Langages fonctionnels et parallélisme
Une réalisation pour le système CAML

Francis Dupont

11 Avril 1990

Remerciements

Je remercie Monsieur Patrick Cousot d'avoir accepté de présider ce jury et Messieurs Jean-Pierre Banatre, Christian Queinnec, Gérard Huet, Bernard Lang et Jean-Jacques Lévy d'avoir bien voulu en faire partie.

J'exprime toute ma gratitude à Monsieur Jérôme Chailloux et à l'équipe de Le-Lisp pour m'avoir initier aux joies des parenthèses et envers les membres du projet Formel pour m'avoir fait découvrir au autre facette prometteuse des langages fonctionnels avec le système CAML.

Je souhaite aussi remercier la communauté de chercheurs connue sous le nom de bâtiment 8 où j'ai trouvé un environnement de recherche exceptionnel.

Je profite de l'occasion pour rendre hommage à nos aînés dont les résultats ont été fort injustement oubliés ce qui nous a obligé à les redécouvrir péniblement.

Enfin je témoigne de la reconnaissance à la Direction Générale pour l'Armement qui a fait preuve d'une bienveillance toute paternelle à mon égard.

Introduction

Dans l'évolution de la technologie des ordinateurs vers des machines de plus en plus rapides, une des composantes est le développement des multiprocesseurs. Si ceux-ci existent depuis longtemps, les progrès de la technologie permettent d'envisager des machines peu coûteuses et d'accès facile, disposant d'une dizaine de processeurs réunis par un bus autour d'une mémoire partagée.

Notre principale idée est de s'appuyer sur le point fort des langages fonctionnels comme CAML, qui est leurs sémantiques mathématiques, pour construire une extension parallèle en ajoutant des constructions entièrement intégrables. Une conséquence sera la conservation des propriétés sémantiques pour le sous-ensemble le plus large possible du langage.

Un langage de programmation peut avoir des primitives parallèles explicites comme des sémaphores ou des canaux de communication. Nous avons choisi une autre voie, en mettant l'accent sur l'accroissement des performances au lieu d'augmenter le pouvoir d'expression du langage avec de nouvelles primitives.

Développons maintenant quelques points : la gestion de la mémoire, l'appel spéculatif et les exceptions.

Une nouvelle méthode de GC

Les langages fonctionnels comme CAML nécessitent un système d'allocation dynamique de la mémoire : le "glaneur de cellules" ou GC ou encore ramasseur de miettes. Ce domaine est bien étudié et beaucoup de méthodes ont été définies.

Nous proposons une nouvelle technique qui marie deux des techniques de GC les plus utilisées :

- le marquage/balayage (*Mark & Sweep*) qui est très simple et efficace,

en particulier dans sa variante parallèle, mais ne réalise pas de compactage.

- la recopie (*Stop & Copy*) qui compacte la mémoire, mais gaspille la moitié de l'espace.

Les performances de ces classes d'algorithmes sont très différentes, car si la durée de la recopie ne dépend que de la quantité d'objets effectivement utilisés, celle du balayage dépend aussi de l'espace total.

La méthode que nous proposons ici (conjointement avec Bernard Lang) essaie de rendre compatible les avantages des deux méthodes de recopie et de marquage/balayage. L'espace mémoire est divisée en trois zones : une zone traitée en marquage/balayage et les espaces de départ et d'arrivée du recopie. A chaque GC une fraction de la mémoire est compactée en ne gaspillant qu'une fraction équivalente. L'intérêt de notre méthode est qu'elle est complètement réglable entre les deux extrêmes que sont la recopie ou le marquage/balayage, et par exemple peut assurer les meilleures performances en fonction du taux de remplissage.

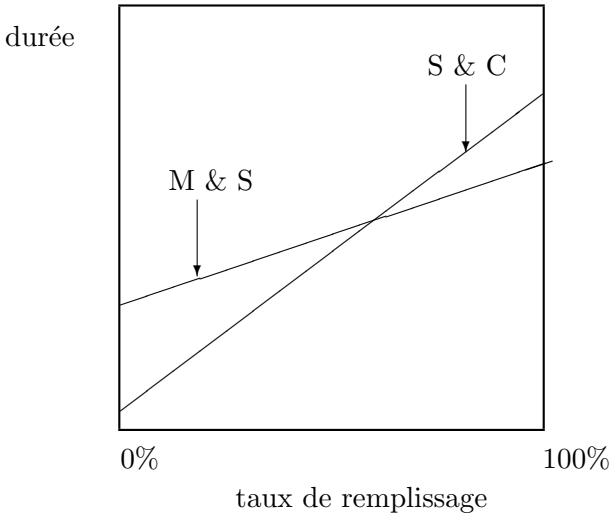


Figure 1 : la durée du GC en fonction du remplissage de la mémoire.

Des raffinements des méthodes de marquage/balayage et de recopie, extensibles immédiatement à notre algorithme mixte, existent pour répondre à des contraintes comme :

- le GC incrémental où l'allocation et la récupération se fait "en même temps" au lieu d'arrêter toutes les activités quand la mémoire est pleine pour procéder à la récupération.
- le GC temps réel qui doit garantir une borne pour le temps d'allocation.
- le GC parallèle où les tâches d'allocation et de récupération s'exécutent simultanément sur des processeurs différents.

ou d'une manière orthogonale :

- le GC conservatif pour un langage utilisant des pointeurs ambigus que le mécanisme de GC ne sait pas distinguer des valeurs, comme dans le cas d'un support d'exécution écrit en C.
- le GC à générations qui distingue les objets selon leur durée de vie. Les systèmes à générations sont très efficaces, et conviennent idéalement à notre nouvelle méthode synthétique car la recopie est appropriée aux jeunes générations alors que les vieilles sont plus stables et doivent plutôt être traitées par marquage/balayage.

Une version à générations de notre méthode a été développée par Damien Doligez pour le futur système d'exécution du langage CAML.

La terminaison de l'appel spéculatif

Notre système propose trois modes d'évaluation : l'appel par valeur, l'appel par nécessité et l'appel spéculatif. Ce dernier produit des suspensions comme l'appel par nécessité, mais au lieu d'effectuer les calculs quand ils sont nécessaires, ils sont exécutés en parallèle.

Dans le Lambda Calcul ou dans le langage CAML, à chaque instant il y a au moins une seule expression nécessaire. Pour garantir la terminaison des programmes et aller directement au résultat, notre système garde la trace du calcul nécessaire courant et lui garantit l'allocation d'un processeur. En revanche des systèmes comme le langage MultiLisp développé par R. Halstead n'assure qu'un ordonnancement équitable : chaque tâche, nécessaire ou non, a l'accès à un processeur au bout d'un temps fini.

Nous considérons l'appel par spéculation comme un dérivé de l'appel par nécessité. Au coût de la gestion du multi-processeur près, le système parallèle est au moins aussi rapide que la version paresseuse (le terme "paresseux" est synonyme d'appel par nécessité). Une autre conséquence est qu'il est

impossible d'exprimer une fonction comme le ou-parallèle (*parallel-OR*) qui n'a pas de sous-expressions nécessaires :

Por	$\perp$	$False$	$True$
$\perp$	$\perp$	$\perp$	$True$
$False$	$\perp$	$False$	$True$
$True$	$True$	$True$	$True$

Figure 2 : la table de vérité du ou-parallèle ($\perp$ est l'indéfini).

Le traitement des exceptions

Le langage CAML dispose d'un système d'exceptions, événements qui provoquent l'abandon de l'exécution de l'expression courante et le lancement d'une routine de traitement. L'utilisation des exceptions s'appuie sur la connaissance de l'ordre d'évaluation.

Prenons par exemple la fonction :

$$f(x) = \texttt{if } x > 0 \texttt{ then } e^{-1/x^2} \texttt{ else } 0$$

Un programme CAML la calculant peut être :

```
type point = {X : num; parallel V : num};;

let f x = {X = x; V = exp(-1/(x*x))};;

let g x =
 let v = f x in
   if v.X > 0 then v.V else 0
;;
```

Un objet de type `point` est un enregistrement à deux champs `X` et `V`. La valeur du champ `V` est calculée en parallèle : l'appel `f(x)` construit une paire avec `x` et une suspension du calcul `exp(-1/(x*x))` qui sera lancé dès qu'un processeur sera disponible. La fonction `g` appelée avec 0 doit renvoyer 0 bien qu'elle transmette 0 comme argument à `f`, et que la division par 0 lève une exception.

Nous avons recherché une solution raisonnable au problème des exceptions dans un système parallèle. D'abord nous avons distingué les exceptions

"ordinaires" des interruptions comme le "*break*" qui sont des événements extérieurs de nature différentes.

Nous reprenons le même comportement que pour l'extension paresseuse de CAML dont le système parallèle est dérivé (même si l'utilisation d'exceptions dans un cadre paresseux n'est pas totalement satisfaisante) : les valeurs sont divisées en valeurs ordinaires et en valeurs exceptionnelles. Les suspensions produisent une valeur d'un des deux genres, les valeurs exceptionnelles ne déclenchent une exception que lorsqu'elles deviennent nécessaires.

Dans l'exemple précédent, l'exception ne s'échappera pas de la suspension et `g(0)` rendra toujours le résultat attendu 0.

La structure du document

Nous parcourons l'éventail des architectures parallèles pour déterminer quelles machines seront les cibles de notre système. Puis nous passons en revue les différents types de langages parallèles existants afin d'établir le genre de constructions que nous proposons. En particulier, nous expliquons pourquoi nous refusons d'introduire le non-déterminisme.

Ensuite nous détaillons les notions de fonction et de liaisons, en décrivant les méthodes d'implantation et en précisant les choix que nous pensons devoir faire. Nous introduisons un nouveau mode d'appel de fonctions, l'appel par spéculation qui est avec l'appel par valeur et l'appel par nécessité l'un des trois modes d'évaluation proposés par notre langage. Nous développons les différentes propriétés de l'évaluation paresseuse et de sa dérivée parallèle.

Après nous présentons la compilation dans la machine CAM du langage, ainsi que les détails de la réalisation du noyau, en particulier pour l'évaluation paresseuse et l'évaluation parallèle. Nous tirons les bénéfices de notre approche en, par exemple, donnant un sens aux situations d'étreinte fatale. Puis nous décrivons les extensions syntaxiques de la version parallèle de CAML, ainsi que celles qui devraient être intégrées dans les versions futures.

Enfin nous exposons les premiers résultats et comparons notre système à MultiLisp et à ses dérivés, en montrant les avantages de notre technique originale soutenue par des considérations sémantiques.

Nous faisons une digression pour traiter le problème délicat du ramasseur de miettes (ou glaneur de cellules) en proposant une nouvelle classe d'algorithmes prometteurs qui concilient les avantages des méthodes classiques de marquage/balayage et de recopie.

Pour finir, nous redémontrons la séquentialité de ML qui est une pro-

priété très importante, surtout pour un système parallèle (même s'il peut sembler paradoxal de présenter un système parallèle muni d'une sémantique séquentielle, c'est-à-dire qui ne permet pas d'exprimer des fonctions comme le ou-parallèle). En seconde annexe, nous détaillons le problème des structures de contrôle manipulant des continuations ou des contextes.

En conclusion, nous essayons de justifier la démarche suivie.

Chapitre 1

Architectures Parallèles

Depuis très longtemps, les architectes d'ordinateurs ont découvert la notion du parallélisme, c'est-à-dire l'idée de faire plusieurs choses en même temps plutôt qu'en séquence. Ils exploitent ce parallélisme à tous les niveaux du matériel et maintenant du logiciel. Ce qui nous intéresse directement est la possibilité d'utiliser simultanément plusieurs unités centrales (en anglais *CPU*[1]).

Il existe ainsi toute une gamme de machines plus ou moins parallèles et plus ou moins faciles à programmer que nous décrirons rapidement. En effet comme nous le verrons dans le prochain chapitre, l'architecture de la machine "cible" conditionne en partie les possibilités parallèles d'un langage.

Les machines SIMD

En suivant un degré de parallélisme croissant, on trouve d'abord les machines mono-processeurs "classiques", puis les SIMD (pour *Simple Instruction Multiple Data*) où tous les processeurs exécutent en même temps le même code, et enfin les MIMD (pour *Multiple Instructions Multiple Data*) où chaque processeur exécute un flot d'instructions indépendant.

Il existe deux grandes catégories de SIMD :

- les *pipelines* : les unités fonctionnelles sont découpées en sous-unités (étages) qui effectuent successivement une partie de l'opération.

[1] Comme l'informatique est une science principalement anglo-saxonne, les termes usuels sont tous anglais et n'ont que parfois un équivalent en français (qui de plus n'est pas toujours très utilisé, ni compris). C'est pourquoi nous donnerons systématiquement pour tout terme technique l'original.

- les machines massivement parallèles, généralement utilisées dans des applications très régulières comme le traitement d'images, où un grand nombre de processeurs très simples forme une grille (d'où le nom d'*array-processors*).

Les machines "pipelines"

Les *pipelines*[2] sont utilisés depuis très longtemps dans les mono-processeurs, et sont généralement invisibles pour le programmeur, sauf dans le cas de certains processeurs RISC (pour *Reduced Instruction Set Computer*). Comme cette technique connaît actuellement un grand succès, nous allons entrer plus dans les détails.

Un exemple typique est le Mips R2000 qui possède un *pipeline* à 5 étages, la traversée de chaque étage prenant un cycle d'horloge :

IF (*Instruction Fetch*) recherche de l'instruction suivante, divisée en un accès au cache de la table de page (TLB) et la lecture de 32 bits dans le cache d'instructions.

RD (*Read*) lecture des opérandes nécessaires depuis les registres (RF = *Register Fetch*).

ALU exécution de l'opération demandée. En parallèle à la fin de l'opération, accès au cache de pagination.

MEM accès à la mémoire (d'abord dans le cache de données).

WB (*Write Back*) écriture des résultats de l'unité arithmétique et logique (ALU) ou de la valeur chargée dans le cache de données dans le registre destination.

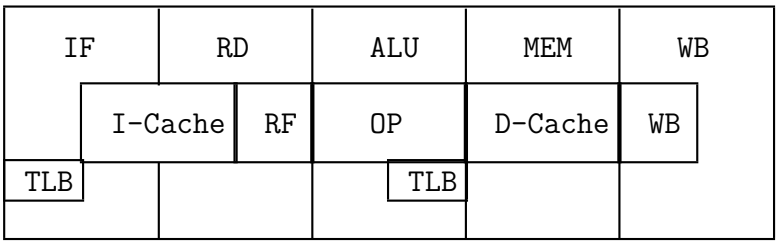


Figure 1 : les 5 étages du pipeline du processeur Mips.

[2]Notre première expérience en informatique a été la conception d'un processeur pipeline spécialisé pour le langage Le-Lisp.

Le flot d'instructions se déroule en effectuant à chaque cycle une partie d'une instruction différente dans chaque étage :

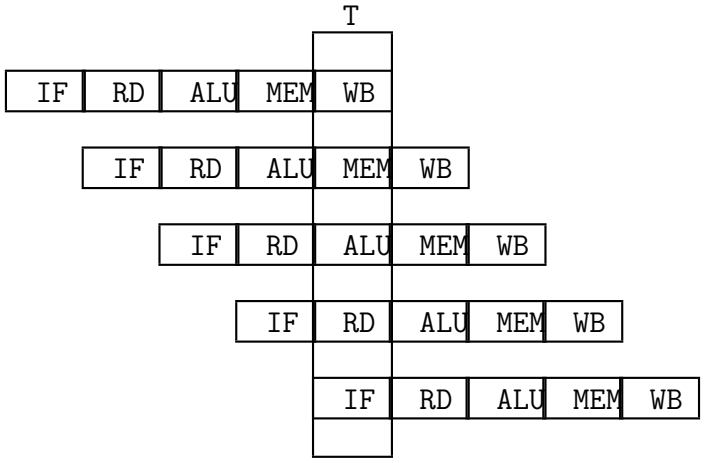


Figure 2 : le déroulement des instructions sur le processeur Mips.

Deux complications apparaissent immédiatement quant à l'utilisation directe d'un *pipeline* :

- les ruptures de séquence forcent le vidage du *pipeline*, car l'instruction à l'adresse suivante est déjà décodée alors qu'elle n'est pas celle qu'il faut exécuter. Souvent un artifice est employé : on exécute quand même les instructions qui suivent les ruptures de séquence (dans le *delay slot*), quitte à insérer des instructions vides (NOP) s'il n'y a rien d'utile à faire.

- l'accès mémoire prend au moins un cycle, et donc une valeur lue de la mémoire vers un registre n'est pas disponible immédiatement. Trois techniques peuvent être utilisées :

  - des cycles d'attente sont systématiquement exécutés.
  - un dispositif matériel (*score-board*) empêche l'utilisation prématurée d'un registre en figeant l'exécution d'une instruction.
  - cette contrainte est renvoyée au programmeur qui doit explicitement rajouter des NOPs ...

Il est souvent plus intéressant qu'il n'y paraît de reporter les contraintes sur le logiciel, car il est généralement possible de réarranger les instructions (*scheduling*) pour utiliser au mieux la machine. Mais le *pipeline* garde deux faiblesses congénitales :

- les accès mémoires qui prennent beaucoup de temps (trivialement le processeur ne peut pas aller plus vite que la mémoire, mais la bande passante entre le ou les processeurs et la ou les mémoires est toujours le facteur qui limite les performances dans les super-ordinateurs). Les caches permettent une amélioration, mais pour qu'ils soient efficaces il faut que les accès présentent une bonne localité.
- les ruptures de séquence, surtout conditionnelles, limitent le nombre d'étages du *pipeline*.

Les processeurs vectoriels

Il est possible quand l'opération à effectuer elle-même prend beaucoup de temps (comme c'est le cas pour les calculs sur les nombres flottants), de diviser une unité arithmétique (au lieu du processeur entier) en plusieurs sous-unités.

A chaque cycle, chaque sous-unité traite un jeu d'opérandes qu'elle passe ensuite à l'étage suivant. La traversée complète, et donc l'opération, prend le même temps, mais comme à chaque cycle il est possible de commencer une nouvelle opération, chaque opération semble ne prendre que le temps de traversée d'un étage.

Cette technique s'applique très bien aux vecteurs (d'où le nom de processeurs vectoriels). Voici par exemple une addition de deux vecteurs sur un pipeline à 5 étages aux instants T et $T+1$:

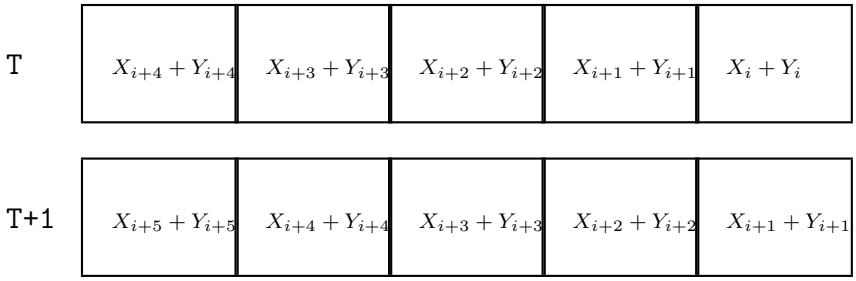


Figure 3 : l'addition de deux vecteurs dans un pipeline à 5 étages.

Les processeurs massivement parallèles

Dans le cas d'une machine massivement parallèle, les données sont découpées suivant le même schéma que l'interconnexion des processeurs. Ces machines sont donc souvent assez spécialisées, mais en contre partie peuvent présenter des performances inégalées pour certains problèmes (ainsi l'ENIAC a été pendant très longtemps la machine la plus rapide pour certains algorithmes, meilleure que les membres de la première génération de superordinateurs comme le Cray 1).

$X_1 + Y_1$	$X_2 + Y_1$	$X_3 + Y_1$	$X_4 + Y_1$	$X_5 + Y_1$
$X_1 + Y_2$	$X_2 + Y_2$	$X_3 + Y_2$	$X_4 + Y_2$	$X_5 + Y_2$
$X_1 + Y_3$	$X_2 + Y_3$	$X_3 + Y_3$	$X_4 + Y_3$	$X_5 + Y_3$
$X_1 + Y_4$	$X_2 + Y_4$	$X_3 + Y_4$	$X_4 + Y_4$	$X_5 + Y_4$

Figure 4 : l'addition de deux matrices avec un tableau de processeurs.

La contrainte d'exécution d'un seul flot d'instructions sur tous les processeurs ou éléments de processeurs est très restrictive. Les architectes ont cherché à augmenter les possibilités des SIMD (sans tomber dans une complexité d'un autre ordre de grandeur) par différentes méthodes :

- avec beaucoup de sous-unités fonctionnelles contrôlées grâce à des jeux d'instructions adaptés[3]. C'est une forme de machines à flots de données au niveau de l'unité centrale.
- en rendant l'exécution des instructions dépendante de conditions locales à chaque processeur, et en compliquant énormément le réseau d'interconnexion qui devient alors un élément très important dans la programmation de ces machines. L'exemple le plus marquant de ces ordinateurs "neuronaux" est la Connection Machine.

[3] les VLIW (pour *Very Large Instruction Word*)

Les machines MIMD

Les architectures MIMD permettent un usage plus général, car ces ordinateurs peuvent effectuer des tâches plus ou moins liées, du programme réparti sur tous les processeurs au service en temps partagé pour un grand nombre d'utilisateurs. C'est intéressant à la fois pour les utilisateurs, qui peuvent acquérir une machine pour un projet précis et à la fin de celui-ci la réutiliser dans de bonnes conditions, et pour les constructeurs qui peuvent ainsi viser plusieurs marchés différents avec la même gamme[4]. Cette souplesse d'utilisation est sûrement la raison du succès grandissant de cette architecture.

La principale caractéristique d'un MIMD est l'interconnexion des processeurs et de la ou des mémoires, plutôt que le nombre ou la puissance des unités centrales. Elle influe souvent directement sur la programmation, malheureusement en rendant généralement certaines optimisations nécessaires pour une exploitation rentable du parallélisme.

Les machines à mémoire partagée

Les machines les plus simples n'ont que quelques processeurs sur un bus ordinaire, et utilisent un système d'exploitation asymétrique, c'est-à-dire un système où l'un des processeurs (le maître) est privilégié, par exemple il est le seul à faire des entrées/sorties. Ce genre d'architecture est simple et ne nécessite pas de logiciels spécifiques : c'est devenu la norme pour les ordinateurs à usage général de bonne puissance, car elle permet d'augmenter à bas coût la puissance par simple ajout de nouveaux processeurs.

Mais dernièrement des architectures plus ambitieuses sont apparues avec un nombre plus important (couramment 8 à 30) d'unités centrales connectées à un bus à très grand débit. Comme le principal facteur limitant les performances sur ces machines est la contention sur le bus, des systèmes de caches mémoires sophistiqués ont été développés pour la réduire en diminuant le nombre de requêtes sur le bus. Mais ces caches doivent assurer la cohérence des données (il faut que la modification d'une valeur soit répercutée instantanément dans tous les caches où cette valeur est présente), ce qui les rend compliqués et coûteux pour l'instant.

[4]Les considérations de nature commerciale sont importantes à nos yeux, car pour pouvoir acheter une machine il faut une certaine offre. De plus le succès commercial est toujours un signe de maturité pour une technologie.

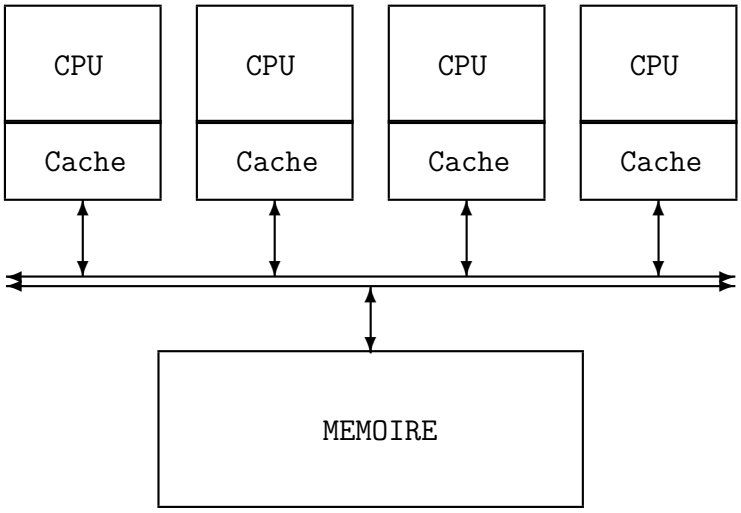


Figure 5 : structure d'un multi-processeur à bus global.

Les systèmes d'exploitation de ces MIMD sont symétriques, c'est-à-dire que tous les processeurs ont le même statut et peuvent exécuter le code de n'importe quelle partie du système. Il n'y a donc pas de maître qui puisse être un goulot d'étranglement pour les esclaves, en revanche le logiciel système est plus difficile à écrire, surtout quand la base est très orientée mono-processeur comme Unix. Mais les performances sont meilleures que celles d'un système asymétrique lorsque le nombre d'unités centrales augmente, surtout en ce qui concerne les entrées/sorties ou le service des interruptions[5].

Le logiciel de base supporte les applications parallèles avec des compilateurs adaptés, des bibliothèques de routines de synchronisation et de partage de la mémoire, et parfois un système de mise au point de programmes parallèles, outil indispensable s'il en est !

Ces ordinateurs se montrent extrêmement utiles pour les recherches touchant le parallélisme, et maintenant rencontrent des succès dans des domaines comme les bases de données ou les télécommunications pour lesquelles leur rapport performances/prix est inégalé.

Pour éviter les limitations d'un bus central, certains architectes l'ont remplacé par un réseau d'interconnexion qui n'a pas de problème de contention,

[5]Un bon exemple de ce genre d'ordinateurs est le Sequent Balance sur lequel nous avons développé la version parallèle de CAML. L'architecture de cette machine est décrite dans [9, Beck].

mais dont la traversée est plus longue (de l'ordre du logarithme du nombres d'unités connectées). Ces machines sont rarement commercialisées et ne peuvent pas raisonnablement être exploitées aujourd'hui comme des ordinateurs d'usage général, en particulier à cause de l'énorme écart de performances entre un logiciel classique séquentiel et un logiciel parallèle. C'est peut-être pourquoi elles sont souvent utilisées comme coprocesseur sans tous les outils traditionnels nécessaires pour un développement agréable d'applications.

Les machines distribuées

Enfin la dernière catégorie de MIMD est celle des machines distribuées, sans mémoire partagée entre tous les processeurs.

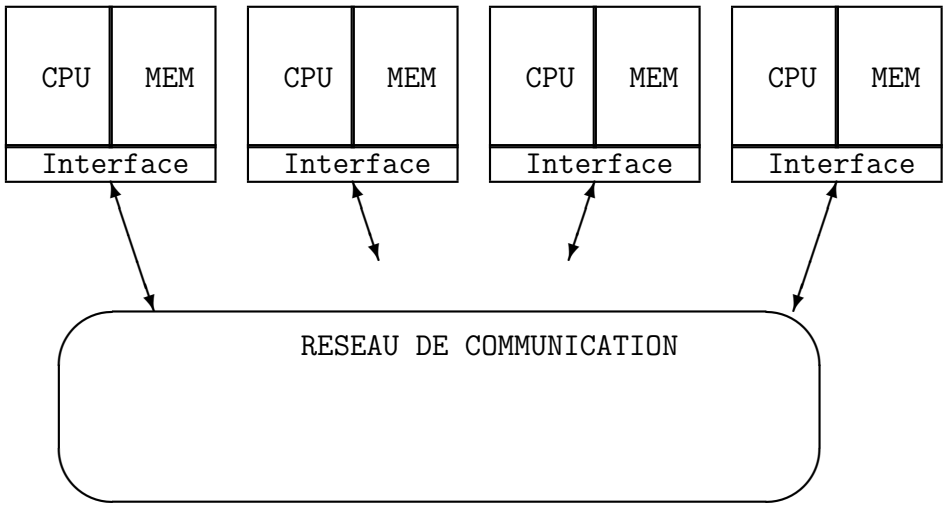


Figure 6 : structure d'une machine distribuée.

Le réseau d'interconnexion peut avoir de nombreuses topologies différentes, la plus en vogue étant l'hypercube (dans un N hypercube, il y a 2^N processeurs, chacun connecté aux N processeurs dont le numéro binaire n'a qu'un bit de différence. Il y a donc une quantité en C_N^x de mémoire à "distance" x dans un hypercube d'ordre N).

Voici par exemple un 4-hypercube :

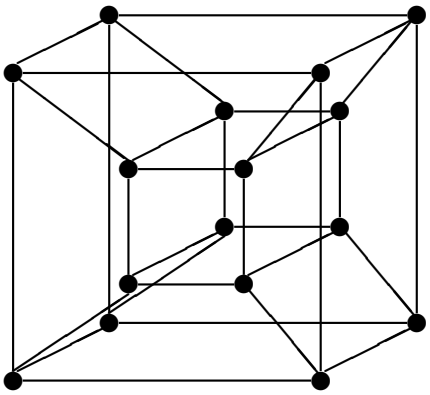


Figure 7 : le 4-hypercube.

Il y a très peu de systèmes d'exploitation distribués commerciaux, car la plupart des systèmes comme Unix nécessitent une complète réécriture pour être adaptés à une architecture distribuée (il faut remplacer par exemple les appels de procédures par des envois de messages).

Mais la possibilité d'obtenir une puissance quasi-illimitée avec certaines architectures distribuées en ajoutant simplement des machines dans un réseau reste très attractive. Des processeurs spécialisés comme le fameux "Transputer" d'INMOS, où l'aspect communication devient dominant ont même été développés.

De manière plus classique, certains ont remarqué que les réseaux locaux de stations de travail recelaient énormément de puissance de calcul inutilisée, et veulent récupérer les unités centrales non occupées pour exécuter en arrière plan des applications distribuées (souvent gigantesques, car à la mesure des réseaux des campus d'universités américaines).

