Une présentation fonctionnelle de la compaction de code

Marc Pouzet

Laboratoire VERIMAG
Miniparc-ZIRST, Rue Lavoisier
38330 Montbonnot St-Martin, France
Adresse électronique : Marc.Pouzet@imag.fr

RÉSUMÉ. *Cet article présente le problème de la compaction de programmes pour les architectures parallèles à grain fin de type VLIW (Very Long Instruction Word) ou super-scalaires. Nous proposons une formulation nouvelle de la compaction, appliquée directement à la partie de premier ordre d'un langage fonctionnel avec appel par valeur. Par rapport aux travaux précédents, la compaction est définie formellement comme un système de transformations de programmes, chaque transformation "améliorant" le programme pour une machine donnée. La compaction est ici directement appliquée à des expressions fonctionnelles plutôt qu'à une représentation graphique (graphes de dépendances ou graphes de contrôle) donnant ainsi une présentation uniforme et simple.*

ABSTRACT. *This paper presents a general method to compact the first-order part of functional languages with call-by-value semantics for fine-grain parallel machines like VLIW or super-scalars. Compared to previous works, compaction is defined in a formal way by a set of program transformations, each of them "improving" the program for a given machine. The compaction is directly applied to functional expressions instead of graph based representations (control flow or dependence flow based representations) leading to a uniform and simple presentation.*

MOTS-CLÉS : *Parallélisme à grain fin, ordonnancement d'instructions, pipeline logiciel, langages fonctionnels, transformations de programme.*

KEY WORDS : *Instruction Level Parallelism (ILP), instruction scheduling, software pipelining, functional languages, program transformations.*

1. Introduction

Les progrès technologiques permettent de proposer aujourd'hui des processeurs capables d'exécuter en parallèle plusieurs calculs élémentaires. Le micro-parallélisme offert dans les nouvelles architectures peut être simplement de type "pipeline" ou "super-pipeline" : dans ce cas, une nouvelle instruction est exécutée avant la fin de la précédente. Le temps de calcul moyen par instruction est alors divisé par la taille du pipeline (de l'ordre de 5 à 10 pour les processeurs actuels). Mieux encore, plusieurs exécutions pipelines peuvent être chevauchées. On obtient alors un fonctionnement dit "super-scalaire" où plusieurs instructions scalaires successives sont exécutées en parallèle. L'efficacité de ces architectures va alors reposer sur la présence de "micro-parallélisme" dans le code déterminant la pleine utilisation des ressources de la machine. En effet, comment lancer en parallèle deux instructions si la deuxième dépend de la première ? De même, quelle nouvelle instruction exécuter si l'instruction en cours d'exécution est un test qui doit déterminer le chemin d'exécution ? Ce problème d'exploitation du parallélisme fin des programmes est ancien et, historiquement, deux points de vues assez radicaux ont été défendus : le point de vue "logiciel" et le point de vue "matériel".

Le point de vue "logiciel" repose sur l'utilisation d'un compilateur réordonnant statiquement les instructions de manière à garantir la bonne utilisation des différentes unités de calcul. Le processeur exécute le code tel quel sans aucun réordonnancement dynamique : le comportement de l'architecture est entièrement défini statiquement. Si une instruction dépend de l'instruction qui la précède, le compilateur devra insérer entre les deux une autre instruction indépendante (remplissage des *delay-slots*) de manière à ce que le pipeline reste pleinement utilisé. Le point de vue "logiciel" s'appuie sur des travaux ayant montré à la fois l'importance du parallélisme à grain fin des programmes mais surtout, la possibilité de l'exploiter automatiquement dans un compilateur [NIC 81, FIS 84, THE 92, LAM 92, FIS 92]. Le point de vue "logiciel" s'est illustré dans les premières machines RISC (Reduced Instruction Set Computer) de MIPS [KAN 88] puis des machines VLIW (Very Long Instruction Word) [FIS 81, FIS 84, MUL 87]. En reportant la gestion du parallélisme sur le compilateur, il a permis de définir des architectures pipeline à très haute fréquence et les premières architectures exécutant plusieurs instructions en parallèle.

Presque à l'opposé du point de vue "logiciel" (au moins à l'origine), le point de vue "matériel" fait le pari que le parallélisme à grain fin peut être géré par le processeur durant l'exécution. Pratiquement, il consiste à intégrer un contrôleur au sein du processeur capable d'analyser durant l'exécution les dépendances entre instructions ainsi que l'occupation des différentes ressources. À chaque cycle, le processeur charge plusieurs instructions (moins de dix) qui sont distribuées aux unités dédiées fonctionnant en parallèle. On se rapproche en quelque sorte d'une machine data-flow : l'ordre initial des instructions n'est alors pas nécessairement respecté (*out-of-order* execution). La solution matérielle est apparue plus récemment, historiquement dans la série Power d'IBM [IBM 90].

Des deux points de vue défendus, il est difficile — encore aujourd'hui — d'affirmer lequel est le meilleur. Le concept du "tout logiciel" illustré dans les premières architectures RISC et VLIW a — au moins dans sa forme initiale — été abandonné, ceci étant principalement dû à l'absence de compatibilité binaire qu'il entraînait, un nouvel exécutable devant être produit pour une nouvelle version de l'architecture. D'où l'idée de faire évoluer les architectures VLIW vers des architectures super-scalaires *statiques* avec exécution dans l'ordre (*in-order* execution) : le processeur ne réordonne pas les instructions et exécute en parallèle le préfixe maximal d'instructions successives indépendantes [DIG 92]. Un tel mécanisme reste simple à implanter, permet d'assurer la compatibilité binaire, est extensible, et repose sur un ordonnancement statique des instructions. Le concept du "tout matériel" où l'ordonnancement des instructions est effectué par le processeur, bien qu'il assure la compatibilité binaire et simplifie la tache du compilateur, semble aujourd'hui peu extensible : le nombre d'instructions analysées à chaque cycle reste faible et il est difficile d'intégrer un contrôleur capable d'analyser à chaque cycle de très nombreuses instructions, ce qu'un compilateur, lui, n'a aucun mal à faire. De plus, le processeur est limité par les dépendances de contrôle [LAM 92] : certaines transformations facilement réalisables par le compilateur sont impossibles durant l'exécution. D'où la nécessité de "préparer le code" en vue d'une meilleure utilisation des ressources, même dans le cas d'une architecture pleinement super-scalaire avec exécution dans le désordre [1].

