela correspond, pour le thread considéré, aux antécédents de true pour le prédicat knows de la Table 4. Les engagements du processus sont enregistrés dans le tableau commits. Finalement, le compteur de programme pc pointe sur l'instruction courante dans le segment de code.

Le jeu d'instruction de la machine PCM est relativement réduit. En dehors des instructions permettant de manipuler des types de base et d'effectuer des calculs, un nombre réduit d'instructions sont proposées pour permettre de gérer les processus et les canaux de communication. L'interprétation de ces instructions se fait par requête/modification de l'état de la machine PCM. Par faute de place nous ne donnons pas le détail de ces instructions.

4.2. *Ordonnanceur*

L'ordonnanceur de la machine PCM est extrêmement simple, ce qui est selon nous sa caractéristique fondamentale. Il possède deux étages distincts :

1) L'ensemble des threads prêts à être exécutés, dans le champ ready de la structure d'agent.

2) L'ensemble wait des threads en attente de la satisfaction d'un de leurs engagements.

Pour qu'un thread $[\Gamma; \delta] : P$ soit accepté dans la liste ready, il est nécessaire de vérifier $\Gamma = \emptyset$ c'est-à-dire que le processus n'a pas d'engagement. De façon complémentaire, pour que le thread soit accepté dans wait ou sleep il faut qu'il ait des engagements et donc $\Gamma \neq \emptyset$. Cet invariant est extrêmement simple mais néanmoins très utile.

Le fonctionnement de base de l'ordonnanceur suit très précisément

les règles de la sémantique. Au démarrage, nous choisissons un processus dans ready et nous effectuons un changement de contexte. L'instruction pointée par le pc est ensuite exécutée. Si cette instruction est un calcul, un **tau** ou toute autre instruction exécutable de suite, alors l'exécution pour ce thread se poursuit. Un système de *fuel* permet de forcer les changements de contexte mais sinon - outre un yield explicite - il existe trois façons de parvenir à un changement de contexte :

– si le thread effectue une émission et qu'aucun partenaire n'est disponible (cf. règle (out) de la sémantique). Cette information est obtenue de façon immédiate puisqu'il suffit de vérifier que la liste des engagements de réception est vide pour le canal considéré (cf. champ incommits de la structure Channel). Dans ce cas un nouvel engagement d'émission est enregistré et le processus est placé dans la liste wait. Un autre processus est ensuite choisi dans ready.

– de façon symétrique si le thread effectue une réception et qu'aucun partenaire n'est disponible (cf. règle (in)).

– dans le cas d'un choix non-déterministe, le procédé est le suivant. Les branches du choix sont testées dans l'ordre de leur description. Pour la branche courante, si le garde est évalué à true et que l'action gardée est exécutable, alors la branche est choisie immédiatement. Sinon un engagement est enregistré et on passe à la branche suivante (cf. règle $(next)$). Si l'on a parcouru toutes les branches et qu'aucune n'est exécutable, alors le thread est mis en attente.

La principale caractéristique de cet algorithme d'ordonnancement est qu'il fonctionne en temps constant. Ceci représente une avancée certaine par rapport à notre précédente implémentation [15] qui demandait une phase (linéaire) de recherche de partenaire.

4.3. *Ramasse-miettes*

L'implémentation de l'algorithme de ramasse-miettes occupe souvent une part importante de la problématique globale d'implémentation d'une machine virtuelle, non seulement en nombre de lignes de code mais également en ce qui concerne la complexité de ce code. Un bon exemple est celui de la machine virtuelle *HotSpot* pour Java [5]. Nous

proposons dans cet article un changement assez radical de point de vue en passant des cellules (mémoire) aux processus et canaux. Comme le suggèrent les règles $(reclaim)$ et $(stuck)$ de la Table 4, les principes du GC de la machine PCM sont étonnamment simples. Tout d'abord, de façon classique, tout canal dont le comptage de référence atteint 0 est directement récupéré. Il n'y a pas d'algorithme séparé pour traiter ce cas, ce qui est très avantageux en matière de concurrence. En fait, la seule information partagée et à synchroniser est le compteur de références globales.