Ainsi sont apparus des logiciels d'envoi de messages ou de RPC (pour *Remote Procedure Call* : appel de procédures distantes). Ces techniques sont typiques des architectures faiblement couplées et peuvent être facilement ajoutées à un langage existant sous forme d'une bibliothèque, ce qui ne permet pas des utilisations parallèles très générales (d'après l'expérience que nous avons d'un système de ce genre pour le langage Le-Lisp). En revanche les mécanismes de RPC permettent de construire une interface sure avec d'autres langages car les espaces d'adresses sont disjoints et aucune interférence (catastrophique avec un glaneur de cellules) n'est à craindre.

Mais certains problèmes inhérents aux aspects distribués subsistent :

- le temps d'accès d'un processeur à une ressource dépend des positions respectives dans le réseau d'interconnexion, ce qui oblige le programmeur à prendre en compte la notion de localité.
- l'allocation d'une tâche à un processeur et son éventuelle migration est à ce jour pratiquement impossible à automatiser dans des cas non-spécifiques.

En conclusion les machines fortement couplées avec bus sont peut-être limitées, mais elles restent une bonne base pour développer des langages parallèles. Tous les outils indispensables sont disponibles et au besoin une architecture plus élaborée peut être très facilement simulée.

Chapitre 2

Langages Parallèles

Certains langages de programmation proposent des constructions de type parallèle pour exprimer des activités concurrentes ou bien pour exploiter un matériel potentiellement parallèle, même s'ils ne sont pas toujours destinés à l'utilisation d'une architecture parallèle. En effet il peut être indispensable de répondre à des contraintes de pseudo-parallélisme.

Le cas le plus caractéristique des applications nécessitant de telles possibilités est le système d'exploitation où se rencontrent à la fois toutes les contraintes provenant des différents niveaux de parallélisme et toutes les ressources matérielles pouvant aider à les résoudre.

Les techniques de base de la programmation parallèle ont donc été développées dans le cadre des systèmes d'exploitation, toutefois sans un succès total. Par exemple nous ne savons toujours pas aujourd'hui démontrer la correction de parties significatives d'un tel logiciel, ce qui peut inciter à restreindre la puissance des langages de programmation (et malheureusement leurs domaines d'emploi) pour les rendre plus "maniables".

Chaque dispositif matériel étant souvent associé à une construction syntaxique, les différents types de machines parallèles ont leurs pendants dans les langages. Avant de présenter CAML, il est bon de décrire brièvement l'éventail des langages parallèles.

Les langages destinés aux machines SIMD

Pour exploiter les processeurs vectoriels des instructions spécifiques sont utilisées. Par exemple en Fortran 8x la somme de deux vecteurs s'écrit :

```
      C(1:N) = A(1:N) + B(1:N)
```

au lieu de la boucle classique :

```
      DO 10 I = 1, N
          C(I) = A(I) + B(I)
10    CONTINUE
```

La boucle est en quelque sorte "passée dans l'indice", ce qui est l'expression naturelle d'une opération vectorielle. De plus, ce genre de construction convient aussi bien à un processeur où les opérations élémentaires sont chaînées (de type *pipeline*) qu'à un tableau de processeurs. Car pour ces derniers le découpage des vecteurs et tableaux est libre, mais doit être très bien adapté à la topologie et au nombre de processeurs de la machine pour obtenir une bonne efficacité.

Il est possible de transformer automatiquement la boucle en une construction vectorielle : c'est la tâche des "vectoriseurs" qui analysent les dépendances dans les boucles en faisant des calculs formels sur les indices.

Utilisée dans les compilateurs Fortran pour super-calculateurs, cette technique a connu récemment un énorme développement qui peut donner à penser que la vectorisation/parallélisation de "boucles DO" fait maintenant partie du domaine industriel.

En revanche à notre connaissance, il y a peu de supports logiciels pour les architectures systoliques qui sont souvent très spécialisées. Il est en effet inutile de concevoir un langage de programmation complet pour une machine n'exécutant qu'un unique algorithme. Il existe pourtant au moins une exception notable : la "*Connection Machine*" qui supporte plusieurs langages dont l'un, CM Lisp ([37, Hillis]), possède des constructions d'application et de réduction de structures de données potentiellement infinies qui sont très intéressantes du point de vue conceptuel, même si elles correspondent à une idéalisation de l'architecture qui suppose un nombre infini de processeurs. Nous reviendrons sur les structures de données infinies.

Un exemple de langage faussement parallèle

Avant d'aborder les langages de programmation associés aux MIMD, on peut remarquer qu'il y a deux buts à la programmation parallèle :

- la puissance d'expression, certains algorithmes s'écrivant plus simplement de façon parallèle.

- la vitesse d'exécution, car on peut espérer raisonnablement que N processeurs effectuent un travail donné en moins de temps qu'un processeur de même puissance, avec un rapport des temps d'exécution (en anglais on parle de *speedup*) de l'ordre de N pour les cas favorables.

Mais si la vitesse d'exécution nous semble plus intéressante, il ne faut pas négliger la puissance d'expression qui concerne un bon nombre d'applications importantes. Par exemple les systèmes réactifs qui décrivent une "boîte noire" soumise à des commandes extérieures (l'exemple canonique du langage Esterel[1] est une montre Seiko) nécessitent des programmes parallèles qu'il est possible de compiler en des automates d'état fini, ce qui est très efficace et pas du tout parallèle ! Or des systèmes réactifs se présentent dans des domaines aussi divers que la gestion de processus industriels ou les interfaces homme-machine à base de systèmes de fenêtres.

Ce n'est pas un exemple isolé car il est fréquent de rencontrer des algorithmes qui s'écrivent beaucoup plus facilement de façon parallèle tout en s'exécutant généralement de manière séquentielle.

Ce mélange a priori curieux de constructions parallèles et d'implantations séquentielles nous a conduit à l'idée d'étudier une option symétrique : l'implantation parallèle d'un langage qui possède une sémantique séquentielle ...

les systèmes de programmation multi-tâche

L'idée d'effectuer concurremment plusieurs actions différentes est assez ancienne. Le dispositif logiciel le plus simple est la co-routine : plusieurs sous-programmes s'exécutent alternativement sur le même processeur.

[1] Esterel [10, Berry, Cosserat] est un langage possédant des constructions parallèles destinées à la programmation de systèmes réactifs.

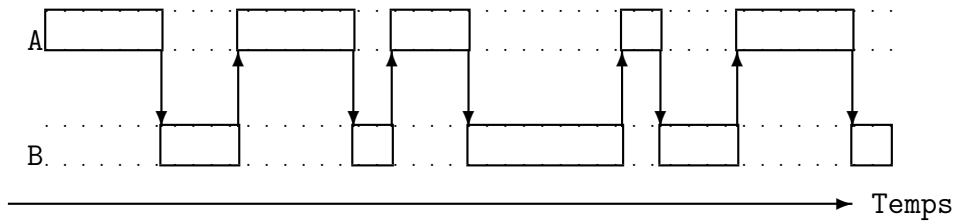


Figure 1 : un exemple de co-routines.

Il s'étend immédiatement aux multi-processeurs :

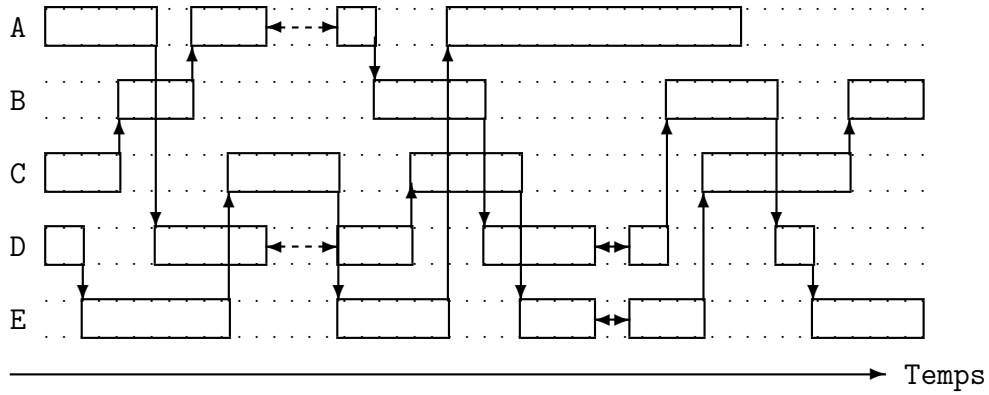


Figure 2 : un système multi-tâche avec des sections critiques.

Certaines parties d'un programme ne doivent être exécutées que par un seul processeur à la fois : les "sections critiques". Dans des conditions semblables un système d'exploitation pour mono-processeurs masque les interruptions (qui constituent la seule manière d'entrelacer arbitrairement l'exécution de sous-programmes indépendants).

Les multi-processeurs disposent d'instructions spéciales qui permettent de construire des verrous qu'un processeur doit poser avant d'entrer dans une section critique et lever en sortant. L'instruction spéciale la plus courante est le *test-and-set* : dans un cycle indivisible, un bit en mémoire est lu, comparé avec zéro et positionné à un. Si l'ancienne valeur est un, alors le verrou est déjà pris et la routine doit refaire des tentatives jusqu'au succès (c'est l'attente active, en anglais ce genre de verrous est appelé *spin-lock*) ou bien abandonner la tâche courante.

Les verrous et les déclarations de variables partagées ou privées forment le niveau le plus bas du support de la programmation parallèle que proposent les multi-processeurs fortement couplés. Ils sont rarement exploitables directement, et servent plutôt à construire des systèmes de plus haut niveau.

Les co-routines peuvent se généraliser en des systèmes de tâches ou de processus "légers" (par opposition aux processus du système d'exploitation qui sont plus coûteux à manipuler) que ce soit pour un multi-processeur ou un mono-processeur. Quand l'accent est mis sur l'existence de plusieurs flots de contrôle à l'intérieur d'une même application, on utilise le terme anglo-saxon de *multi-thread*.

Ce genre de systèmes connaît actuellement un grand succès car il permet par exemple une programmation aisée de systèmes interactifs de fenêtrage (très important dans les interfaces homme-machine modernes), et commence à être proposé en standard dans les dernières versions d'Unix. La réalisation en est assez simple car le noyau d'un *multi-thread* consiste en une manipulation des piles d'exécution qui contiennent l'essentiel de l'état d'un *thread*.

Le noyau d'un système multi-tâche

Les opérations importantes d'un tel système de tâches sont :

- la création
- la terminaison
- la commutation
- la synchronisation
- la communication.

La création peut être déterminée à la compilation (statiquement) ou à l'exécution (dynamiquement) ce qui est plus général, mais ne permet pas de déterminer à l'avance la répartition des tâches sur les processeurs.

La terminaison peut être considérée comme un cas particulier inévitable de synchronisation. Une des structures les plus simples de ce genre de programmation parallèle est le *fork/join* où un flot de contrôle se divise en N flots indépendants (le *fork*) qui s'unissent après un certain temps pour redonner un seul flot (le *join*). Le dispositif de synchronisation associé est la barrière : les tâches s'arrêtent en atteignant cette barrière qui ne s'ouvre que lorsque les N tâches l'ont atteinte. Cette barrière est essentiellement un compteur protégé par un verrou.

La commutation des processus peut se faire selon deux modes :

préemptive : une tâche peut perdre le CPU à n'importe quel moment. Généralement un ordonnanceur (universellement connu sous le terme imprononçable de *scheduler*) alloue à chaque tâche active un quantum de temps, et utilise une interruption d'horloge pour reprendre le contrôle à intervalles réguliers et donner la main à une autre tâche. C'est la copie du comportement d'un système d'exploitation classique.

non-préemptive : une tâche garde le CPU tant qu'elle ne le libère pas explicitement (généralement à l'occasion d'une synchronisation). Ce mode est plus simple, car une tâche ne peut être suspendue qu'à des endroits bien particuliers, ce qui rend le système de multi-programmation bien plus facile à développer. Bien sûr, il faut s'assurer que toutes les tâches "rendent la main" de temps en temps.

Les langages à la CSP

Les communications sont souvent couplées avec les synchronisations (ce qui donne des communications synchrones plus faciles à gérer que les asynchrones) pour former le système de canaux de la programmation par envoi de messages[2]. Ainsi un processus peut lire et écrire dans un ou des canaux (et dans le cas synchrone le lecteur est bloqué tant que le canal est vide, et réciproquement l'écrivain doit attendre qu'un lecteur se manifeste pour repartir).

Pour multiplexer les différents canaux Dijkstra [23, Dijkstra] a inventé les commandes gardées. En voici un cas simple, pour ne pas dire simpliste, en CSP ([39, Hoare]), le langage et modèle théorique ancêtre du système OCCAM pour Transputer, un des langages parallèles avec canaux et commandes gardées les plus typiques :

```
( x=true -> P | x=false -> Q)
```

L'idée est d'offrir un choix entre différentes commandes préfixées par une garde : une condition booléenne ou une action sur un canal ou, dans certains langages, d'autres conditions comme des délais, etc ... Le processeur "choisit" une garde possible et exécute la branche associée.

[2]Ces messages ne sont pas les messages des langages orientés objets quoiqu'il y ait des analogies.

En ADA (le langage de haut niveau promu par le *Department Of Defense*, très bon exemple d'un langage à usage général moderne) la construction de commandes gardées est le `select` dont voici un exemple typique (décrivant la boucle interne de gestion d'un tampon circulaire de taille bornée) :

```
select
  when COUNT < POOL_SIZE =>
    accept WRITE(C: in CHARACTER) do
      POOL(IN_INDEX) := C;
    end;
    IN_INDEX  := IN_INDEX mod POOL_SIZE + 1;
    COUNT     := COUNT + 1;
or when COUNT > 0 =>
    accept READ(C: out CHARACTER) do
      C := POOL(OUT_INDEX);
    end;
    OUT_INDEX  := OUT_INDEX mod POOL_SIZE + 1;
    COUNT      := COUNT - 1;
or
   terminate;
end select;
```

Le non-déterminisme

Malheureusement ce type de construction a un effet immédiat : comme le flot de contrôle peut "choisir" entre plusieurs possibilités, il n'est pas garanti que le choix sera toujours le même. Ce genre de construction est "non-déterministe" (comme ce terme peut avoir plusieurs significations, nous ne l'emploierons que dans ce sens).

Voici un exemple en OCCAM qui met en valeur cette propriété :

```
PAR
  a ! 0
  b ! 1
  ALT
    a ? x
      c ! x
    b ? x
      c ! x
```

L'indentation en OCCAM définit la structure de blocs, donc le bloc commençant par le mot clé `PAR` comprend trois tâches exécutées en parallèle :

- le premier envoie la valeur 0 sur le canal `a`
- le second la valeur 1 sur le canal `b`
- le troisième (l'alternative) attend une valeur sur l'un des canaux `a` et `b` et la recopie sur le canal `c`.

En conclusion le canal `c` reçoit la valeur 0 ou 1. La sémantique d'OCCAM ne permet pas de déterminer laquelle des deux valeurs possibles sera choisie (en fait, cela dépend de la réalisation ou même de conditions extérieures).

Le non-déterminisme a une conséquence extrêmement fâcheuse : à chaque choix l'exécution peut prendre plusieurs chemins différents. Si un programme passe N fois dans l'exemple précédent, il y a 2^N résultats possibles.

Pour un système de preuves de programmes, cette explosion exponentielle est catastrophique. Pour la mise au point, c'est encore pire car il est possible de passer un très grand nombre de fois un jeu de test sans explorer toutes les possibilités, ce qui rend en pratique l'utilisation des langages non-déterministes très problématique pour les applications où les erreurs peuvent avoir des conséquences graves.

Le non-déterminisme n'est pas une propriété intrinsèque du parallélisme, or ses conséquences sur la programmation sont dramatiques. Comme il est possible d'écrire des programmes parallèles déterministes, à notre avis, le non-déterminisme doit être absolument évité. Cette idée n'est pas nouvelle[3], mais elle mérite sûrement un renouveau d'attention.

Nous avons participé au développement du multi-tâche proposé en standard par le système Le-Lisp et cette expérience nous a décidé à explorer une autre voie pour intégrer le parallélisme dans CAML.

Les autres langages parallèles

Il existe quelques classes de langages parallèles assez différents car non-impératifs.

[3] voir le très vieil article [41, Kahn, MacQueen]

Les systèmes d'acteurs

L'envoi de messages employé ici est assez proche de celui des langages "orientés objets". Il est donc naturel que les deux constructions fusionnent en une seule. Les langages résultants sont dits d'"acteurs" d'après la terminologie des premiers membres de cette catégorie.

Certaines réalisations de ces langages semblent très atteintes par le problème de la granularité : la taille des acteurs doit être assez petite pour que leur nombre ne soit inférieur à celui des processeurs, et suffisamment gros pour que le coût de gestion des acteurs ne devienne pas prohibitif.

Ce problème de dimensionnement des tâches est commun à tous les langages parallèles et n'est résolu de façon complètement satisfaisante par aucun. Les langages d'acteurs présentent une des plus mauvaises interactions entre le problème de dimensionnement des processus et la commodité de programmation qui nécessite la liberté la plus complète dans la taille des acteurs.

MultiLisp

Une autre forme de parallélisme a été proposée pour des dialectes de Lisp : la construction "*future*".

Quand une expression `(future <e>)` est évaluée, un objet que nous noterons `f` est rendu en résultat et en même temps une tâche est créée pour calculer la valeur du corps `<e>`. Si une tâche essaye d'utiliser la valeur de `<e>` avant la fin de son calcul, elle est bloquée. Quand le calcul de `<e>` est terminé, l'objet `f` est mis à jour et les tâches en attente du résultat relancée.

Nous reviendrons sur cette idée avec une approche assez différente de celle utilisée dans la présentation de MultiLisp [33, Halstead] car nous nous appuierons sur les différents modes d'évaluation, en particulier nous insisterons sur les relations entre la construction "*future*" et l'appel par nécessité.

Les systèmes à réduction

La réduction de chaînes ou de graphes est une autre technique "exotique" qui se prête bien au parallélisme.

Le programme est représenté comme un terme à réduire sous forme d'une chaîne ou plus souvent d'un graphe. Cette structure est parcourue par un ou plusieurs processeurs qui recherchent et effectuent les réductions possibles, soit en copiant des portions, soit en effectuant des substitutions physiques.

Ce genre de technique convient bien aux langages fonctionnels dans un

cadre d'architectures distribuées où le graphe est réparti dans les mémoires locales des processeurs. La communication se fait par la structure de données de façon plutôt implicite, ce qui donne le plus souvent au programmeur une vision du parallélisme assez éloignée des réalités de l'implantation.

La programmation logique

Un langage non conventionnel, Prolog[4], présente énormément de parallélisme intrinsèque :

et : les différents membres de la conjonction d'une clause sont résolus en parallèle. Cette méthode est simple et conservative.

ou : toutes les clauses pouvant s'appliquer sont traitées simultanément. Cette méthode ne conserve pas l'ordre des résultats. Plus grave encore, elle change les propriétés de terminaison !

La programmation logique est donc un terrain favorable pour les machines MIMD ce qui explique l'apparition de stations de travail spécialisées multi-processeurs comme celles développées dans le cadre du projet japonais de cinquième génération. Mais ce paradigme est très éloigné des habitudes de la quasi totalité des programmeurs, et il est sûrement illusoire de vouloir imposer un langage aussi différent dans lequel la plupart des algorithmes ne peuvent pas s'exprimer naturellement.

Le cliché : "comment faire une inversion de matrice en Prolog ?" est excessif, mais contient un fond de vérité. Un langage de programmation doit rester utilisable, et il ne faut pas sacrifier des constructions un peu discutables sur l'autel de la pureté théorique.

Conclusion

Nous avons rapidement fait le tour du paysage des langages parallèles. En général, la construction parallèle proposée est un multi-tâche si la machine cible est un mono-processeur ou un système avec canaux de communication et rendez-vous avec un multi-processeur. Nous avons déjà exploré ces possibilités pour des extensions parallèles du langage Le-Lisp.

Pour CAML, une voie plus originale s'appuyant sur les fondements des langages fonctionnels nous semble plus prometteuse. Notre but n'est pas

[4] [32, Gregory] présente des extensions parallèles de Prolog

d'ajouter des constructions pour exprimer de nouveaux programmes, mais en contraire d'accélérer l'exécution de programmes déjà existants au moindre coût.

Chapitre 3

Langages Fonctionnels

Une classe de langages de programmation est particulièrement digne d'intérêt : les langages fonctionnels.

Dans un langage "classique" les différentes opérations à effectuer sont décrites par des commandes (d'où le qualificatif d'impératif) du genre :

```
a := b + 1;
c := 3 * a;
```

Quand une même séquence est dupliquée à plusieurs endroits du programme, elle peut être remplacée par un appel de procédure ou de fonction (le terme fonction désigne d'ordinaire une procédure qui retourne une valeur). Ces fonctions correspondent à la notion intentionnelle (au sens de règle de calcul qui produit un résultat à partir d'un argument) de fonction dans les mathématiques, et ne sont donc pas qu'un artifice permettant un partage du code. La notion de fonction est la base des langages de programmation fonctionnels.

Le Lambda Calcul

Les fonctions supportent deux types d'opérations :

abstraction – la création de l'objet fonctionnel avec ses paramètres et son corps.

application – l'appel de fonctions. Les paramètres formels sont substitués par les arguments dans le corps qui est ensuite exécuté.

Il existe un modèle théorique basé sur la notion de fonction : le Lambda Calcul[1]. Toutes les fonctions du Lambda Calcul ont un et un seul argument et s'écrivent : (λ <*paramètre*> . <*corps*>) L'application se note : (<*fonction*> <*argument*>) Par exemple la fonction qui à x associe $x+1$ se note ($\lambda x.x+1$) et son application à 2, $((\lambda x.x+1)2)$.

Les termes du Lambda Calcul peuvent être décrits par le type[2] concret récursif CAML :

```
type term =
     Var of string
   | App of term * term
   | Abs of string * term
;;
```

Ce type concret possède donc 3 constructeurs de base :

Var – les variables, le nom étant représenté par une chaîne de caractères

App – l'application de deux termes

Abs – l'abstraction par rapport à une variable.

Une variable ne figurant pas dans une abstraction englobante est libre, sinon elle est dite liée. Les ensembles FV et BV des variables libres et liées du terme M sont définissables par récurrence de la façon suivante :

$$\begin{array}{ll} FV(v) = \{v\}; & BV(v) = \emptyset \\ FV(MN) = FV(M) \cup FV(N); & BV(MN) = BV(M) \cup BV(N) \\ FV(\lambda v.M) = FV(M) - \{v\}; & BV(\lambda v.M) = BV(M) \cup \{v\} \end{array}$$

Le nom des variables n'a aucune importance, et on peut changer le nom d'une variable liée par un autre dans toutes les occurrences qui correspondent à ce nom :

$$(\lambda x.M) \Rightarrow \lambda y.M[x \setminus y] \;\; \textit{where } y \notin FV(M)$$

ce qui s'exprime par la fonction CAML :

[1] l'ouvrage de référence sur le Lambda Calcul est [8, Barendregt]. Une introduction pour non-spécialiste est [38, Hindley, Seldin].

[2] Le langage CAML possède un syntaxe très naturelle. Nous supposons donc que les exemples en CAML sont compréhensibles, dans le cas contraire il faut se reporter à la documentation de CAML [58, Projet Formel].

```
let subst new_name (Abs(old_name,body)) =
  Abs(new_name,subst_aux body)
    where rec subst_aux =  function
        Var(v) -> if v=old_name then Var(new_name) else Var(v)
      | App(t1,t2) -> App(subst_aux t1, subst_aux t2)
      | Abs(p,b) -> if p=old_name
                       then Abs(p,b)
                       else Abs(p,subst_aux b)
;;
```

En effet il faut prendre des précautions pour éviter les "captures" de variables, ce qui incite certaines présentations du Lambda Calcul à utiliser des systèmes comme les indices de De Bruijn, qui remplacent les noms de variables par une numérotation abstraite. Classiquement le renommage est appelé α-conversion dans le Lambda Calcul.

La règle de calcul (nommée β-conversion) précise la façon d'effectuer l'application $((\lambda x.M)N)$ qui se réduit en M où x est substitué par N. Là aussi, les difficultés techniques sont évitées en supposant les identificateurs tous différents.

Ce modèle a été abondamment étudié et les langages fonctionnels en reflètent les propriétés théoriques (terminaison, canonisation, confluence, etc). Nous ferons donc souvent allusion à des théorèmes du Lambda Calcul qui se généralisent aux langages fonctionnels [11, Berry, Lévy] , ou à un sous-ensemble "pur" d'un de ces langages.

La réduction de graphe

Deux méthodes d'exécution sont utilisées dans les langages fonctionnels :

1. la réduction de graphe
2. les machines à environnement

Dans les systèmes à réduction de graphe[3] (comme le langage LML [6, Augustsson]), le terme est d'abord transformé en fonctions sans variables libres appelées combinateurs. En effet normalement il faut recopier le corps

[3]La réduction de graphes est une technique passionnante. Avec Luc Maranget nous avons programmé (en CAML bien sûr !) les différentes réalisations de la réduction de graphes décrites dans [51, Peyton Jones], ce qui a été le point de départ du système GAML [44, Maranget], version à réduction de graphes de CAML.

des abstractions à chaque application car il ne faut pas qu'une modification physique, par exemple la substitution du paramètre d'appel, puisse changer un terme qui n'a pas de rapport. En fait, seuls les sous-termes où le paramètre apparaît libre doivent être recopiés, d'où l'idée de les éliminer et donc de transformer l'expression en combinateurs auxquels seront associés les codes construisant les graphes correspondants.

L'ensemble des combinateurs utilisés peut être fixe, comme dans les systèmes SKI ([56, Turner], d'après les noms des combinateurs les plus simples) ou être dérivé du programme à compiler (on parle alors de "super-combinateurs" d'après [40, Hughes]). Les variables libres des fonctions sont éliminées à l'aide d'un procédé d'extraction des lambdas (nommé en anglais *lambda lifting*). Par exemple $(\lambda y.+y\,x)$ est transformé en $(\lambda z.\lambda y.+y\,z)x$. Malheureusement le *lambda lifting* semble ne pas permettre aisément la construction d'un système interactif[4], car il n'existe pas à notre connaissance un langage interactif l'utilisant.

L'arête gauche (*spine*) du terme (considéré comme un graphe) est explorée afin d'y trouver des radicaux (termes réductibles) qui sont exécutés. Cette méthode est assez simple, mais nécessite beaucoup d'allocations mémoire pour la construction du graphe et impose un ordre d'évaluation. En revanche elle présente d'excellentes occasions pour un traitement parallèle sur des architectures distribuées car la taille des processus est assez petite et les communications limitées.

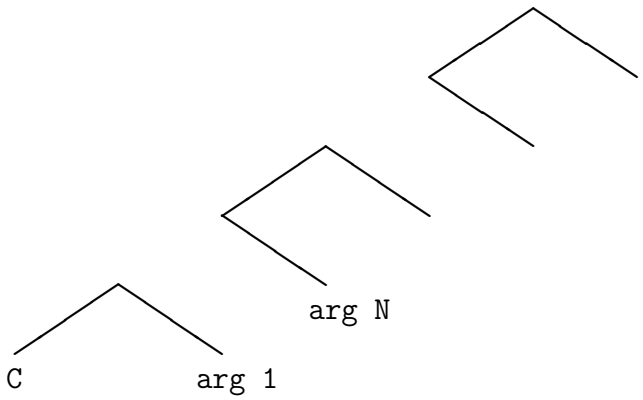


Figure 1 : réduction d'un combinateur C à N arguments.

[4]La plupart des systèmes sont limités à la transformation d'un programme en un exécutable. A contraire un langage interactif comme CAML possède une boucle d'interaction (*top-level*) qui lit une phrase tapée par l'utilisateur, l'exécute, imprime le résultat et enfin attend la phrase suivante.

Les machines à environnement

L'autre technique consiste à conserver un environnement qui contient les valeurs des variables liées. A l'appel d'une fonction les associations paramètres/valeurs sont ajoutées dans cette structure, et lorsqu'une variable est rencontrée dans le corps de la fonction sa valeur est recherchée dans l'environnement. C'est une méthode très classique et utilisée depuis longtemps, pour laquelle ont été développées beaucoup d'optimisations (concernant les additions dans l'environnement, que ce soit en place ou en temps, ou la recherche des valeurs (*lookup*).

L'ancêtre des langages fonctionnels est Lisp[5], un langage applicatif (c'est à dire un langage où les notions de fonction et surtout d'application sont importantes, mais qui ne possède pas de valeur fonctionnelle proprement dite (l'expression usuelle est : les fonctions ne sont pas des valeurs à part entières (*first class citizen*)).

Mais Lisp a énormément évolué depuis les premières versions et certains de ses "dialectes" sont des langages fonctionnels. La plus grande part des concepts qui nous intéressent ont été expérimentés avec Lisp, à commencer par ceux du Lambda Calcul, le père de la notation Lisp d'une fonction anonyme : `(lambda (x) (+ x 1))`.

Les liaisons

Une des notions importantes d'un langage de programmation et en particulier de Lisp est la liaison : le procédé qui associe un nom de variable à une valeur dans un environnement. Le tableau des valeurs peut donc être accédé en suivant les deux indices (le nom de la variable et l'environnement) dans les deux ordres possibles :

- l'environnement, puis la variable, c'est la liaison profonde qui est assez naturelle mais inefficace s'il y a beaucoup de couches entre l'environnement courant et celui qui contient la valeur de la variable, car les deux principales options sont :
  - chaîner les environnements (*static link*). Or le parcours d'un chaînage est proportionnel à sa longueur.

[5] J'ai travaillé au développement du système Le-Lisp [14, Chailloux, Devin, Hullot] dont le système d'exécution est utilisé dans la version actuelle de CAML

  - utiliser un vecteur pointant chaque environnement intéressant (*display*). Si la taille de ce vecteur n'est pas fixée une fois pour toute, sa gestion peut prendre beaucoup de temps.

  Si une pile est utilisée et que la position (relative au sommet de pile, c'est-à-dire à l'environnement courant) peut être déterminée à la compilation, alors cette méthode est la préférée car elle permet de ne pas conserver les noms de variables à l'exécution.

