Cahiers GUTenberg n ° 9 — Juillet 91

Programmer dans un langage fonctionnel parallèle avec continuations

Luc Moreau

Service d'Informatique, Institut d'Electricité Montefiore, B28
Université de Liège, Sart-Tilman, 4000 Liège, Belgique
moreau@montefiore.ulg.ac.be

Résumé

Dans cet article, nous montrons que la méthodologie de programmation fonctionnelle avec continuations peut être appliquée pour réaliser des applications parallèles. Cette approche est rendue possible grâce à des opérateurs pour le parallélisme qui sont transparents. La définition de ces opérateurs se base sur une notion de métacontinuation représentant un ordre d'évaluation séquentiel de gauche à droite. Une définition d'un langage fonctionnel avec opérateurs transparents pour le parallélisme est donnée sous forme d'une traduction vers un langage fonctionnel possédant 4 primitives pour le parallélisme du type CCS.

1. Introduction

On considère habituellement qu'il existe deux tendances pour introduire explicitement le parallélisme dans les langages fonctionnels. D'une part, des constructions du type `future`, `pcall` peuvent être ajoutées au langage tout en conservant ses caractéristiques fonctionnelles (MultiLisp [2], [3]). Cela signifie que la méthodologie de programmation fonctionnelle ne doit pas être modifiée pour développer des applications parallèles. D'autre part, on peut ajouter des constructions pour créer des processus et des canaux de communication ainsi que pour communiquer entre processus d'une façon semblable à CCS [14]; différentes implémentations de ce type furent réalisées pour ML [15] (PFL [9], CML [22]). Parmi les avantages de cette dernière approche, l'on peut citer une sémantique non seulement intuitive mais aussi assez simple [13]. Par contre, elle impose au programmeur d'adopter une nouvelle méthodologie de programmation pour développer des applications parallèles car ce langage n'est plus fonctionnel.

Afin de pouvoir conserver une méthodologie de programmation fonctionnelle, il est important que les programmes utilisant les constructions pour le parallélisme retournent les mêmes résultats que les versions *séquentialisées* de ces programmes, c'est-à-dire les mêmes programmes desquels les constructions pour le parallélisme ont été retirées. On dénomme *transparence* cette propriété des opérateurs de parallélisme. Dès lors, on peut considérer les constructions transparentes comme de simples annotations pour une exécution parallèle.

Le langage Scheme [21] fait lui aussi l'objet d'une parallélisation. L'approche couramment adoptée est celle qui consiste à conserver les caractéristiques fonctionnelles en ajoutant la construction `future`. En outre, Scheme possède des continuations de première classe dont l'expressivité pour des applications séquentielles a été montrée dans [5], [6], [7]. Malheureusement, l'introduction du parallélisme dans Scheme soulève le problème de la définition de la notion de continuation dans un langage parallèle. Parmi les différentes solutions rencontrées ([11], [8], [10], [4] et [18]), deux retiennent particulièrement l'attention. D'une part, Katz et Weise [10] proposent une implémentation d'un opérateur `future` transparent en introduisant une notion de légitimité. D'autre part, Queinnec [18] donne une sémantique dénotationnelle pour un dialecte de Scheme appelé Polyscheme, sans pour autant avoir la propriété de transparence mais en proposant un nouveau style de continuations.

Dans [16], nous avons donné une sémantique opérationnelle pour un langage fonctionnel avec continuations et constructions transparentes pour le parallélisme. Cette sémantique est basée sur une notion de continuation d'ordre supérieure. D'autre part, dans [17], nous faisons une comparaison entre une sémantique dénotationnelle basée sur une notion de continuation d'ordre supérieur et une sémantique dénotationnelle basée sur une notion de légitimité semblable à celle proposée par Katz et Weise dans leur implémentation. La notion de continuation d'ordre supérieur permet de réduire le calcul spéculatif par rapport à la notion de légitimité en étant plus conservative lors de l'application de continuations.

Dans ce papier, nous donnons différents exemples de programmation avec les continuations et montrons comment ils peuvent être parallélisés. En particulier, nous reprenons le style de programmation par coroutines [6], [7] qui se prête particulièrement bien à une parallélisation. En plus, nous montrons quelles différences peuvent exister entre un langage avec opérateurs transparents et un langage non transparent tel que PolyScheme proposé

par Queinnec. Dans une seconde partie, nous introduisons brièvement la notion de continuation d'ordre supérieur et grâce à elle, nous donnons une sémantique opérationnelle d'un langage fonctionnel parallèle avec continuations.

2. Λ_C : un langage fonctionnel parallèle avec continuations

Cet article présente une méthodologie de la programmation dans un langage avec continuations et parallélisme que nous appellerons Λ_C. Il est défini comme le sous-ensemble fonctionnel de Scheme auquel on ajoute `call/cc` et des constructions pour le parallélisme. Par défaut, on suppose que l'évaluation est séquentielle dans un ordre de gauche à droite sauf lorsque le parallélisme est explicitement introduit par une des trois constructions suivantes :

Un processus p_1 évaluant `(fork exp)` dans une séquence, crée un processus p_2 évaluant `exp`, la valeur de l'expression `fork` est non spécifiée. Le processus p_1 continue son évaluation en parallèle avec p_2.

Un processus p_1 évaluant l'expression `(pcall M N)` crée un processus p_2 évaluant `M` et un processus p_3 évaluant `N`. Quand les deux valeurs sont calculées, l'application de la valeur de `M` à la valeur de `N` est faite[1].