De façon similaire, avant de vérifier si une émission ou une réception est possible, nous pouvons tester la valeur du champ globalrc sur le canal concerné. Si ce compte est à 1 alors cela veut dire que le canal n'est connu que du processus qui essai d'émettre ou de recevoir. Dans une structure de choix, si tous les canaux sont de la même façon connus uniquement du processus courant, alors le processus est indéfiniment bloqué et on peut le récupérer. On exploite ici la régle dérivée suivante :

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

Encore une fois, l'algorithme de GC associé est mis en œuvre de façon purement locale par chaque processus. Notons qu'il s'agit d'une optimisation car c'est un cas particulier de la règle $(stuck)$ pour un processus unique. En particulier, $(stuck_1)$ ne couvre pas les problèmes de détection de cycles. Dans les algorithmes basés sur le comptage de références, la principale problématique concerne justement le ramassage efficace des structures cycliques. En général, les algorithmes sont basé sur de la détection de cycle à l'intérieur d'un graphe [12]. Plutôt que de détecter explicitement des cycles, nous nous intéressons dans notre cas à la détection de terminaison (partielle) d'un ensemble de processus. Dans notre contexte composé uniquement de canaux et de processus, la terminaison partielle est un problème plus général que la détection de cycle - et sa solution plus simple !

D'un point de vue algorithmique, le principe de la règle $(stuck)$ est de calculer une clique de processus vérifiant une propriété sur les compteurs de références des canaux sur lesquels ils attendent. L'hypothèse de

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

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

Tableau 6 – Algorithme : mise en attente et réveil de thread

la règle concerne le calcul de la somme des références vers les canaux bloqués dans l'ensemble des processus qui n'attendent pas sur ces canaux. On doit obtenir un compte de 0 c'est-à-dire que les autres processus ne connaissent pas les canaux qui bloquent la clique, et donc celle-ci ne peut être réveillée. Il serait très coûteux de calculer à chaque fois la somme des références locales pour tous les processus à l'extérieur de la clique. En revanche, il est très facile de maintenir le compte du nombre de processus qui attendent sur les canaux. C'est le rôle du champ waitrc de la structure de donnée implémentant les canaux dans la PCM (cf. Table 5). Lors de la mise en attente ou le réveil d'un thread, il faut donc mettre à jour ce compte, comme indiqué sur la Table 6.

Considérons la proposition suivante :

Proposition 4.1 *Pour tout canal ch, si ch.waitrc=ch.globalrc alors tous les processus qui connaissent le canal sont en attente.*

Quoique évidente, cette proposition se trouve au cœur de notre algorithme de GC de seconde génération, dont le pseudo-code est indiqué sur la Table 7. Le but du jeu est de construire de façon incrémentale - et de façon concurrente - une clique de processus de la forme

```
procedure gc2(a :Agent,th :Thread) {
  var clique :Set[Thread] := ∅
  var candidates :Set[Thread] := { th }
  do {
    var candidate :Thread := 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 :Thread ∈ clique {
    reclaim(stuck)
  }
}
```

Tableau 7 – Algorithme : collecteur de seconde génération

$\prod_i [\Gamma_i; \delta_i, \mathsf{pid} \triangleright \hat{p}_i] : \sum \mathbf{wait}$ qui soit isolée en termes de canaux. L'algorithme procède ainsi. Il faut tout d'abord choisir un candidat parmi les threads en attente dans l'agent. Des heuristiques intéressantes existent (notamment, choisir un processus en attente «depuis longtemps») mais concentrons-nous sur l'essentiel en considérant que le choix peut se faire de façon arbitraire. Il s'agit donc d'un paramètre de l'algorithme : la variable th. A l'initialisation, on part d'une clique vide dans la variable clique. Le premier candidat th est placé dans un ensemble candidates qu'il faut analyser pour construire la clique. Dans la boucle principale de l'algorithme, on essaye de «vider» l'ensemble des candidats afin de «remplir» la clique. Pour cela, on choisit d'abord un thread quelconque dans candidates. L'ensemble des canaux qui nous intéresse est celui référencé par l'ensemble des engagements du thread bloqué. Si l'un de ces canaux est connu par un processus qui n'est pas en attente, alors on est sûr qu'au moins un thread à l'extérieur de la clique en construction connaît le canal et peut donc potentiellement la

réveiller. Dans ce cas, on abandonne cette instance de l'algorithme[8]. En revanche, si l'équation ch.waitrc=ch.globalrc est vérifiée alors tous les threads qui ont des engagements pour le canal ch sont en attente. Ils sont donc candidats pour notre clique et sont ajoutés à candidates sauf s'ils sont déjà présents dans clique. Si l'on a réussi à vider candidates alors on a construit une clique valide vis-à-vis de $(stuck)$. La clique est donc intégralement ramassée.