La parallélisation à grain fin pour ce type d'architecture, appelée *compaction* à l'origine, consiste notamment à ordonner statiquement des groupes d'instructions qui seront ensuite exécutés en parallèle. Cet ordonnancement statique n'est en fait qu'une des composantes de la compaction. De même que le terme "vectorisation" désigne les techniques de compilation pour les machines vectorielles, le terme "compaction" va ici désigner l'ensemble des méthodes de compilation permettant d'exploiter le parallélisme offert par les architectures de type VLIW ou super-scalaires. Ce terme regroupe donc le problème de placement des calculs mais aussi celui de l'allocation des noms (registres, caches...) et plus généralement toute méthode de compilation spécifique à ces architectures. Traditionnellement, on distingue plusieurs types de compaction. La compaction peut être *locale* — elle est alors limitée aux expressions sans branchement, ou *globale* — et elle traite les expressions conditionnelles. Les boucles sont compilées par *pipeline logiciel*.

Dans cet article, nous proposons une présentation formelle de la compaction, appliquée directement à des expressions fonctionnelles (termes) en montrant qu'elle peut être vue comme un ensemble de transformations de programmes. Elle peut alors être combinée avec d'autres transformations de programme.

[1] Le lecteur trouvera dans la nouvelle édition de [HEN 90] une présentation détaillée de l'ensemble de ces architectures et montrant leur évolution possible.

Cette formulation permet de raisonner sur les transformations. Elle pourrait être utilisée dans la définition d'un méta-paralléliseur pour ces architectures mais aussi comme une première étape vers des implantations rapides et efficaces des langages fonctionnels pour les architectures parallèles à grain fin.

Cet article est organisé de la manière suivante. Dans la première partie nous justifierons le point de vue adopté dans ce travail. Il sera ensuite illustré par quelques exemples. La deuxième partie présentera une formalisation des idées avancées : nous définirons tout d'abord un langage fonctionnel et sa sémantique opérationnelle. La notion classique de dépendances sera définie par une relation d'équivalence appelée *équivalence structurelle* mettant en relation les "programmes ayant les mêmes dépendances". Cette équivalence servira de cadre à la formalisation : toute transformation de programme devra préserver cette équivalence. Ensuite, nous construirons un système de transformations de termes améliorant le programme pour une machine donnée.

2. Motivations

Les compilateurs compactant le code, et plus généralement, les paralléliseurs, se décomposent en deux familles. Dans la première famille, la compilation s'applique à une représentation de bas niveau du programme sous forme de graphe de flot de contrôle (graphe de blocs de base) où des instructions élémentaires "remontent" dans le graphe [FIS 84, ELL 85, AIK 88, NIC 85, EBC 87]. Le pipeline logiciel est obtenu par des méthodes itératives avec un dépliage contrôlé des boucles en recherchant un motif répétitif [AIK 88, EBC 87]. Dans ce cadre, l'occupation des ressources de la machine est exprimée en rassemblant dans un même noeud les instructions s'exécutant en parallèle. Les compilateurs de la deuxième famille s'appliquent à une représentation graphique du programme en étendant les méthodes d'ordonnancement classiques aux programmes contenant des structures de contrôle [TOU 84, LAM 88]. Le pipeline logiciel est alors appliqué aux graphes cycliques: on parle alors d'ordonnancement cyclique. Dans ce cadre, l'occupation des ressources est représentée par des matrices de réservation, indiquant les ressources utilisées pour chaque cycle. L'opposition de ces deux familles est entre la *généralité* et l'*efficacité*. La représentation basée sur le contrôle et associée aux méthodes itératives est le cadre le plus général puisque c'est la représentation de bas niveau de tous les programmes. Cependant, elle entraîne souvent des implantations coûteuses — les déplacements d'instructions ont un coût linéaire — et souffre de problèmes de convergence et de croissance de code. Les représentations basées sur les graphes de dépendances permettent des implantations efficaces — les déplacements d'instructions ont un coût constant — mais seul un sous-ensemble (sans tests) d'un vrai langage est efficacement traité dans cette représentation. En particulier, le pipeline logiciel est appliqué à des boucles élémentaires do sans test ou boucle imbriquée. Lorsque le contrôle est ajouté (par exemple, la *réduction hiérarchique* de Lam [LAM 88]), le contrôle est enfermé (une expression conditionnelle est assimilée à une instruction élémentaire), ce qui empêche d'atteindre les meilleurs résultats. Enfin, le graphe de dépendances ne peut pas être modifié durant la compaction limitant la possibilité de combiner l'ordonnancement avec d'autres transformations [2].

Cet article étudie une formalisation de la compaction, ayant donc à définir un langage intermédiaire et une notion abstraite de machine parallèle à grain fin. Nous devons donc résoudre l'opposition suivante : le langage intermédiaire doit être suffisamment général pour représenter un sous-ensemble réel des langages classiques, mais il doit également être utilisable en pratique (la représentation doit permettre des implantations efficaces). De plus, cette formalisation doit être suffisamment simple pour permettre de raisonner sur la compaction. Ensuite, certaines transformations de programme (renommage, duplication, dépliage, etc.) doivent être possibles. Enfin, pour être suffisamment générale, elle doit être capable de traiter le cas des programmes fonctionnels et récursifs : pour compacter de tels programmes, il est inutile de les transformer dans une représentation sous forme de graphe de contrôle de bas niveau pour ensuite retrouver leurs dépendances, puisque dans un langage fonctionnel les dépendances sont directement exprimées dans la syntaxe. De plus, dans un langage fonctionnel, des parties de programmes sont garanties sans effet de bord par le typage [3] et ont la propriété d'assignation unique (Static Single Assignment property). Parce qu'elle ne peut représenter le contrôle de n'importe quel programme, la représentation sous forme de graphe de dépendances seule n'est pas une représentation de programme satisfaisante. La solution proposée ici est en quelque sorte à mi-chemin entre la représentation par graphe de contrôle et la représentation par graphe de dépendances.