Un processus p_1 évaluant l'expression `(future exp)` crée un processus p_2 évaluant `exp`; la valeur de la forme `future` est un *placeholder* qui est une structure de donnée destinée à recevoir une valeur, la valeur calculée par le processus p_2. Lorsque le placeholder contient une valeur on dit qu'il est *résolu*. Dans MultiLisp, on fait une distinction entre les fonctions strictes demandant des placeholders résolus afin d'utiliser la valeur qu'ils contiennent et les fonctions non strictes ne demandant pas que les placeholders soient résolus. Si un processus applique une fonction stricte à un placeholder non résolu, il est suspendu jusqu'à sa résolution. Dans Λ_C, nous supposerons que toutes les fonctions sont non strictes (à la manière des fonctions Scheme vis-à-vis d'une valeur retardée par `delay`); on introduit une seule fonction stricte forçant la suspension d'un processus lorsqu'un placeholder n'est pas résolu : `touch` (l'équivalent de `force` pour les streams).

[1]Comme on le verra plus loin, p_1 est arrêté après avoir créé p_2 et p_3; un de ceux-ci réalisera l'application et l'autre sera aussi arrêté.

3. Exemples de programmation

Nous allons donner quelques exemples de programmes fonctionnels et montrer comment ils peuvent être étendus en des versions parallèles.

3.1. La fonction `search-first`

Le premier exemple consiste en une recherche en profondeur de gauche à droite d'une S-expression à la recherche d'un atome satisfaisant un prédicat donné. Une version séquentielle est donnée par

```
(define (search-first tree pred)
  (call/cc
   (lambda (exit)
     (letrec ((loop (lambda (tree)
                      (cond ((atom? tree) (if (pred tree)
                                              (exit tree)
                                              '()))
                            (else (begin (loop (car tree))
                                         (loop (cdr tree))))))))
       (loop tree)))))
```

Lorsqu'un atome satisfaisant le prédicat est trouvé, la continuation `exit` existant à l'entrée de la fonction `search-first` est appliquée. Cette version peut être légèrement modifiée en tenant compte du caractère de première classe et de durée de vie infinie des continuations. Lorsqu'on trouve un atome satisfaisant le prédicat, on le retourne ainsi que la continuation existant à ce moment :

```
(define (search-first tree pred)
  (call/cc
   (lambda (exit)
     (letrec ((loop (lambda (tree)
                      (cond ((atom? tree) (if (pred tree)
                                              (call/cc
                                               (lambda (next)
                                                 (exit (list tree next))))
                                              '()))
                            (else (begin (loop (car tree))
                                         (loop (cdr tree))))))))
       (loop tree)))))
```

L'intérêt de cette version est que, après avoir trouvé le premier atome vérifiant le prédicat, il est possible de reprendre le calcul et de trouver le second en appliquant la continuation `next`.

Afin de paralléliser ce programme, on peut effectuer des recherches sur le sous-arbre de gauche et le sous-arbre de droite en parallèle au lieu de les faire séquentiellement. Cela peut être fait en annotant le premier appel de la fonction `loop` sur le sous-arbre de gauche à l'aide de la construction `fork`.

```
(define (search-first tree pred)
  (call/cc
   (lambda (exit)
     (letrec ((loop (lambda (tree)
                      (cond ((atom? tree) (if (pred tree)
                                              (call/cc
                                               (lambda (next)
                                                 (exit (list tree next))))
                                              '()))
                            (else (begin (fork (loop (car tree)))
                                         (loop (cdr tree))))))))
       (loop tree)))))
```

Lorsqu'un processus évalue l'expression `(fork (loop (car tree)))`, un nouveau processus est créé pour évaluer l'expression `(loop (car tree))` en parallèle. Dans cet exemple, la construction `fork` est plus appropriée que la construction `future` car on n'attend pas que l'expression `(loop (car tree))` dans la forme `begin` retourne une valeur.

D'après sa définition, la construction `fork` crée un processus pour évaluer son argument. Il se pourrait que les conditions de course soient telles qu'un atome soit trouvé dans un sous-arbre de droite avant qu'un atome ne soit trouvé dans un sous-arbre de gauche et que la continuation `exit` soit appliquée à un atome qui n'est pas le premier dans l'ordre gauche-droite. En fait, il n'en n'est rien : `fork` a été défini comme une construction transparente et un programme utilisant la construction `fork` doit retourner le même résultat que le même programme sans la construction `fork`. Par conséquent, une continuation ne sera réellement appliquée dans un sous-arbre de droite que si la recherche dans le sous-arbre de gauche correspondant s'est terminée par un échec. La version parallèle du programme conserve donc la même propriété que la version séquentielle : c'est *toujours* l'atome le plus à gauche satisfaisant le prédicat qui est retourné.

De plus, comme nous le verrons plus tard, lorsqu'on applique une continuation, les processus qui s'exécutent en parallèle ne sont pas

suspendus. Ainsi, dans cet exemple, lorsqu'on a trouvé un atome satisfaisant le prédicat `pred`, les processus lancés en parallèle continuent la recherche d'autres atomes. On est en présence d'un calcul spéculatif : bien que l'on ne sache pas si les atomes suivants seront demandés, leur recherche se fait quand même parallèlement au calcul obligatoire.

A présent, pour afficher tous les atomes d'une S-expression satisfaisant un prédicat, on définit la fonction :

```
(define (display-atoms tree pred)
  (let ((a-leaf (search-first tree pred)))
    (if (null? a-leaf)
        'end
        (begin (display (car a-leaf))
               ((cadr a-leaf) '())))))
```

Dans cette fonction la recherche dans l'arbre est lancée une première fois. Tant que le résultat obtenu est une liste formée d'un atome et d'une continuation pour reprendre le calcul, l'atome est affiché et la continuation est appliquée pour obtenir l'atome suivant.

Il est à noter que la fonction `display-atoms` peut tout aussi bien faire appel à la version séquentielle de `search-first` qu'à la version parallèle. Dans le premier cas, l'affichage des atomes et leur recherche sont entrelacés à la manière de coroutines. Dans le second cas, la version parallèle de `search-first` autorise le calcul spéculatif. Les atomes seront donc cherchés en parallèle avec leur affichage et la recherche pourra devancer l'affichage (suivant la stratégie de scheduling utilisée).