- la variable, puis l'environnement, c'est la liaison superficielle, qui est la technique classique d'une ancienne génération de dialectes Lisp qui associaient à chaque variable une cellule (C-value) contenant la valeur courante, et qui sauvait les anciennes valeurs des paramètres dans la pile à l'appel de fonctions pour les restituer au retour, ce qui est efficace et très peu adapté aux langages parallèles car avoir plusieurs tâches signifie avoir plusieurs environnements courants au même instant.

Portée et durée de vie

Deux propriétés définissent la validité d'une liaison :

la durée de vie ou extension (*extend*) de l'association nom/valeur. En général la durée de vie est soit infinie soit limitée à la durée d'activation de la procédure (donc de l'appel jusqu'au retour), elle est dite alors dynamique.

la portée (*scope*) décrit les endroits dans le texte du programme où la liaison est visible. Le problème de la portée est extrêmement important tant pour ces implications théoriques et pratiques que pour les styles de programmation sous-jacents.

Il y a deux genres de portées :

dynamique (qualifiée en Lisp de spéciale ou de fluide). Une liaison est créée à l'appel d'une procédure et reste visible jusqu'au retour de la procédure. Le seul cas où elle peut être masquée est celui où une fonction avec un paramètre portant le même nom est appelée, mais dans le cas général la visibilité se confond avec la durée de vie dynamique. Schématiquement les environnements forment une pile, l'appel de fonction

empilant et le retour dépilant. L'environnement courant est au sommet et la recherche d'une variable se fait du sommet jusqu'au bas de la pile.

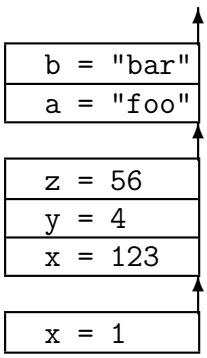
Voici par exemple la forme que peuvent prendre les environnements pour ce morceau de programme LISP :

```
(defun f (a b)
   (print a b)
   (g 123 4 56)
   ... )

(defun g (x y z)
   (if (< y z)
       (h (mod z 5))
       ...))

(defun h (x) ...)

(f "foo" "bar")
```



Liaison dynamique avec pile d'environnement

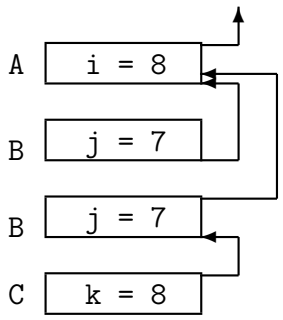
En Lisp, la liaison dynamique est associée avec la liaison superficielle et a prévalu pendant longtemps, au contraire des autres langages, jusqu'à l'émergence récente de la liaison statique. Les derniers dialectes de Lisp proposent souvent les deux types de liaisons, mais la portée statique est désormais la portée par défaut.

statique (qualifiée en Lisp du terme très parlant de lexicale). La visibilité suit exactement la forme textuelle du programme : une variable est visible à partir du début de la fonction jusqu'à la fin. Si au milieu une

fonction définie ailleurs (c'est-à-dire une fonction qui n'est pas locale à celle qui définie la liaison) est appelée, alors dans son corps la liaison n'est pas valide.

Ce type de portée est très facile à compiler. Si le langage n'a pas de valeurs fonctionnelles, alors une simple pile suffit avec un chaînage "statique" entre les différents environnements.

```
let A i =
  let rec B j =
    let C k = k +1
    in if i > j then B (j+1) else C j
  in B (i-1)
in A 8
;;
```

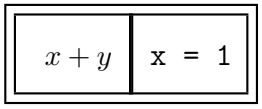


Liaison statique avec chaînage

C'est pourquoi la liaison statique est de très loin la plus employée. Le cas des valeurs fonctionnelles est plus compliqué, car il faut associer à une fonction l'environnement dans laquelle elle a été créée (l'objet résultant est appelé fermeture en anglais) car on "enferme" la valeur des variables libre dans une structure de données représentant l'environnement associée au code de la fonction). Par exemple dans :

```
let f x = fun y -> x + y in
  let g = f 1 in
     g 2
;;
```

l'objet fonctionnel correspondant à la fonction `g` est :



Implantations des environnements

Un choix se présente entre deux techniques d'implantations :

- ou bien les objets fonctionnels sont rares, et alors il faut utiliser une pile qui pourra être copiée dans les cas où c'est indispensable.
- ou bien le style de programmation est très fonctionnel[6]. Sauf dans des cas particuliers (qui peuvent être détectés par une analyse) où il reste possible de mettre une valeur dans la pile, il faut allouer les environnements dans le tas. Il y a deux raffinements à cette méthode :
  - Les valeurs des variables libres sont mises dans un vecteur représentant la partie environnement d'une fermeture. En effet un identificateur peut :
    - ne pas être présent dans le corps et donc être omis.
    - être un des paramètres de la fonction et alors sa valeur est accessible directement.
    - être libre dans le corps de la fonction, mais liée par une fonction à un niveau lexical englobant, alors il faut mettre sa valeur associée dans la fermeture.
    - être une variable globale, et sa valeur est directement accessible.

    L'accès aux valeurs est en temps constant, par contre le partage est très mauvais. Au passage remarquons qu'un combinateur donne des fermetures avec un environnement vide, et se compile donc de façon plus efficace. En revanche la transformation de fonctions en combinateurs par *lambda lifting* ne constitue pas une optimisation car elle remplace les variables libres qui doivent aller dans les environnements par des applications partielles.
  - L'environnement utile est mis sous forme d'un arbre, ce qui permet un partage (et donc une utilisation de l'espace) optimal, mais les temps d'accès aux variables peuvent se dégrader quand il faut parcourir de longues branches de l'arbre. Cette méthode est sûrement la meilleure pour du code très fonctionnel, d'après les performances d'un programme de test réalisant la sémantique dénotationnelle de la suite de Fibonacci en Pascal.

[6] L'article de base sur la compilation des langages fonctionnels est [13, Cardelli]

Pour mettre en valeur les subtilités de la liaison lexicale, voici un exemple :

```
let x = 1;;
Value x = 1 : num

let f () = x;;
Value f = <fun> : (unit -> num)

let x = [1.23];;
Value x = [1.23] : num list

f();;
1 : num
```

L'environnement le plus externe (l'environnement global) est souvent géré d'une manière différente des autres, en particulier dans les systèmes interactifs où chaque phrase peut définir de nouvelles variables globales. Si l'environnement global est à portée statique, alors en fait il peut être considéré comme une suite de blocs emboîtés, un par phrase du *top-level*. Bien sûr l'implantation est plus efficace que pour les autres environnements (généralement des tables de hachages sont utilisées).

```
let (val,next) =
  let counter = ref 0 in
    (fun () -> !counter),
    (fun () -> counter := !counter + 1; ())
;;
Value val = <fun> : (unit -> num)
Value next = <fun> : (unit -> unit)

next(); next();;
() : unit

val();;
2 : num
```

La variable `counter` n'est visible que dans `next` et `val`, et correspond à l'ancienne notion de variables rémanentes.

Un langage fonctionnel non-typé : Scheme

Les langages fonctionnels se divisent en deux grandes classes selon qu'ils soient typés ou non. Le membre le plus important de la seconde est Scheme décrit dans [16, Clinger, Rees]. Un très grand nombre de programmes Scheme illustrant le caractère agréable de l'utilisation d'un langage fonctionnel peuvent être trouvés dans l'excellent ouvrage [1, Abelson, Sussman].

Scheme est un descendant direct de Lisp suivant une approche plus proche de la théorie et donc avec la liaison lexicale. Par exemple le vieux problème de la surcharge de nil, à la fois valeur booléenne faux et liste vide a été résolu en introduisant des symboles spéciaux réservés à des usages uniques. De même la notion de forme spéciale a été supprimée, et les macros ne sont plus dans la partie "essentielle" en attendant qu'un consensus se dégage sur la sémantique qu'il faut leur attribuer.

En Scheme le compilateur ne fait aucune vérification de types. Celles-ci ne sont effectuées qu'à l'exécution. Ainsi l'addition d'une liste et d'une chaîne de caractères ne peut provoquer une erreur que pendant l'exécution. Cette méthode est plus souple et a un plus grand pouvoir d'expression, mais introduit des inefficacités car les tests de types dynamiques prennent beaucoup de temps lorsqu'ils sont systématiques (et ils doivent l'être pour que le langage reste sûr, c'est-à-dire qu'il ne donne pas inopinément des erreurs systèmes comme des violations de segment ou des instructions illégales). De plus certaines erreurs de programmation peuvent ne pas apparaître si le code en cause n'est jamais exécuté dans les jeux de tests.

Le système de type de CAML

A cause de ces facilités de détection d'erreurs et des optimisations possibles, la quasi-totalité des langages fonctionnels sont fortement typés.

A la suite de la première version de ML un grand nombre de ces langages utilisent des systèmes de types effectuant la synthèse de types : avec cette technique l'utilisateur n'écrit que les déclarations de types nécessaires, le compilateur inférant le type des expressions. Par exemple :

```
let f x =  x + 1;;
Value f = <fun> : (num -> num)
```

la primitive d'addition `+` travaillant sur des objets de type *num*, le vérificateur de type en déduit que la variable `x` doit être un nombre, ...

Ce genre de méthode est très agréable pour le programmeur, mais restreint le système de types car la synthèse n'est pas toujours possible. Heureusement elle autorise quand même la construction de systèmes extrêmement puissants comme celui de ML qui possède des caractéristiques originales par rapport aux langages classiques comme Pascal ou C.

Les ML modernes disposent de types de base comme les booléens (*bool*), les nombres (*num*), les chaînes de caractères (*string*), ... et des types non atomiques comme le produit cartésien ($\tau * \sigma$), les vecteurs (σ *vect*), et la fonction ($\tau \to \sigma$).

Des types concrets peuvent être définis : ils consistent en une "somme" de constructeurs prenant des paramètres ou non (dans le dernier cas ces constructeurs sont dits constants). La notion mathématique sous-jacente est l'"union disjointe" : les éléments de $A + B$ sont tous les éléments de A et de B marqués avec une étiquette qui indique s'ils proviennent de A ou de B. Le "produit" correspondant est l'enregistrement plus classique qu'on retrouve par exemple en C ou en Pascal.

Beaucoup de fonctions sont plus ou moins indifférentes aux types des paramètres. Par exemple l'identité peut prendre un argument de n'importe quel type :

```
let id x = x;;
```

donc on pourrait écrire : `id` $:\forall\sigma.\sigma \to \sigma$. ML résout ce problème en introduisant des types polymorphes qui se comportent comme des variables de type notées α, β, etc ... (en CAML `'a`, `'b`, ...). Ainsi un type est synthétisé par CAML :

```
let id x = x;;
Value id = <fun> : ('a -> 'a)

(id id) 1;;
1 : num
```

dans l'expression `(id id) 1`, le premier `id` a pour type $(num \to num) \to (num \to num)$, et le second $(num \to num)$.

Ce polymorphisme paramétrique [50, Milner] permet beaucoup de souplesse car une fonction peut être utilisée dans beaucoup de contextes différents avec exactement le même code. Il peut être utilisé pour construire de nouveaux types, par exemple en CAML le type des listes homogènes (c'est-à-dire dont tous les éléments sont du même type) est défini comme :

```
type 'a List = Nil | Cons of 'a * 'a List;;
```

et toutes les fonctions sur les listes qui ne mettent pas en jeu le type des éléments sont écrites une fois pour toutes, contrairement à ce qu'il aurait fallu faire dans un langage comme Pascal.

Un des principaux avantages du contrôle de type à la compilation est la possibilité de reconnaître des motifs (*pattern matching*). Par exemple la fonction qui renvoie le premier membre d'une liste est :

```
let Hd = fun
    (Cons(x,r)) -> x
  | Nil -> failwith "Hd"
;;
Value Hd = <fun> : ('a List -> 'a)
```

Dans le cas de types avec beaucoup de constructeurs, ce système est incroyablement pratique, permettant par exemple d'écrire un traducteur avec une facilité déconcertante[7]. De plus, la compilation du *pattern matching* est très efficace car seuls les tests nécessaires sur les représentations des valeurs sont produits.

Les avantages des langages fonctionnels

Les langages fonctionnels forment un terrain très favorable au parallélisme. Leur base sémantique, le Lambda Calcul, est naturellement parallélisable, leur pouvoir d'expression se situe au moins au même niveau que celui des langages traditionnels dès que des constructions "impures" mais fort utiles comme les références, les objets modifiables (*mutables*) et les exceptions ont été ajoutées à un noyau "pur".

De plus ils autorisent une grande liberté dans les styles de programmations parallèles, par exemple l'accent peut être mis sur le découpage de structures de données ou sur la présence de plusieurs flots de contrôle. Aucune construction parallèle n'est imposée dès le départ, et nous verrons plus tard qu'en fait aucune construction de communication ou de synchronisation n'est nécessaire.

La source des avantages des langages fonctionnels sur la plupart des autres classes de langages de programmation est leur sémantique très simple

[7] Comme la description du système de méta-compilation SAM [59, Weis] le démontre.

et très rigoureuse[8]. Il est donc naturel d'étudier plus en détail cette sémantique pour en déduire la "bonne" technique pour transformer un langage comme CAML en un langage parallèle.

[8]La description de la sémantique de Scheme tient en quelques pages, et permet l'implantation complète du langage [15, Clinger]

Chapitre 4

Les modes d'évaluation

Syntaxes et sémantiques

La définition d'un langage de programmation se résume à deux composantes :

syntaxe : les règles d'écriture des programmes. La syntaxe est constituée de règles lexicales qui décrivent les mots clés, les nombres, les identificateurs, etc, reconnaissables à l'aide d'automates et d'une grammaire du langage.

Il existe des techniques pour fabriquer un analyseur syntaxique à partir de la grammaire. En CAML, la construction "grammar" permet de déclarer un analyseur produisant des arbres de syntaxe (sous forme de valeurs ou de programmes) [46, Mauny] dont les tables seront extraites automatiquement.

sémantique : la signification des constructions du langage. Elle peut prendre en gros deux formes :

opérationnelle : le langage est spécifié sous forme de règles de réécritures décrivant une étape élémentaire du calcul, et de règles d'inférence précisant comment elles se combinent. Ce type de sémantique permet aisément d'écrire un interprète du langage car il suffit de réaliser (généralement à l'aide d'une machine abstraite) chaque pas de calcul.

dénotationnelle : à chaque terme du langage est associée une fonction dans un formalisme abstrait (un dérivé du Lambda Calcul

typé, surtout dans le cas de langages fonctionnels). Généralement l'ordre d'évaluation n'est pas complètement précisé, sauf quand des "continuations" (objets abstraits qui décrivent les calculs restants à faire) sont utilisées. Ce genre de sémantique est très adapté à la compilation, et en particulier lorsque le calcul correspondant est facile à implanter.

La sémantique dénotationnelle plus abstraite convient à CAML qui permet très aisément de construire les systèmes de calcul nécessaires. Ainsi on peut écrire des méta-compilateurs qui prennent en entrée les spécifications d'un langage et un programme, et qui produisent le code correspondant. Cette technique qui paraît à première vue assez inefficace se marie tellement bien avec les langages fonctionnels qu'elle est utilisée pour produire des compilateurs (applications partielles des méta-compilateurs) commerciaux.

Les fonctions sémantiques sont définies dans des domaines ad hoc. A un ensemble de valeurs comme les nombres naturels est ajoutée la valeur indéfinie *bottom* (notée $\perp$). Cette valeur sémantique particulière sert à donner une topologie adéquate à l'ensemble des fonctions en introduisant une relation d'ordre "plus définie que".

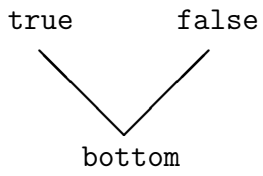


Figure 1 : Le domaine sémantique des booléens.

Il est possible de donner la sémantique de CAML sous forme de Lambda Calcul typé. Un formalisme s'appuyant sur les bases théoriques de CAML est sûrement plus adapté.

La théorie des catégories cartésiennes fermées [20, Curien] (avec le produit cartésien, et l'exponentielle qui se traduit par la notion de fonction) fournit un calcul de combinateurs dits catégoriques équivalent au Lambda Calcul. La réduction faible des combinateurs catégoriques peut être réalisée par une machine : la CAM (*Categorical Abstract Machine*), dont nous parlerons au chapitre suivant.

L'appel par valeur

La sémantique habituelle de l'application :

$$(M\,N)(\rho) \Rightarrow M'(\rho.(x = N')) \texttt{ avec } M \Rightarrow (\lambda x.M')\,et\,N \Rightarrow N'$$

est moins immédiate qu'il n'y paraît à première vue. Car l'ordre d'évaluation est imposé (ce qui correspond dans le Lambda Calcul à choisir l'ordre de réductions des radicaux).

Cet ordre est communément appelé "appel par valeur" parce que la valeur des arguments est passée à la fonction.

L'appel par nom

La partie vraiment importante d'une technique d'appel de fonction est l'ordre des opérations. Ainsi on peut choisir de ne pas évaluer les arguments avant de procéder à l'appel, et de n'effectuer le calcul que lorsque la valeur d'un argument est utilisée : c'est l'appel par nom [52, Plotkin].

Un programme écrit dans un langage fonctionnel peut être transformé pour utiliser l'appel par nom. A chaque appel de fonction il faut créer une suspension (*thunk*) pour chaque paramètre, avec le code qui permet de calculer sa valeur et l'environnement dans lequel il faudra exécuter ce code (donc l'environnement courant à l'appel). Ces suspensions ont la même structure que des fermetures (couple code et environnement), il est donc naturel d'utiliser des fermetures pour les implanter.

La transformation pour un langage du premier ordre se fait donc sur le schéma :

```
  (Appl (M,N)) -> (Appl (M, Lambda (Void, N)))
| (Var x)      -> (Appl (x, Void))
```

Le principal défaut de l'appel par nom apparaît immédiatement : il faut créer et évaluer des suspensions, ce qui prend du temps. En plus, l'appel par nom rend souvent impossible certaines optimisations comme l'utilisation d'une pile pour placer la valeur d'une variable (au lieu d'allouer de l'espace dans un environnement ordinaire).

L'appel par nécessité

L'appel par nom proprement dit n'est presque jamais utilisé car il en existe une optimisation immédiate : au lieu de calculer la valeur à chaque

fois, il suffit de ne la calculer qu'une fois et de réutiliser le résultat.

Donc les suspensions sont remplacées par un objet plus complexe nommé habituellement glaçon. A l'appel de fonction un glaçon (un code et un environnement) est associé à chaque paramètre. Quand une valeur devient nécessaire, le code est exécuté dans l'environnement du glaçon et le résultat est noté dans le glaçon une fois pour toutes. Cette technique où le calcul n'est effectué que lorsqu'on en a besoin est nommée l'appel par nécessité. Il se généralise aux structures de données, etc ... en l'"évaluation paresseuse" (*lazy evaluation*)[1].

Les réalisations de l'appel par nécessité

Les glaçons créés par la primitive `freeze` et consultés par `force` peuvent être décrits par[2] :

```
(* les types des résultats et des glaçons *)

type result = Not_evaluated | Evaluated of val
and  thunk  = { Code : code;
                Env : env;
                mutable Result : result }
;;

(* création d'un glaçon *)

let freeze (code, env) = {Code = code;
                          Env = env;
                          Result = Not_evaluated}
;;

(* dégel d'un glaçon *)

let force x =
  match x.Result with
     (Evaluated v) -> v
   | Not_evaluated ->
       let v = eval(x.Code, x.Env) in
         x.Result <- v; v
;;
```

[1] L'appel par nécessité a été introduit dans le monde des langages fonctionnels par [29, Friedman, Wise].

[2] L'annotation `mutable` précise que le champ peut être modifié par l'opération `<-`

Le test peut être évité en utilisant le code de “retourne immédiatement l'environnement réduit à une valeur” :

```
type code' = Return | Code of code
and  env'  = Env of env | Val of val
and  thunk = {Code : code' ; Env : env'}
;;

let freeze (code, env) =
  {Code = (Code code); Env = (Env env)}
;;

(* dégel d'un glaçon sans test *)

let force x =
  let v = eval'(x.Code, x.Env) in
    x.Env <- (Val v); v
;;
```

Généralement on procède à une substitution directe du glaçon :

```
type value_or_thunk = Val of val | Thunk of thunk
and  thunk = {Code : code ; Env : env}
;;

let freeze (code, env) =
  ref (Thunk {Code = code; Env = env})
;;

(* dégel d'un glaçon avec mise à jour immédiate *)

let force x =
  match !x with
     (Val v) -> v
   | (Thunk {Code = code; Env = env}) ->
      let v = eval(code, env) in
        x := (Val v); v
;;
```

Mais cette méthode peut poser des problèmes, car il est impossible de remplacer physiquement une valeur par une autre valeur prenant plus de place. Il faut donc que tous les objets aient la même taille (soit réellement,

soit en utilisant un en-tête de taille fixe avec un pointeur vers la partie de taille variable), ou bien se résoudre à mettre à jour le glaçon par un pointeur d'indirection vers la valeur (mais alors il faut tester sa présence à chaque accès, ce qui est extrêmement coûteux).

La version mixte de CAML

La version mixte (appel par valeur et appel par nécessité) de CAML emploie une technique plus subtile : les constructions paresseuses sont restreintes pour garantir que le "père" d'un glaçon soit toujours un vecteur ou une paire (soit explicitement dans une structure de donnée, soit implicitement dans un environnement représenté comme un arbre, donc formé de paires). Le pointeur vers un glaçon est toujours unique, car lorsqu'une composante d'un objet qui peut être un glaçon est accédée, elle est automatiquement dégelée s'il le faut. En effet, il suffit d'assurer l'unicité à la création et alors il est impossible de dupliquer un pointeur sur un glaçon car l'accès provoque le dégel.

Cette méthode ne restreint pas trop les possibilités d'utilisation de l'évaluation paresseuse, et est très efficace. Les seules parties du système à modifier sont les fonctions d'accès aux composantes des paires et des vecteurs.

L'appel par spéculation

L'appel par nécessité peut être étendu pour les langages parallèles. Au lieu d'évaluer un glaçon au dernier moment, il est possible d'utiliser des processeurs pour dégeler à l'avance les glaçons : c'est l'appel par spéculation, car on spécule sur le fait que les valeurs calculées à l'avance seront utiles.

Les glaçons dans le cadre de l'appel par spéculation (nommés souvent *futures* d'après la terminologie de MultiLisp [33, Halstead]) peuvent avoir trois états, successivement :

froid : le glaçon n'a pas encore attiré l'attention sur lui. Il est conservé dans une "glacière" où un processeur pourra le prendre.

tiède : un processus est en train d'évaluer le glaçon, soit parce qu'il l'a retiré de la glacière, soit parce que sa valeur est nécessaire à un calcul.

fondu : le glaçon a été entièrement évalué et remplacé par sa valeur. Il disparaîtra définitivement quand il ne restera plus de pointeurs (transitoires avec un système comme CAML) vers lui.

Les propriétés des modes d'évaluation

L'appel par nom, par nécessité et par spéculation ont tous les mêmes propriétés. A part pour les effets de bord, l'appel par spéculation a exactement la même sémantique que les appels par nécessité et par nom, c'est-à-dire qu'un programme donnera le même résultat ou la même erreur ou ne terminera pas.

Ces propriétés théoriques sont remarquables et ont été étudiées dans le cadre de l'appel par nom (et restent valables pour l'appel par nécessité ou par spéculation) ([57, Vuillemin] et [11, Berry, Lévy]). En effet dans le Lambda Calcul il existe des réductions (plus exactement des ordres de réduction) standards : si un terme M se réduit complètement en N, alors il existe une réduction standard qui donne N à partir de M. Toutes les réductions possibles peuvent être réalisées pour une réduction standard.

L'ordre normal qui réduit les radicaux les plus à gauche et à l'extérieur (*leftmost-outermost*) est un ordre standard, et correspond à l'appel par nom dans les langages de programmation.

L'appel par valeur est moins satisfaisant, car l'ordre équivalent dans le Lambda Calcul ne donne pas toujours la forme normale quand elle existe. Ce résultat a des conséquences directes dans les langages de programmations.

Par exemple :

terminaison : certains programmes ne terminent pas ("bouclent") en appel par valeur alors que l'appel par nom rend un résultat. Voici une démonstration du phénomène :

```
(fun x -> 1) (while true do () done);;
```

conditionnelle : les constructions conditionnelles n'évaluent pas toutes les branches. Il est impossible d'écrire une fonction en appel par valeur équivalente à une conditionnelle, par exemple la fonction :

```
let IF (test,vrai,faux) = if test then vrai else faux;;
```

évalue toujours les arguments `vrai` et `faux`, comme le montre l'exemple :

```
let x = 0 in
  IF(x = 0, 1, 1 mod x)
;;
```

récursion : elle ne peut pas être réalisée dans sa pleine généralité avec un appel par valeur. Souvent la récursion est réservée aux fonctions et interdite pour les valeurs d'autres types comme en Scheme ou en Standard ML.

Le caractère strict

Dans certains cas il est pourtant possible d'utiliser de façon équivalente les modes d'appels. Si un programme termine avec l'appel par valeur, alors d'après le théorème de standardisation l'appel par nom (ou par nécessité ou spéculation) termine aussi et avec le même résultat.

En utilisant la valeur sémantique spéciale $\perp$, les choses deviennent claires : le cas de non-terminaison correspond à la valeur indéfinie. Donc les deux appels sont équivalents pour une fonction qui vérifie :

$$f \perp = \perp$$

Ces fonctions sont dites "strictes". Une interprétation abstraite [19, Cousot et Cousot] (nommée en anglais *strictness analysis*) donne à la compilation une approximation du caractère strict des fonctions, qui autorise le remplacement par l'appel par valeur qui est généralement bien plus efficace.

La *strictness analysis* est très employée dans les langages fonctionnels qui ne pratique pas l'appel par valeur, car elle permet d'obtenir des performances avouables [2, Abramsky, Hankin]. Malheureusement cette analyse est très difficile à réaliser en général car le problème de fond est indécidable (il se réduit trivialement au problème de l'arrêt), la récursion engendre d'énormes difficultés techniques, et les structures de données comme les listes nécessitent des domaines très compliqués. Finalement cette technique n'est pas encore assez développée pour être assurée de combler en pratique l'écart de performances entre un système qui ne pratique que l'appel par nom ou ses dérivés et un système qui ne l'utilise qu'à l'occasion.

Des applications de l'évaluation paresseuse

La sémantique supérieure de l'évaluation paresseuse autorise de nouvelles techniques de programmation qui produiraient des calculs ne se terminant pas avec une évaluation stricte. Nous illustrons ces nouveaux procédés avec quelques exemples typiques.

Les structures potentiellement infinies sont possibles avec l'évaluation

paresseuse. Par exemple les flots (*streams*), listes infinies :

```
type 'a stream = { lazy Hd : 'a ; lazy Tl : 'a stream };;
```

Avec un appel par valeur, un objet de type `stream` ne peut pas généralement être créé car le système chercherait à calculer toutes les composantes, or leur nombre n'est pas borné (sauf dans des cas particuliers comme :

```
let rec x = 1::x;;
```

qui peut être représenté par un graphe fini).

Les flots sont une structure de données typiquement associée à l'appel par nécessité et sont très agréables à utiliser dans certains domaines comme :

ZF expression : c'est un des noms des listes ou des flots notés comme des ensembles $\{x \in E\ P(x)\}$. Par exemple les longueurs des cotés des triangles rectangles :

```
let pythagore n =
  [ (a,b,c) | a,b,c <- [1..n]; a+b+c<=n; a*a+b*b=c*c ]
;;
```

Le tri rapide s'écrit très facilement avec ce genre de notation (`@` est la concaténation des listes) :

```
let rec qsort = fun
    []     -> []
  | (x::r) ->
      (qsort [y | y <- r; y<x])@[x]@(qsort [y | y <- r; y>x])
;;
```

séries formelles : elles se programment aisément, et on peut même coder des fonctionnelles. Par exemple l'intégration :

```
let integrate l = intrec 0 l
 where rec intrec n (x::r) = (x/(n+1))::(intrec (n+1) r)
;;
```

L'appel par nécessité et ses dérivés n'évaluant un sous-ensemble d'une structure de données que lorsque c'est nécessaire, la récursion peut être implémentée de façon parfaitement correcte, et des objets comme le flot des nombres naturels (en notant les flots comme les listes) peuvent être définis facilement :

```
let rec map f (x::r) = (f x)::(map f r);;

let rec Nat = 0::map (fun x -> x+1) Nat;;
```

Des propriétés pour le parallélisme

Le sous-langage de CAML sans les références et les objets modifiables possède deux propriétés intéressantes pour le parallélisme qui peuvent être dérivées de ses sémantiques [53, Plotkin] :

déterminisme (en anglais *determinacy*). Un programme donne toujours le même résultat. Il n'y a pas d'opérations comme le choix dont le déroulement peut prendre une voie ou une autre sous l'influence de paramètres extérieurs. C'est à notre avis une propriété extrêmement importante pour la mise au point et pour la correction des transformations de programme.

séquentialité : cette propriété semble assez paradoxale, surtout avec un langage possédant une extension parallèle.

Cette terminologie provient du Lambda Calcul : si on représente les termes à calculer comme des arbres où les radicaux sont remplacés par des Ω marquant les endroits où peut se poursuivre un calcul, alors pour un programme il existe au moins un Ω nécessaire (c'est-à-dire un Ω qu'il faut calculer pour obtenir le résultat). Il est donc impossible de programmer par exemple le "parallel-OR" dont la définition sémantique est :

pOR	$\perp$	$False$	$True$
$\perp$	$\perp$	$\perp$	$True$
$False$	$\perp$	$False$	$True$
$True$	$True$	$True$	$True$

car dans $pOR(\Omega, \Omega)$ aucun des deux Ω n'est nécessaire. Il faut les calculer en parallèle, d'où le nom de "parallel-OR".