La compaction sera définie par un ensemble de transformations directement appliquées à des *termes* fonctionnels, chacune d'entre elles *améliorant* le terme en un certain sens. Le langage est bien une extension des graphes de dépendances — un graphe de dépendance est un cas particulier d'expression fonctionnelle [4] — et est également capable d'exprimer le contrôle des programmes. Avec la représentation fonctionnelle, le compilateur manipule en même temps les dépendances et le contrôle du programme. Grâce à la propriété d'assignation unique, les transformations (renommage, duplication, simplifications...) y sont plus simples et peuvent être combinées entre elles avec le réordonnancement. Avec cette représentation, l'occupation des ressources sera représentée d'une manière implicite : une machine sera vue comme un automate lisant directement le programme. Afin de définir un système formel pour la compaction de programmes, nous proposons une sémantique opérationnelle

---

[2] Pour terminer, le lecteur trouvera une vue d'ensemble du domaine dans [RFI 93].

[3] Bien qu'elles manipulent des structures de données (listes,...) représentées avec des pointeurs.

[4] Par exemple, la définition locale let dans une expression fonctionnelle est l'écriture syntaxique du partage, i.e, le moyen de définir des Graphes Acycliques Dirigés (DAGs).

donnant une mesure aux expressions — toute transformation devra donc améliorer les programmes pour cette mesure — et une notion de *dépendances*. La sémantique opérationnelle définit la manière dont une architecture parallèle à grain fin exécute le programme. La notion de dépendances est importante également. En effet, de même que la compaction respecte habituellement cette notion dans le cadre classique, on peut se demander comment cette notion s'applique dans le cas fonctionnel de manière à conserver la même expressivité. En d'autres termes, quelles sont les transformations raisonnables, voire minimales, dont un compilateur doit disposer ? Ici, la notion de dépendances sera définie par une relation d'équivalence, appelée *équivalence structurelle* et permettra les transformations classiques de la compaction.

3. Exemples

Essayons d'abord de compacter le programme P_1 donné dans la partie gauche de la figure suivante. Il exécute une addition $(+_1)$ entre une variable et une constante. Il lie le résultat à la variable x et exécute ensuite l'opération $+_2$ puis $-_3$, la condition et l'une des deux branches. Enfin, il calcule $*_5$ entre la valeur de $(x +_2 4) -_3 x$ et la valeur de l'expression conditionnelle.

$$
\begin{array}{ll}
P_1 = & \textsf{let}\ x = \overline{y +_1 2} \\
& \textsf{in}\ ((x +_2 4) -_3 x) *_5\ \textsf{if}\ \overline{z =_2 0} \\
& \qquad\qquad\qquad\qquad \textsf{then}\ \ \textsf{let}\ m = \overline{4 +_3 y} \\
& \qquad\qquad\qquad\qquad\qquad\quad \textsf{in}\ m *_4 n \\
& \qquad\qquad\qquad\qquad \textsf{else}\ x *_3 2
\end{array}
$$

$$
\begin{array}{ll}
P_2 = & \textsf{let}\ x = y +_1 2 \\
& \textsf{in}\ \ \textsf{if}\ z =_2 0 \\
& \quad \textsf{then}\ ((x +_2 4) -_3 x) *_5 (\ \textsf{let}\ m = \overline{4 +_3 y}\ ) \\
& \qquad\qquad\qquad\qquad\qquad\quad \textsf{in}\ m *_4 n \\
& \quad \textsf{else}\ ((x +_2 4) -_3 x) *_5 (x *_3 2)
\end{array}
$$

$$
\begin{array}{ll}
P_3 = & \textsf{let}\ x = y +_1 2 \\
& \quad\ \ m = 4 +_3 y \\
& \textsf{in}\ \ \textsf{if}\ z =_2 0 \\
& \quad \textsf{then}\ \ \textsf{let}\ x_1 = x +_2 4 \\
& \qquad\qquad\qquad x2 = m *_4 n \\
& \qquad\qquad \textsf{in}\ \ \textsf{let}\ x_1 = x_1 -_3 x \\
& \qquad\qquad\quad\ \ \textsf{in}\ x_1 *_5 x_2 \\
& \quad \textsf{else}\ \ \textsf{let}\ x_1 = x +_2 4 \\
& \qquad\qquad\quad\ \ x_2 = x *_3 2 \\
& \qquad\qquad \textsf{in}\ \ \textsf{let}\ x_1 = x_1 -_3 x \\
& \qquad\qquad\quad\ \ \textsf{in}\ x_1 *_5 x_2
\end{array}
$$