5. Travaux connexes

Il existe une grande variété d'abstractions linguistiques pour la programmation parallèle. Nous nous intéressons principalement aux langages et implémentations basées sur une métaphore de processus communicants via des canaux de communication. Concernant le π-calcul - notre objet d'étude et d'expérimentation - l'implémentation la plus connue est Pict, qui correspond en fait à une variante asynchrone et sans opérateur de choix [17]. En comparaison, nos implémentations sont plus «fidèles» au calcul original car synchrones et proposant un opérateur de choix - un outil selon-nous vraiment indispensable en terme d'expressivité. Une autre implémentation du π-calcul est notre propre CubeVM [15] qui possède elle aussi un opérateur de choix. Ce dernier est non ordonné mais favorise les émissions, ce qui entraîne un déséquilibre très artificiel. Surtout, l'algorithme d'ordonnancement associé est peu efficace. La nouvelle sémantique opérationnelle que nous proposons, à base d'engagements explicites, ne souffre pas de ce déséquilibre. En complément, l'algorithme d'ordonnancement qui en résulte est bien plus efficace. Notons également l'introduction d'une seconde génération pour le ramasse-miettes concurrent. Le ramassage efficace des cycles dans les algorithmes basés sur le comptage de références est toujours une problématique assez complexe et d'actualité [12]. L'originalité de notre approche est de nous attaquer en fait à un autre problème, celui de la détection de terminaison partielle d'un ensemble de processus concurrents. Nous l'avons vu, la solution à ce problème est

8. De façon intéressante, il est possible de faire tourner plusieurs instances de l'algorithme de GC de seconde génération. Les différentes instances peuvent coopérer en mixant leurs cliques si elles ont une intersection non-vide. Nous n'avons pas encore implémenté cette variante parallèle.

plus simple (dans une machine où tout est processus ou canal) que la détection de cycle explicite.

L'extension CML (*Concurrent ML*) pour SML/NJ offre un mode de programmation basé sur les interactions via des canaux synchrones ainsi qu'une implémentation particulièrement véloce [18]. Une variété d'opérateurs de choix est encodable grâce à des primitives de sélection et de composition. Il est même possible de proposer un choix dynamique. Bien sûr, cette généralisation est coûteuse, en particulier si l'on pense à sa caractérisation formelle. L'opérateur de choix du π-calcul, malgré son expressivité, est à la fois plus simple et plus sûr. Notre compilateur, par exemple, peut étudier certaines propriétés des choix, notamment l'absence de cycle et donc de «*deadlock* par construction» [9]. Une autre différence fondamentale est que CML ne franchit pas le pas du «tout processus». La gestion des ressources (processus, canaux) doit être intégrée dans l'implémentation hôte qui possède ses propres mécanismes, notamment pour le ramasse-miettes. Dans notre cas, tout est processus ou canal, ce qui uniformise et simplifie la gestion des ressources.

Récemment, la firme *Google* a lancé le langage de programmation Go qui supporte la concurrence à l'aide de *goroutines* (sic) et intègre des primitives de synchronisation inspirées par CSP [8, 3]. Il est un peu tôt pour pouvoir comparer les deux approches mais au moins une caractéristique de Go nous intéresse. L'implémentation utilise la notion de *pile segmentée* pour permettre de «plaquer» les co-routines sur des threads système. Le principe est simple : la pile principale est découpée en zones de quelques kilo-octets, chaque zone étant attribuée à une co-routine donnée. Si cette zone est dépassée (par un nombre trop important d'appel récursifs par exemple), alors le tas est utilisé pour simuler ce qu'il manque de pile. Cette technique peut également être employée avec les π-threads pour pré-allouer l'environnement local des processus sur la pile du processeur. Le grand avantage des π-threads est que la taille nécessaire est connue au moment de la compilation.

La *programmation fonctionnelle monadique* permet de composer de façon élégante les traits purement fonctionnels et les traits moins «purs» comme les effets de bord, les entrées/sorties et même la concurrence. Du point de vue de la métaphore de programmation, les *M-vars* [10] ne

permettent pas les choix non-déterministes et restent donc limitées en terme d'expressivité. Ce n'est pas le cas du choix transactionnel [7] qui est de fait plus «puissant» que le choix «un coup en avance» du π-calcul. Mais il est également selon nous beaucoup plus lourd à implémenter et surtout plus difficile à caractériser et analyser formellement. Pour les langages à processus communicants, le gain en terme d'expressivité ne nous semble pas évident non plus. En revanche, la construction de structures transactionnelles au dessus des primitives simples du π-calcul sont tout à fait envisageables.