3.2. Le problème du producteur et du consommateur

Dans l'exemple précédent, la fonction `display-atoms` n'est pas considérée comme une coroutine par la fonction `search-first` car c'est toujours la même continuation `exit` qui y est appliquée. Par contre, `search-first` est bien reprise par la continuation `next` là où elle avait été interrompue.

Considérons à présent le problème du producteur et du consommateur dont une version séquentielle est écrite dans un style coroutine. Contrairement à [6], [7], nous travaillons dans un langage purement fonctionnel. Une coroutine est appelée à l'aide de la fonction `resume` qui lui transmet une

paire (continuation de la coroutine appelante et valeur à transmettre à la coroutine appelée).

```
(define (resume coroutine value)
  (call/cc (lambda (k)
             (coroutine (list k value)))))

(define producer
  (lambda (producer-job)
    (lambda (consumer)
      (letrec ((loop (lambda (n pair)
                       (let* ((pair (resume (car pair) n))
                              (new-value (producer-job n)))
                         (loop new-value pair)))))
        (loop 0 consumer)))))

(define consumer
  (lambda (producer consumer-job)
    (letrec ((loop (lambda (producer)
                     (let* ((pair (resume producer 'any))
                            (producer (car pair))
                            (n (cadr pair)))
                       (consumer-job n)
                       (loop  producer)))))
      (loop producer))))
```

Le système de coroutines affichant les nombres de `1` à l'infini est lancé par la fonction `run` :

```
(define (run)
  (consumer (producer (lambda (n) (+ n 1)))
            (lambda (n) (display n) (newline))))
```

Il existe différentes façons de paralléliser le code. Une possibilité est de reprendre la coroutine consommateur en parallèle avec le calcul de l'élément suivant dans le producteur. On remarquera qu'il est nécessaire d'utiliser explicitement l'opérateur `touch`. En effet, avec la sémantique des continuations d'ordre supérieur que nous donnons plus loin, on garantit la transparence de `pcall` mais `future` n'est transparent que lorsque combiné avec `touch`. Dans cet exemple, lorsqu'on appelle une coroutine en parallèle avec `future`, la valeur `pair` retournée par celle-ci devra être précédé de `touch` lorsqu'elle sera effectivement utilisée.

```
(define producer
  (lambda (producer-job)
    (lambda (consumer-value)
      (letrec ((loop (lambda (n pair)
                       (let* ((pair (future (resume (car (touch pair)) n)))
                              (new-value (producer-job n)))
                         (loop new-value pair)))))
        (loop 0 consumer-value)))))
```

Cette solution autorise du calcul spéculatif (dépendant des stratégies de scheduling) mais il est aussi possible d'introduire du parallélisme sans avoir de calcul spéculatif. En effet, on peut effectuer le `producer-job` en parallèle avec la reprise du consommateur dans le producteur mais les calculs des éléments suivants ne seront faits qu'à la demande du consommateur.

```
(define producer
  (lambda (producer-job)
    (lambda (consumer-value)
      (letrec ((loop (lambda (n pair)
                       (pcall loop (producer-job n)
                                   (resume (car pair) n)))))
        (loop 0 consumer-value)))))
```

Le programmeur a donc la possibilité de déterminer le type de parallélisme qu'il veut (spéculatif ou non) en choisissant les expressions à évaluer en parallèle et les constructions de parallélisme.

3.3. `visit` : une comparaison des styles de programmation de Λ_C et de PolyScheme

Dans [18], Queinnec donne un exemple de recherche dans un arbre dans le style PolyScheme. Nous le reprenons mais, contrairement à la version originale, nous faisons apparaître explicitement les appels parallèles à l'aide de la notation `pcall*` indiquant par là qu'elle n'a pas la même sémantique que notre `pcall` car elle n'est pas transparente.

```
(define visit
  (lambda (tree do)
    (if (atom? tree)
        (call/cc (lambda (return)
                   (pcall* list (return tree)
                                (return (do tree)))))
        (begin (visit (car tree) do)
               (visit (cdr tree) do)))))

(call/cc (lambda (exit)
           (prog2 (visit a-tree exit)
                  (local-end))))
```

La fonction `visit` applique son second argument `do` à chaque atome de la S-expression `tree`. L'exploration se fait en profondeur, de gauche à droite mais il n'est pas garanti que la fonction `do` sera appliquée aux feuilles dans cet ordre là. En effet, dans `(pcall* list (return tree) (return (do tree)))` les deux applications se font en parallèle, et le résultat de l'application de `do` à `tree` n'est pas attendu avant de reprendre l'exploration à l'aide de `(return tree)`. Par conséquent, `(return tree)` pourrait lancer l'évaluation d'un `do` sur la feuille suivante alors que l'application sur la feuille courante n'a pas encore commencé.

La solution proposée par Queinnec consiste à poursuivre l'exploration après avoir appliqué `do`. C'est donc au programmeur d'appliquer *explicitement* la continuation `c` reçue en argument dans `do`.

```
(define visit
  (lambda (tree do)
    (if (atom? tree)
        (call/cc (lambda (return)
                   (return (do tree return))))
        (begin (visit (car tree) do)
               (visit (cdr tree) do)))))

(define (print-square tree)
  (visit tree (lambda (leaf c)
                (begin (fork (display (square leaf)))
                       (c leaf)))))
```

Dans notre approche des coroutines, la visite d'un arbre est faite par la fonction `producer`; elle transmet toutes les valeurs au consommateur dans l'ordre de la recherche et il est possible de faire la recherche en parallèle et de façon spéculative :

```
(define (producer tree)
  (lambda (consumer-value)
    (letrec ((loop (lambda (tree consumer-pair)
                     (if (atom? tree)
                         (future (resume (car (touch consumer-pair)) tree))
                         (loop (cdr tree)
                               (future (loop (car tree) consumer-pair)))))))
      (resume (car (touch (loop tree consumer-value))) 'eot)))))
```