En annexe nous démontrons que dans le Lambda Calcul typé avec la conditionnelle, les booléens et la récursion, c'est-à-dire dans le noyau d'un langage comme CAML, le "parallel-OR" ne peut pas être programmé. Donc une construction de ce type n'est pas exprimable dans une extension parallèle de CAML où elle n'est pas ajoutée explicitement sous une forme ou une autre.

L'excellente assise théorique que donne à CAML sa sémantique aide à définir l'implémentation de telle ou telle construction, dès que le problème peut être posé en terme de signification.

Chapitre 5

Compilation

Les combinateurs catégoriques

Le calcul des combinateurs catégoriques [20, Curien] [35, Hardin] est la base de la machine abstraite CAM. Généralement le premier pas du passage du Lambda Calcul vers un système de combinateurs consiste à éliminer les variables. Dans ce cas, le formalisme de de Bruijn est utilisé : il consiste à remplacer les occurrences des variables par une distance (le nombre de λ entre ceux qui lient les variables et leurs occurrences). Plus précisément avec l une liste de variables libres $[v_0; v_1; \ldots; v_p]$:

- $[v]_l \rightarrow i$ où i est le plus petit entier tel que $v_i = v$
- $[\lambda v.M]_l \rightarrow \lambda.[M]_{v::l}$

Les nombres de de Bruijn sont bien connus en compilation, car ils correspondent aux indices lexicaux de variables (qui forment généralement un couple avec la distance entre l'environnement courant et celui de la variable et le numéro d'ordre de la variable dans son environnement).

Le point de départ du support théorique de CAML est un Lambda Calcul avec les indices de de Bruijn, le produit cartésien $<\ ,\ >$ et les deux projections Fst et Snd. Il se traduit dans la logique combinatoire catégorique forte par les règles $(CCL\beta\eta SP)$:

$$\begin{array}{rcl}
[\![i]\!] & \Rightarrow & Snd \circ Fst^i \\
[\![MN]\!] & \Rightarrow & App \circ < [\![M]\!], [\![N]\!] > \\
[\![\lambda.M]\!] & \Rightarrow & \Lambda([\![M]\!]) \\
[\![Fst(M)]\!] & \Rightarrow & Fst \circ [\![M]\!] \\
[\![Snd(M)]\!] & \Rightarrow & Snd \circ [\![M]\!] \\
[\![< M, N >]\!] & \Rightarrow & < [\![M]\!], [\![N]\!] >
\end{array}$$

Cette logique étant définie par les équations ((SP) et $(S\Lambda)$ étant des conséquences des autres) :

$$\begin{array}{lrcl}
(Ass) & (x \circ y) \circ z & = & x \circ (y \circ z) \\
(IdL) & Id \circ x & = & x \\
(IdR) & x \circ Id & = & x \\
(Fst) & Fst \circ < x, y > & = & x \\
(Snd) & Snd \circ < x, y > & = & y \\
(Dpair) & < x, y > \circ z & = & < x \circ z, y \circ z > \\
(FSI) & < Fst, Snd > & = & Id \\
(SP) & < Fst \circ x, Snd \circ x > & = & x \\
(D\Lambda) & \Lambda(x) \circ y & = & \Lambda(x \circ < y \circ Fst, Snd >) \\
(Beta) & App \circ < \lambda(x), y > & = & x \circ < Id, y > \\
(AI) & \Lambda(App) & = & Id \\
(S\Lambda) & \Lambda(App \circ < x \circ Fst, Snd >) & = & x
\end{array}$$

Ces équations avec les opérateurs composition $\circ$, couple $<\ ,\ >$, identité Id, projections Fst et Snd, curryfication Λ et applicateur App sont directement dérivées des axiomes définissant les Catégories Cartésiennes Fermées.

En introduisant $A^> = \Lambda(A \circ Snd)$ et $A^< = App \circ < A, Id >$ on peut définir l'application $AB = (A \circ B^>)^<$ et le couple $(A, B) =< A^>, B^> >^<$ dont les propriétés sont décrites par la logique combinatoire catégorique faible :

$$\begin{array}{lrcl}
(id) & Idx & = & x \\
(ass) & (x \circ y)z & = & x(yz) \\
(fst) & Fst(x, y) & = & x \\
(snd) & Snd(x, y) & = & y \\
(dpair) & < x, y > z & = & (xz, yz) \\
(app) & App(x, y) & = & xy \\
(d\Lambda) & \Lambda(x)yz & = & x(y, z)
\end{array}$$

La dernière équation expliquant le nom de curryfication donné à Λ. Ces logiques ne sont intéressantes que parce que la machine catégorique abstraite

CAM réalise une réduction faible, et que les règles fortes permettent de justifier des optimisations (que la machine ne sait pas simuler) comme :

$$\lambda x.(\lambda y.y)x \to \lambda x.x$$

qui correspond au calcul catégorique fort :

$$\Lambda(App \circ < \Lambda(Snd), Snd >) \overset{Beta}{\to} \Lambda(Snd \circ < Id, Snd >) \overset{Snd}{\to} \Lambda(Snd)$$

mais en fait le plus important est la compilation vers la machine CAM et le fonctionnement de celle-ci.

La machine abstraite catégorique

La machine CAM [18, Cousineau, Curien, Mauny] est constituée de :

un accumulateur contenant un terme (généralement un graphe) sur lequel sont effectués les calculs.

un code sous forme d'une liste d'instructions.

une pile servant à sauver des résultats partiels et des états.

et suit les règles (données comme un fragment de code CAML transformant une configuration de la machine `(accu,code,stack)` en la configuration suivante :

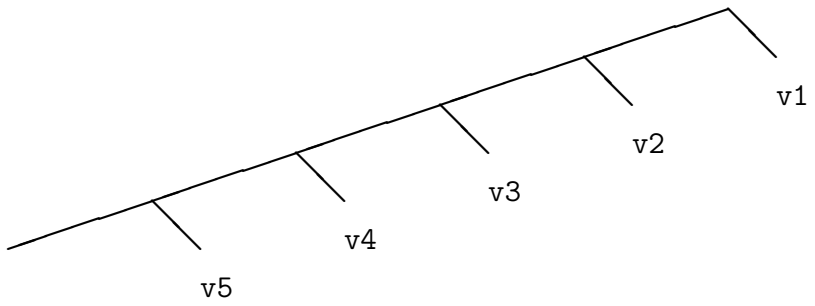
```
   ((s,t),     Car::C,           S)     ->  (s,       C,  S)
 | ((s,t),     Cdr::C,           S)     ->  (t,       C,  S)
 | (s          (Cur C)::C',      S)     ->  ((C,s),   C', S)
 | (s,         Push::C,          S)     ->  (s,       C,  s::S)
 | (t,         Swap::C,          s::S)  ->  (s,       C,  t::S)
 | (t,         Cons::C,          s::S)  ->  ((s,t),   C,  S)
 | (((C,s),t), App::C',          S)     ->  ((s,t),   C,  C'::S)
 | (s,         [],               C::S)  ->  (s,       C,  S)

 | (t,         Pop::C,           s::S)  ->  (s,       C,  S)
 | (s,         (Quote c)::C,     S)     ->  (c,       C,  S)
 | (s,         PrimOp::C,        S)     ->  (Op(s),   C,  S)
 | ((s,t),     BinOp::C,         S)     ->  (Op(s,t), C,  S)
 | (true,      Branch(C1,C2)::C, s::S)  ->  (s,       C1, C::S)
 | (false,     Branch(C1,C2)::C, s::S)  ->  (s,       C2, C::S)
 | ((s,t),     Rplac::C,         u::S)  ->  ((s,u),   C,  S)
```

Les constantes (avec l'instruction `Quote`) ainsi que les primitives d'arités 1 ou 2, la conditionnelle et une modification physique ont été ajoutées.

La traduction de la logique catégorique forte en code CAM est claire, les projections étant notées `Car` et `Cdr` comme en Lisp au lieu de Fst et Snd, sauf pour le produit cartésien, $<,>$ se traduisant en `Push`, `Swap` et enfin `Cons` [47, Mauny, Suárez].

L'environnement est constitué d'un arbre gauche de la forme :



Le schéma de compilation strict

Il est temps de passer à la compilation proprement dite. Les constantes sont transformées en `Quote`, les variables en une chaîne de `Car` suivie d'un `Cdr`, en suivant la forme de l'environnement qui est calculé en même temps.

$$
\begin{array}{rcl}
[\![c]\!]\rho & \Rightarrow & [\texttt{Quote}\, c] \\
[\![v]\!]\rho & \Rightarrow & \texttt{access}\, v\, \rho \\
[\![\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3]\!]\rho & \Rightarrow & [\texttt{Push}]@[\![e_1]\!]\rho@[\texttt{Branch}([\![e_2]\!]\rho, [\![e_3]\!]\rho)] \\
[\![(e_1, e_2)]\!]\rho & \Rightarrow & [\texttt{Push}]@[\![e_1]\!]\rho@[\texttt{Swap}]@[\![e_2]\!]\rho@[\texttt{Cons}] \\
[\![e_1;\, e_2]\!]\rho & \Rightarrow & [\texttt{Push}]@[\![e_1]\!]\rho@[\texttt{Pop}]@[\![e_2]\!]\rho \\
[\![\texttt{let}\, x = e_1\, \texttt{in}\, e_2]\!]\rho & \Rightarrow & [\texttt{Push}]@[\![e_1]\!]\rho@[\texttt{Cons}]@[\![e_2]\!]\rho.x \\
[\![\texttt{let rec}\, x = e_1\, \texttt{in}\, e_2]\!]\rho & \Rightarrow & [\texttt{Push}; \texttt{Quote}(); \texttt{Cons}; \texttt{Push}]@[\![e_1]\!]_{\rho.x} \\
 & & @[\texttt{Swap}; \texttt{Rplac}]@[\![e_2]\!]_{\rho.x} \\
[\![\lambda x.e]\!]\rho & \Rightarrow & [\texttt{Cur}([\![e]\!]_{\rho.x})] \\
[\![\texttt{fst}\, e]\!]\rho & \Rightarrow & [\![e]\!]\rho@[\texttt{Car}] \\
[\![\texttt{snd}\, e]\!]\rho & \Rightarrow & [\![e]\!]\rho@[\texttt{Cdr}] \\
[\![\texttt{PrimOp}\, e]\!]\rho & \Rightarrow & [\![e]\!]\rho@\texttt{Code}(Op) \\
[\![\texttt{BinOp}\, e]\!]\rho & \Rightarrow & [\![e]\!]\rho@[\texttt{Push}; \texttt{Car}; \texttt{Swap}; \texttt{Cdr}]@\texttt{Code}(Op) \\
[\![(e_1\, e_2)]\!]\rho & \Rightarrow & [\texttt{Push}]@[\![e_1]\!]\rho@[\texttt{Swap}]@[\![e_2]\!]\rho@[\texttt{Cons}; \texttt{App}]
\end{array}
$$

Les variables closes dans un terme peuvent être mises en pile et accédées :

- par un déplacement par rapport au sommet de pile, ce qui oblige à calculer précisément la forme qu'aura la pile à l'exécution.
- dans un bloc placé dans la pile et pointé par un registre (c'est la méthode utilisée actuellement par CAML), ce qui est la seule solution quand le déplacement par rapport au sommet de pile n'est pas calculable (à cause d'une récursion par exemple).
- par des blocs chaînés en pile, comme pour un langage classique. Ce n'est pas très rentable, car il y a peu de cas pratiques où une technique aussi compliquée est réellement utilisable.

Dans les `let` récursifs ou non et les abstractions (le `let` peut être considéré pour la compilation comme un cas particulier de l'abstraction car `let x=e1 in e2` est équivalent à `(fun x -> e2) e1`, mais le `let` présente beaucoup plus d'opportunité pour les optimisations, et en particulier on effectue dès la compilation la β-réduction), la variable de liaison peut être remplacée par des motifs, et la forme `and` est tout simplement traité comme une paire :

```
let x=a and y=b in e;;
```

devient

```
let (x,y) = (a,b) in e;;
```

La compilation du filtrage

Le filtrage se traduit en une série de tests sur la représentation des valeurs pour choisir une branche qui est ensuite compilée avec les variables du motif placées dans l'environnement.

Dans le cas d'une évaluation paresseuse, il ne faut évaluer que les parties d'une valeur qui permettent de choisir entre les motifs. Par exemple dans :

```
match x with
    (_,true) -> true
  | (_,_)    -> false
;;
```

il ne faut pas commencer par la première composante de la paire. Dans certains cas il n'y a pas de partie nécessaire comme pour :

```
match x with
    (true,true,false) -> true
  | (true,false,true) -> true
  | (false,true,true) -> true
  | (_,_,_)           -> false
;;
```

en revanche [43, Laville] propose un algorithme qui détermine s'il existe une façon "paresseuse" de scruter la valeur et produit le code correspondant. Ce problème peut donc être considéré comme réglé.

La récursion en CAML strict est compilée grâce à une modification physique de l'environnement qui permet à la fermeture d'une fonction récursive de figurer dans son propre environnement. Cette méthode a l'avantage de construire certaines valeurs récursives, mais n'est pas correcte dans le cas général de l'appel par valeur. Pour des fonctions, une boucle dans le code est bien plus efficace. En revanche, cette technique est parfaitement valable pour un système paresseux.

Le schéma de compilation paresseux

La machine CAM paresseuse [45, Mauny] est très proche de la version stricte, seules changent les règles pour `Car` et `Cdr`, et une nouvelle instruction `Fre` construisant des glaçons `{ }` (des paires munies d'un drapeau permettant de les distinguer) est ajoutée :

```
   (({s},t), Car::C,      S)  ->  (Eval{s}, C,  S)
      avec ({s},t) <- (Eval{s},t)
 | ((s,t),   Car::C,      S)  ->  (s,       C,  S)
 | ((s,{t}), Cdr::C,      S)  ->  (Eval{t}, C,  S)
      avec (s,{t}) <- (s,Eval{t})
 | ((s,t),   Cdr::C,      S)  ->  (t,       C,  S)
 | (s,       (Fre C)::C', S)  ->  ({C,s},   C', S)
```

Les règles de compilation sont modifiées en :

$$
\begin{aligned}
[\![(e_1, e_2)]\!]\rho &\Rightarrow [\texttt{Push}; \texttt{Fre}[\![e_1]\!]\rho; \texttt{Swap}; \texttt{Fre}[\![e_2]\!]\rho; \texttt{Cons}] \\
[\![\texttt{let}\, x = e_1\, \texttt{in}\, e_2]\!]\rho &\Rightarrow [\texttt{Push}; \texttt{Fre}[\![e_1]\!]\rho; \texttt{Cons}]@[\![e_2]\!]\rho.x \\
[\![\texttt{let rec}\, x = e_1\, \texttt{in}\, e_2]\!]\rho &\Rightarrow [\texttt{Push}; \texttt{Quote}(); \texttt{Cons}; \texttt{Push}; \\
&\quad \texttt{Fre}[\![e_1]\!]\rho; \texttt{Swap}; \texttt{Rplac}]@[\![e_2]\!]\rho.x \\
[\![(e_1\, e_2)]\!]\rho &\Rightarrow [\texttt{Push}]@[\![e_1]\!]\rho@[\texttt{Swap}; \texttt{Fre}[\![e_2]\!]\rho@[\texttt{Cons}; \texttt{App}]
\end{aligned}
$$

Les constructions paresseuses sont donc les paires explicites, l'application (seul l'argument est gelé) et les différents `let`. Le pointeur vers le glaçon est bien créé en un seul exemplaire, et comme les instructions d'accès `Car` et `Cdr` forcent le dégel, il ne peut être dupliqué.

Dans l'environnement, les glaçons ne peuvent être présents que comme feuilles, donc dans les chaînes d'accès aux valeurs des variables la présence d'un glaçon n'est testé qu'à la dernière indirection. Mais il est beaucoup moins fréquent de pouvoir mettre une valeur dans la pile à cause des glaçons : si un glaçon utilise la valeur d'une variable, alors cette valeur ne peut pas être empilée à un emplacement qui ne sera peut-être plus valide quand on dégèlera le glaçon.

Les deux compilateurs sont utilisables dans le même système sans difficulté, il suffit de garder la définition paresseuse pour `Car` et `Cdr` et d'introduire les constructions `lazy` et `strict` pour changer de schéma de compilation :

$$\begin{array}{rcl} [\![lazy\ e]\!]^{xxx} & \Rightarrow & [\![e]\!]^{lazy} \\ [\![strict\ e]\!]^{xxx} & \Rightarrow & [\![e]\!]^{strict} \end{array}$$

pour changer de mode de compilation.

Certains termes triviaux ne produisent jamais de glaçons (c'est un embryon de *strictness analysis* : il est clair par exemple qu'il ne faut pas geler une constante). Les primitives sont par définition strictes, et pour forcer l'évaluation complète une nouvelle primitive `force_all` est fournie : elle parcourt récursivement (en prenant garde aux cycles) son argument pour forcer le dégel de tous les glaçons qui s'y trouvent.

Le schéma de compilation parallèle

Cette machine CAM paresseuse s'étend très facilement à une machine parallèle en ajoutant l'instruction `Fut` (pour *future*) qui se comporte exactement comme `Fre` si ce n'est que le glaçon créé est placé dans une glacière d'où un processeur disponible pourra le retirer pour l'évaluer.

La CAM parallèle possède donc deux genres de glaçons : les glaçons spéculatifs et les glaçons paresseux ou ordinaires qui ne sont pas différents sinon au moment de la création.

La compilation ne change presque pas, les règles créant des glaçons dans le schéma $[\![\ ]\!]^{lazy}$ sont dupliquées dans le schéma $[\![\ ]\!]^{parallel}$ en remplaçant `Fre` par `Fut`, et une nouvelle construction `parallel` similaire à `strict` et `lazy` permet de prendre en compte ce nouveau mode de compilation.

Donc avec très peu de modifications, il est possible de transformer le compilateur du CAML mixte en un compilateur parallèle admettant trois modes de compilation :

1. l'appel par valeur en mode *strict*
2. l'appel par nécessité et plus généralement l'évaluation paresseuse en mode *lazy*
3. l'appel par spéculation par une simple extension du mode précédent produisant le mode *parallel*

La maintenance du système devient très aisée, et toutes les améliorations apportées au mode *lazy* profitent immédiatement au mode *parallel*, en particulier les analyses comme la *strictness analysis*.

Chapitre 6

Noyau du système paresseux

Si une machine abstraite comme la CAM fournit un modèle pour la compilation, elle ne correspond pas au jeu d'instructions du processeur qui va effectivement exécuter le programme, et donc il faut faire une traduction. Deux grandes méthodes peuvent être utilisées :

- un interprète de "*bytecode*" : la machine cible est une machine virtuelle avec un jeu d'instructions ad hoc (généralement les codes d'opérations tiennent sur un octet, d'où le nom de *bytecode*). Ce code est ensuite interprété.

  Cette technique permet une très grande portabilité et une génération de code très facile. Le code est souvent très compact, mais le surcoût dû à l'interprétation est souvent important (on peut constater une dégradation de performances dans un facteur de 10 à 20). En revanche la mise au point est très aisée, ce qui explique l'utilisation fréquente de *bytecode* pour les prototypes.

- la production directe de code natif. Cette méthode autorise de plus grandes performances, mais au prix de la réécriture du générateur de code pour chaque processeur, tâche qui peut être extrêmement difficile et coûteuse. La mise au point est pénible et chaque modification même mineure demande une grande expertise.

L'implantation de CAML

Actuellement le système CAML produit du code LLM3, machine virtuelle de Le-Lisp [14, Chailloux, Devin, Hullot], qui permet un portage facile de

CAML pour un processeur où le système Le-Lisp a déjà été porté. Mais cette machine LLM3 n'est pas très bien adaptée à CAML, car elle est optimisée pour les opérations les plus fréquentes de Le-Lisp (comme le test de type) qui ne sont pas les plus utilisées par CAML. De plus, le système de ramassage de miettes est complètement inadapté à un langage fonctionnel qui alloue des structures éphémères pour réaliser des fermetures.

Elle possède quatre accumulateurs $A1$, $A2$, $A3$, $A4$, et une pile. Le jeu d'instructions permet les mouvements indirects et indexés pour les chaînes et les vecteurs, une arithmétique entière sur 16 bits et une arithmétique flottante (au choix sur 64 bits avec allocations ou avec 31 bits de valeurs immédiates), des tests de type Lisp, des primitives d'entrées-sorties et de manipulations de chaînes et de vecteurs.

En prenant un modèle intermédiaire avec un nombre non borné de registres, et en affectant ces registres virtuels dans les vrais registres, l'excédent étant placé sur la pile (technique de *spill*), ce qui est la méthode standard des compilateurs pour des langages comme C ou Pascal, il est assez facile de produire directement un code efficace pour une machine possédant beaucoup de registres. En revanche il est beaucoup plus difficile de passer d'une machine à pile comme la CAM à un processeur avec des registres. En effet il faut "cacher" le sommet de pile dans des registres, ce qui permet au mieux l'utilisation de quelques accumulateurs.

Ainsi le code CAM est transformé en un code LLM3, certaines instructions CAM étant remplacées par du code en ligne et d'autres par un appel à un sous-programme LLM3. Par exemple l'application appelle un sous-programme correspondant au pseudo-code CAML (qui réalise la séquence [Cons ; App] du chapitre précédent) :

```
(* le type des fermetures générales *)

type closure = Comb of code | Clos of code * env;;

(* l'application d'une fermeture *)

let appl (func,arg) =
  match func with
     (Comb code)       -> eval(code,arg)
   | (Clos(code, env)) -> eval(code,(env,arg))
;;
```

En fait les combinateurs (fermeture avec un environnement vide) et les fermetures ordinaires ont une représentation différente, respectivement `((),code)` et `(code,env)`. Deux optimisations immédiates semblent possibles :

- faire l'expansion du code de l'application, surtout dans le cas d'une fonction globale (qui est remplacée par une constante au chargement du code).

- représenter les combinateurs par l'adresse du code, et pas par une paire.

Mais elles ne sont pas faites car les systèmes de mise au point, statistiques, traces, etc, substituent physiquement les paires représentant les fermetures.

L'implantation des glaçons

L'évaluation paresseuse se résume en un retard sur le calcul et en mise à jour du glaçon par la valeur. Le code le plus simple pour l'instruction CAM `Car` en mode paresseux est :

```
(* le type des paires modifiables *)

type ('a, 'b) mpair = {mutable Fst : 'a; mutable Snd : 'b };;

(* le type des glaçons *)

type thunk = {Code : code; Env : env};;

(* Car avec dégel *)

let car_unf p =
  if is_thunk (p.Fst)
   then
     (let v = eval(p.Fst.Code, p.Fst.Env) in
         (* mise à jour dans p *)
         p.Fst <- v; v)
   else p.Fst
;;
```

Les valeurs indéfinies

Une amélioration importante peut être faite : il est possible que dans le calcul de la valeur du glaçon cette valeur soit elle-même nécessaire, comme dans :

```
let rec x=x in x;;
```

où la variable `x` se trouve dans le second élément de l'environnement :

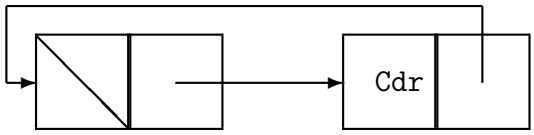


Figure 1 : l'environnement du glaçon de `x` dans `let rec x=x in x ; ;`

Sans précaution, le dégel boucle jusqu'au débordement de pile, or cette situation a un sens ! En effet la récursion correspond en sémantique dénotationnelle au plus petit point fixe, ce qui se calcule dans un domaine avec la valeur indéfinie $\perp$ et une topologie adéquate en prenant pour une fonction f la borne supérieure des itérés $f^i(\perp)$. Il est tout à fait possible que ce point fixe soit $\perp$, et donc, à une définition récursive qui donne un calcul qui ne termine pas, il faut associer une valeur indéfinie.

En CAML cette valeur indéfinie est représentée par l'exception `bottom`, le pseudo-code de l'instruction `Car` devenant :

```
(* l'exception des valeurs indéfinies *)

exception bottom;;

(* le type des paires modifiables *)

type ('a, 'b) mpair = {mutable Fst : 'a; mutable Snd : 'b };;

(* le type des glaçons *)

type thunk = {mutable Code : code; Env : env};;

(* Car avec dégel *)

let car_unf p =
  (* est-ce un glaçon ? *)
  if is_thunk (p.Fst)
   then
     (* on le dégele *)
     (let code = p.Fst.Code in
        (* le prochain dégel lèvera bottom *)
```

```
        p.Fst.Code <- code_of(raise bottom);
        (let v = eval(p.Fst.Code, p.Fst.Env) in
            p.Fst <- v; v))
   else p.Fst
;;
```

Ainsi un second dégel provoque la levée de l'exception `bottom`, et dans :

```
let rec x=x in x;;
```

le système réussit à calculer le point fixe de l'identité ... Nous avons rencontré ce phénomène de façon assez inattendue, la fonction classique `map` :

```
let map f = map_f
  where rec map_f = fun
     []     -> []
   | (a::l) -> (f a::map_f l)
;;
```

ayant été optimisée pour les singletons en :

```
let map' f = map_f'
  where rec map_f' = fun
     []     -> []
   | [a]    -> [f a]
   | (a::l) -> (f a::map_f' l)
;;
```

des définitions comme celles des nombres naturels produisaient une valeur indéfinie :

```
let rec Nat = 0::map' succ Nat;;
```

En effet l'interdépendance des éléments de `Nat` est très simple : l'élément d'indice N se calcule à partir de l'élément d'indice $N-1$. Or pour différencier le cas du singleton `[a] -> [f a]` du cas général `(a::l)` il faut savoir s'il existe un élément suivant, donc déterminer l'élément d'indice $N+1$ alors que l'élément d'indice N qui est indispensable pour $N+1$ est en cours de calcul, donc l'élément d'indice N est indéfini.

Les cas de valeurs récursives indéfinies sont très importants pour la version parallèle, car c'est tout simplement la situation connue sous le nom

d'étreinte fatale (ou plutôt, d'après le terme anglais, *deadlock*). Les *deadlocks* ont donc une signification, et nous savons enfin ce qu'il faut faire lorsqu'un *deadlock* est détecté. Rien que cela justifie entièrement l'approche par la sémantique, surtout lorsqu'il existe une littérature très abondante sur ces problèmes d'étreintes fatales.

Les exceptions

La sémantique sous forme de combinateurs catégoriques de CAML est une sémantique sans continuations, et il faudrait une sémantique avec continuations, le contrôle prenant une forme d'arbre plutôt qu'une forme de pile, pour que le traitement des exceptions soit parfait.

Mais les exceptions peuvent quand même en partie être modélisées de façon satisfaisante pour notre propos. Considérons les valeurs de CAML comme l'union des valeurs proprement dites et des valeurs exceptionnelles (comme c'est quasiment le cas dans les dernières versions). Seules les fonctions d'échappement peuvent manipuler les valeurs exceptionnelles, en les filtrant ou en renvoyant une valeur exceptionnelle.

Donc quand le calcul d'un glaçon rend une valeur exceptionnelle, le glaçon doit être mis à jour avec cette valeur. Comme c'est impossible à réaliser directement, il faut remplacer le glaçon dégelé par un glaçon levant l'exception :

```
(* le type des glaçons *)

type thunk = {mutable Code : code; mutable Env : env};;

(* Car avec dégel *)

let car_unf p =
  if is_thunk (p.Fst)
   then
     (let code = p.Fst.Code in
        p.Fst.Code <- code_of(raise bottom);
        (* protection de l'évaluation *)
        try let v = eval(p.Fst.Code, p.Fst.Env) in
              p.Fst <- v; v
         with ve ->
              (* valeur exceptionnelle *)
              p.Fst.Code <- code_of(raise);
              p.Fst.Env  <- ve;
```

```
                raise ve)
    else p.Fst
 ;;
```

La construction `try` n'est pas parfaitement traitée à cause de sa nature dynamique, mais les exceptions dans un cadre paresseux restent utilisables pour des usages comme le traitement des erreurs (par exemple la division par 0).

Beaucoup de langages fonctionnels éliminent les exceptions pour des arguments de "pureté", or les mécanismes de contrôle qu'elles permettent sont très utiles.

Par exemple soit `f` une fonction transformant une structure dans une structure du même type, et qui fréquemment renvoie son argument sans modification. Il est possible de faire le maximum de partage entre l'argument et le résultat en utilisant une exception :

```
exception Identity;;

let share f x = try f x with Identity -> x;;
```

et en modifiant `f` pour lever l'exception `Identity` quand l'argument n'est pas modifié. Cette propriété peut être étendue aux paires et aux listes en donnant des versions spéciales de `dist_pair` et de `map` :

```
 let dist_pair f (x,y) = (f x,f y);;

 let map f = map_f
   where rec map_f = fun
      []     -> []
    | (a::l) -> (f a::map_f l)
 ;;
```

en utilisant cette exception :

```
 let pair_share f (x,y) =
   try (f x, share f y)
    with Identity -> (x,f y)
 ;;

 let map_share f = map_share_f
   where rec map_share_f = fun
```

```
      []     -> raise Identity
    | (x::l) ->
        try (f x::share map_share_f l)
         with Identity -> x::map_share_f l
;;
```

Mais il est clair que cette technique ne convient pas aux structures paresseuses, ce qui ne doit pas être un prétexte pour l'interdire.

Les interruptions

Certains événements sont assez proches des exceptions : les interruptions, comme l'interruption clavier, le déclenchement du ramasse-miette, etc ... Contrairement aux exceptions qui dépilent, les interruptions sauvent l'état courant, sont traitées, et le calcul peut être repris à l'endroit où il a été interrompu. En plus, leurs causes peuvent être externes au programme, et donc se produire n'importe où dans celui-ci.