Ce programme peut être vu comme un programme de très bas niveau de type *code trois-adresses* où les primitives exécutées sont celles de la machine et où tous les noms de registres n'auraient pas été affectés. Considérons donc une machine $\mathcal{M}$ de type super-scalaire capable d'exécuter ce programme suivant un ordre gauche-droite et telle que trois instructions successives et indépendantes puissent être exécutées en parallèle. Les opérations sont ici annotées avec un indice donnant l'ordre d'exécution des instructions sur $\mathcal{M}$. En effet, au premier cycle, $+_1$, $+_2$ et $-_3$ sont chargées et seule $+_1$ peut être exécutée à cause des dépendances des deux autres. Au deuxième cycle, seul le test est chargé. Il est exécuté en parallèle avec $+_2$, etc. Finalement, le temps d'exécution de P_1 sur $\mathcal{M}$ est 5 si la condition est vraie. Regardons maintenant comment une simple transformation peut diminuer cette durée d'exécution. Considérons les calculs élémentaires qui peuvent être exécutés immédiatement et en parallèle. Nous appelons ces calculs des *sous-termes prêts*. Ils sont surlignés dans P_1. $+_3$ peut être exécuté spéculativement avant le test parce que c'est un calcul *sain* [5]. Nous appliquons alors une règle de remontée au test pour obtenir P_2. Maintenant $+_3$ peut être remonté. Nous continuons la compaction avec les deux branches pour obtenir P_3. Ce programme ne peut plus être compacté et sa durée d'exécution sur $\mathcal{M}$ est 4. On peut signaler maintenant la forte interaction entre le problème de l'ordonnancement et celui des noms (l'allocation de registre) : le code produit correspond cette fois très précisément à du code de type *trois-adresses* où le nom de variable désigne un registre. La compaction peut être appliquée indépendamment de l'allocation — le compilateur supposant disposer d'une infinité de noms ou bien n'autorisant jamais les renommages — ou appliquée en concurrence avec l'allocation de registre — en gérant à la fois le nombre d'unités de calculs et un nombre limité de noms de registres. Notons également que les fins communes des deux branches seront partagées pour limiter la croissance de la taille du code.

Considérons maintenant le cas d'une fonction récursive simple F écrite à la PCF [GUN 92]. C'est une itération sur une liste où $[\,]$ désigne la liste vide, hd et tl désignent respectivement la tête et la queue de la liste (x et a sont les arguments). Nous considérons ici que ces primitives sont élémentaires.

$$F = \mathsf{rec}_f \left(\begin{array}{l} \lambda\,[x;a].\ \mathsf{if}\ x = [\,] \\ \qquad\quad \mathsf{then}\ a \\ \qquad\quad \mathsf{else}\ f[\overline{tl(x)}; -(\overline{hd(x)}) + a] \end{array} \right)$$

La compaction globale de ce programme compacterait le corps de la fonction en ignorant le chevauchement possible entre itérations, et donc le parallélisme. La durée d'exécution du corps de la boucle est donc de l'ordre de 4 cycles [6]. Pour cette raison, nous appliquons d'abord une règle de dépliage : $\mathsf{rec}_f(a) \longrightarrow a[f \backslash \mathsf{rec}_f(a)]$ signifiant que toutes les occurrences de la variable libre f ont été remplacées par sa définition. Nous obtenons alors le terme donné ci-dessous. Maintenant, le seul sous-terme prêt est le test. Nous remontons ensuite les deux sous-termes prêts surlignés de la branche où le test est faux, pour obtenir le programme de droite.

$$\begin{array}{l} \lambda\,[x;a].\ \mathsf{if}\ x = [\,] \\ \qquad\quad \mathsf{then}\ a \\ \qquad\quad \mathsf{else}\ F[\overline{tl(x)}; -(\overline{hd(x)}) + a] \end{array} \qquad\qquad \begin{array}{l} \lambda\,[x;a].\ \mathsf{if}\ x = [\,] \\ \qquad\quad \mathsf{then}\ a \\ \qquad\quad \mathsf{else}\ \ \mathsf{let}\ x_1 = tl(x) \\ \qquad\qquad\qquad\ \ x_2 = hd(x) \\ \qquad\qquad\quad\ \mathsf{in}\ \underline{F[x_1; -x_2 + a]} \end{array}$$

Le terme restant à compacter est $F[x_1; -x_2+a]$. Parce que le premier argument est calculé, le dépliage de F va permettre de faire apparaître de nouveaux

[5] Un calcul est *sain* lorsqu'il peut être exécuté spéculativement sans modifier la sémantique.

[6] Pour être exact, il faut y ajouter le coût du branchement.

sous-termes prêts pouvant être exécutés en parallèle avec les sous-termes prêts courants. Après le dépliage, le programme suivant doit être compacté.

$$(\lambda[x;a].\mathsf{if}\ x=[\,]\ \mathsf{then}\ a\ \mathsf{else}\ F[tl(x);-(hd(x))+a])([x_1;-x_2+a])$$

Une simple règle de renommage est utilisée pour simplifier l'application. Nous obtenons le nouveau terme :

$$(\lambda a.\mathsf{if}\ \overline{x_1=[\,]}\ \mathsf{then}\ a\ \mathsf{else}\ F[tl(x_1);-(hd(x_1))+a])(\overline{-x_2}+a)$$

Nous pouvons remonter les sous-termes prêts. Le programme résultant est donné ci-dessous, à gauche. Dans ce terme, la sous-expression $(\lambda a.a)(x_3+a)$ peut être simplifiée en x_3+a. Après la première étape de compaction de la branche fausse, nous obtenons le terme donné à droite.

$$\begin{array}{l}
\mathsf{let}\ x_3=-x_2\\
\mathsf{in}\ \ \mathsf{if}\ x_1=[\,]\\
\quad\ \mathsf{then}\ (\lambda a.a)(x_3+a)\\
\quad\ \mathsf{else}\ (\lambda a.F[\overline{tl(x_1)};-(\overline{hd(x_1)})+a])(\overline{x_3+a})
\end{array}
\qquad
\begin{array}{l}
\mathsf{let}\ x_3=-x_2\\
\mathsf{in}\ \ \mathsf{if}\ x_1=[\,]\\
\quad\ \mathsf{then}\ x_3+a\\
\quad\ \mathsf{else}\ \ \mathsf{let}\ x_3=x_3+a\\
\qquad\qquad\quad x_2=tl(x_1)\\
\qquad\qquad\quad x_4=hd(x_1)\\
\qquad\quad\ \ \mathsf{in}\ (\lambda a.F[x_2;-x_4+a])x_3
\end{array}$$