La fonction `loop` du `producer` a pour valeur un placeholder qui, lorsqu'il sera résolu, contiendra une paire (continuation du consommateur/valeur transmise par le consommateur). Lorsqu'un `producer` arrive au bout d'un arbre, il produit le symbole `eot` supposé différent de tous les autres.

```
(define consumer
  (lambda (producer pred do)
    (letrec ((loop (lambda (producer)
                     (let* ((pair (resume producer 'any))
                            (producer (car pair))
                            (n (cadr pair)))
                       (if (pred n)
                           (begin (do n)
                                  (loop  producer)))))))
      (loop producer))))

(define (run)
  (consumer (producer '((1 . 2) . 3))
            (lambda (x) (not (eq? x 'eot)))
            (lambda (n) (display (square n)) (newline))))
```

Ici, il est garanti que les feuilles sont transmises dans l'ordre à la fonction `do` car comme les problèmes de production et consommation sont indépendants, ils peuvent être exécutés séparément. On ne crée donc pas un processus pour exécuter en parallèle la fonction `do` : la coroutine `consumer` exécute *séquentiellement* la fonction `do` sur toutes les feuilles tandis que la coroutine `producer` parcourt l'arbre en parallèle et de façon spéculative.

3.4. Le problème `samefringe`

Un dernier exemple de programmation parallèle est le problème `samefringe` où l'on compare les feuillages de deux arbres binaires. La solution proposée est une parallélisation de la version séquentielle consistant en trois coroutines : une coroutine de comparaison et deux coroutines

retournant le feuillage des arbres. Les recherches dans les arbres sont effectuées par la coroutine `producer` vue à la section précédente et la comparaison est faite grâce à `compare` dont voici le code :

```
(define compare
  (lambda (co1 co2)
    (letrec ((loop (lambda (co1 co2)
                     (let* ((result1 (resume co1 'any))
                            (co1 (car result1))
                            (a1 (cadr result1))

                            (result2 (resume co2 'any))
                            (co2 (car result2))
                            (a2 (cadr result2)))
                       (cond ((and (eq? 'eot a1) (eq? 'eot a2)) #t)
                             ((equal? a1 a2) (loop co1 co2))
                             (else #f)))))))
      (loop co1 co2))))

(define samefringe
  (lambda (t1 t2)
    (compare (producer t1)
             (producer t2))))
```

4. $\Lambda_{//}$: un langage fonctionnel avec des primitives de communication à la CCS

Λ_C est le nom du langage fonctionnel avec continuations et opérateurs transparents pour le parallélisme dont nous venons de donner des exemples de programmation. Dans [16], nous donnons une sémantique opérationnelle de Λ_C par une traduction de ce langage vers un langage fonctionnel que l'on a appellé $\Lambda_{//}$; il ne possède pas de continuations mais des primitives de base pour le parallélisme dans la philosophie CCS [14]. Une sémantique opérationnelle pour un ML parallèle avec des primitives de communication semblables est donnée dans [13]. Ces primitives sont au nombre de quatre :

`(fork thunk)` La fonction `fork` prend en argument un thunk (fonction sans argument) et crée un nouveau processus appliquant ce thunk. La valeur de `fork` est non spécifiée.

`(channel)` Les processus échangent des données sur des canaux. La fonction `channel` retourne un *nouvel* objet appelé *channel identifier* sur lequel des communications peuvent être faites.

`(send channel value)` Les communications sont synchrones comme dans CCS. Afin de réaliser une communication, il doit y avoir un processus envoyant une valeur sur un canal et un processus attendant une valeur sur le même canal. La valeur de la fonction `send` est non spécifiée.

`(receive channel)` La valeur de la fonction `receive` est la valeur transmise sur `channel` par le processus ayant envoyé une donnée durant une communication synchrone.

5. Des continuations "symétriques" pour le parallélisme

Dans une première étape, on donne à la figure 1 une traduction du sous-ensemble séquentiel de Λ_C `(M N)`, `(lambda (x) M)`, `x`, `(call/cc M)` auquel on ajoute la construction pour le parallélisme `(pcall M N)`. Une traduction est donnée par un ensemble de règles ayant le schéma : $[\![$`Term`$]\!]$=`exp`. Le membre de gauche de la règle est un terme de Λ_C entre crochets et le membre de droite est une expression dans $\Lambda_{//}$. Une telle règle doit être lue comme "le texte de la traduction de `Term` est `exp`, dans lequel toute occurrence de $[\![e]\!]$ doit être remplacée par le texte de la traduction de e et toutes les nouvelles variables introduites dans `exp` sont supposées ne pas entrer en collision avec celles déjà existantes".