Deux méthodes peuvent être employées pour récupérer les interruptions :

- l'interruption est traitée immédiatement. Mais alors il faut prendre beaucoup de précautions pour assurer la reprise du calcul dans tous les cas. Les contraintes sur le code sont importantes, par exemple l'accès au troisième caractère dans une chaîne dans le système Le-Lisp se fait par les instructions pour NS32x32 :

  ```
  movd    0(Rstring),R0
  movzbd  3+8(R0),Rvalue
  ```

  Si une interruption se produit entre les deux instructions, et que pendant son traitement le ramasseur de miettes est appelé et déplace la chaîne, alors le contenu du registre `R0` est invalide.

- l'autre technique est que la fonction de traitement système se contente de positionner un drapeau qui sera régulièrement testé (au moins une fois par boucle pour assurer la prise en compte au bout d'un temps fini). Ces tests sont à rapprocher de ceux nécessaires pour éviter le débordement de pile.

  Il faut donc modifier le générateur de code (ou l'interprète de *byte-code*), mais cette technique est moins spécifique et convient bien aux multi-processeurs (avec un drapeau partagé protégé par des sections critiques).

Dans le cas de CAML le *break* peut être pris en compte dans le dégel des glaçons, par exemple pour effacer les calculs en cours (ce qui permet d'obtenir plus tard le même résultat si le programme est sans effet de bord).

Nous pouvons donc enfin donner le pseudo-code CAML de la routine `car_unf` définitive :

```
(* Car avec dégel *)

let car_unf p =
  (* est-ce un glaçon ? *)
  if is_thunk (p.Fst)
   then
     (* c'est un glaçon : on le dégèle *)
     (let code = p.Fst.Code in
        (* le prochain dégel lèvera bottom *)
        p.Fst.Code <- code_of(raise bottom);
        (* protection de l'évaluation *)
        try let v = eval(p.Fst.Code, p.Fst.Env) in
              p.Fst <- v; v
         with
            break ->
              (* break a été levé : on défait *)
              p.Fst.Code <- code;
              raise break
          | ve ->
              (* valeur exceptionnelle *)
              p.Fst.Code <- code_of(raise);
              p.Fst.Env  <- ve;
              raise ve)
   (* cas ordinaire *)
   else p.Fst
;;
```

En fait le code du glaçon, le bloc d'échappement et la fonction qui réalise la mise à jour sont placés dans la pile. Dans les cas simples le lancement et le retour du dégel d'un glaçon se font une vingtaine d'instructions LLM3.

Chapitre 7

Parallélisme

Si la compilation du système parallèle est extrêmement proche de celle du système paresseux, la réalisation du noyau est plus compliquée. Comme elle constitue la partie essentielle du système PCAML, nous allons détailler les algorithmes ainsi que des améliorations possibles.

La glacière

L'unique différence entre les glaçons spéculatifs et les glaçons paresseux se situe dans la routine de création, car les glaçons spéculatifs sont placés dans une "glacière" où les processus disponibles peuvent les trouver.

Dans le système CAML, il n'y a qu'un seul pointeur vers un glaçon donné, situé dans un champ de la structure "père" qui est écrasé après le dégel. Il est possible :

- d'ajouter avec un glaçon son "père" et le type d'accès associé de façon à faire la mise à jour après le dégel (car c'est le père qui sera physiquement modifié, et donc il faut connaître son adresse).
- de ne mettre que le glaçon spéculatif dans la "glacière". Alors le processus qui prend en charge son dégel ne termine pas par la mise à jour. Il se contente de remplacer le code par un "retour immédiat" et l'environnement par la valeur. Le glaçon se présente comme s'il n'était pas dégelé, mais le calcul est déjà fait. Si ensuite le glaçon est accédé de façon normale, la mise à jour sera effectuée immédiatement.

La glacière doit être protégée par des sections critiques pour l'insertion et le retrait des glaçons spéculatifs. Avant de commencer le dégel d'un glaçon

qu'il vient de retirer dans la glacière, un processus doit s'assurer qu'il n'est pas déjà calculé (donc en vérifiant le pointeur du père, ou en contrôlant que le code se soit pas une routine particulière comme le retour immédiat).

La gestion et l'organisation même de la glacière sont assez libres :

- la taille peut être bornée, ce qui permet de limiter les risques d'explosion du nombre de glaçons spéculatifs, qui sont alors dégradés en glaçons ordinaires (qui ne seront dégelés qu'à la demande) ou bien évalués directement (mais alors il faut qu'une analyse à la compilation ou des annotations l'autorisent, car cette optimisation consiste à transformer un appel par spéculation ou nécessité en appel par valeur).
- il peut y avoir plusieurs priorités, les plus prioritaires étant évalués en premier.
- un glaçon peut être ajouté en tête ou en queue, et donc la glacière être gérée comme une queue ou une pile, selon les caractéristiques de tel ou tel algorithme.
- la glacière peut être découpée en plusieurs parties correspondant chacune à un processeur ou à un groupe de processeurs, afin d'améliorer la localité (c'est utile avec certains types de caches même avec une mémoire globale).

Les tâches

Le calcul des glaçons nécessite des ressources privées pour l'évaluation (principalement une pile) qui associées à un flot de contrôle forment une tâche. Plusieurs stratégies sont utilisables :

- ne pas employer de pile, et gérer le contrôle en allouant dynamiquement une structure en forme d'arbre. C'est le cas pour certains systèmes utilisant les continuations, comme la dernière version du Standard ML du New Jersey [4, Appel]. C'est une solution quasiment idéale quand elle est applicable.
- allouer la pile par petits morceaux, mais alors il faut que le ramasseur de miettes puissent les récupérer. Un simple schéma semblable du compteur de référence sur un bit permet de récupérer au fur et à mesure l'espace même avec des manipulations de continuations dans le langage [21, Danvy]. C'est la méthode favorite des systèmes Scheme.

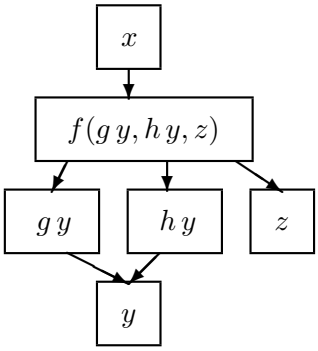
- allouer statiquement un certain nombre de piles. Beaucoup d'espace (en mémoire virtuelle heureusement) est gaspillé, et le nombre total de tâches actives ou bloquées est limité, ce qui peut être considéré comme une bonne chose (car toutes les ressources du système ne risquent pas d'être consommées pour un travail spéculatif) ou une mauvaise (car dans certains cas la plupart des processeurs risquent de ne pas trouver de tâches actives, et donc d'attendre qu'on veuille bien leur donner du travail à faire). Pour des raisons techniques liées à l'implantation actuelle de CAML, c'est la solution qui a été retenue.

Les tâches peuvent être actives et donc affectées à un processeur ou bien inactives, c'est-à-dire bloquées en attente d'une valeur ou d'un processeur libre, ou inutilisées. En plus, comme nous allons le voir, une tâche peut être prioritaire (nous dirons avoir le témoin) ou non. Une tâche peut dégeler au même instant plusieurs glaçons, et il faut donc distinguer les ressources attachées à un glaçon ou à une tâche.

Le témoin

Le calcul des glaçons peut être modélisé par le graphe de dépendance dont les nœuds sont les glaçons, les arcs symbolisent la relation de dépendance "pour obtenir la valeur de la source, il faut d'abord dégeler le but" et la racine n'étant autre que le programme à calculer (qu'on peut toujours pour nos besoins mettre dans un glaçon).

Par exemple le graphe de `x = f (g y, h y, z)` est :



Les parties non connexes avec la racine sont des calculs qui ne participent en rien au résultat final, les boucles correspondent à des valeurs indéfinies comme nous l'avons vu au cours du chapitre précédent (il existe d'autres sortes de valeurs indéfinies par exemple les branches infinies de l'arbre de

dépendance, sans parler des boucles en mode strict ...).

Le calcul normal (dans le sens commun et dans le sens "dans l'ordre normal") est le parcours en profondeur d'abord, que nous décrirons à l'aide d'un programme CAML :

```
(* le type des arbres binaires *)

type tree = Leaf of label | Node of node
and  node = {Left : tree; Label : label; Right : tree}
;;

(* la traversée en profondeur d'abord *)

let rec f = fun
   (Leaf lab) -> fl lab
 | (Node {Left=left; Label=lab; Right=right}) ->
     f left; f right; fl lab
;;
```

Dans ce modèle, chaque tâche est un évaluateur qui parcourt l'arbre en parallèle avec les autres tâches. Les points de départ secondaires sont les glaçons spéculatifs. A un moment donné une et une seule tâche doit être en train d'effectuer le calcul inachevé le plus à gauche du graphe (comme pour l'ordre normal). Comme dans une course de relais, cette tâche possède le "témoin" et est la seule importante, les autres accélérant le calcul mais ne participant pas à cet instant directement à l'élaboration du résultat final.

Pour conserver la sémantique de l'appel par nécessité, cette tâche associée à l'évaluateur principal doit avoir un statut privilégié : par exemple en lui garantissant d'être toujours active, la propriété de terminaison est maintenue.

Les queues

Une queue de tâches est associée à chaque glaçon en cours de dégel. Quand une tâche rencontre une valeur en cours de dégel, elle est ajoutée dans la queue et prend l'état bloqué. Si cette tâche a le témoin, elle le passe à la tâche qui travaille sur ce glaçon, et elle est notée comme devant reprendre le témoin quand la valeur sera calculée.

Le passage du témoin se produit donc de deux façons différentes :

1. lorsque la tâche principale se bloque sur un glaçon tiède

2. à la fin du dégel d'un glaçon ayant bloqué la tâche principale.

Après le calcul d'un glaçon, la mise à jour est effectuée et les tâches en attente dans la queue prennent l'état disponible (*runnable*). Si la tâche T qui vient de finir le calcul a le témoin et qu'une tâche de la queue est marquée comme devant le reprendre, alors la tâche T prend l'état disponible et lui donne le témoin et le processeur.

Le cas de valeurs indéfinies peut être considéré comme exceptionnel, et nous nous limitons uniquement à la détection pour le calcul principal.

La structure décrivant un glaçon comprend les champs :

- le code à exécuter pour calculer la valeur
- l'environnement
- la tâche qui effectue le dégel
- un drapeau notant la présence d'une tâche prioritaire dans la queue
- la queue (seule la tête de la queue est dans cette structure, la queue étant chaînée sans allocations dynamiques par les tâches)

Nous pouvons maintenant donner le pseudo-code. Les différentes déclarations de type des objets manipulés sont :

```
(* le type des paires modifiables *)

type ('a, 'b) mpair = {mutable Fst : 'a; mutable Snd : 'b };;

(* le type des actions à lancer *)
(* éléments de la glacière *)
(* Code est car_unf, cdr_unf, ... *)
(* Father est le père du glaçon *)

type work = {Code : code; Father : product};;

(* le type des tâches *)

type task  = { Stack : stack;
               mutable State : state;
               mutable Queue : queue;
               mutable Pri : bool}
(* le type des états des tâches *)
and  state = Unused | Blocked of thunk | Runnable | Running
```

```
(* le type des queues *)
and  queue = Empty | Next of task
;;

(* le type des glaçons *)

type thunk = {mutable Code : code;
              mutable Env : env;
              mutable InProgress : wtask;
              mutable PriInQ : bool;
              mutable Queue : queue}
and  wtask = No | Task of task
;;
```

La gestion élémentaire de la glacière `future_pool` se fait par :

```
(* la queue des tâches disponibles *)

let runnable_task_queue = ref Empty;;

(* la glacière *)

let future_pool = ref ([] : work list);;
```

avec comme fonctions de création des glaçons ordinaires et spéculatifs :

```
let mk_thunk (code, env) =
  {Code=code; Env=env; InProgress=No; PriInQ=false; Queue=Empty}
;;

let mk_future (code, env) =
    LOCK(future_pool);
    future_pool := mk_thunk(code, env)::!future_pool;
    UNLOCK(future_pool)
;;
```

Les processeurs libres attendent une tâche à reprendre ou un glaçon spéculatif à dégeler. Un protocole est utilisé pour prendre les verrous afin de réduire la taille des sections critiques.

```
let process_loop () =
 (* boucle principale *)
 while true do
```

```
    (* recherche d'une tâche disponible *)
    if !runnable_task_queue <> Empty
     then
       (LOCK(runnable_task_queue);
        if !runnable_task_queue = Empty
         then UNLOCK(runnable_task_queue) (* course perdue *)
         else
           (let (Next first_task) = !runnable_task_queue in
               (* relance de la première tâche *)
              runnable_task_queue := first_task.Queue;
              UNLOCK(runnable_task_queue);
              first_task.Queue <- Empty;
              first_task.State <- Running;
              resume(first_task)))
     (* recherche d'un glaçon spéculatif *)
     else if !future_pool <> []
           then
            (LOCK(future_pool);
             if !future_pool = []
              then UNLOCK(future_pool) (* course perdue *)
              else
                (let ({Code=code; Father=father}::rest) =
                     !future_pool in
                   (* dégel le premier glaçon *)
                   future_pool := rest;
                   UNLOCK(future_pool);
                   eval_and_update(code,father)))
;;
```

Notre exemple de fonction d'accès `car_unf` doit gérer la queue à la fois à l'appel si le glaçon est tiède (c'est-à-dire en cours d'évaluation), et au retour pour relancer d'éventuelles tâches bloquées.

```
(* Car avec dégel *)

let car_unf p =
 let fst = p.Fst in
  (* est-ce un glaçon ? *)
  if not (is_thunk(fst))
   (* cas ordinaire *)
```

```
    then fst
    else
      (* cas du glaçon *)
      (LOCK(fst);
       if not (is_thunk(p.Fst)) (* course perdue *)
        then (UNLOCK(fst); p.Fst)
        else (* verrou pris *)
             if fst.InProgress = No
              then
                (* glaçon froid *)
                (let curtask = current_task() in
                   (* c'est notre glaçon *)
                   fst.InProgress <- Task curtask;
                   UNLOCK(fst);
                   (* évaluation *)
                   let v = eval(fst.Code, fst.Env) in
                     LOCK(fst);
                     (* mise à jour *)
                     p.Fst <- v;
                     (* avons-nous bloqué big brother *)
                     if fst.PriInQ
                      then
                        (suspend curtask;
                        (* la continuation est 'retry car_unf' *)

                         switch_with_pri_task_in_queue (curtask,fst));
                     LOCK(runnable_task_queue);
                     (* réveillons les autres *)
                     signal_waiting(fst.Queue);
                     UNLOCK(runnable_task_queue);
                     UNLOCK(fst);
                     v)
              (* glaçon tiède *)
              else enqueue_on fst)
;;
```

La première fonction auxiliaire ajoute la tâche courante dans la queue d'un glaçon tiède en transmettant le témoin s'il le faut :

```
(* met la tâche courante sur la queue du glaçon *)

let enqueue_on thunk =
  let curtask = current_task()
  and (Task wrktask) = thunk.InProgress in
    curtask.Queue <- thunk.Queue;
    thunk.Queue <- Next curtask;
    curtask.State <- Blocked thunk;
    suspend curtask; (* la continuation est 'retry cxr_unf' *)
    (* donne le témoin si on l'a *)
    if curtask.Pri then (thunk.PriInQ <- true; promote wrktask);
    UNLOCK(thunk);
  return_to_process_loop()
 ;;
```

Le témoin est passé à la tâche qui dégèle le glaçon qui bloque, et si cette tâche est elle-même bloquée sur un autre glaçon, le témoin est transmis récursivement jusqu'à une tâche active :

```
(* donne récursivement le témoin *)

let rec promote task =
  if task.Pri then (* cas de bottom *) ...;
  task.Pri <- true;
  match task.State with
     Running -> () (* fait *)
   | Runnable -> switch task (* donne le processeur *)
   | Blocked thunk ->
       (* transmet le témoin à la tâche bloquante *)
       if thunk.PriInQ then (* autre cas de bottom *) ...;
       LOCK(thunk);
       if task.State <> Blocked thunk
        then
          (UNLOCK(thunk); promote task)
        else
          (thunk.PriInQ <- true;
           let (Task task') = thunk.InProgress in
             promote task';
             UNLOCK(thunk))
;;
```

Au retour du dégel, une tâche prioritaire bloquée reprend le témoin et hérite du processeur :

```
(* donne tout à big brother (la tâche prioritaire) *)


let switch_with_pri_task_in_queue (curtask,thunk) =
 let rec scan_queue (Next task') =
   if task'.Pri then task' else scan_queue task'.Queue in
  let pritask = scan_queue thunk.Queue in
    pritask.Pri <- false;
    exchange_stack (pritask, curtask);
    simulate_return curtask
;;
```

Enfin les tâches ordinaires bloquées prennent l'état "en attente d'un processeur libre" :

```
(* marque les tâches bloquées comme disponibles *)


let rec signal_waiting = fun
    Empty -> () (* fait *)
  | (Next task) ->
       signal_waiting task.Queue;
       task.State <- Runnable;
       task.Queue <- !runnable_task_queue;
       runnable_task_queue := Next task
;;
```

Le traitement des exceptions et des interruptions se fait en ajoutant un code du même genre que pour le système paresseux que nous omettons ici. S'il est indispensable de conserver la même sémantique que pour l'appel par nécessité aussi avec des effets de bord (et donc pour la totalité du système), il suffit d'ajouter une primitive qui bloque la tâche qui l'exécute tant qu'elle n'a pas le témoin.

Il est intéressant de fournir à l'utilisateur une primitive `purge` qui vide la glacière, arrête toutes les tâches et rend aux glaçons en cours de dégel leurs formes primitives (c'est-à-dire efface les calculs en cours). Cette primitive est associée à l'interruption clavier pour permettre à l'utilisateur interactif de "reprendre la main" dans de bonnes conditions.

Le gain dû au parallélisme

L'existence et l'unicité d'une seule tâche prioritaire active (celle qui a le témoin) est une conséquence du caractère séquentiel du système. Une fonction du type ou-parallèle produirait simultanément deux processus équivalents quant à l'obtention du résultat.

L'écriture du noyau est rendue plus simple, et la source des gains que peut procurer le parallélisme apparaît clairement : les tâches qui ont le témoin effectuent exactement le même calcul que dans un système paresseux, mais quand certains glaçons deviennent nécessaires, une autre tâche a déjà commencé ou mieux fini le dégel. Si le surcoût du parallélisme pouvait être négligé, alors au pire (quand tout le travail spéculatif des autres tâches ne sert pas) le gain serait nul.

L'implantation doit favoriser au maximum les tâches prioritaires : si une optimisation permet d'enlever quelques instructions pour le cas prioritaire, alors elle doit être préférée. Il est inutilement coûteux de répartir les tâches non prioritaires sur les processeurs selon une stratégie avec préemption, car la trace de la tâche qui calcule le résultat est conservée à chaque instant.

Notre système est à notre connaissance le seul à appliquer cette technique qui pourtant est la conséquence naturelle de la filiation entre l'appel par nécessité et l'appel par spéculation. Nous assurons ainsi la conservation de la propriété de terminaison sans avoir besoin de recourir à des arguments de nature statistique du genre : "l'ordonnancement étant équitable, la tâche principale ne peut pas rester indéfiniment bloquée par manque de processeurs libres".

Les optimisations par analyse à la compilation

Le modèle du graphe de dépendance permet de dégager les analyses à la compilation qui permettent d'améliorer les performances. Par exemple :

- souvent la valeur d'un glaçon est utilisée plusieurs fois dans une même tâche, alors il est possible de n'effectuer le test pour le dégel que la première fois.

- les glaçons spéculatifs qui ne servent pas (et donc qui ne sont pas dans la composante connexe de la racine) peuvent être dégradés en glaçons paresseux et ne pas être placés dans la glacière.

- par contre, les glaçons spéculatifs qui vont servir doivent être dégelés le plus tôt possible et placés en conséquent dans la glacière.
- les glaçons dont il est prouvé que leurs valeurs sont nécessaires (précaution indispensable pour ne pas modifier le comportement du programme) peuvent être remplacés par le code en appel par valeur correspondant.

L'analyse du caractère strict (*strictness analysis*) permet justement de réaliser ces deux optimisations, mais il ne faut pas oublier que la suppression d'un appel par spéculation enlève du parallélisme.

Malheureusement la plupart de ces analyses sont difficiles à faire et doivent être réservées comme options pour la compilation non interactive de fichiers. Mais elles ont généralement aussi un intérêt pour les systèmes paresseux, et les efforts pour les développer dans un cadre paresseux profitent immédiatement à un système parallèle.

La granularité

Le problème principal du parallélisme est la taille des tâches. En effet il faut avoir suffisamment de tâches pour occuper tous les processeurs, et donc il ne faut pas un trop petit nombre de grosses tâches. Et au contraire si les tâches sont trop petites, alors leur nombre est beaucoup plus grand que le nombre de processeurs et toute la puissance de calcul disponible sera absorbée par la gestion des tâches.

Notre système ne permet pas de résoudre dans sa généralité le problème, mais plusieurs sortes de parallélismes (sur les données ou sur le contrôle par exemple) peuvent être directement exploitées. La sémantique très simple du système autorise des transformations de programmes dans de bonnes conditions sans risque de changer de façon inattendue le comportement. Il est même raisonnable de vouloir méta-compiler des constructions parallèles introduites dans le cadre de problèmes particuliers. Cette souplesse ainsi que la facilité de mise au point permet l'adaptation à ce système de programmes déjà existants et donc écrits sans souci d'une éventuelle parallélisation.

Chapitre 8

Programmation du parallélisme

La description des différentes constructions contrôlant le parallélisme est volontairement assez brève, car premièrement elles pourrait être qualifiées facilement de “sucre syntaxique”, et deuxièmement nous nous sommes attachés à suivre les syntaxes proposées dans les versions successives de CAML pour l'évaluation paresseuse.

Nous présentons ensuite sur un exemple la mise en œuvre du parallélisme dans le style de la dernière version de CAML.

Les anciens CAML paresseux

Les anciennes versions de CAML disposent de deux systèmes :

1. un système classique en appel par valeur qui est le système distribué.
2. un système dit “mixte” expérimental.

Ce système permet l'appel par valeur et l'appel par nécessité à l'aide de deux compilateurs, l'un produisant une évaluation ordinaire stricte, l'autre une évaluation paresseuse sur les appels de fonctions (y compris les `let` que nous considérons ici comme des cas particuliers de fonctions anonymes) et les structures mettant en jeu un produit cartésien.

Les mots clés `strict` et `lazy` préfixent une expression et sélectionnent le mode de compilation à utiliser, et des directives autorisent le choix d'un mode par défaut.

Par exemple la fonction produisant la paire de suspensions quotient/reste, les glaçons étant imprimés sous forme d'étoiles * ou d'ellipse ... :

```
let quorem q = function x -> lazy (quotient x, remainder x)
  where quotient x = strict x quo q (* primitives strictes *)
  and remainder x = strict x mod q
;;
Value quorem = - : (num -> num -> num & num)

quorem 12 148;;
(*,*) : (num & num)

fst it;;
12 : num
```

Le CAML parallèle

Ces constructions ad hoc donnent un contrôle très fin sur le mode de compilation utilisé pour chaque expression, et sont en fait équivalentes à des annotations.

Nous avons ajouté un troisième mot clé `parallel` pour le mode d'appel par spéculation comme nous l'avons suggéré. Une grande partie des constructions `parallel` ont une portée limitée par des `strict` ou des `lazy` dans les sous-expressions. Deux constructions supplémentaires `freeze` et `future` ont été introduites : le mode de compilation n'est changé que pour le premier niveau de l'expression argument.

De plus, ces constructions autorisent des formes `let` avec une partie des liaisons réalisées avec un mode d'appel différent du mode courant.

Un nouveau module travaillant sur la syntaxe profonde de CAML vérifie la cohérence des constructions paresseuses et parallèles, expanse les `freeze` et les `future`, et procède à une *strictness analysis* élémentaire. Nous présentons dans le prochain chapitre des exemples d'utilisation de ces constructions.

Comme dans le cas général il est impossible de déterminer les modes utilisés pour les appels à la définition des fonctions (tout simplement parce que les appels peuvent être dans une autre phrase), les arguments sont systématiquement considérés comme pouvant être des glaçons, et donc seules les fonctions paresseuses d'accès des produits cartésiens sont utilisables. Ces systèmes sont donc principalement des systèmes paresseux ou parallèles, ce qui se ressent sur les performances.

Les nouvelles versions de CAML

La version paresseuse que nous venons de présenter n'est pas assez efficace pour être choisi comme version de base de CAML, et donc sa maintenance est devenu trop coûteuse.

Pour éviter de proposer deux systèmes assez différents, la dernière (V2–6) version de CAML propose en standard des possibilités d'évaluation paresseuse en annotant les définitions de types (en suivant la même syntaxe que pour les objets modifiables qui sont une généralisation des références).

Ainsi le mot clé `lazy` peut apparaître :

- pour un constructeur d'un type concret. En fait la fonction associée au constructeur qui produit des valeurs est appelée par nécessité. Le type "paresseux" le plus simple, celui des valeurs "gelées" se définit donc par :

  ```
  type 'a frozen = lazy Freeze of 'a;;
  ```

  La construction d'une valeur gelée se fait en appliquant la fonction de construction

  ```
  Freeze : 'a -> 'a frozen
  ```

  et le dégel peut être forcé simplement en accédant à la valeur comme dans :

  ```
  let Unfreeze = fun (Freeze x) -> x;;
  ```

- l'autre possibilité est d'annoter une étiquette d'un type record. Par exemple les paires paresseuses (sur les deux éléments) et les flots peuvent être définis par :

  ```
  type ('a, 'b) lpair = {lazy Fst : 'a; lazy Snd : 'b };;

  type 'a stream = {lazy Hd : 'a; lazy Tl : 'a stream };;
  ```

Une version parallèle se déduit simplement (du moins pour l'aspect extérieur) des constructions paresseuses en introduisant le mot clé `parallel` qui s'emploie exactement comme `lazy` avec une signification un peu différente. Ainsi le type des glaçons spéculatifs est :

```
type 'a future = parallel Future of 'a;;
```

Mais le CAML parallèle disponible à l'heure où sont écrites ces lignes est dérivé d'une ancienne version (V2–5), et donc les exemples que nous donnerons dans le chapitre suivant suivront la vieille syntaxe.

Il est possible de retrouver l'appel par nécessité classique en transformant les programmes par l'insertion de `Freeze` et `Unfreeze`. Par exemple :

```
let d x y = x*(x + y) in d 5 6;;
```

se transforme en :

```
let d x y = Unfreeze x * (Unfreeze x + Unfreeze y)
  in d (Freeze 5) (Freeze 6)
;;
```

L'argument n'est pas spécieux car il est très facile d'écrire une transformation sur la syntaxe de CAML grâce à l'appel par filtrage et à l'interface avec l'analyseur syntaxique. Les deux aspects de l'évaluation paresseuse :

1. l'évaluation retardée des arguments des appels de fonctions
2. les types de données paresseux

sont dépendants l'un de l'autre. Mais il faut disposer des deux pour que le système soit complet pour un langage fonctionnel comme CAML.

Une proposition

Une voie que nous proposons pour les prochaines versions de CAML est d'annoter les définitions de fonctions (alors que dans les anciennes versions ceux sont les appels qui peuvent être annotés). Déjà, l'appel des primitives est un cas traité de manière spécifique pour les compilateurs, et certaines optimisations proposées utilisent une analyse de la partie fonction des appels.

Ainsi la fonction `IF` paresseuse sur ses second et dernier arguments s'écrirait :

```
let IF test (lazy vrai) (lazy faux) =
 if test then vrai else faux
;;
```

au lieu de :

```
let IF test vrai faux =
 (fun (Freeze v) -> v) (if test then vrai else faux)
;;
```

en n'oubliant pas de remplacer tous les `IF t v f` par :

```
IF t (Freeze v) (Freeze f)
```

Cette technique est plus efficace et sûrement plus naturelle que l'introduction systématique de constructeurs paresseux ou parallèles dans les appels de fonctions. De plus elle ne reproduit pas l'inconvénient des anciennes versions, car seuls les arguments de certaines fonctions ou de certains constructeurs reconnaissables pendant la compilation peuvent être des glaçons.

Un exemple détaillé

Prenons comme exemple de programme la vérification d'un cas particulier du théorème de Ramsey. Considérons N points joints deux à deux par des arcs, chaque arc étant soit de couleur bleu, soit de couleur rouge. Par exemple :

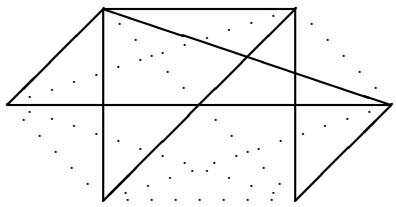


Figure 1 : le problème de Ramsey.

Ramsey a démontré que s'il y a 6 points ou plus, alors il existe trois points reliés par des arcs de même couleur (donc soit un triangle bleu, soit un triangle rouge). Nous nous proposons de vérifier cette proposition.

6 points sont joints par $\sum_{i=1}^{5} i = 15$ arcs qui peuvent former $C_6 3 = 20$ triangles. Le problème se résout donc en la recherche d'un des 20 triangles d'une des deux couleurs dans chacune des $2^{15} = 32768$ configurations possibles.

Pour obtenir des performances raisonnables, nous codons chaque configuration dans un petit entier, un arc étant associé à un élément binaire. L'énumération naturelle des configurations commence avec tous les arcs d'une

couleur pour finir avec l'ensemble des arcs de l'autre couleur : il faut donc grouper pour une configuration et un triangle le test tout bleu et tout rouge (c'est-à-dire les trois éléments binaires à 0 ou à un 1). De plus le problème est symétrique : chacune des deux couleurs joue le même rôle, et nous pouvons nous limiter aux configurations avec la couleur d'un des arcs fixée arbitrairement à 0.