La nouvelle expression à compacter est $(\lambda a.F[x_2;-x_4+a])x_3$. Par la règle de renommage, ce terme se simplifie en $F[x_2;-x_4+x_3]$. Nous avons déjà vu un terme semblable. Il est égal, modulo un renommage de ses variables libres, au terme $F[x1;-x2+a]$. Parce que l'on peut considérer que la compaction ne dépend pas des noms, elle produira le même résultat pour cette entrée, et le processus de compaction entre donc dans une boucle infinie. Pour arrêter la compaction, nous donnons un nom — *ff* — au terme et le remplaçons par un appel. Le programme final est le suivant.

$$\begin{array}{l}
\mathsf{let}\ \mathit{ff}=\mathsf{rec}_{\mathit{ff}}(\ \lambda[x_1;x_2;a].\\
\qquad\qquad\qquad\qquad \mathsf{let}\ x_3=-x_2\\
\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{if}\ x_1=[\,]\\
\qquad\qquad\qquad\qquad\quad\ \mathsf{then}\ x_3+a\\
\qquad\qquad\qquad\qquad\quad\ \mathsf{else}\ \ \mathsf{let}\ x_3=x_3+a\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad x_2=tl(x_1)\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad x_4=hd(x_1)\\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathsf{in}\ \mathit{ff}[x_2;x_4;x_3])\\
\mathsf{in}\ \lambda\,[x;a].\ \mathsf{if}\ x=[\,]\\
\qquad\qquad\ \ \mathsf{then}\ a\\
\qquad\qquad\ \ \mathsf{else}\ \ \mathsf{let}\ x_1=tl(x)\\
\qquad\qquad\qquad\qquad\quad x_2=hd(x)\\
\qquad\qquad\qquad\ \ \mathsf{in}\ \mathit{ff}[x_1;x_2;a]
\end{array}$$

Ce programme ne peut plus être compacté. C'est une version *pipeline* du programme initial, i.e. un dépliage supplémentaire suivi de la compaction produirait le même résultat puisque F a été dépliée dès que possible. Le calcul du corps de la boucle interne *ff* dure deux cycles au lieu de quatre.

Parce que *ff* est une fonction récursive terminale, l'appel récursif à *ff* peut être implanté par un branchement direct et les noms de variables peuvent être vus comme des noms de registres. Ainsi, l'allocation de registre peut être effectuée à la volée avec la compaction. Dans ce cas, les instructions de déplacements (instructions move) entre x_1 et x_2, x_2 et x_4, a et x_3 nécessaires avant le saut sont éliminées. Ceci dit, sur une machine VLIW, tous les move peuvent être exécutés en parallèle avec le saut, ce qui montre qu'il est très raisonnable de rendre équivalents deux termes égaux à un renommage prêt.

Cet exemple soulève les questions suivantes. Dans ce procédé, certaines transformations de programme ont été utilisées telles que le renommage, la remontée d'instructions, l'élimination de calculs inutiles. On peut donc se demander quelles sont les transformations *raisonnables* qu'un compilateur peut faire. Pourquoi le programme final est meilleur que le premier ? Est-ce que la compaction termine toujours et comment peut-elle être formalisée ? Nous allons tacher maintenant de répondre à ces questions.

4. Formalisation

Nous présentons une formalisation complète de la compaction. La première partie présente un langage récursif, appelé $\mathcal{F}$. Il sera vu comme un langage de bas niveau. Nous présentons ensuite le modèle de machine permettant d'exprimer qu'"un programme est meilleur qu'un autre pour une machine donnée". La définition proposée permet de caractériser à la fois les machines *séquentielles* et les machines *super-scalaires* ou *VLIW*. La partie suivante présente le modèle sémantique de la parallélisation à grain fin en définissant la notion de *dépendances* dans $\mathcal{F}$. Nous verrons pourquoi les dépendances sont définies intentionnellement par une relation d'équivalence appelée *équivalence structurelle*. La dernière partie est la définition du *pipeline logiciel* et du système permettant de le produire.

4.1. Le langage $\mathcal{F}$

Le langage $\mathcal{F}$ est un langage récursif avec une syntaxe à la ML mais où la granularité est proche de celle d'une machine, i.e, les constructions du langage (primitives, abstraction...) sont considérées comme atomiques.

Définition 1 (Le langage $\mathcal{F}$) *La définition du langage est la suivante.*

$$\begin{array}{rcl} a & ::= & i \mid \mathsf{true} \mid \mathsf{false} \mid x \mid \mathsf{let} x = a \;\mathsf{in}\; a \mid a[a; ...; a] \mid \lambda[x_1; ...; x_n].a \\ & & \mid op[a; ...; a] \mid \mathsf{rec}_f(a) \mid \mathsf{if}\; a \;\mathsf{then}\; a \;\mathsf{else}\; a \\ op & ::= & \mathsf{add\_int} \mid \ldots \end{array}$$

Les primitives (appelées op) sont celles de l'architecture. a désigne les termes. $\mathcal{F}$ permet la manipulation de scalaires (i), de variables (x), de définir des valeurs locales (let), des fonctions n-aires $\lambda[x_1;...;x_n].a$, des fonctions récursives $\mathsf{rec}_f(a)$, des primitives n-aires (op) et des applications $a[a_1;...;a_n]$. Tous les accès mémoires sont faits explicitement par des primitives et il n'y a pas d'*effet de bord* au sens où le langage a la propriété d'assignation unique (Static Single Assignment property) [CYT 89]. Dans cet article, nous notons $\vec{a}$ au lieu de $[a_1;...;a_n]$ et $\vec{a}|_i$ pour $[a_1;...;a_{i-1};a_{i+1};...;a_n]$.