Les quatres premières règles définissent la sémantique des expressions séquentielles et sont habituellement appelées traduction "*continuation passing style*" [1]. On y remarque la forme des continuations de `M` et `N` dans l'application séquentielle : la continuation de `M` évalue `N` et la continuation de `N` réalise l'application. Ces continuations spécifient un ordre total d'évaluation de gauche à droite. Par contre, pour une application parallèle de `M` à `N`, deux processus sont créés pour évaluer `M` et `N` en parallèle. Les continuations sont définies selon la même technique que celles proposées Queinnec dans sa définition de PolyScheme [18]. La continuation de `M` (resp. `N`) sauve dans un emplacement `cm` (resp `cn`) une fonction, lit une autre fonction `fn` (resp. `fm`) dans l'emplacement `cn` (resp. `cm`) et applique cette fonction `fn` (resp. `fm`) à la valeur de `M` (resp. `N`) et la continuation courante κ. Si `N` (resp. `M`) n'est pas encore évalué, cette fonction `fn` (resp. `fm`) est la valeur initiale se trouvant dans l'emplacement `cn` (resp. `cm`) et le processus évaluant `M` (resp. `N`) est arrêté faute de code à exécuter. Il est garanti qu'un seul des deux processus fera l'application car les emplacements `cm` et `cn` sont considérés comme une section critique. Les canaux `cn` et `cm`

```
⟦x⟧             = (lambda (κ) (κ x))
⟦(lambda (x) M)⟧= (lambda (κ) (κ (lambda (x c) (⟦M⟧ c))))
⟦(call/cc M)⟧   = (lambda (κ) (⟦M⟧ (lambda (vm) (vm (lambda (v κ') (κ v)) κ))))
⟦(M N)⟧         = (lambda (κ) (⟦M⟧ (lambda (vm) (⟦N⟧ (lambda (vn) (vm vn κ))))))
⟦(pcall M N)⟧   =
  (lambda (κ)
    (let ((cn (channel)) (cm (channel)) (sem (channel)))
      (begin (fork (lambda () (⟦M⟧ (lambda (vm)
                                    (begin (receive sem)
                                           (write cm (lambda (vn x) (vm vn x)))
                                           (let ((fn (read cn)))
                                             (begin (send sem 'any)
                                                    (fn vm κ))))))))
             (fork (lambda () (⟦N⟧ (lambda (vn)
                                    (begin (receive sem)
                                           (write cn (lambda (vm x) (vm vn x)))
                                           (let ((fm (read cm)))
                                             (begin (send sem 'any)
                                                    (fm vn κ))))))))
             (make-store cm (lambda(vn κ) nil))
             (make-store cn (lambda(vm κ) nil))
             (make-store sem 'any))))
```

Figure 1. Définition de Λ_C avec des continuations symétriques

sont nouvellement créés et représentent des emplacements mémoire dont la définition est donnée à la figure 2.

Selon cette technique, les continuations de `M` et de `N` sont symétriques contrairement à celles de l'application séquentielle qui sont asymétriques. Dans la suite du texte, on parlera de continuations symétriques ou asymétriques pour référencer ces deux types de continuations.

Les caractéristiques de Λ_C défini suivant la technique des continuations symétriques (figure 1) sont les mêmes que celles de PolyScheme. Elles peuvent se résumer par :

- tout programme n'appliquant pas de continuation réifiée par `call/cc` retourne toujours le même résultat que la version séquentialisée de ce programme,
- quand les continuations sont utilisées, des réponses multiples peuvent être retournées; il existe toujours une exécution du programme

```
(define (make-store c init-value)
  (fork (lambda ()
          (letrec ((loop (lambda (v)
                           (begin (send c v)
                                  (receive c))))
            (loop init-value)))))


(define (read c)                (define (write c v)
  (let ((value (receive c)))      (begin (receive c)
    (begin (send c value)                (send c v))))
           value)))
```

Figure 2. Définition d'un store dans $\Lambda_{//}$ à l'aide de processus

qui retourne la même solution que celle retournée par sa version séquentialisée,

- pour certains programmes, le nombre de solutions retournées peut varier d'une exécution à l'autre.

Si l'on considère l'exemple suivant,

```
(pcall f1 (call/cc (lambda (k)
                     (pcall (pcall f2 (k 1))
                            (k 2)))))
```
(1)

la continuation `k` est appliquée[2] à `1` et `2`. Ce programme peut retourner quatre réponses différentes: le résultat de l'application de `f1` à `1`, le résultat de l'application de `f1` à `2`, le résultat de l'application de `f1` à `1` suivi du résultat de l'application de `f1` à `2`, le résultat de l'application de `f1` à `2` suivi du résultat de l'application de `f1` à `1`, les deux dernières réponses étant des résultats multiples.

[2]Lorsqu'on applique une continuation, on ne suspend pas les processus s'exécutant en parallèle. En effet, une continuation reifiée est du type `(lambda (v` κ'`) (`κ `v))`; elle abandonne uniquement la continuation du processus courant.

6. Définition de Λ_C

6.1. La construction `pcall`

On souhaite que les programmes de Λ_C utilisant des constructions parallèles soient équivalents aux versions séquentialisées de ces programmes. Par conséquent, le programme 1 doit être équivalent au programme

```
(f1 (call/cc (lambda (k)
               ((f2 (k 1)) (k 2)))))
```
(2)

dont la valeur est le résultat de l'évaluation de `(f1 1)`. Dans la sémantique séquentielle, la continuation `k` est seulement appliquée à `1` car on a un ordre d'évaluation de gauche à droite des expressions dans une application.

Nous allons à présent modifier la sémantique donnée dans la section précédente afin que la continuation `k` soit appliquée seulement à `1` et non pas à `2` dans l'exemple 1.

C'est parce que les continuations du "*continuation passing style*" définissent un ordre total entre expressions qu'il n'y a pas de parallélisme. D'autre part, c'est parce que les continuations sont totalement symétriques dans la définition de `pcall` (figure 1) qu'elle peuvent toujours être appliquées sans respecter la sémantique séquentielle. Le seul ordre d'évaluation est un ordre partiel : le corps d'une fonction est évalué après les arguments. La solution que nous cherchons est un compromis entre ces deux extrêmes : nous conservons les continuations symétriques car elles offrent un maximum de parallélisme et nous introduisons une notion de *continuation d'ordre supérieur* qui représente l'ordre gauche droite; nous l'appelons aussi *métacontinuation*.

Sachant que suivant la sémantique donnée dans la section précédente, seuls les programmes parallèles utilisant explicitement des continuations peuvent retourner des résultats différents de leurs versions séquentialisées, la notion de métacontinuation sera utilisée lorsqu'une continuation réifiée doit être appliquée. Au moment de l'application d'une telle continuation dans une expression parallèle, la métacontinuation vérifie que toutes les sous-expressions qui sont évaluées dans la version séquentialisée de cette

expression au moment de l'application de cette même continuation sont bien évaluées. Si cette condition est satisfaite, l'application de la continuation peut se faire sinon, l'application "est suspendue" jusqu'à ce qu'elle soit valide. Pour ce faire, on définit une notion de *left expressions* concernant l'ordre d'évaluation gauche droite.