Nous pouvons maintenant présenter quelques fonctions auxiliaires. La fonction `n_sets` produit la liste des ensembles à n éléments compris entre 1 et N :

```
let rec n_sets N = function
   0 -> [[]]
 | n -> if n > N then [] else
        n_sets (N-1) n @ map (cons N) (n_sets (N-1) (n-1))
;;
```

Ensuite la fonction `pairs` produit les paires (représentant les arcs) associées :

```
let rec pairs = function
   [] -> []
 | [_] -> []
 | x::l -> let compound accu y = (y,x)::accu in
             it_list compound (pairs l) l
;;

let pN = pairs [6; 5; 4; 3; 2; 1];;
```

Puis il faut traduire ces paires dans un masque d'éléments binaires :

```
let mask =
  it_list (fun accu p ->
             lor (accu, lshift (1,pred (index p pN))))
          0
;;
```

Le test élémentaire est fait par `test` :

```
let mono m n =
 eq (land (m,n), m) or eq (land (m,n), 0)
;;

let test x n_set = mono (mask (pairs n_set)) x;;
```

Le problème est donc maintenant décomposé en 16384 fois 20 tests. Proposons une solution avec un éventuel processus pour chaque configuration utilisant le dernier style de constructions parallèles. Pour chaque configuration il faut appeler une fonction `try_conf` qui vérifie l'existence d'un triangle d'une couleur. Pour gérer les 16384 appels, nous utilisons une liste parallèle de booléens :

```
type jobs = Done | Link of link
and link = {parallel ToDo: bool; parallel Rest: jobs}
;;
```

qui sera réduite par :

```
let rec do_jobs = fun
   Done -> true
 | (Link {ToDo=j;Rest=r}) -> j & do_jobs r
;;
```

La paire dans cette liste est parallèle pour le premier élément afin d'effectuer les calculs pour chaque configuration indépendamment, et aussi pour le second élément (le chaînage de la liste) dans le but de rendre un résultat sans attendre la fin de la construction de la liste si la fonction de réduction rencontre une valeur fausse.

Malheureusement les performances prouvent que le nombre de processus potentiels est trop élevé (16384 !) et leur taille trop petite. Il faut donc aborder le problème en le décomposant en un processus par triangle, soit en tout 20 processus éventuels.

A nouveau ces processus peuvent être organisés en une liste :

```
type jobs = Done | Link of link
and link = {parallel ToDo: num; Rest: jobs}
;;
```

construite par :

```
let jobs =
  let rec build = function
     [] -> Done
   | set::rest ->
       Link {ToDo=set_test set; Rest=build rest}
   in build (n_sets 6 3)
;;
```

et réduite par :

```
let rec reduce = function
   Done -> 0
 | Link {ToDo=n; Rest=rest} -> n+reduce rest
;;
```

Le résultat des tests pour un triangle n'est pas directement utilisable et la fonction `reduce` peut être utilisée par exemple pour recueillir des statistiques sur le déroulement du programme. En fait, le but principal de `reduce` est de détecter la terminaison des glaçons en utilisant leurs résultats.

A chaque configuration est associée un élément d'un vecteur de booléens qui est mis à vrai quand un triangle y est découvert :

```
let col = vector 16384 of false;;
```

La fonction de test est :

```
let set_test n_set =
  let x = ref 16384 in
    while !x<>0 do
      decr x;
      if not(col.(!x)) & test !x n_set
       then col.(!x) <- true
    done;
    !x
;;
```

Le résultat final est donné par la réduction de ce vecteur après la fin des calculs :

```
let ramsey () =
  reduce jobs;
  it_vect (fun a b -> a or b) false col
;;
```

La partie importante de ce programme est la fonction `set_test`. Si la configuration possède déjà un triangle, alors la fonction passe à la configuration suivante bien que le vecteur d'état des configurations soit géré sans section critique. La technique proposée est correcte car les modifications physiques de ce vecteur sont monotones, mais il est clair qu'un peu d'efficacité

est perdue si deux processus découvrent simultanément un triangle dans la même configuration.

Le gain en vitesse en fonction du nombre de processeurs est assez bon, mais s'il reste loin de l'idéal (gain linéaire) :

nombre de processeurs	1	2	4	6	8	10
temps d'exécution	60s	34s	21s	18s	15s	13s

Une comparaison avec d'autres parallélismes

La forme de parallélisme offert par notre système est très différente de celle de langages plus traditionnels comme ADA.

Le parallélisme peut être présent dans la partie "contrôle" d'un programme, ou au contraire suivre les structures de données. De même, les constructions parallèles sont fondamentalement implicites ou explicites, un simple sucre syntaxique permettant d'en adapter la présentation. Le caractère plutôt explicite de nos programmes est un trait du style employé, rendu obligatoire par des contraintes matériel, plutôt qu'une propriété du système. Le problème est le même que pour un système paresseux : il est possible d'utiliser par défaut l'appel par nécessité, mais généralement ce n'est pas considéré comme raisonnable.

Il reste certaines limites aux possibilités proposées, car les glaçons spéculatifs sont restreints comme les glaçons paresseux aux produits cartésiens. Par exemple, même si l'appel est "par nécessité", les conditionnelles comme les `if` n'évaluent pas en parallèle les deux branches. Il faut placer une conditionnelle dans une fonction faisant apparaître explicitement un produit comme pouvoir forcer un calcul concurrent du `then` et du `else`, et les résultats ne seront disponibles qu'après la fin des deux évaluations.

La transformation de l'appel par nécessité en appel par spéculation est sûre, sauf pour les effets de bord qui peuvent être classés en deux catégories :

- les affectations et autres modifications physiques. Comme l'ordre d'évaluation de l'évaluation paresseuse n'est pas vraiment immédiat, ce genre d'effets de bord n'est pas utilisé en pratique dans un cadre non-strict.

- les entrées/sorties. Généralement le programme peut être vu comme une "boite noire" qui prend un argument en entrée et rend un résultat en sortie. Ce type d'effets de bord peut être donc placé à l'extérieur

d'une telle "boite noire" et le parallélisme restreint à l'intérieur de la boite, le reste étant évaluée en mode strict.

En conclusion le but poursuivi n'est pas d'étendre la puissance d'expression du langage, mais bien d'augmenter la vitesse d'exécution des programmes, l'introduction du parallélisme se faisant avec le moins de modifications possible.

Chapitre 9

Premiers résultats

Nous avons réalisé un prototype à partir du système paresseux de la version 2.5 sur une machine Sequent Balance équipée de 10 micro-processeurs NS 32032 d'environ 0.75 Mips chacun et de 16 M octets de mémoire centrale, avec un Unix symétrique comme système d'exploitation.

La fonction de Fibonacci

La série de Fibonacci a été choisie comme exemple d'étude. La fonction CAML réalisant la série $u_{n+1} = u_n + u_{n-1}$ avec $u_1 = u_2 = 1$ est :

```
let rec fib = fun
    1 -> 1
  | 2 -> 1
  | n -> fib(n-1) + fib(n-2)
;;
```

L'opérateur `+ : (num * num) -> num` est strict et est compilé sans construire la paire argument, ce qui oblige à introduire une fonction `plus` appelant la primitive `+` et construisant explicitement la paire :

```
let plus = prefix +;;

directive infix "plus";;
```

La paire argument de type `num * num` peut maintenant avoir pour éléments des glaçons.

La version séquentielle stricte s'écrit donc :

```
let rec fibs = fun
    1 -> 1
  | 2 -> 1
  | n -> (fibs (n-1)) plus (fibs (n-2))
;;
```

Il est possible de paralléliser un ou les deux appels récursifs, ce qui donne `fib1` ou `fib2` dans le style de la version 2.5 avec par défaut le mode de compilation séquentiel :

```
let rec fib1 = fun
    1 -> 1
  | 2 -> 1
  | n -> (future fib1 (n-1)) plus (fib1 (n-2))
;;

let rec fib2 = fun
    1 -> 1
  | 2 -> 1
  | n -> (future fib2 (n-1)) plus (future fib2 (n-2))
;;
```

Mais ces fonctions présentent un parallélisme gigantesque qui ne convient pas, car elles produisent un grand nombre (de l'ordre de $fib(n)$) de tâches se réduisant à quelques instructions. Il faut donc assurer une taille raisonnable des tâches en introduisant un second paramètre qui sert à couper la production de nouvelles tâches :

```
let rec fib' = fun
    (1,_) -> 1
  | (2,_) -> 1
  | (n,0) -> fib n (* la fonction fib rapide *)
  | (n,p) ->
      (future fib' (n-1, p-1)) plus (future fib' (n-2, p-1))
;;
```

Les différentes versions de `fib` exécutées par les systèmes :

CAML : la version 2.5 standard avec seulement un seul mode de compilation possible : le mode strict

LCAML : le système 2.5 paresseux avec les deux modes strict et paresseux

PCAML : le système expérimental 2.5 parallèle avec les trois modes de compilation

donnent comme temps d'exécution :

Fibonacci (`fib 20`) en secondes		
CAML (strict, 1 CPU)	`fib 20`	0.52
CAML (strict, 1 CPU)	`fibs 20`	0.73
CAML (strict, 1 CPU)	`fib' (20,0)`	0.84
LCAML (paresseux, 1 CPU)	`fib 20`	1.21
LCAML (paresseux, 1 CPU)	`fibs 20`	1.83
LCAML (paresseux, 1 CPU)	`fib' (20,0)`	2.32
PCAML (parallèle, 1 CPU)	`fib 20`	1.21
PCAML (parallèle, 1 CPU)	`fibs 20`	1.83
PCAML (parallèle, 1 CPU)	`fib1 20`	3.97
PCAML (parallèle, 1 CPU)	`fib2 20`	2.72
PCAML (parallèle, 1 CPU)	`fib' (20,0)`	2.32
PCAML (parallèle, 2 CPU)	`fib1 20`	2.20
PCAML (parallèle, 2 CPU)	`fib2 20`	1.48
PCAML (parallèle, 2 CPU)	`fib' (20,3)`	0.45
PCAML (parallèle, 3 CPU)	`fib' (20,4)`	0.30
PCAML (parallèle, 4 CPU)	`fib1 20`	1.40
PCAML (parallèle, 4 CPU)	`fib2 20`	0.90
PCAML (parallèle, 4 CPU)	`fib' (20,4)`	0.25
PCAML (parallèle, 6 CPU)	`fib' (20,5)`	0.19
PCAML (parallèle, 8 CPU)	`fib' (20,5)`	0.15

L'introduction de la fonction `plus` et de l'argument supplémentaire grève lourdement les performances, mais c'est le prix à payer pour obtenir un niveau de parallélisme raisonnable.

La fonction `fib1` est très lente car l'appel sur `fib1 (n-2)` est souvent plus rapide que celui sur `future fib1 (n-1)`, ce qui rend séquentielle l'exécution. Il ne reste plus que le coût de la production et du dégel des glaçons.

La fonction `fib2` (la version parallèle naturelle) ne présente pas cet inconvénient, mais son parallélisme débridé ne permet pas un accroissement des performances linéaire en fonction du nombre des processeurs, en raison des problèmes de synchronisation d'un très grand nombre de tâches dans le noyau. En plus, cette version consomme énormément d'espace.

La variante `fib'` est très satisfaisante. Avec seulement deux processeurs, elle est déjà plus rapide que la version standard sur le système CAML le plus optimisé. Le gain est quasiment proportionnel au nombre de processeurs, sauf avec l'ensemble des CPU car :

- le système d'exploitation a besoin d'un processeur libre pour le service des interruptions. Un artifice permet de sélectionner le processeur le moins chargé, mais quand une application utilise tous les processeurs, il est possible qu'une tâche perde la main inopinément (dans une section critique de quelques instructions par exemple).
- le système PCAML ne possède que 16 piles préallouées, ce qui est trop juste pour plus de quelques processeurs (mais le nombre et la taille des piles sont des paramètres configurables).

La profondeur du parallélisme de `fib'` (qui n'est autre que le second paramètre) doit être réglée en fonction du nombre de processeurs disponibles. Pour `fib'` il faut environ 4 tâches par processeur.

Le problème des reines

Un autre test usuel est le problème des reines : il faut placer N reines sur un échiquier $N \times N$ de telles façons qu'aucune paire de reines soit en prise (c'est-à-dire sur une même ligne, colonne ou diagonale). La fonction de test renvoie le nombre de solutions possibles :

```
(* map avec concaténation des résultats *)

let concmap f =
 let rec cmr = fun
      [] -> []
    | (x::l) -> f x @ cmr l
  in cmr
;;
Value concmap = <fun> : (('a -> 'b list) -> 'a list -> 'b list)

(* filtre une liste *)

let filter p =
 let rec fr = fun
      [] -> []
    | (x::l) -> if p x then x::(fr l) else fr l
```

```
  in fr
;;
Value filter = <fun> : (('a -> bool) -> 'a list -> 'a list)

(* teste si une position est sans prise *)

let rec safe x =
 let rec sr d = fun
      [] -> true
    | (q::l) -> x <> q & x <> q+d & x <> q-d & sr (d+1) l
  in sr
;;
Value safe = <fun> : (num -> num -> num list -> bool)

(* teste une solution partielle *)

let ok = fun [] -> true | (x::l) -> safe x 1 l;;
Value ok = <fun> : (num list -> bool)

(* produit la liste des solutions *)

let rec gen size =
 let rec gr = fun
      0 -> [[]]
    | n ->
        concmap
         (fun b ->
           filter ok (map (fun q -> q::b) (interval 1 size)))
         (gr (n-1))
  in gr
;;
Value gen = <fun> : (num -> num -> num list list)

(* compte les solutions *)

let test size = length (gen size size);;
Value test = <fun> : (num -> num)

(* le lancement du test *)

test 7;;
```

Ce programme n'a aucune prétention d'être une manière efficace ou même raisonnable de résoudre le problème.

Les temps pour $N = 7$ sont :

Sept Reines (en secondes)	
CAML (strict, 1 CPU)	4.70
PCAML (parallèle, 1 CPU)	14.6
PCAML (parallèle, 2 CPU)	10.8
PCAML (parallèle, 3 CPU)	9.4
PCAML (parallèle, 4 CPU)	8.1
PCAML (parallèle, 6 CPU)	6.6

La version 2.5 montre sur ce test certaines déficiences :

- les fonctions récursives sont assez mal compilées dans certains cas car une fonction récursive étant placée dans son propre environnement, elle ne peut pas être considérée comme un combinateur. Donc chaque appel provoque l'allocation d'une paire.

- les fonctions à plusieurs arguments sont curryfiées ce qui entraîne la construction de toutes les fermetures intermédiaires même quand elles sont inutiles.

- la fonction `concmap` doit effectuer des effets de bord pour éviter des recopies.

Des analyses locales permettraient de procéder aux optimisations, mais dans un système interactif comme CAML, chaque définition au top-level est indépendante. Par exemple il n'est pas possible de considérer tous les usages d'une fonction globale pour les remplacer par le code en-ligne de la fonction.

La version parallèle se comporte assez bien, mais ce test met en évidence la nécessité d'éliminer à la compilation les tests des valeurs gelées dans une grande partie des occurrences des primitives d'accès dans les paires.

Le test de Boyer

Un autre test intéressant est le "Boyer" [30, Gabriel] : c'est un système de preuves de théorèmes constitué d'une base de règles de simplification du

genre :

$$(x > y) \rightarrow (y < x)$$
$$(p \Rightarrow q) \rightarrow (\texttt{if}\, p\, \texttt{then}\, (\texttt{if}\, q\, \texttt{then}\, true\, \texttt{else}\, false)\, \texttt{else}\, false)$$

et d'une routine de démonstration. Le test consiste à montrer :

$$(x \Rightarrow y \wedge (y \Rightarrow z \wedge (z \Rightarrow u \wedge u \Rightarrow w))) \Rightarrow (x \Rightarrow w)$$

où les propositions sont :

$$x \equiv f((a+b)+(c+0))$$
$$y \equiv f((a \times b) \times (c+d))$$
$$z \equiv f(reverse(append(append(a,b),nil)))$$
$$u \equiv (a+b) = (x-y)$$
$$w \equiv (a\, rem\, b) < member(a, length(b))$$

Le théorème à démontrer est entièrement réécrit en suivant les règles de simplification, puis est passé à la routine de test qui prend trois arguments :

1. le terme courant
2. une liste T de termes vrais initialisée à $true$
3. une liste F de termes faux initialisée à $false$

Le seul connecteur logique reconnu est le $\texttt{if}\, p\, \texttt{then}\, q\, \texttt{else}\, r$ qui a la valeur de vérité de :

si $p \in T$ alors q ;

sinon il faut que q et r sont vrais avec respectivement $T', F' = T \cup p, F$ et $T, F \cup p$

Ce test est excellent pour juger de la qualité d'un système Lisp, mais la phase de réécriture fait beaucoup trop de travail (il suffit de se limiter à la transformation des $\Rightarrow$ et des $\wedge$ en `if` pour obtenir le résultat). Un langage paresseux se montre bien plus efficace que n'importe quelle version stricte, et un système parallèle retourne le résultat avant la fin des calculs ! Le test de Boyer est donc peu intéressant pour juger des performances de PCAML, mais il montre tout l'intérêt d'un système qui peut combiner aisément plusieurs modes d'évaluations ...

D'autres systèmes comparables

MultiLisp [33, Halstead] est le premier langage possédant la construction `future`. C'est un dérivé de Scheme. Le noyau est organisé autour d'un interprète de *bytecode*, ce qui assure une excellente portabilité et d'exécrables performances. Mais MultiLisp est un prototype qui n'est pas destiné à des utilisations où les performances importent. Le système de GC de MultiLisp est assez original, car il emploie un algorithme parallèle temps-réel.

D'autres systèmes ont été développés à partir de l'expérience de MultiLisp :

- MultiScheme [49, Miller] qui est une version spéciale du MIT Scheme avec la construction `future` et ses dérivés.

- Mul-T [34, Halstead, Kranz, Mohr] est une modification profonde d'un des systèmes Scheme les plus performants. Il est intéressant car il est possible de juger le coût des glaçons spéculatifs dans un système de très haut niveau de performances. Le GC n'est autre qu'un algorithme classique parallélisé (où donc l'activité s'arrête sur tous les processeurs pour reprendre après la récupération).

  La gestion de la glacière essaie de préserver une bonne localité des tâches sans autres raffinements. Le compilateur élimine les tests des valeurs gelées : en moyenne le coût de ces tests est de 100% en temps d'exécution supplémentaire par rapport à une version sans les tests, après optimisation il se réduit à 65% (les valeurs pour PCAML sont proches, et cette optimisation est donc effectivement importante).

  Quand le nombre de tâches dépasse un certain seuil, Mul-T exécute directement le code des glaçons. Malheureusement cette transformation change un appel spéculatif en appel par valeur, et donc peut poser des problèmes. En Mul-T, l'appel par nécessité n'apparaît pas en temps que tel, ce qui sûrement l'origine de l'incorrection de certaines optimisations proposées. Nous évitons ces problèmes en PCAML grâce à notre approche originale qui essaie de coller au maximum à la sémantique des systèmes séquentiels de départ CAML et LCAML.

- le dernier système est QLisp [31, Gabriel, Goldman]. Ce langage résout le problème de l'explosion du parallélisme en ajoutant un prédicat aux constructions parallèles. S'il est vrai, les tâches correspondantes sont crées, sinon la construction est exécutée comme son homologue séquentiel (c'est la technique utilisée pour dans la fonction `fib'`).

La forme `future` existe en QLisp, mais c'est la construction dérivée `qlet` (équivalente à un `let` PCAML en mode parallèle) qui est la construction de base.

Ces langages sont assez proches de PCAML et partagent avec lui la même restriction d'utilisation : il faut que les tâches ne soient pas trop petites, et donc contrôler par un moyen quelconque le niveau de parallélisme. Mais si cette condition est remplie, les performances sont bonnes même comparées à un système strict très rapide.

Des systèmes à réduction de graphes ont été aussi proposés comme systèmes fonctionnels parallèles [51, Peyton Jones]. Ainsi il est possible de dériver une machine parallèle de la G-machine, avec un fonctionnement interne assez proche de PCAML.

L'orientation vers des tâches plus petites permet de simplifier la gestion de la glacière et des queues de processus, en évitant des sections critiques aux prix de l'exécution possible du même travail par plusieurs processeurs. Mais les problèmes fondamentaux du noyau restent identiques.

Malheureusement les langages supportées par ces machines sont exclusivement paresseux, sans exceptions ou effets de bord. Le résultat immédiat de ce choix est un niveau de performances bien inférieur à celui des langages fonctionnels moins purs comme ML ou Scheme, et une diffusion assez faible des systèmes.

Déjà il semble que les langages fonctionnels les plus généraux doivent se cantonner à un usage de prototypage : beaucoup d'applications écrites d'abord dans un langage fonctionnel sont réécrites dans un langage plus classique comme C avant de devenir “industrielles”. Les marchands de traducteurs Lisp vers C font désormais fortune !

Chapitre 10

Le glaneur de cellules

Les langages fonctionnels produisent beaucoup d'objets comme les fermetures qu'il faut allouer dynamiquement. La libération explicite par le programmeur de l'espace occupé est pénible et guère envisageable pour les objets produits par le système lui-même.

Donc un ramasseur de miettes ou "glaneur de cellules" est indispensable. Mais la présence d'un GC change beaucoup la programmation du support d'exécution. En fait les langages peuvent être divisés en deux classes :

- ceux sans GC où les utilisateurs doivent allouer et libérer l'espace mémoire explicitement.
- les systèmes avec GC[1], dits à gestion automatique de l'espace mémoire. Cette technique est potentiellement moins efficace et impose de grosses contraintes au générateur de code, mais est incomparablement plus souple.

Les différentes classes d'objets

Un objet est accessible grâce à un "pointeur" sur cet objet qui s'implante physiquement comme une adresse. Un objet peut être :

- une valeur immédiate codée dans les bits de l'adresse. Ce genre d'objets ne prend pas de place en mémoire, et donc n'est jamais récupérable.
- un objet alloué en mémoire et donc récupérable.

[1] La référence en matière de GC est l'article [17, Cohen].

En anglais les termes *unboxed* et *boxed* (emboîté) qualifient les deux possibilités pour les petits objets. Par exemple les petits entiers peuvent être mis directement dans les bits d'une adresse et un "pointeur" vers un entier contiendra la valeur de cet entier, ou bien un espace peut être réservé en mémoire et renfermera la valeur. Il faut remarquer que les objets physiquement plus long qu'une adresse sont toujours alloués.

Le GC doit pouvoir reconnaître les objets emboîtés des valeurs immédiates, et plus généralement le système doit pouvoir distinguer les différents classes d'objets comme les chaînes de caractères, les vecteurs, etc. Ces classes ou types ne sont pas les types du langage, mais ceux de la machine de base. En CAML, ils sont accessibles sous la forme de constructeurs du type concret des représentations des valeurs, le type CAML `obj` qui a pour définition :

```
type obj =
    obj_int of int
  | obj_float of float
  | obj_string of string
  | obj_atom of atom
  | obj_vect of obj vect
  | obj_cons of obj * obj
;;
```

A chaque classe est associée une instruction qui teste l'appartenance ou non de la représentation d'une valeur à la classe. Ces instructions sont utilisées par le compilateur pour coder les tests des filtrages.

Avec un GC, il n'est plus possible d'allouer les objets n'importe où, car il faut pouvoir réaliser les tests de types. Plusieurs techniques d'implantations sont possibles :

- l'espace des adresses (virtuelles) est divisé en zones. Chacune des zones est associée à un type, et le test consiste à vérifier l'encadrement de l'adresse par les bornes de la zone. C'est une méthode très simple, assez efficace (une ou deux comparaisons pour le test de type) mais qui a tendance à figer statiquement la taille des zones.

- l'espace est divisé en pages. Chaque page ne contient que des objets d'un même type (cette technique est connue sous le nom anglais de *big bag of pages*). Le test de type s'effectue en déterminant à quelle page appartient une valeur, puis en consultant une table, ce qui manque d'efficacité. Par contre l'allocation de la mémoire aux différents types est très souple.

- des bits des pointeurs contiennent un codage du type. Cette méthode est très utilisée malgré ces nombreux inconvénients :
  - l'espace total adressable est réduit
  - l'indirection sur un pointeur nécessite le masquage des bits de types (l'étiquette) sauf des cas particuliers astucieusement choisis ou avec un dispositif matériel ad hoc
  - l'utilisation des valeurs immédiates peut exiger aussi le masquage des bits
- chaque objet contient un champ de type. Cette technique doit être associée à une des autres sauf si tous les objets sont emboîtés ce qui entraîne généralement un gaspillage éhonté de la mémoire. Le test de type s'effectue par un accès mémoire et une comparaison.

Si en plus un objet doit pouvoir être substitué par un objet d'un autre type, il faut éviter que le type soit codé d'une manière ou d'une autre dans le pointeur, et donc seule la dernière technique est utilisable en général. Ici le terme substitution recouvre au moins deux cas :

1. une substitution physique, comme dans un système paresseux où une suspension peut être accédée de plusieurs endroits à la fois
2. un code qui doit effectuer des opérations indépendamment du type de représentation, comme pour un type abstrait

Le travail du GC consiste à récupérer tous les objets qui ne sont pas accessibles à partir des racines (registres de travail, pile d'exécutions, environnement global, etc ...) qu'on peut considérer comme constituées d'un seul objet pour simplifier notre propos. Si la mémoire est modélisée comme un graphe dirigé où les pointeurs sont les arcs, alors le GC doit calculer la composante connexe contenant la racine.

Pour certaines applications il peut être utile d'introduire des pointeurs faibles (en anglais *weak pointers*) qui permettent dans certaines conditions d'accéder à un objet, mais qui ne sont pas pris en compte par le GC. Ils permettent de construire des tables de hachage, et par exemple la glacière du système CAML parallèle doit être implantée avec des pointeurs faibles vers les glaçons spéculatifs pour qu'un glaçon déconnecté de la racine soit récupérable.

Les compteurs de références

La première technique est celle des compteurs de références : à chaque objet est associé le nombre de pointeurs (les références) le désignant. A chaque fois qu'un nouveau pointeur est créé, alors le compteur de références de l'objet pointé est augmenté d'une unité. Au contraire si un pointeur est détruit, alors le compteur est diminué d'une unité. A la création d'un objet, le compteur est initialisé à un (car il n'y a alors qu'un seul pointeur vers lui). Quand un compteur descend à la valeur zéro (c'est-à-dire quand un objet n'est plus référencé), alors cette objet est récupérable et est ajouté dans une liste des emplacements mémoires libres (*free list*).

Mais cette technique ne permet pas directement de récupérer les objets circulaires, ce qui impose des contraintes importantes au langage ou l'utilisation d'une autre méthode de GC pour le traitement des structures circulaires.

Les compteurs de références utilisent beaucoup d'espace mémoire pour conserver pour chaque objet la valeur du compteur associé. En plus, la valeur maximale de ces compteurs doit être grande pour éviter le problème du débordement. Le compilateur doit produire du code pour gérer ces compteurs à chaque manipulation de pointeurs, ce qui se révèle en pratique comme extrêmement coûteux en temps (lenteur d'exécution) et en place (très mauvaise densité du code obtenu), mais cet algorithme est naturellement incrémental, c'est-à-dire que la récupération de la mémoire s'effectue en même temps (pour l'utilisateur) que les autres activités, ce qui est un avantage important pour des utilisations interactives ou orientées temps-réel.

Deux optimisations ont été proposées pour cette technique :

1. l'utilisation de compteurs sur un bit. En effet la quasi-totalité des objets ne sont pas partagés, et les rares objets qui possèdent plusieurs références peuvent être gérés grâce à une autre table ou à une autre méthode de GC.

2. la mise à jour des compteurs peut se faire de façon différée, ce qui permet de dissocier la gestion des compteurs de l'activité du programme principal, et de simplifier éventuellement des transactions dans des sens contraires sur les compteurs.

Les compteurs de références ne sont pas très utilisés dans le monde des langages fonctionnels à l'exception d'InterLisp à cause du manque d'efficacité, bien que cette technique semble être bien adaptée aux systèmes distribués

et dans les contextes temps-réel. Nous ne reviendrons plus sur les systèmes à compteurs de références.

Le marquage-balayage

Les autres algorithmes explorent l'espace mémoire pour y déterminer l'ensemble des objets inaccessibles et donc récupérables. Le plus ancien d'entre eux est connu sous le nom anglais de *Mark & Sweep* et comprend deux phases :

marquage : les objets sont explorés récursivement à partir de la racine en suivant les pointeurs, chaque objet rencontré étant marqué au moyen d'un bit de marquage.

```
let rec explore_list l =
  if not (is_marked l)
   then
    (mark l;
     match l with
        [] -> ()
      | x::r -> explore x; explore_list r)
and explore x =
  if not (is_marked x) then mark x
;;
```

Le gros avantage du marquage est qu'il s'effectue par une opération idempotente sur un bit.

balayage : l'espace complet est balayé séquentiellement et tous les objets non marqués sont récupérés, généralement en les plaçant dans une liste des objets libres. Cette phase peut en fait être accomplie en différé tant que les bits des marquages subsistent. En particulier il est intéressant de construire la liste libre à la demande.

Les phases de marquage (à cause de l'utilisation du OU) et de balayage (en découpant l'espace à balayer) se parallélisent très bien. Il existe même une extension incrémentale et parallèle due à Kung et Song [42, Kung, Song] qui possède la propriété remarquable de ne pas nécessiter d'exclusion mutuelle entre le mutateur (le processus qui exécute le programme utilisateur) et le collecteur (le processus qui récupère la mémoire).

Mais les objets ne sont pas déplacés et donc ne peuvent pas être compactés. Les extensions proposées pour réaliser le compactage utilisent des manipulations de pointeurs (trop) compliquées et ne sont pas parallélisables de façon efficace. Cette technique est donc parfaite pour des objets de taille fixe, mais ne convient absolument pas pour les objets de taille variable où un compactage est nécessaire.