$\mathcal{F}$ est un langage typé *à la ML* et nous supposons l'existence d'une fonction de typage — appelée *type* — retournant pour chaque expression de $\mathcal{F}$, son type pris dans le langage de type suivant pour lequel un algorithme de typage classique peut être utilisé [DAM 82].

$$t \quad ::= \quad \tau \mid t \rightarrow t \mid [t;...;t]$$

τ désigne les types scalaires (int , bool ,...). $t \rightarrow t$ est le type fonctionnel et $[t;...;t]$ est le type produit. L'information sur le type qui sera utilisée ici est sa complexité, c'est à dire, le nombre de flèches à gauche.

Définition 2 (Complexité du type)*Nous définissons la fonction ord, rendant la complexité de son argument.*

$$ord(t_1 \rightarrow t_2) = max(1 + ord(t_1), ord(t_2)) \quad ord(\tau) = 0$$
$$ord([t_1;...;t_n]) = max_{1 \leq i \leq n}(ord(t_i))$$

Par exemple, la complexité de $(\mathsf{int} \rightarrow \mathsf{int}) \rightarrow \mathsf{int}$ est 2 alors que la complexité de $\mathsf{int} \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$ est 1. Elle donne donc l'ordre de fonctionnalité. Nous avons besoin maintenant d'une définition préliminaire avant de définir la notion de sous-terme prêt.

Définition 3 (Coût)*Le coût d'un terme a est défini par la fonction* $||.||$*, rendant un élément de* $I\!N \cup \{\infty\}$.

$$||\mathsf{if}\ a_1\ \mathsf{then}\ a_2\ \mathsf{else}\ a_3|| = \sum_{i=1}^{3}(||a_i||) \qquad ||op[a_1;...;a_n]|| = 1 + \sum_{i=1}^{n}(||a_i||)$$

$$||\lambda\vec{x}.a|| = 1 \qquad ||\mathsf{let} x = a_1\ \mathsf{in}\ a_2|| = ||a_1|| + ||a_2||$$

$$||x|| = ||i|| = 0 \qquad ||a([b_1;...;b_n])|| = \infty \qquad ||\mathsf{rec}_f(a)|| = ||a||$$

Le coût est une approximation grossière du temps d'exécution d'une expression. Nous supposons que les calculs primitifs sont exécutés en un cycle et que le temps d'exécution des expressions contenant des appels de fonctions est infini. Le coût de l'abstraction est 1 car nous considérons que c'est le coût de la construction d'une fermeture [7]. Maintenant nous pouvons définir les sous-termes prêts.

---

[7] Une opération sur plusieurs cycles $op(x)$ peut être modélisée en la remplaçant par $op_1(op_2(...(op_n(x))...))$.

Définition 4 (Sous-termes prêts)*Soit a un terme de $\mathcal{F}$. $s \prec a$ signifie que le sous-terme s de a est prêt dans a.*

$$\frac{}{s \prec s} \qquad \frac{s \prec a_i}{s \prec op([a_1;...;a_n])} \qquad \frac{s \prec a \quad x_i \not\prec s \quad ||s|| < \infty}{s \prec \lambda[x_1;...;x_n].a}$$

$$\frac{s \prec \lambda f.a}{s \prec \mathsf{rec}_f(a)} \qquad \frac{s \prec a_1}{s \prec \mathsf{let} x = a_1 \;\mathsf{in}\; a_2} \qquad \frac{s \prec a_1}{s \prec \mathsf{if}\; a_1 \;\mathsf{then}\; a_2 \;\mathsf{else}\; a_3}$$

$$\frac{s \prec a_i}{s \prec a_0([a_1;...;a_n])} \qquad \frac{s \prec a_2 \quad x \not\prec s}{s \prec \mathsf{let} x = a_1 \;\mathsf{in}\; a_2} \qquad \frac{s \prec a_i \quad ||s|| < \infty}{s \prec \mathsf{if}\; a \;\mathsf{then}\; a_1 \;\mathsf{else}\; a_2}$$

Un sous-terme prêt est un sous-terme qui peut être exécuté immédiatement. Par exemple, $1 + x \prec \lambda y.(1 + x) * y$. La notion de sous-terme prêt est une extension des variables libres (une variable libre est toujours prête). Pour cette raison, nous utiliserons la notation classique $FV(a)$ définissant l'ensemble des variables libres de a, i.e, $x \in FV(a)$ ssi $x \prec a$. Les sous-termes prêts particulièrement intéressants ici seront ceux dont le coût est unitaire, autrement dit les calculs élémentaires.

Définition 5 (La substitution de sous-termes prêts)*Soit t tel que $t \prec a$. La substitution de toutes les occurrences de t dans a par b, noté $a[t\backslash b]$, est définie par :*

$$\begin{array}{rcll}
t[t\backslash b] & = & b & \\
x[t\backslash b] & = & x & \textit{si } x \neq t \\
(op[a_1;...;a_n])[t\backslash b] & = & op[[a_1[t\backslash b];...;a_n[t\backslash b]] & \\
(a_0[a_1;...;a_n])[t\backslash b] & = & (a_0[t\backslash b])([a_1[t\backslash b];...;a_n[t\backslash b]]) & \\
(\lambda x.a)[t\backslash b] & = & \lambda x.a & \textit{si } x \prec t \\
(\lambda x.a)[t\backslash b] & = & \lambda z.(a[x\backslash z][t\backslash b]) & \textit{sinon.} \\
 & & & z \not\prec a,\ z \not\prec b \\
(\mathsf{let} x = a_1 \;\mathsf{in}\; a_2)[t\backslash b] & = & \mathsf{let} x = a_1[t\backslash b] \;\mathsf{in}\; a_2 & \textit{si } x \prec t \\
(\mathsf{let} x = a_1 \;\mathsf{in}\; a_2)[t\backslash b] & = & \mathsf{let} z = a_1[t\backslash b] \;\mathsf{in}\; a_2[z\backslash x][t\backslash b] & \textit{sinon.} \\
 & & & z \not\prec a,\ z \not\prec b \\
\mathsf{rec}_f(a)[t\backslash b] & = & \mathsf{rec}_f(a) & \textit{si } f \prec t \\
\mathsf{rec}_f(a)[t\backslash b] & = & \mathsf{rec}_g(a[f\backslash g][t\backslash b]) & \textit{sinon.} \\
 & & & g \not\prec a,\ g \not\prec b \\
(\mathsf{if}\; a_1 \;\mathsf{then}\; a_2 \;\mathsf{else}\; a_3)[t\backslash b] & = & \mathsf{if}\; a_1[t\backslash b] \;\mathsf{then}\; a_2[t\backslash b] \;\mathsf{else}\; a_3[t\backslash b] &
\end{array}$$