On dénomme par le terme *left expressions* relatives à l'application d'une continuation, toutes les expressions qui doivent être évaluées avant d'appliquer cette continuation. Si `k` désigne une continuation et `(k v)` est une sous-expression `e` de `E`, on peut calculer $\mathcal{L}_{\texttt{e}}(\texttt{E})$ l'ensemble des *left expressions* relatives à l'application d'une continuation `(k v)` dans l'expression `E` par l'induction suivante:

$$\mathcal{L}_{\texttt{e}}(\texttt{e}) = \emptyset \tag{3}$$

$$\mathcal{L}_{\texttt{e}}(\texttt{E}) = \alpha \quad \supset \quad \begin{cases} \mathcal{L}_{\texttt{e}}((\texttt{E N})) = \alpha & (a) \\ \mathcal{L}_{\texttt{e}}((\texttt{M E})) = \alpha & (b) \\ \mathcal{L}_{\texttt{e}}((\texttt{pcall E N})) = \alpha & (c) \\ \mathcal{L}_{\texttt{e}}((\texttt{pcall M E})) = \alpha \cup \{\texttt{M}\} & (d) \\ \mathcal{L}_{\texttt{e}}((\texttt{call/cc (lambda (x) E)})) = \begin{cases} \emptyset & \textit{si } (\texttt{eq? k x}) \\ \alpha & \textit{sinon} \end{cases} & (e) \end{cases} \tag{4}$$

$$\mathcal{L}_{\texttt{e}}(\texttt{E}) = \alpha \quad \supset \quad \begin{cases} \mathcal{L}_{\texttt{e}}(((\texttt{lambda (x) E) M})) = \alpha \\ \mathcal{L}_{\texttt{e}}((\texttt{pcall (lambda (x) E) M})) = \alpha \end{cases} \tag{5}$$

La règle 4.d ajoute `M` à l'ensemble des left expressions de `E` dans l'expression `(pcall M E)` et la règle 4.e limite les left expressions à la portée dynamique du `call/cc` qui a capturé `k`. Cela permet à une fonction d'utiliser de façon interne une continuation sans devoir se synchroniser avec le reste du programme dans lequel elle est appelée. Dans l'exemple 1, cela permet d'appliquer `k` sans que `f1` soit évalué. La règle 5 lie la notion de left expression à la règle d'application d'une fonction.

On peut donc exprimer les contraintes que l'on impose lors de l'application d'une continuation :

> *Une continuation peut être appliquée dans la sémantique parallèle si et seulement si toutes les "left expressions" ont été évaluées et ont retourné une valeur.*

Cette contrainte définit un nouvel ordre partiel d'évaluation, sous-ensemble de l'ordre partiel défini par les continuations symétriques, et

contenant l'ordre total défini par la traduction "*continuation passing style*" : non seulement toutes les sous-expressions d'une application doivent être évaluées avant le corps de la fonction mais toutes les left expressions relatives à l'application d'une continuation doivent aussi être évaluées avant d'appliquer cette continuation. Il faut remarquer que les contraintes introduites pour respecter la sémantique séquentielle n'interviennent que lorsqu'on applique une continuation.

La traduction de Λ_C avec la construction `pcall` est donnée à la figure 3. A présent, le résultat de la traduction est une fonction à deux arguments : une continuation et une métacontinuation. La première représente "la suite du calcul" dans le sens défini à la section précédente et la seconde est chargée de vérifier les contraintes en cas d'application d'une continuation. Une continuation réifiée est de la forme `(lambda (v c` γ`) ((`γ κ`) v))`, c'est-à-dire que lorsqu'on applique une telle continuation, on compose la métacontinuation courante γ avec la continuation implicite capturée κ. La métacontinuation γ procède de la façon suivante: elle recherche d'abord la plus proche des left expressions qui ne soit pas évaluée. S'il n'y en a pas, le résultat de cette composition est κ. Sinon, soit `E` cette expression, `E` faisant partie de l'expression `(pcall E N)`. Elle sauve comme valeur de `N`, la fonction `(lambda (vm c` γ`) ((`γ κ`) v))` dans `cn` associé à `N`. Lorsque `E` sera évalué, la continuation de `E` applique la fonction `fn` rangée dans `cn` et reprend l'application de la continuation en poursuivant la vérification des left expressions suivantes.

6.2. La construction `fork`

`fork` est une construction parallèle qui doit apparaître dans une séquence. Elle crée un processus qui évalue son argument. Dans la séquence `(begin (fork exp1) exp2)`, `exp1` est évaluée en parallèle avec `exp2` et la valeur de `exp1` n'est pas utilisée. Mais notre sémantique doit garantir que si une continuation est appliquée dans `exp2`, elle peut échapper de `exp2` si et seulement si elle est appliquée dans la définition séquentielle, c'est-à-dire si `exp1` est évaluée et a retourné une valeur.