L'arrêt-recopie

L'algorithme le plus utilisé effectue une recopie et est nommé en anglais *Stop & Copy*. La mémoire est divisée en un espace de départ (*from-space*) et un espace d'arrivée (*to-space*) initialement vide. La mémoire est explorée à partir de la racine, un objet rencontré est recopié dans le *to-space* et remplacé par un pointeur d'indirection vers la copie. Un pointeur vers un objet déjà copié est remplacé par la valeur de ce pointeur d'indirection ce qui assure la cohérence.

```
let rec explore_list l =
  if (have_forward l)
   then (follow_forward l)
   else
    (match l with
        [] -> []
      | x::r ->
         let (x'::r' as p) = new_cons () in
           put_forward (p,l);
           x' <- explore x;
           r' <- explore_list r)
and explore x =
  if (have_forward x)
   then (follow_forward x)
   else (let y = new_objet x in put_forward (y,x))
;;
```

après le GC, les deux espaces *from-space* et *to-space* sont échangés.

Cette méthode laisse certaines libertés quand à la façon dont est effectuée l'exploration. En effet il est possible d'utiliser l'espace de recopie *to-space* comme une séquence implicite de pointeurs sur les objets accessibles, et de le balayer linéairement, ce qui a l'avantage d'éviter un parcours récursif,

mais change l'ordre de recopie (de profondeur d'abord en largeur d'abord). De plus il peut être utile de jouer sur l'ordre de recopie pour améliorer la localité des objets.

L'allocation se fait linéairement dans l'espace vide à la fin du dernier *to-space*, ce qui est généralement plus rapide que l'utilisation d'une liste libre. Ce GC augmente la localité des objets : deux objets liés par des pointeurs sont recopiés dans des positions proches en mémoires, et compacte complètement l'espace. Mais il utilise le double d'espace (heureusement généralement virtuel). Cette technique est sensée être très favorable aux systèmes à mémoire virtuelle, mais nous n'avons jamais vu une démonstration de cette allégation et en pratique nous avons pu constater que le gaspillage de la moitié de la mémoire amène des déboires avec certains de ces systèmes.

Par contre, cette méthode a un avantage important : le temps d'exécution est proportionnel à l'espace accessible et pas à l'espace total. Donc si seulement une très petite part de la mémoire totale est utilisée, la part du GC dans le temps d'exécution tend vers zéro. Cet argument a été utilisé pour justifier l'utilisation d'un tas par rapport à une pile [3, Appel] :

- l'allocation dans une pile ou dans un tas géré en *Stop & Copy* est de même nature : il suffit d'avancer un pointeur marquant le début de l'espace libre.

- la récupération dans la pile se fait par des instructions "dépile" ou des ajustements du pointeur de pile. Disons qu'il faut une instruction pour deux objets mis en pile.

- le temps que prend un GC *Stop & Copy* ne dépend que du nombre d'objets accessibles, disons 5 instructions par objet.

- or il n'y a qu'une portion ρ de l'espace total utilisé, donc le coût d'un GC est de 5ρ.

- donc si $5\rho < 1/2$, c'est-à-dire $\rho < 1/10$, alors le tas est plus efficace que la pile !

Malheureusement pour des raisons conjoncturelles la mémoire est assez chère, et ce raisonnement n'est valable que pour des mini-ordinateurs avec vraiment beaucoup de mémoire ...

Un algorithme synthétique

Il est possible de concilier les avantages respectifs des GC en *Mark & Sweep* et en *Stop & Copy* [26, Dupont, Lang]. En effet, il suffit de découper à chaque GC l'espace mémoire en trois zones :

1. une zone (*MS-space*) qui sera traitée en *Mark & Sweep*
2. une zone source pour la copie (*from-space*) qui sera entièrement libérée à la fin du GC
3. une zone vide (*to-space*) où seront recopiés les objets de la zone précédente.

Pour le GC suivant, le *from-space* deviendra le nouveau *to-space*, et le *MS-space* sera décalé. Ainsi chaque morceau de la mémoire sera compacté au bout de quelques phases. La mémoire inutilisée n'est plus que d'une fraction de l'espace total, au lieu de la moitié.

L'intérêt de cette technique est que la proportion entre l'espace géré en *Mark & Sweep* et celui en *Stop & Copy* peut être réglée entre chaque appel du GC pour s'adapter au comportement du programme ou à des contraintes du gestionnaire de mémoire virtuelle [54, Spir].

Le GC sur un multi-processeur

Les différentes techniques de GC s'adaptent aux multi-processeurs. La méthode la plus simple est d'arrêter l'activité de tous les processeurs, puis de procéder à la récupération et enfin de reprendre l'activité. La seule contrainte est de réaliser une synchronisation globale à l'entrée dans le GC.

Pour un *Mark & Sweep*, cette technique donne :

1. synchronisation globale de tous les processeurs, que nous considérons comme un cas particulier des interruptions.
2. phase de marquage en parallèle. Comme le marquage proprement dit monte à la valeur 1 les bits de marquage, il n'est pas nécessaire d'utiliser des verrous pour les protéger. La racine est généralement constituée d'objets comme la table de hachage de l'environnement global, ce qui permet une distribution aisée des tâches aux différents processeurs.
3. une synchronisation globale à la fin du marquage (sous forme du passage d'une barrière).

4. on peut alors relancer l'activité ou bien procéder immédiatement au balayage sur plusieurs processeurs.

Il y a très peu d'attentes sur les verrous (seulement pour la distribution des tâches), et les performances sont très bonnes avec un gain proche du nombre de processeurs. En plus, la réalisation est relativement facile.

Pour un *Stop & Copy*, il faut prendre un verrou pour chaque objet au moment de la recopie, ce qui est assez coûteux en performances. Un GC mixte est donc très intéressant même s'il est assez complexe à écrire.

Les techniques incrémentales

Une autre contrainte qui peut être exigée pour un GC est de récupérer l'espace pendant l'activité normale du programme, ce qui permet d'obtenir des temps de réponse peu dépendants du GC. En effet la durée d'un GC ordinaire est de l'ordre de quelques secondes, ce qui est intolérable dans certains contextes comme le temps-réel.

Les différents algorithmes possèdent parfois une version "incrémentale" où la récupération (le "collecteur") s'effectue au fur et à mesure de l'activité principale (le "mutateur"). Généralement la technique est pratiquement identique dans le cas de deux tâches collecteur/mutateur entrelacées ou d'un vrai parallélisme.

Les *Mark & Sweep* incrémentaux utilisent un système de marquage à plusieurs couleurs, chacune correspondant à un état :

blanc : les objets qui n'ont pas été atteints par le collecteur. Au début du cycle, tous les objets sont blancs, et à la fin du cycle, seuls les objets inaccessibles sont blancs.

gris : les objets atteints par le collecteur, mais pas encore explorés. Au début du cycle la racine prend cette couleur.

noir : les objets qui ont été complètement explorés par le collecteur. A la fin du cycle, tous les objets accessibles sont noirs.

Le collecteur explore tous les objets gris, un objet passe à l'état noir une fois qu'il a été exploré et que donc tous les objets qu'il référence sont au moins gris. Quand le mutateur modifie dans un objet noir un pointeur vers un objet blanc, cet objet prend la couleur grise. D'ailleurs tous les algorithmes incrémentaux imposent une contrainte spéciale en cas de modification physique. Le cycle s'achève quand tous les objets gris ont disparu.

La méthode de Kung & Song consiste à implanter la couleur grise par une file d'attente entre les deux processus contenant au début du cycle la racine. A une extrémité le mutateur peut insérer des éléments. Le collecteur peut insérer ou extraire des éléments à l'autre extrémité. Ainsi aucune exclusion mutuelle n'est nécessaire entre les deux processus !

Les GC en *Stop & Copy* demandent en revanche des exclusions mutuelles et ne sont pas intéressants du point de vue performances pour un système parallèle. Mais dans le cas incrémental il est possible d'éliminer cette difficulté en interrompant le mutateur à des moments précis, c'est l'algorithme de Baker [7, Baker].

Le collecteur pendant la phase d'exploration copie les structures atteintes à partir de la racine dans l'espace d'arrivée. Un balayage linéaire de cet espace explore ces structures et copie les objets atteints. Les nouveaux objets sont aussi alloués dans le *to-space*.

Le mutateur doit à chaque accès mémoire en lecture à partir d'un pointeur tester un des trois cas possibles :

1. l'objet est vieux (dans le *from-space*). Alors il faut le recopier et prendre le pointeur sur sa copie.

2. l'objet a été copié et le mutateur a obtenu un pointeur d'indirection. Il doit le suivre de façon transparente (d'où le nom de "pointeurs invisibles").

3. l'objet est dans le *to-space*.

Ainsi le mutateur n'a accès qu'à des objets situés dans le *to-space* (et donc accessibles). Malheureusement la présence de "pointeurs invisibles" est très mauvaise pour les performances, en effet il faut faire un test à chaque indirection ou comme [12, Brooks] l'a proposé ajouter à chaque objet un pointeur invisible qui sera systématiquement suivi, et gaspiller un mot par objet !

Le système d'exploitation

Heureusement dans certains cas le système d'exploitation (en particulier la gestion de mémoire virtuelle) permet d'effectuer le test avec un coût plus acceptable. L'exemple le plus simple de support par le système est le débordement de la zone d'allocation. Normalement pour chaque objet alloué il faut vérifier que le pointeur d'allocation reste valide. Si l'espace bordant la

zone est invalide, alors il suffit d'attendre une erreur de segmentation ce qui représente quelques dizaines d'instructions par débordement au lieu d'une comparaison par allocation. Malheureusement il faut pouvoir reprendre le cours des opérations au même endroit et donc que le système offre à l'utilisateur le même genre de possibilité de continuation ou de reprise que le processeur lui fournit, ce qui est très loin d'être un cas fréquent. Dans le cas de l'algorithme de Baker l'échange des tests contre une manipulation de la pagination apporte un gain énorme [5, Appel, Ellis, Li], mais à part pour le système lui-même ou avec du matériel spécialisé, le programmeur restera sur sa faim.

Les systèmes à générations

Les GC incrémentaux ne sont donc pas utilisables en pratique avec des performances raisonnables. Il a donc fallu trouver d'autres méthodes pour éviter les arrêts brutaux des programmes pour cause de GC, surtout dans des systèmes interactifs où les utilisateurs finissent par "interagir" avec les implémenteurs ...

L'analyse de la durée de vie des objets montre que les objets se classent en deux catégories [36, Hewitt, Liebermann] :

- des objets qui ont une durée de vie très faible et qui ont peu de chances de survivre à un GC (en gros des objets temporaires)
- des objets qui ont une durée de vie beaucoup longue et qui peuvent survivre à grand nombre de GC.

Il faut donc séparer les objets en générations, plaçant un objet dans une génération selon son âge, et en effectuant les GC préférentiellement sur les jeunes générations où il y a plus de place à récupérer.

Ainsi classiquement dans un système à génération il y a trois zones avec :

- les objets éphémères qui ont été alloués depuis le dernier GC
- les objets dynamiques qui ont survécu aux GC
- les objets statiques dont le système assure qu'ils ont une durée de vie très longue.

A chaque GC les objets éphémères accessibles sont recopiés dans la zone dynamique : c'est la phase mineure qui prend très peu de temps car la fraction

active est petite (de l'ordre de quelques pour cent). Quand la mémoire est pleine, il faut faire un GC sur les générations plus anciennes pour récupérer le maximum d'espace : c'est la phase majeure, aussi longue qu'un GC ordinaire, mais assez rare si le système a un bon comportement.

Les objets modifiables posent un problème, car le GC à génération repose sur l'hypothèse que les objets vieux ne pointent pas sur les objets jeunes, et donc qu'il est possible de restreindre l'exploration aux dernières générations. Il faut donc en cas de modifications physiques produisant des pointeurs de vieux vers des jeunes utiliser des tables annexes. Heureusement dans la plupart des langages fonctionnels les modifications physiques sont des extensions impératives peu utilisées, et donc ce problème ne grève pas les performances.

Il est clair que les objets jeunes doivent être gérés par un GC en *Stop & Copy*, alors qu'un GC en *Mark & Sweep* est plus intéressant pour les objets vieux, surtout pour les objets statiques qu'il est souvent préférable de garder immobiles. Donc les algorithmes mixtes sont idéalement adaptés aux techniques à générations, en changeant la balance marquage/copie pour chaque génération.

Dans le cadre de CAML un tel système a été proposé et réalisé [24, Doligez]. Pour l'extension parallèle, il est clair qu'il faut se contenter de paralléliser le code et ne pas se lancer dans des systèmes incrémentaux. En plus la partie en *Stop & Copy* est dans un cadre favorable avec peu d'objets à recopier pour la zone éphémère et seulement une partie des zones plus anciennes à gérer, donc avec peu de sections critiques et très peu de chances de contentions.

Conclusion

L'architecture choisie, une machine multi-processeur à mémoire partagée, s'est révélée bien adaptée à un langage parallèle supportant d'assez grosses tâches. Les mécanismes internes de verrous ont permis la réalisation d'un noyau du langage qui suit une logique assez simple, et nous sommes convaincus qu'il se porte sans difficulté sur d'autres machines d'architectures comparables.

Nous avons rejeté les constructions parallèles non-déterministes classiques pour explorer une voie nouvelle, s'appuyant sur le caractère fonctionnel du langage. Nous y avons gagné un système qui a nécessité très peu de modifications du compilateur, et donc qui s'intègre parfaitement. La version paresseuse qui nous a servi de point de départ a été remise à jour et complétée, ce qui a donné les moyens de proposer dans les dernières versions de CAML des constructions paresseuses.

Nous avons obtenu un système unique en son genre car il propose trois modes d'évaluation : strict, paresseux et parallèle. Ils peuvent être mêlés librement dans les programmes, et l'utilisateur peut choisir le mode le plus approprié pour chaque expression. La sémantique du mode paresseux est conservée dans le mode parallèle lorsqu'aucun effet de bord n'est utilisé, ce qui autorise une mise au point d'un programme à partir d'une version séquentielle sans craindre des surprises désagréables au passage à la version parallèle.

Les premiers résultats sont encourageants, et la comparaison avec des systèmes descendants de MultiLisp démontre l'intérêt de notre approche originale basée sur la sémantique des langages fonctionnels.

Nous avons proposé une nouvelle classe d'algorithmes pour le glaneur de cellules. Il apparaît maintenant clairement que le glaneur de cellules a une influence dramatique sur les performances globales d'un système, et donc en est une pièce maîtresse.

Finalement nous disposons maintenant d'un prototype parallèle d'un sys-

tème CAML, c'est-à-dire d'un système complet et pas d'un langage jouet taillé sur mesure. Les techniques que nous avons développées ont déjà fait leurs preuves, et donc pourrons être employées pour d'autres systèmes.

Annexe A

Séquentialité de PCF

Le langage PCF utilisé pour les démonstrations de propriétés de ML est un sous-ensemble de ML (car le langage complet est trop lourd pour notre propos, et n'apporte rien de plus pour les propriétés théoriques). Suivons la démonstration de [53, Plotkin].

Le langage PCF

Les types sont :

- les booléens *bool* avec les constantes $false$ et $true$
- les entiers naturels *nat* avec toutes les constantes correspondantes k_n
- les fonctions de type $\sigma \to \tau$. Les types fonctionnels sont dits composés, les autres (ici *bool* et *nat*) sont des types de base.

L'abstraction et l'application sont notées de façon usuelle, respectivement $(\lambda x.M)$ et (MN), avec comme types $(\lambda x_\sigma.M_\tau) : \sigma \to \tau$ et $(M_{\sigma\to\tau}N_\sigma) : \tau$.

En plus les fonctions suivantes appartiennent au langage :

- $\texttt{if}_{bool} : bool \to bool \to bool \to bool$
- $\texttt{if}_{nat} : bool \to nat \to nat \to nat$
- les opérateurs de point fixe : $\texttt{Y}_\sigma : (\sigma \to \sigma) \to \sigma$
- $\texttt{succ} : nat \to nat$
- $\texttt{pred} : nat \to nat$

- $\texttt{null} : nat \to bool$

La sémantique opérationnelle de PCF

Deux sémantiques peuvent être définies sur ce langage :

- une sémantique opérationnelle qui décrit chaque pas de calcul
- une sémantique dénotationnelle qui représente la signification d'un programme par une fonction entre domaines.

Décrivons d'abord la sémantique opérationnelle symbolisée par la flèche $\Rightarrow$ définie par les équations :

$$\begin{array}{ll}
\texttt{if}\, true\, M_\sigma N_\sigma \Rightarrow M_\sigma & (\sigma\, type\, de\, base) \\
\texttt{if}\, false\, M_\sigma N_\sigma \Rightarrow N_\sigma & (\sigma\, type\, de\, base) \\
\texttt{Y}_\sigma M \Rightarrow M(\texttt{Y}_\sigma M) & \\
((\lambda x.M)N) \Rightarrow M[x \backslash N] & \\
\texttt{succ}\, k_n \Rightarrow k_{n+1} & (n \geq 0) \\
\texttt{pred}\, k_{n+1} \Rightarrow k_n & (n \geq 0) \\
\texttt{null}\, k_0 \Rightarrow true,\ \texttt{null}\, k_{n+1} \Rightarrow false & (n \geq 0)
\end{array}$$

et les règles :

$$\begin{array}{ll}
\dfrac{M \Rightarrow M'}{(MN) \Rightarrow (M'N)} & M, M' : \sigma \to \tau; N : \sigma \\
\dfrac{M \Rightarrow M'}{\texttt{if}\, M \Rightarrow \texttt{if}\, M'} & M, M' : bool \\
\dfrac{M \Rightarrow M'}{\texttt{succ}\, M \Rightarrow \texttt{succ}\, M'} & M, M' : nat \\
\dfrac{M \Rightarrow M'}{\texttt{pred}\, M \Rightarrow \texttt{pred}\, M'} & M, M' : nat \\
\dfrac{M \Rightarrow M'}{\texttt{null}\, M \Rightarrow \texttt{null}\, M'} & M, M' : nat
\end{array}$$

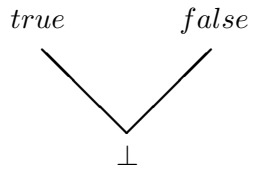
Un programme de PCF est un terme clos de type de base (l'idée est qu'un programme doit produire une valeur) et sa sémantique opérationnelle est donnée par la fermeture transitive de $\Rightarrow$. Donc un programme M a pour résultat la constante c si et seulement si :

$$M \stackrel{*}{\Rightarrow} c$$

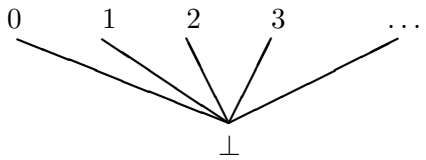
La sémantique dénotationnelle de PCF

La sémantique dénotationnelle est définie comme une application des termes de PCF dans le domaine D de Scott qui est défini comme $D = \cup D_\sigma$ avec :

- pour *bool*, $D_{bool} = \{\bot, False, True\}$ ordonné en :



- pour *nat*, $D_{nat} = \{\bot\} \cup \{0, 1, 2, \ldots\}$ avec :



- sinon $D_{\sigma\to\tau}$ est l'ensemble des fonctions continues de D_σ dans D_τ.

Cette sémantique notée traditionnellement par $[\![\ ]\!]$ indicé par l'environnement courant. La définition est :

$$
\begin{array}{l}
[\![false]\!]\rho = False \\
[\![true]\!]\rho = True \\
[\![k_n]\!]\rho = n \\
[\![x]\!]\rho = lookup(x,\rho) \\
[\![(MN)]\!]\rho = ([\![M]\!]\rho[\![N]\!]\rho) \\
[\![(\lambda x.M)]\!]\rho = v \rightsquigarrow [\![M]\!]extend(\rho,x,v) \\
[\![\texttt{succ}]\!]\rho = \left\{ \begin{array}{ll} x \rightsquigarrow x+1 & (x \geq 0) \\ x \rightsquigarrow \bot & (x = \bot) \end{array} \right. \\
[\![\texttt{pred}]\!]\rho = \left\{ \begin{array}{ll} x \rightsquigarrow x-1 & (x \geq 1) \\ x \rightsquigarrow \bot & (x = 0) \\ x \rightsquigarrow \bot & (x = \bot) \end{array} \right. \\
[\![\texttt{null}]\!]\rho = \left\{ \begin{array}{ll} x \rightsquigarrow True & (x = 0) \\ x \rightsquigarrow False & (x \geq 1) \\ x \rightsquigarrow \bot & (x = \bot) \end{array} \right. \\
[\![\texttt{if}]\!]\rho = \left\{ \begin{array}{ll} p\,x\,y \rightsquigarrow x & (p = True) \\ p\,x\,y \rightsquigarrow y & (p = False) \\ p\,x\,y \rightsquigarrow \bot & (p = \bot) \end{array} \right. \\
[\![\texttt{Y}_\sigma]\!]\rho = f \rightsquigarrow \bigcup_{n \geq 0} f^n(\bot)
\end{array}
$$

Les fonctions auxiliaires *lookup* et *extend* peuvent être définies en CAML par :

```
let rec lookup x = fun
    ((n,v)::r) -> if n=x then v else lookup x r
  | _ -> failwith "lookup"
;;

let extend rho x v = (x,v)::rho;;
```

La terminaison

Le premier théorème intéressant est :

> pour tout programme M et constante c $M \stackrel{*}{\Rightarrow} c$ si et seulement si $[\![M]\!]\emptyset = [\![c]\!]$ ($\emptyset$ est l'environnement vide).

Il se démontre par induction structurelle sur les termes de PCF, la seule difficulté étant la récursion qui nécessite des propriétés topologiques du domaine (c.p.o.), mais ce n'est pas notre propos.

Une conséquence est intéressante : la non-terminaison d'un programme P correspond à $[\![M]\!]\emptyset = \bot$, ce qui justifie le nom de cette valeur sémantique particulière. On peut exhiber un cas utile de tels programmes :

$$\Omega_\sigma = \mathtt{Y}_\sigma(\lambda x_\sigma.x_\sigma)$$

L'opérateur OU parallèle

Malheureusement la sémantique dénotationnelle accepte des termes qui nécessitent des réductions en parallèles qui ne peuvent pas être appréhendés par la sémantique opérationnelle, en particulier le OU parallèle (*parallel-OR*).

Soit les termes $M_i (i = 0, 1)$ encodant sa table de vérité :

$$\begin{array}{lll} M_i = \lambda x.(\mathtt{if} & (x\, true\, \Omega_{bool}) & \\ & (\mathtt{if} & (x\, \Omega_{bool}\, true) \\ & & (\mathtt{if} \quad (x\, false\, false) \\ & & \qquad \Omega_{nat} \\ & & \qquad k_i) \\ & & \Omega_{nat}) \\ & \Omega_{nat}) & \end{array}$$

qui a pour type $(bool \to bool \to bool) \to nat$.

La sémantique dénotationnelle de Por peut être résumé par :

Por	$\bot$	$False$	$True$
$\bot$	$\bot$	$\bot$	$True$
$False$	$\bot$	$False$	$True$
$True$	$True$	$True$	$True$

et il est facile de vérifier que $([\![M_i]\!]\emptyset)Por = i$. Mais dans tout contexte $C[\ ]$ acceptant un terme du bon type, les $C[M_i]$ ne peuvent pas être opérationnellement différenciés.

Le lemme d'activité

Définissons la notion de sous-programme actif (qui précise où sera faite la prochaine réduction) :

1. si le programme M a une des formes $((\lambda x.N)\ldots)$, $(\mathtt{Y}\ldots)$, $(\mathtt{succ}\, c)$, $(\mathtt{pred}\, c)$, $\mathtt{null}\, c$ ou $\mathtt{if}\, c\ldots$ alors le terme actif de M est M.

2. si le programme M a pour forme $\texttt{succ}\, N$, $\texttt{pred}\, N$, $\texttt{null}\, N$ ou $\texttt{if}\, N \ldots$ où N n'est pas une constante. Alors le terme actif de M s'il existe est celui de N.

D'après cette définition, les programmes qui n'ont pas de sous-programme actif sont les constantes, et si un programme termine et a un terme actif, alors celui-ci termine aussi.

Lemme d'activité : Soit $C[M_1, \ldots, M_n]$ un programme terminant avec la valeur c contenant les termes clos $M_1, \ldots, M_n$. Alors pour tous les termes clos $N_1, \ldots, N_n$ du bon type, ou :

1. $C[N_1, \ldots, N_n]$ termine avec la valeur c, ou
2. il existe un contexte $D[\ ]$, un entier i ($1 \leq i \leq n$) et des indices $j_1, \ldots, j_m$ tels que $C[N_1, \ldots, N_m] \stackrel{*}{\Rightarrow} D[N_{j_1}, \ldots, N_{j_m}]$ et la partie active de $D[N_{j_1}, \ldots, N_{j_m}]$ existe et est d'une des formes $(N_i\, M)$, $\texttt{succ}\, N_i$, $\texttt{pred}\, N_i$, $\texttt{null}\, N_i$ ou $\texttt{if}\, N_i$.

Preuve : Notons $N_0 = C[M_1, \ldots, M_n] \Rightarrow N_1 \Rightarrow \ldots \Rightarrow N_p = c$ les étapes du calcul du programme. Chaque terme N_i peut être considéré comme un contexte D prenant comme arguments des M_j, le dernier $N_p = c$ n'ayant plus d'arguments et vérifiant trivialement le lemme.

Chaque étape réduit un radical dans le contexte D ou dans un des M_j, selon la position du sous-programme actif. Si elle ne met pas en jeu un des M_j, c'est-à-dire qu'il y a un D' et des $k_1, \ldots, k_q$ tels que pour tous les termes clos du bon type M'_j on a $D[M'_j] \Rightarrow D'[M'_k]$, alors si le lemme est vrai pour D', alors il est vrai pour D.

Donc les cas intéressants sont ceux qui mettent en jeu un M_j qu'on notera simplement M. Passons en revue les cas possibles où le sous-programme actif (où se fera la réduction) est dans le contexte immédiat de M :

- $\texttt{if}\, true\, M\, N$: ce terme se réduit en M indépendamment de M et donc ne le met pas en jeu.
- $\texttt{if}\, true\, N\, M$: ce terme fait disparaître M et ne le met pas en jeu. Les cas en $\texttt{if}\, false$ sont identiques.
- $\texttt{if}\, M$: le sous-programme actif est alors $\texttt{if}\, M$ et est l'un des cas prévu dans le lemme.
- $\texttt{Y}\, M$: ce terme se transforme en $M(\texttt{Y}\, M)$ pour tout M, et donc ne met pas en jeu M.

- $\texttt{succ}\, M$: c'est le sous-programme actif et aussi l'un des cas du lemme. Les cas $\texttt{pred}\, M$ et $\texttt{null}\, M$ se traitent de la même façon.

- $(M\, N)$: c'est aussi un cas du lemme.

- $((\lambda x.M)N)$: M est clos, donc la variable x n'est pas libre dans M, d'où pour tous les termes clos M' du bon type ce terme se réduit en M'. Donc c'est encore un cas où M n'est pas mis en jeu.

- $((\lambda x.(N))M)$: pour tous termes clos M' du bon type, ce terme se transforme en un $C[M', \ldots, M']$ avec un M' pour chaque occurrence libre de x dans le corps de l'abstraction. Ce cas ne met pas en jeu M.

- $((N)\, M)$: la partie active est dans (N) et donc ce cas n'est pas à envisager.

Donc dans $N_0 = C[M_1, \ldots, M_n] \Rightarrow N_1 \Rightarrow \ldots \Rightarrow N_p = c$, deux cas sont possibles :

1. aucun des M_i n'est mis en jeu et donc de proche en proche on a pour tous les termes clos M_i du bon type :

$$C[M_i'] \Rightarrow C_1[M_{j_1}'] \Rightarrow \ldots \Rightarrow c$$

   et donc on est dans le premier cas du lemme.

2. au moins un des M_i est mis en jeu, considérons le premier $N_p = D[M_{j_1}, \ldots, M_{j_m}]$ pour lequel ca se produit, et le J indice du M_j mis en jeu dans D. On a donc pour le calcul (éventuellement vide) de N_0 à N_p, pour tous les termes clos M_i du bon type :

$$C[M_1', \ldots, M_n'] \Rightarrow \ldots \Rightarrow D[M_{j_1}', \ldots, M_{j_m}']$$

   et nous avons vu que les seuls cas possibles pour D sont ceux où il y a un sous-programme actif en $(M_J \ldots)$, $\texttt{succ}\, M_J$, $\texttt{pred}\, M_J$, $\texttt{null}\, M_J$ ou $\texttt{if}\, M_J$. Donc nous sommes dans le deuxième cas du lemme.

Le théorème du OU parallèle

Théorème : il n'existe pas de contexte $C[\ ]$ tel que les programmes $C[M_0]$ et $C[M_1]$ terminent avec des valeurs différentes.

Preuve : par l'absurde, considérons un contexte C contre-exemple du théorème. Appliquons le lemme, nous ne sommes pas dans le premier cas car les valeurs pour deux arguments distincts sont différentes. Donc $C[M_0] \stackrel{*}{\Rightarrow} D[M_0, \ldots, M_0]$, D a au moins un argument M_0 et le sous-programme active a une des formes du lemme. Or le type de M_0 impose qu'il existe un L tel que cette forme soit $(M_0\, L)$, avec en plus car le programme termine pour L :

$$\begin{array}{llll} L & true & \Omega_{bool} & \stackrel{*}{\Rightarrow} \quad true \\ L & \Omega_{bool} & true & \stackrel{*}{\Rightarrow} \quad true \\ L & false & false & \stackrel{*}{\Rightarrow} \quad false \end{array}$$

Considérons le contexte $C'[true, \Omega_{bool}] = L\, true\, \Omega_{bool}$ et appliquons à nouveau le lemme. Soient M et N des termes clos quelconques du type booléens, alors :

1. $(L\, M\, N) \stackrel{*}{\Rightarrow} true$: c'est impossible car $(L\, false\, false) \stackrel{*}{\Rightarrow} false$.

2. $(L\, M\, N)$ se récrit en un terme qui utilise N, c'est-à-dire un contexte $C'[]$ où (car N est un booléen) le sous-programme actif existe et est de la forme `if` N. Mais alors si on prend Ω_{bool} pour N, ce sous-programme ne termine pas, et donc $(L\, M\, \Omega_{bool})$ ne termine pas. Or ce cas n'est pas possible car $(L\, true\, \Omega_{bool})$ termine avec la valeur $true$.