La substitution des sous-termes prêts est très proche de la substitution classique de variables libres. Par exemple, $(\lambda x.(1 + y) * x)[1 + y\backslash y] = \lambda x.y * x$. La substitution est donc un moyen d'extraire — de remonter — des calculs d'une expression.

4.2. Sémantique opérationnelle

La sémantique opérationnelle a deux objectifs. Comme d'habitude, elle définit le lien entre des expressions et des valeurs en donnant un ordre d'évaluation.

Ici, elle va permettre également d'exprimer comment une architecture à grain fin consomme un programme donné. Elle donne ainsi une mesure sur les programmes.

Le langage a une évaluation par valeur suivant un ordre gauche-droite. Une machine peut être vue comme une fonction sélectionnant puis évaluant un sous-ensemble de sous-termes prêts. Cependant, tous les sous-termes prêts ne peuvent être sélectionnés. C'est le cas, par exemple, des sous-termes prêts à droite d'un appel de fonction. En effet, une machine exécute du code *en ligne* sans connaissance de la structure du programme : elle suit un seul chemin d'exécution (en fait, on pourrait dire qu'elle n'exécute qu'un processus). Pour exécuter en avance une instruction suivant un saut, le contrôleur devrait alors disposer d'une pile interne non bornée où stocker l'effet de l'instruction jusqu'à la fin de l'exécution de ce saut, ce qui est irréaliste. Prenons un exemple : dans le terme $f(x+1)+(2*y)$ dont la version plus "trois-adresses" serait :

$$\begin{array}{l} \mathsf{let}\, x_1 = x+1 \\ \mathsf{in}\;\; \mathsf{let}\, x_1 = f(x_1) \\ \quad\;\; \mathsf{in}\;\; \mathsf{let}\, x_2 = \overline{2*y} \\ \quad\quad\;\; \mathsf{in}\;\; x_1 + x2 \end{array}$$

le terme surligné est prêt mais ne peut être exécuté avant l'appel de fonction, en parallèle avec $x+1$, par exemple. Une machine super-scalaire pourra éventuellement lire les instructions du code de la fonction f mais pas le code qui la suit. On voit immédiatemment l'intérêt de la compaction, même pour les architectures super-scalaires idéales : la compaction statique permet des transformations difficiles sinon impossibles à faire dynamiquement.

Définition 6 (Occurrence et profondeur)*L'ensemble des occurrences (chemins d'accès), $\mathcal{O}(a)$ d'un terme a contient des mots sur des entiers (Λ désigne le mot vide). Il est défini par :*

$\Lambda \in \mathcal{O}(a)$ et pour toutes les constructions C du langage,
$i.o_i \in \mathcal{O}(C(a_1, ..., a_n))$ si $o_i \in \mathcal{O}(a_i)$

La profondeur d'une occurrence dans un terme est telle que :

$$\mathcal{D}_{\mathsf{let}\, x=a\, \mathsf{in}\, b}(\Lambda) = 0 \qquad \mathcal{D}_{\mathsf{if}\, a\, \mathsf{then}\, b\, \mathsf{else}\, c}(\Lambda) = ||a|| \qquad \mathcal{D}_{\mathsf{true}}(\Lambda) = 0$$

$$\mathcal{D}_{op[a_1;\ldots;a_n]}(\Lambda) = \sum_{i=1}^{n} ||a_i|| \quad \mathcal{D}_{a_1[a_2;\ldots;a_n]}(\Lambda) = \sum_{i=1}^{n} ||a_i|| \quad \mathcal{D}_{\mathsf{false}}(\Lambda) = 0$$

$$\mathcal{D}_x(\Lambda) = 0 \qquad \mathcal{D}_i(\Lambda) = 0 \qquad \mathcal{D}_{\lambda\vec{x}.a}(\Lambda) = 0 \qquad \mathcal{D}_{\mathsf{rec} f.a}(\Lambda) = 0$$

$$\mathcal{D}_{a_1[a_2;\ldots;a_n]}(1.o) = \sum_{i=2}^{n} ||a_i|| + \mathcal{D}_{a_1}(o) \textit{ pour l'application}$$

$$\mathcal{D}_{C(a_1,\ldots,a_n)}(i.o) = \mathcal{D}_{a_i}(o) + \sum_{j<i} ||a_j|| \textit{ pour les autres construction } C$$

Par exemple, le sous-terme $(1+y)$ de $(\lambda x.x*(1+y))(2+z)$ est à l'occurrence $1.1.2.\Lambda$ et le sous-terme $(2+x)$ de $(\mathsf{if}\ x=0\ \mathsf{then}\ 1+y\ \mathsf{else}\ (2+y)*x)+(2+x)$ est à l'occurrence $2.\Lambda$. La *profondeur* d'une occurrence est le nombre d'instructions à lire jusqu'à elle (elle exceptée), dans l'ordre d'évaluation gauche-droite. Ainsi, la profondeur de $(1+y)$ est 1 parce que l'argument est exécuté avant le corps de la fonction. La profondeur de $(2+x)$ est ici 4 et non 2. En effet, le contrôleur du processeur ne peut décider de remonter l'opération que s'il dispose du code complet des deux branches. En particulier, si l'une des branches contient un appel de fonction, l'exécution spéculative est impossible. La profondeur d'un sous-terme à droite d'une application est toujours égal à ∞. Cela va nous permettre de rendre ces sous-termes prêts inatteignables par la machine.