Les structures de synchronisation *join patterns* de JoCaml [6] offrent un mode de composition orthogonal au choix non-déterministe. Si ce dernier joue le rôle d'un «OU», les joins sont des «ET» et les deux sont clairement combinables. Nous avons ajouté le support natif des join patterns (une variante synchrone contrairement à JoCaml asynchrone) dans notre bibliothèque LuaPi (élaborée au dessus des coroutines du langage de scripts Lua) mais il est possible de les encoder dans le langage de base. Par exemple le *join pattern* {c ?(x) & d !v}.P (réception sur c et émission sur d de façon simultanée, puis continuation en P) peut être encodé par : **def** J = c ?(x),{c !x,J+d !v.P}

Le langage Erlang [2] propose des abstractions de type «acteurs» [1] pour la programmation concurrente. Les processus communiquent de façon asynchrone par passage de message sans notion explicite de canal. La métaphore employée est donc assez différente du π-calcul. Le fait en π-calcul de ne pas nommer les processus - et d'utiliser à la place des identificateurs de canaux - offre selon nous un meilleur découplage entre les processus. JoCaml et Erlang adoptent une sémantique de communication asynchrone. Nous favorisons l'interprétation synchrone du π-calcul pour plusieurs raisons. Tout d'abord, la sémantique synchrone est plus naturellement caractérisable formellement, et les systèmes de files d'attentes nécessaires pour simuler l'asynchronisme sont coûteux de ce point de vue. De nombreux protocoles de synchronisation s'expriment beaucoup plus naturellement avec des primitives atomiques pour la synchronisation. Il est de plus beaucoup plus naturel d'encoder l'asynchronisme (s'il est nécessaire) en utilisant ces primitives que d'effectuer le travail inverse. En revanche, on perd l'extension naturelle aux systèmes répartis qui eux sont essentiellement asynchrones.

6. Conclusion

Nous motivons dans cet article l'utilisation d'une variante appliquée du π-calcul – les π-threads – comme «langage noyau» pour un environnement de programmation concurrente à la fois *sûr* et *efficace*. Pour appuyer cette idée, nous proposons une sémantique opérationnelle assez simple et «naturelle» selon-nous, basée sur la notion d'engagements explicites faits par les processus sur les canaux de communications qui leur servent pour se synchroniser et communiquer. Nous donnons également un aperçu de la machine virtuelle PCM qui concrétise ce langage noyau. Cette machine innove selon-nous sur deux aspects : (1) son ordonnanceur à base d'engagements explicites à la fois simple et efficace (en théorie tout du moins même si nos premières expérimentations sont assez concluantes) et (2) son algorithme de ramasse-miettes basé sur le comptage de références ramassant également les cycles et autres structures bloquantes.

Nous disposons de trois implémentations du calcul des π-threads : deux bibliothèques de programmation (pour Lua et Java) ainsi qu'un compilateur qui cible directement la machine PCM. La bibliothèque Java permet d'exploiter les architectures multi-cœurs. Les performances de ces implémentations sont encourageantes (cf. notre site pour le projet[9]). Nous concevons à l'heure actuelle une passe supplémentaire de compilation pour l'analyse statique de programme par inférence de types à base de processus [9]. L'idée est de détecter de façon statique certains cas d'erreur non triviaux (ex. certaines situations de *deadlock*) mais également de permettre certaines optimisations comme la détection automatique des *canaux linéaires* [19]. Nous travaillons également sur une variante de la machine PCM pour les architectures multi-cœurs inspirée par notre bibliothèque Java. Finalement, nous aimerions proposer à plus long terme un compilateur certifié.

9. `http://www-poleia.lip6.fr/~pesch/pithreads`

Références

[1] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1) :1–69, January 1997.

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

[3] The Go authors. *The Go Programming Language Specification*, 2009. `http://golang.org`.

[4] Gérard Boudol. The π-calculus in direct style. *Higher-Order and Symbolic Computation*, 11(2) :177–208, 1998.

[5] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM '04 : Proceedings of the 4th international symposium on Memory management*, pages 37–48, New York, NY, USA, 2004. ACM.

[6] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer.

[7] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8) :91–100, 2008.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[9] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3) :121–163, 2004.

[10] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL*, pages 295–308, 1996.

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

[12] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. An efficient on-the-fly cycle collection. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

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

[14] Frédéric Peschanski and Joël-Alexis Bialkiewicz. Modelling and verifying mobile systems using pi-graphs. In *SOFSEM*, volume 5404 of *Lecture Notes in Computer Science*, pages 437–448, 2009.

[15] Frédéric Peschanski and Samuel Hym. A stackless runtime environment for a pi-calculus. In *VEE '06 : Proceedings of the 2nd international conference on Virtual execution environments*, pages 57–67, New York, NY, USA, 2006. ACM Press.

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

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

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

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