A présent que nous avons défini `pcall` dans notre langage, il est simple de définir `fork`. `fork` doit apparaître dans une séquence qui est en fait du "sucre syntaxique" pour une application d'une lambda. La définition donnée à la figure 4 empêche tout échappement de `N` à moins que `M` ne soit évalué et ait retourné une valeur. La valeur de la séquence (la valeur de `M`) est

retournée seulement lorsque `N` est évalué.

```
⟦x⟧              = (lambda (κ γ) (κ x))
⟦(lambda (x) M)⟧ = (lambda (κ γ) (κ (lambda (x κ γ) (⟦M⟧ κ γ))))
⟦(call/cc M)⟧    = (lambda (κ γ)
                     (let ((f (lambda (v c γ) ((γ κ) v)))
                           (γ' (lambda (cont) (if (eq? cont κ) cont (γ cont))))))
                       (⟦M⟧ (lambda (vm) (vm f κ γ')) γ))
⟦M N⟧            = (lambda (κ γ)
                     (⟦M⟧ (lambda (vm) (⟦N⟧ (lambda (vn) (vm vn κ γ))
                                            γ))
                          γ))
⟦(pcall M N)⟧    =
(lambda (κ γ)
  (let ((cn (channel)) (cm (channel)) (sem (channel)))
    (begin (fork (lambda () (⟦M⟧ (lambda (vm)
                                   (begin (receive sem)
                                          (write cm (lambda (vn κ γ) (vm vn κ γ)))
                                          (let ((fn (read cn)))
                                            (begin (send sem (lambda (cont s f)
                                                               (lambda (s)
                                                                 (begin(s v)
                                                                       ((γ cont) v)))))
                                                   (fn vm κ γ)))))
                                 γ)))
           (fork (lambda () (⟦N⟧ (lambda (vn)
                                   (let ((f (receive sem)))
                                     (begin (write cn (lambda (vm κ γ) (vm vn κ γ)))
                                            (let ((fm (read cm)))
                                              (begin (send sem f)
                                                     (fm vn κ γ))))))
                                 (lambda (cont)
                                   (let ((f (receive sem)))
                                     (f cont
                                        (lambda (v) (send sem f))
                                        (lambda (v)
                                          (begin (write cn (lambda (vm κ γ)
                                                             ((γ cont) v)))
                                                 (send sem f)))))))))
           (make-store cm (lambda(vn κ γ) nil))
           (make-store cn (lambda(vm κ γ) nil))
           (make-store sem (lambda (cont s f) f)))))
```

Figure 3. Définition de Λ_C avec les métacontinuations

```
⟦(begin (fork M) N)⟧ = ⟦((lambda (x) N) (fork M))⟧
                     = ⟦(call/cc (lambda (k)
                                   (pcall (let ((x  M)) (lambda (u) u)) (k N))))⟧
```

Figure 4. Translation rule for `fork`

6.3. La construction `future`

A la figure 5, on trouve la traduction de l'expression `(M (future N))`. Contrairement à `(pcall M N)`, c'est toujours la continuation de `M` qui fait l'application. Si `N` n'est pas encore évalué, l'application se fait à un placeholder qui est ici une structure de données contenant une fonction. Lorsque l'opérateur `touch` est appliqué à un placeholder, la fonction qui y est contenue est appliquée et force la lecture d'une valeur sur le canal `val`. Une même valeur est constamment envoyée sur ce canal par un processus `emitter` : il s'agit de la première valeur transmise à la continuation de `N` et envoyée sur `vali`. Les autres valeurs envoyées sur `vali` sont simplement réceptionnées par un processus `sink`. `emitter` et `sink` sont deux fonctions qui, respectivement, envoie toujours la même valeur sur un canal et reçoit des valeurs sur un canal.

Si `N` retourne plus d'une fois, la continuation de `N` applique la valeur de `M` à la valeur de `N` et non pas au placeholder. Cela correspond à la sémantique donnée par [10] et préserve bien la propriété qu'un placeholder ne peut contenir qu'une seule valeur.

Queinnec dans [20] donne aussi une définition de `future`. Notre approche diffère de la sienne car le mécanisme de gestion de la queue associée à un placeholder n'est pas explicite mais se trouve caché dans la sémantique de la fonction `receive` qui suspend les processus l'exécutant si aucune valeur n'est envoyée sur le canal.

7. Comparaisons avec d'autres travaux

Dans cet article, nous reprenons un style de programmation avec coroutines. Ce style avait déjà été proposé comme application des continuations dans [6] et [7]. Il existe deux différences essentielles par rapport à ces articles : sachant que nous travaillons avec un langage purement

fonctionnel, il n'est pas possible d'utiliser la fonction `make-coroutine` retournant une closure qui altère son état local. D'autre part, dans [6], [7], les coroutines sont étudiées dans un cadre séquentiel alors que nous les

```
⟦(M (future N))⟧  =
(lambda (κ γ)
  (let ((cn (channel)) (cm (channel)) (sem (channel))
        (val (channel)) (vali (channel)))
    (begin (fork (lambda () (⟦M⟧ (lambda (vm)
                                   (begin (receive sem)
                                          (write cm (lambda (vn κ γ) (vm vn κ γ)))
                                          (let ((fn (read cn)))
                                            (begin (send sem (lambda (cont s f)
                                                               (lambda (s)
                                                                (begin(s v)
                                                                      ((γ cont) v)))))
                                                   (fn vm κ γ)))))
                                 γ)))
           (fork (lambda () (⟦N⟧ (lambda (vn)
                                   (let ((f (receive sem)))
                                     (begin (send vali vn)                      ; ***
                                            (write cn (lambda (vm κ γ) (vm vn κ γ)))
                                            (if (not(eq? (receive val) vn))    ; ***
                                                (let ((fm (read cm)))
                                                  (begin (send sem f)
                                                         (fm vn κ γ)))
                                                (send sem f)))))
                                 (lambda (cont)
                                   (let ((f (receive sem)))
                                     (f cont
                                        (lambda (v) (send sem f))
                                        (lambda (v)
                                          (begin (write cn (lambda (vm κ γ)
                                                             ((γ cont) v)))
                                                 (send sem f)))))))))
           (make-store cm (lambda(vn κ γ) '()))
           (make-store cn (lambda(vm κ γ)
                            (vm (make-placeholder (lambda () (receive val))) κ γ)))
           (make-store sem (lambda (cont s f) f))
           (let ((the-future-value (receive vali)))                           ; ***
             (begin (fork (lambda () (emitter val the-future-value)))
                    (fork (lambda () (sink vali))))))))

(define (touch object) (if (placeholder? object) (touch ((cdr object))) object)))
```

Figure 5. Traduction de `(M (future N))` et définition de `touch`

présentons dans un cadre parallèle.