3. donc c'est M qui est utilisé, mais si on prend Ω_{bool} pour M on aboutit à la même contradiction car $(L\, \Omega_{bool}\, true) \stackrel{*}{\Rightarrow} true$.

On obtient donc une contradiction dans tous les cas, donc le contexte C n'existe pas. Les deux types de sémantiques n'ont pas la même puissance d'expression si nous introduisons des équivalences entre les programmes (du genre : deux termes sont équivalents s'ils se comportent de la même façon dans n'importe quel contexte).

En conclusion un langage comme CAML présente la propriété de "séquentialité" qui interdit l'écriture de programme où il faut simultanément calculer deux termes. Cette propriété est vraie aussi pour l'extension parallèle de CAML.

Annexe B

Continuations et Contextes

Un langage fonctionnel “pur” ne permet pas d'exprimer certaines structures de contrôles pourtant bien utiles agissant sur le contexte du terme courant. Le partage maximal peut être réalisé à l'aide de telles structures dans la fonction qui enlève les occurrences du terme `e` dans la liste `l`, contrairement à la version élémentaire :

```
let delete e l =
 let rec loop = function
     [] -> []
   | x::r -> if x=e then loop r else x::loop r
 in
  loop l
;;
```

Les exceptions

La forme la plus simple de la manipulation du contrôle est l'exception, qui est créée par la construction `raise` qui prend en argument un couple nom d'exception et valeur et lève l'exception avec cette valeur. L'autre primitive est le `try` qui prend une expression et un filtrage d'exceptions comme arguments. Le `try` évalue l'expression et retourne sa valeur. Si l'expression lève une exception, alors le couple nom d'exception/valeur est filtré, et la branche correspondante est exécutée. Dans le cas par défaut où aucun filtre n'est approprié, par convention l'exception est levée avec la même valeur (construction `reraise`).

Notre exemple s'écrit avec l'exception `Identity` qui est levée en levant une exception au lieu de renvoyer une copie de la fin de la liste :

```
exception Identity;;

let delete e l =
 let rec loop = function
     [] -> raise Identity
   | x::r ->
      if x=e
       then try loop r with Identity -> r
       else x::loop r
 in
  try loop l with Identity -> l
;;
```

La réalisation des exceptions

Les constructions `try` sont gérées en pile, l'entrée dans un `try` provoque la pose d'un bloc spécial dans une pile, la sortie du `try` son dépilage. Ces blocs sont chaînés entre-eux, le `raise` fabrique le couple exception/valeur, recherche le dernier `try` dans la pile et lance la fonction associée au filtre après avoir retiré le bloc (afin qu'un `reraise` ou un nouveau `raise` accède au `try` suivant).

Le contrôle de type des exceptions

Ce mécanisme d'exceptions est donc dynamique, ce qui impose quelques restrictions pour son typage. Un type de valeurs exceptionnelles peut être introduit : les exceptions peuvent être considérées comme des constructeurs d'un type concret (en SML New Jersey ce type s'appelle `exn`). Ce type est un peu particulier car les déclarations d'exceptions y ajoutent de nouveaux constructeurs.

La construction `raise` est assez simple à typer : comme elle ne retourne jamais de valeur, elle peut être placée n'importe où dans le code, et a donc un type polymorphe `exn -> 'a`. Le `try` est plus compliqué, si l'expression dans le `try` a le type `t`, alors le filtrage du `try` doit renvoyer un objet du même type, donc être en `exn -> t`.

Si l'exception n'est pas polymorphe, le typage est donc aisé. Mais il est interdit de construire des valeurs au top-level de type polymorphe `'a`, car

s'il existe `x : 'a`, alors la phrase suivante "casse" le système : `x + (x 1)` avec comme instances du type de `x num` et `num -> num` !

Il n'est pas permis de déclarer des exceptions de type polymorphe, ce qui est la rançon habituelle d'un caractère dynamique ... Cette contrainte peut être assouplie en introduisant des types polymorphes faibles, ce qui permet d'écrire l'utilitaire qui essaye d'appliquer une fonction en retournant sa valeur et en cas d'échec en demandant s'il faut ressayer. Si oui, on recommence, sinon l'exception rendue est levée, ce qui donne en reprenant la notation de SML pour ce genre de type :

```
let ask_retry f =
 exception Success of '1a in ask_rec
  where rec ask_rec x =
   try
     try f x with _ ->
       if ask "Evaluation failed, retry [y/n]? " = "y" then
        (raise Success (ask_rec x)) reraise
   with Success x -> x
;;
Value ask_retry = <fun> : (('a -> '1b) -> 'a -> '1b)
```

Le type `'1a` est un type polymorphe faible de degré 1, ce qui signifie que `ask_retry` doit être appliquée au moins une fois, et qu'après cette application le type `'1a` doit être instancié par un type non polymorphe (ce type peut être considéré comme `'0a`, l'abstraction élevant le degré, l'application le diminuant. Un type fortement polymorphe ayant un degré infiniment grand).

Le call-with-current-continuation

Certains langages proposent une manipulation des continuations avec des opérateurs comme le `call-with-current-continuation` noté plus brièvement `call/cc` [16, Clinger, Rees]. Une continuation est l'abstraction d'un contexte, par exemple la continuation associée au contexte $C[\ ]$ est la fonction $\lambda v.C[v]$.

Dans ML il est possible d'introduire un type des continuations `'a cont`, le paramètre `'a` étant le type de l'argument de la continuation, c'est-à-dire du "trou" du contexte. Une opération sur les continuations est `call/cc f` qui capture la continuation courante et la passe en argument à `f`, et le lancement `throw k v` qui reprend l'exécution à l'endroit où a été créée la continuation `k` en retournant la valeur `v`.

Il est facile de remplacer les exceptions avec des `call/cc` et une abstraction à portée dynamique `fluid x -> e` (le terme *fluid* est traditionnellement employé pour la liaison dynamique, de même le terme *lexical* pour la liaison à portée statique). Ainsi un programme du genre :

```
exception foo of T;;
...
try e1 with foo x -> x;;
...
raise foo v;;
```

devient :

```
...
call/cc (fluid foo -> e1);;
...
throw foo v;;
```

Par exemple, la fonction `tak` bien connue [30, Gabriel] :

```
let rec tak x y z =
  if y >= x then z else
     tak (tak (x-1) y z)
         (tak (y-1) z x)
         (tak (z-1) x y)
;;
```

peut s'écrire avec une exception retournant la valeur :

```
exception Val of num;;

let rec aux x y z =
 if y >= x then raise Val z else
    aux (try aux (x-1) y z with Val n -> n)
        (try aux (y-1) z x with Val n -> n)
        (try aux (z-1) x y with Val n -> n)
;;

let tak x y z = try aux x y z with Val n -> n;;
```

La traduction avec le `call/cc` demande de passer les continuations en arguments pour simuler une liaison dynamique qui n'existe pas en CAML :

```
let rec aux k x y z =
 if y >= x then throw k z else
  call/cc (fun k -> aux k (call/cc (fun k -> aux k (x-1) y z))
                          (call/cc (fun k -> aux k (y-1) z x))
                          (call/cc (fun k -> aux k (z-1) x y)))
;;

let tak x y z = call/cc (fun k -> aux k x y z);;
```

La fonction `delete` aussi peut s'écrire de façon pas trop lisible à notre avis avec des `call/cc` :

```
let delete e l =
 let rec start s =
  call/cc (fun k ->
             let rec go = function
                [] -> throw k s
              | x::r -> if x=e then start r else x::go r
             in go s)
 in
  start l
;;
```

Le type de call/cc

Le type des ces opérateurs pose plus de problèmes ... Dans un contexte qui prend un objet de type `'a` le `call/cc` appelle une fonction avec la continuation courante de type `'a cont` et finit soit par le retour direct d'une valeur de type `'a`, soit par un appel de la continuation qui finalement doit produire un valeur du type `'a`.

Un type acceptable pour le `call/cc` serait donc `('a cont -> 'a) -> 'a`. Le lancement prend une continuation `'a cont`, une valeur du type correspondant `'a` et ne revient jamais, d'où `throw : 'a cont -> 'a -> 'b`.

Malheureusement ces types ne marchent pas si la continuation qui a une durée de vie illimitée est utilisée dans une autre phrase, car alors il est facile de produire des valeurs polymorphes comme dans cet exemple :

```
type foo = Nothing | Cont of num cont;;
 Type foo defined
      Nothing : foo
```

```
    | Cont : (num cont -> foo)

let bar = ref Nothing;;
 Value bar = (ref Nothing) : foo ref

call/cc (fun k -> bar := Cont k; 2);;
 2 : num

let (Cont k) = !bar in throw k 1;;
 1 : 'a
```

Si le type de la valeur finalement renvoyée par l'exécution est pris en compte, il faut ajouter un second paramètre au type des continuations, et le nouveau type de `throw : ('a,'b) cont -> 'a -> 'b` sera trop restrictif !

De toute façon, dans ce genre de continuations rien ne délimite vraiment le contexte manipulé, ce qui suggère de travailler directement sur les contextes pour plus de clarté, et d'introduire des opérations qui définissent exactement la portée et la durée de vie du contexte courant.

Des opérateurs plus généraux

L'opérateur le plus simple est `abort` qui oublie le contexte courant :

$$C[\texttt{abort}\, M] \Rightarrow M$$

qui permet de définir la construction associée au `call/cc` :

$$C[\texttt{escape}\, x\, M] \Rightarrow C[M[x \leftarrow \lambda v.\texttt{abort}\, C[v]]]$$

Une construction primitive plus simple a été introduite par M. Felleisen :

$$C[\texttt{control}\, x\, M] \Rightarrow M[x \leftarrow \lambda v.C[v]]$$

si `x` est libre dans `M`, elle se réduit à `abort`.

Le prompt `#` sert à délimiter le contexte courant, ainsi :

$$C_1[(\#\, C_2[\texttt{control}\, x\, M])] \Rightarrow C_1[M[x \leftarrow \lambda v.C_2[v]]$$

malheureusement l'aspect totalement dynamique de ce genre de construction rend le typage impossible, car le `#` délimite un contexte courant qui ne peut être déterminé en général qu'à l'exécution.

Or il est possible d'introduire des opérateurs du même genre avec une portée totalement lexicale [22, Danvy, Filinski] :

reset E : l'expression `E` est évaluée dans un nouveau contexte vide. Il y a un `reset` implicite devant chaque phrase du *top-level*.

shift I **in** E : le contexte courant est abstrait comme une fonction et lié à l'identificateur `I` avant que l'expression `E` soit exécutée dans un nouveau contexte.

Grâce à la portée statique, ces opérateurs sont enfin compatibles avec le système de type de ML au prix d'une importante complication, car il faut non seulement traiter la valeur d'une expression, mais aussi son action sur le contexte.

Il est donc possible de synthétiser le type d'opérateurs sur les contextes, ce qui lève le principal obstacle à l'introduction d'abstractions fonctionnelles du contrôle dans ML.

Dans ce style, notre exemple est très facile à programmer. la fonction auxiliaire `start` servant à définir le contexte (et donc le morceau de liste) dans lequel on essaye de réaliser le partage :

```
let delete e l =
 let rec start s =
  let rec go = function
      [] -> shift k in s
    | x::r -> if x=e then start r else x::go r
  in reset (go s)
 in
  start l
;;
```

Le style de passage de continuations

Généralement les opérateurs de continuations ne sont pas implantés directement car les programmes subissent au cours de la compilation une traduction dans un style de passage de continuations (en anglais *CPS* pour *Continuation Passing Style*) directement inspiré par la sémantique avec continuations [4, Appel].

Ainsi cette transformation suit en gros le schéma suivant :

$$\begin{array}{rcl}
[\![C]\!] & \Rightarrow & \lambda\kappa.\kappa\, C \\
[\![I]\!] & \Rightarrow & \lambda\kappa.\kappa\, I \\
[\![\lambda I.E]\!] & \Rightarrow & \lambda\kappa.\kappa(\lambda I.[\![E]\!]) \\
[\![\texttt{Prim}]\!] & \Rightarrow & \lambda\kappa.\kappa(\lambda v\,\kappa'.\kappa'(\texttt{Prim}\, v)) \\
[\![(E_1\, E_2)]\!] & \Rightarrow & \lambda\kappa.[\![E_1]\!](\lambda v.[\![E_2]\!](\lambda w.v\, w\, \kappa)) \\
[\![\texttt{if}\, E_1\, \texttt{then}\, E_2\, \texttt{else}\, E_3]\!] & \Rightarrow & \lambda\kappa.[\![E_1]\!](\lambda v.\texttt{if}\, v\, \texttt{then}([\![E_2]\!]\kappa)\texttt{else}([\![E_3]\!]\kappa)) \\
[\![\texttt{call/cc}]\!] & \Rightarrow & \lambda\kappa.\kappa(\lambda v\,\kappa'.v(\lambda w\,\kappa''.\kappa'\, w)\kappa') \\
[\![\texttt{freeze}\, E]\!] & \Rightarrow & \lambda\kappa.\kappa[\![E]\!] \\
[\![\texttt{force}\, E]\!] & \Rightarrow & \lambda\kappa.[\![E]\!](\lambda v.v\, \kappa)
\end{array}$$

L'ordre d'évaluation est entièrement précisé. En particulier une application par nom où l'argument n'est pas évalué donne la transformation :

$$[\![(E_1\, E_2)]\!] \Rightarrow \lambda\kappa.[\![E_1]\!](\lambda v.v\, [\![E_2]\!]\, \kappa)$$

La conversion en CPS permet certaines optimisations (dont nous constatons les résultats, car beaucoup de compilateurs très efficaces utilisent cette méthode depuis Rabbit [55, Steele]), mais nécessite une élimination de tous les radicaux inutiles introduits, et ne conserve pas certaines propriétés sémantiques [48, Meyer, Piecke]. De plus les programmes écrits dans ce style sont à peu près illisibles, ce qui ne constitue pas vraiment une qualité.

Les opérateurs `shift` et `reset` peuvent être traduits dans un CPS plus acceptable, avec des abstractions fonctionnelles du contrôle, et donc utilisables directement ou comme code intermédiaire pour les utilisateurs ordinaires d'un système comme CAML. Notre exemple se traduit dans ce style en :

```
let delete e l =
 let rec start k s =
  let rec go c = function
      [] -> k s
    | x::r ->
       if x=e
        then start c r
        else go (fun z -> c (x::z)) r
  in go k s
 in
  start (fun x -> x) l
;;
```

Une machine abstraite

Du point de vue théorique, à part la traduction en CPS, il est possible d'utiliser des machines abstraites comme la CEK de M. Felleisen [25, Duba, Felleisen, Friedman et Wand]. Cette machine comporte trois registres :

C qui contient l'expression en cours d'évaluation ou le marqueur ‡

E qui contient l'environnement

K qui contient la continuation :

- soit continuation ordinaire d'une des formes :

$$(\mathbf{stop}),\ (\kappa\,\mathbf{cont}),\ (\kappa\,\mathbf{arg}\,N\,\rho),\ (\kappa\,\mathbf{fun}\,V)$$

- soit une continuation de retour $(\kappa\,\mathbf{ret}\,V)$

Les continuations de retour n'apparaissent que dans des états du type $(\ddagger, \emptyset, \kappa)$. Les fermetures sont notées $[fun, \rho]$ et les points de continuations $< \mathbf{p}, \kappa >$.

Les règles de réécriture de cette machine avec les opérateurs $\mathcal{A}$ (`abort`) et $\mathcal{C}$ (du genre `call/cc` qui n'est autre que $\lambda.\mathcal{C}(\lambda\kappa.\kappa(f\,\kappa))$) sont :

$$\begin{array}{rcl}
(x,\ \rho,\ \kappa) & \Rightarrow & (\ddagger,\ \emptyset,\ (\kappa\,\mathbf{ret}\,\rho(x))) \\
(\lambda x.M,\ \rho,\ \kappa) & \Rightarrow & (\ddagger,\ \emptyset,\ (\kappa\,\mathbf{ret}[\lambda x.M, \rho])) \\
(M\,N,\ \rho,\ \kappa) & \Rightarrow & (M,\ \rho,\ (\kappa\,\mathbf{arg}\,N\,\rho)) \\
(\ddagger,\ \emptyset,\ ((\kappa\,\mathbf{arg}\,N\,\rho)\mathbf{ret}\,F)) & \Rightarrow & (N,\ \rho,\ (\kappa\,\mathbf{fun}\,F)) \\
(\ddagger,\ \emptyset,\ ((\kappa\,\mathbf{fun}[\lambda x.M, \rho])\mathbf{ret}\,V)) & \Rightarrow & (M,\ \rho[x \leftarrow V],\ \kappa) \\
(\mathcal{A}\,M,\ \rho,\ \kappa) & \Rightarrow & (M,\ \rho,\ (\mathbf{stop})) \\
(\mathcal{C}\,M,\ \rho,\ \kappa) & \Rightarrow & (M,\ \rho,\ (\kappa\,\mathbf{cont})) \\
(\ddagger,\ \emptyset,\ ((\kappa\,\mathbf{cont})\mathbf{ret}\,[\lambda x.M, \rho])) & \Rightarrow & (M,\ \rho[x \leftarrow< \mathbf{p}, \kappa >],\ (\mathbf{stop})) \\
(\ddagger,\ \emptyset,\ ((\kappa\,\mathbf{cont})\mathbf{ret}\ < \mathbf{p}, \kappa_0 >)) & \Rightarrow & (\ddagger,\ \emptyset,\ (\kappa_0\,\mathbf{ret} < \mathbf{p}, \kappa >)) \\
(\ddagger,\ \emptyset,\ ((\kappa\,\mathbf{fun} < \mathbf{p}, \kappa_0 >)\mathbf{ret}\,V)) & \Rightarrow & (\ddagger,\ \emptyset,\ (\kappa_0\,\mathbf{ret}\,V))
\end{array}$$

Les premières règles définissent un interprète du Lambda Calcul avec appel par valeur, les autres les opérations sur les continuations. Il est clair que le terme courant est exploré en poussant les composants des calculs futurs sur la continuation qui contient donc une représentation fidèle du contexte, et quand la machine produit une valeur, elle est mise dans une continuation de retour et le calcul suivant est déclenché.

D'autres machines de ce type existent avec d'autres variantes des opérateurs sur les continuations, et permettent donc une définition opérationnelle exécutable. Il faut très peu de temps pour écrire le programme correspondant dans un langage comme CAML, il suffit d'écrire dans l'ordre :

1. le type concret des termes à manipuler
2. une définition d'une syntaxe produisant des termes, ce qui permettra d'essayer interactivement de petits exemples
3. un imprimeur avec indentations (qui peut être extrait automatiquement de la syntaxe dans la plupart des cas)
4. les types concrets des valeurs, continuations, ... en bref des objets manipulés
5. quelques fonctions auxiliaires sur les environnements, continuations, etc ...
6. une fonction récursive implantant les règles de réécritures qui se réduit à un filtrage sur les états de la machine
7. une fonction d'interaction pour charger (avec $(M, \emptyset, (\mathbf{stop}))$ et décharger la machine à la fin du calcul dans l'état $(\ddagger, \emptyset, ((\mathbf{stop})\mathbf{ret}\, V)$.

en tout une centaine de lignes de code d'après notre expérience !

Une autre option pour décrire la sémantique est de donner une définition dénotationnelle où les continuations (ou des objets plus complexes s'il le faut) apparaissent. C'est la technique usuelle pour spécifier un compilateur, mais souvent elle n'est pas applicable directement si des performances raisonnables sont nécessaires, car l'introduction des continuations produit des termes d'ordre élevé même pour des expressions extrêmement simples.

Il est possible aussi de définir un Lambda Calcul avec des opérations sur les continuations (et il faut donc restreindre ce calcul aux règles de l'appel par valeur). Ces calculs ont de bonnes propriétés et permettent des raisonnements sur le contrôle [27, Felleisen].

Conclusion

Mais les objets dynamiques, donc les continuations en général sauf avec une portée statique, et les objets modifiables posent des problèmes de typage qui ne sont que le reflet de l'incompatibilité de la portée dynamique et du vrai Lambda Calcul. Ce n'est donc pas étonnant que les constructions dynamiques et les appels par nom ou par nécessité fassent très mauvais ménage.

Il faut donc chercher à éliminer ce genre de constructions pour pouvoir utiliser dans de bonnes conditions un système paresseux. Pour les opérations sur les continuations deux grandes techniques sont possibles :

- traduire le programme en CPS ce qui permet de fixer l'ordre d'appel (en effet un programme transformé peut indifféremment être exécuté en appel par valeur ou en appel par nom/nécessité). Mais il faut que la transformation donne quelque chose de raisonnable, d'où l'attrait des systèmes où seules sont expansées les parties faisant appel directement à des constructions manipulant les continuations.

- l'autre technique consiste à coder la "somme" valeur ordinaire/valeur exceptionnelle dans un type concret à deux constructeurs. Les fonctions ordinaires rangent le résultat dans le premier constructeur, les levées d'exceptions dans le second.

Bibliographie

[1] H. Abelson, G.J. Sussman
"Structure and Interpretation of Computer Programs."
M.I.T. Press, 1985

[2] S. Abramsky, C. Hankin
"Abstract Interpretation of Declarative Languages."
Ellis Horwood, 1987

[3] A.W. Appel
"Garbage Collection can be faster than stack allocation."
Information Processing Letters, volume 25, numéro 4, 1987

[4] A.W. Appel
"Continuation-Passing, Closure-Passing Style."
Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL), Janvier 1989

[5] A.W. Appel, J.R. Ellis, K. Li
"Real-Time Concurrent Collection on Stock Multiprocessors."
Report 25, DEC System Research Center, Février 1988

[6] L. Augustsson
"A Compiler for Lazy ML."
Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, Août 1984

[7] H.G. Baker
"List Processing in Real Time on a Serial Computer."
Communications of the ACM, volume 21, numéro 4, Avril 1978

[8] H.P. Barendregt
"The Lambda Calculus. Its Syntax and Semantics."
Studies in Logic 103, North Holland, 1981

[9] B. Beck, B. Kasten, S. Thakkar
"VLSI Assit for a Multiprocessor."
Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), SIGPLAN Notices, volume 22, numéro 10, Octobre 1987

[10] G. Berry, L. Cosserat
"The ESTEREL Synchronous Programming Language and its Mathematical Semantics."
Rapport de Recherche 327, INRIA, Septembre 1984

[11] G. Berry, J-J. Lévy
"Minimal and Optimal Computations of Recursive Programs."
Journal of the ACM, volume 26, numéro 1, Janvier 1979

[12] R.A. Brooks
"Trading Data Space to Reduce Time and Code Space in Real-Time Garbage Collection on Stock Hardware."
Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, Août 1984

[13] L. Cardelli
"Compiling a Functional Language."
Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, Août 1984

[14] J. Chailloux, M. Devin, J-M. Hullot
"Le-Lisp : a Portable and Efficient Lisp System."
Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, Août 1984

[15] W. Clinger
"The Scheme 311 Compiler. An Exercise in Denotational Semantics."
Proceedings of 1984 ACM Symposium on Lisp and Functional Programming, Août 1984

[16] W. Clinger, J. Rees
"The Revised^3 Report on the Algorithmic Language Scheme."
ACM SIGPLAN Notices, volume 21, numéro 12, Décembre 1986

[17] J. Cohen
"Garbage Collection of Linked Data Structures."
ACM Computing Surveys, volume 13, numéro 3, Septembre 1981

[18] G. Cousineau, P-L. Curien, M. Mauny
"The Categorical Abstract Machine."
Science of Computer Programming 8, 1987

[19] P. Cousot, R. Cousot
"Abstract Interpretation : a unified lattice model for static analysis of programs by construction of approximation of fixpoints."
Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL), 1977

[20] P-L. Curien
"Categorical Combinators, Sequential Algorithms and Functional Programming."
Research Notes in Theorical Computer Science, Pitman, 1986

[21] O. Danvy
"Memory Allocation and Higher-Order Functions."
Programming of Future Generation Computers II, Elsevier, 1988
Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices, volume 22, numéro 7, Juillet 1987

[22] O. Danvy, A. Filinski
"A Functional Abstraction of Typed Contexts."
DIKU Rapport 89/05, University of Copenhagen, Août 1989

[23] E. Dijkstra
"Guarded Commands, Nondeterminacy and Formal Derivation of Programs."
Communications of the ACM, volume 18, numéro 8, Août 1975

[24] D. Doligez
"Réalisation d'un glaneur de cellules de Lang et Dupont à générations."

Rapport de DEA, Université de Paris VII, Septembre 1989

[25] B. Duba, M. Felleisen, D. Friedman, M. Wand
"Abstract Constinuations : A Mathematical Semantics for Handling Full Functional Jumps."
Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Juillet 1988

[26] F. Dupont, B. Lang
"Incremental incrementally compacting garbage collection."
Programming of Future Generation Computers II, Elsevier, 1988
Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices, volume 22, numéro 7, Juillet 1987

[27] M. Felleisen
"The Calculi of λ_v-CS Conversion : a syntactic theory of control and state in imperative higher-order programming languages."
Ph. D. Thesis, Indiana University, Août 1987

[28] D. Friedman, C. Haynes
"Engines Build Process Abstractions."
Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, Août 1984

[29] D. Friedman, D. Wise
"CONS should not evaluate its arguments."
Automata, Languages, and Programming, Edinburgh University Press, 1976

[30] R. Gabriel
"Performace and Evaluation of Lisp Systems."
M.I.T. Press, 1985

[31] R.P. Gabriel, R. Goldman
"Preliminary Results with the Initial Implementation of Qlisp."
Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Juillet 1988

[32] S. Gregory
"Parallel Logic Programming in PARLOG. The Language and its Implementation."
International Series in Logic Programming, Addison-Wesley, 1987

[33] R. Halstead
"MultiLisp : a Language for Concurrent Symbolic Computation."
ACM Transactions on Programming Languages and Systems, volume 7, neméro 4, Octobre 1985

[34] R. Halstead, D. Kranz, E. Mohr
"Mul-T : a High-Performace Parallel Lisp."
Proceedings of the SIGPLAN'89 Conference on Prgramming Language Design and Implementation, SIGPLAN Notices, volume 24, numéro 7, Juillet 1989

[35] T. Hardin-Accart
"Résultats de Confluence pour les Règles Fortes de La Logique Combinatoire Catégorique et Liens avec les Lambda-Calculs."
Thèse de Doctorat, Université de Paris VII, Octobre 1987

[36] C. Hewitt, H. Liebermann
"A Real-Time Garbage Collector Based on the Life-Times of Objects."
Communications of the ACM, volume 26, numéro 6, Juin 1983

[37] W.D. Hillis, G.L. Steele
"Connection Machine Lisp : Fine-Grained Parallel Symbolic Processing."
Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming, Août 1986

[38] J. Hindley, J. Seldin
"Introduction to Combinators and λ-Calculus."
London Mathematical Society Student Text 1, Cambridge University Press, 1986

[39] C.A.R. Hoare
"Communicating Sequential Processes."
Communications of the ACM, colume 21, numéro 8, Août 1978

[40] R.J.M. Hughes
"Super-Combinators : a New Implementation Method for Applicative

Language."
Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Août 1982

[41] G. Kahn, D. MacQueen
"Coroutines and Networks of Parallel Processes."
Information Processing 77, Processings of the IFIP Congress 77, 1977

[42] H.T. Kung, S.W, Song
"An Efficient Garbage Collection System and its Correctness Proof."
Proceedings of the IEEE 18th Symposium on Foundations of Computer Science, Octobre-Novembre 1977

[43] A. Laville
"Evaluation Paresseuse des Filtrages avec Priorité. Application au Langage ML."
Thèse de Doctorat, Université de Paris VII, Février 1988

[44] L. Maranget
"GAML, une implémentation de ML paresseux par réduction de graphe."
Rapport de DEA, Université de Paris VII, Septembre 1988

[45] M. Mauny
"Compilation des langages fonctionnels dans les combinateurs catégoriques. Application au langage ML."
Thèse de troisième cycle, Université de Paris VII, Septembre 1985

[46] M. Mauny
"Parsers and Printers as Streams Destructors and Constructors Embedded in Functional Languages."
FPCA 89

[47] M. Mauny, A. Suárez
"Implementing Functional Languages in the Categorical Abstract Machine."
Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Août 1982

[48] A. Meyer, J. Riecke
"Continuations May Be Unreasonable."
Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Juillet 1988

[49] J. Miller
"MultiScheme : a Parallel Processing System Based on M.I.T. Scheme."
Ph. D. Thesis, M.I.T. E.E.C.S. Department, Août 1987

[50] R. Milner
"A theory of type polymorphism in programming."
Journal of Computer and System Science, volume 17, numéro 3, 1978

[51] S. Peyton Jones
"The Implementation of Functional Programming Languages."
Prentice-Hall International Series in Computer Science, Mai 1986

[52] G.D. Plotkin
"Call-by-name, call-by-value and the lambda calculus."
Theoretical Computer Science, volume 1, numéro 2, 1975

[53] G.D. Plotkin
"LCF considered as a programming language."
Theoretical Computer Science, volume 5, numéro 3, 1977

[54] E. Spir
"Etude et implantation d'un glaneur de cellules adaptatif pour Lisp."
Thèse de Doctorat, Université de Paris VII, Janvier 1989

[55] G.L. Steele
"RABBIT : a compiler for SCHEME."
AI-TR 474, M.I.T., Mai 1978

[56] D.A. Turner
"A New Implementation Technique for Applicative Languages."
Software Practice and Experience, volume 9, 1979

[57] J. Vuillemin
"Correct and optimal implementations of recursion in a simple programming language."
Rapport de Recherche 24, IRIA Loboria, Juillet 1973

[58] P. Weis et all
"The CAML Reference Manual. Version 2.6."
INRIA, Mars 1989

[59] P. Weis
"Le système SAM : métacompilation très efficace à l'aide d'opérateurs sémantiques."
Thèse de Doctorat, Université de Paris VII, Novembre 1987

Table des matières