Nous avons également montré la différence entre les styles de programmation d'un langage avec constructions transparentes pour le parallélisme et d'un langage avec constructions non transparentes tel que PolyScheme. L'avantage de notre approche réside dans le fait qu'il suffit simplement d'annoter des programmes développés selon une approche fonctionnelle classique.

PolyScheme a été initialement proposé par C. Queinnec. La figure 1 donne une sémantique utilisant la même technique de continuations que celle employée dans PolyScheme. Les figures 3, 4 et 5 y ajoutent des métacontinuations pour assurer la transparence des opérateurs de parallélisme. Dans [18] et [20], le caractère "unfair" de PolyScheme est évoqué lorsque plusieurs résultats sont retournés. Des conditions sur l'application de continuations sont donc ajoutées afin de préserver le nombre de résultats. Cette approche est totalement opposée à la notre où l'on ajoute des contraintes sur les continuations afin de s'assurer un seul résultat, le même que celui retourné par la version séquentielle.

Une autre approche pour assurer la transparence des opérateurs est celle proposée par Katz et Weise dans [10]. Ils décrivent une implémentation basée sur une notion de *légitimité*. Un processus est légitime si le code qu'il exécute est exécuté par une implémentation séquentielle en l'absence de future. Le processus initial est légitime.

Nous avons introduit une notion de continuation d'ordre supérieur pour donner une sémantique à Λ_C. Une présentation complète en est faite dans [16]. Cette notion est également comparée à une notion de légitimité semblable à celle de Katz et Weise dans [17]. Ces deux approches diffèrent au niveau du calcul spéculatif. Suivant l'approche de la légitimité, un maximum de calcul se fait spéculativement et lorsqu'un résultat est retourné, on vérifie qu'il correspond à celui qui serait retourné par un ordre d'évaluation gauche droite. Les métacontinuations sont utilisées au cours de l'application d'une continuation afin de vérifier s'il est légitime de la réaliser; elles permettent de déterminer au moment de l'exécution si un calcul est légitime ou non. La notion de métacontinuation permet de définir les constructions `pcall` et `fork` transparentes. Quant à `future`, elle demande à l'utilisateur de la combiner avec la construction `touch`, ce qui peut se faire simplement comme on l'a vu dans certains exemples.

8. Conclusions

Les exemples donnés à la section 3 montrent qu'il est possible de définir un langage fonctionnel avec des constructions pour le parallélisme ne modifiant en rien la sémantique des programmes même si des continuations sont utilisées; ces constructions peuvent être considérées comme des annotations pour le parallélisme.

Une sémantique pour ce type de langage fut donnée. Elle utilise une notion de continuation d'ordre supérieur qui assure qu'une continuation peut être appliquée afin de respecter la sémantique séquentielle.

Enfin, d'un point de vue pratique, on n'a pas étudié le scheduling qui pause des problèmes dans le cadre du parallélisme spéculatif. Des solutions existent telles que les sponsors proposées dans [4] et [19]; elles pourraient être combinées avec notre méthode.

9. Remerciements

Ce travail fut réalisé lors d'une visite du Laboratory for Foundations of Computer Science, University of Edinburgh. Je tiens à remercier Rod Burstall qui a rendu possible cette visite, ainsi que Daniel Ribbens pour les nombreuses discussions que nous avons eues. Je tiens aussi à remercier David N. Turner et Christian Queinnec pour les lectures qu'ils ont faites d'une version précédente.

Références bibliographiques

[1] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–1160, June 1990.

[2] Robert H. Halstead, Jr. Multilisp : A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[3] Robert H. Halstead, Jr. Parallel symbolic computing. *IEEE Computer*, pages 35–43, August 1986.

[4] Robert H. Halstead, Jr. New ideas in parallel lisp : Language design, implementation. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, pages 2–57. Lecture Notes 441 in Computer Science. Springer-Verlag, 1990.

[5] C. T. Haynes and D. P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, October 1987.

[6] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM conference on LISP and functional programming*, pages 293–298. ACM, 1984.

[7] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines with continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.

[8] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.

[9] Soren Holmstrom. PFL : A functional language for parallel programming and its implementation. Technical Report 7, Chalmers University, 1983.

[10] Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 176–184, June 1990.

[11] James S. Miller. *MultiScheme : A parallel processing system based on MIT Scheme.* PhD thesis, MIT, 1987.

[12] James S. Miller and B. S. Epstein. Garbage collection in MultiScheme. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp : Languages and Systems. US/Japan Workshop on Parallel Lisp. Japan.*, pages 138–160. Lecture Notes 441 in Computer Science. Springer-Verlag, 1990.

[13] R. Milner, D. Berry, and D. Turner. A semantics for ML concurrency primitives. In *Proceedings of the nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1992.

[14] Robin Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall, 1989.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, 1990.

[16] Luc Moreau. An operational semantics for a parallel language with continuations. Technical report, Université de Liège, 1991.

[17] Luc Moreau and Daniel Ribbens. Higher order continuations or legitimacy in a denotational semantics of parallel scheme. Technical report, Université de Liège, 1991.

[18] Christian Queinnec. Polyscheme, a semantics for a concurrent scheme. In *High Performance and Parallel Computing in Lisp Workshop*, Twickenham, England, November 1990. Europal.

[19] Christian Queinnec. CD Lisp. Technical report, Ecole Polytechnique, 1991.

[20] Christian Queinnec. Crystal Scheme. A language for massively parallel machines. In *Symposium on High Peformance Computers, Montpellier*, 1991.

[21] Jonathan Rees and William Clinger. Revised^3 report on the algorithmic language scheme. Technical report, MIT AI Lab and Indiana University Comp. Science, 1986.

[22] J. H. Reppy. First-class synchronous operations in Standard ML. Technical report, Cornell University, Department of Computer Science, 1989.