Lorsque ce sera possible, l'occurrence sera omise et la notation $\mathcal{D}_a(t)$ sera préférée à $\mathcal{D}_a(o)$ si o est l'occurrence du sous-terme t dans a. On dit que deux sous-termes ont des occurrences disjointes si aucune n'est préfixe de l'autre. Un sous-terme d'un terme est strict s'il est différent du terme lui-même.

Une machine peut être vue comme une sous-relation de $\prec$: seul un sous-ensemble des sous-termes prêts lui est accessible.

Définition 7 (Machine $\mathcal{M}$) *Une machine $\mathcal{M}$ est une fonction des termes de $\mathcal{F}$ dans les ensembles de termes de $\mathcal{F}$. $\mathcal{M}$ est paramétrée par deux constantes entières $\mathcal{M}.h$ et $\mathcal{M}.w$. $\mathcal{M}$ vérifie les deux conditions suivantes :*

$\mathcal{M}$ est bornée. *On a $\mathcal{M}(a) = \{t_1, ..., t_n\}$ où les t_i sont des sous-termes disjoints et stricts de a tels que $1 < n \leq \mathcal{M}.w$, $D_a(t_i) \leq \mathcal{M}.h$ et $t_i \prec a$.*

$\mathcal{M}$ est d'état fini. *Il existe une fonction base telle que pour tout a, $\mathcal{M}(a) = base(t_1, ..., t_n)$ où les t_i sont les sous-termes de a vérifiant $D_a(t_i) \leq \mathcal{M}.h$*

Le nombre de sous-termes prêts sélectionnés est borné par le nombre d'unités fonctionnelles de l'architecture ($\mathcal{M}.w$). La profondeur des sous-termes sélectionnés doit être bornée par la taille de la fenêtre d'analyse ($\mathcal{M}.h$). $\mathcal{M}$ doit être d'état fini, ce qui est obtenu en imposant que ses choix ne dépendent que d'un ensemble borné (connu à priori) de sous-termes de l'entrée. Notons ici que la fenêtre d'analyse contient au plus un appel de fonction (un branchement inconditionnel) qui est nécessairement à droite des instructions [8]. Une machine *super-scalaire* vérifie cette définition. Une machine *séquentielle* est un cas particulier où $\mathcal{M}.w = \mathcal{M}.h = 1$. La définition donnée est suffisante pour autoriser ou non l'exécution spéculative. En effet, $\mathcal{M}$ est typiquement définie par cas suivant les formes possibles de termes. Ainsi, une machine n'autorisant pas l'exécution spéculative s'obtient en interdisant la sélection des sous-termes prêts apparaissant sous une conditionnelle ou une abstraction. Enfin, une machine *VLIW* est aussi un cas particulier de machine. En effet :

Exemple 1 (Machine VLIW) *Une machine VLIW est une machine $\mathcal{M}$ maximale telle que : si $t \in \mathcal{M}(a)$ et $\mathcal{D}_a(t) = n$ alors $\forall t', \mathcal{D}_a(t') < n \Rightarrow t' \in \mathcal{M}(a)$*

Ici, le préfixe maximal d'instructions indépendantes d'une fenêtre d'instructions séquentielles peut être exécuté en parallèle. Nous appelons cela

[8] Selon la définition, la profondeur d'un appel de fonction est finie si cet appel n'est précédé d'aucun autre appel.

une machine VLIW, même si elle ne suit pas le principe initial — caractérisé par les machines Multiflow [MUL 87] — car aucun ré-ordonnancement dynamique n'est fait par la machine.

Les machines pipelines entrent dans cette description si le compilateur transforme une instruction pipeline $op(x)$ en $\mathsf{nop}(...(\mathsf{nop}(op(x)))...)$. Dans ce cas, toutes les opérations vides nop sont toujours exécutées en parallèle avec une instruction non vide.

Exemple 2 (Machine pipeline) *Une machine pipeline est une machine $\mathcal{M}$ telle que : si $t \in \mathcal{M}(a)$ et $t \neq \mathsf{nop}(y)$ alors $\mathsf{nop}(x) \prec a \Rightarrow \mathsf{nop}(x) \in \mathcal{M}(a)$.*

Une machine sera donc définie par un automate lisant une partie finie du terme. Regardons maintenant la sémantique opérationnelle du langage $\mathcal{F}$ définie d'une manière structurelle [PLO 81].

À chaque étape du calcul, la machine sélectionne un sous-ensemble des sous-termes prêts pouvant être exécutés en parallèle, les exécute puis substitue les sous-termes prêts par leurs valeurs. Malheureusement, à cause de la représentation du programme, un terme de $\mathcal{F}$ contient du *bruit*, c'est à dire, des constructions syntaxiques ne générant aucun calcul (en fait, elles ne correspondent à aucune instruction d'assemblage). Par exemple, le let représente des *dags* et l'expression $\mathsf{let}x = 1 \,\mathsf{in}\, x+x$ a le même temps d'exécution que $1+1$. Nous définissons donc une règle de simplification.

Définition 8 (ϵ-transitions) *Nous définissons l'égalité $=_\mathcal{M}$ entre termes. Elle est appliquée dans tous les contextes $C(a)$ où a est visible pour $\mathcal{M}$, i.e, la profondeur de a est inférieure à $\mathcal{M}.h$.*

$$
\begin{array}{rcl}
\mathsf{if\ true\ then}\ a_2\ \mathsf{else}\ a_3 & =_\mathcal{M} & a_2 \\
\mathsf{if\ false\ then}\ a_2\ \mathsf{else}\ a_3 & =_\mathcal{M} & a_3 \\
\mathsf{let}x = y\ \mathsf{in}\ a & =_\mathcal{M} & a[x\backslash y] \\
\mathsf{let}x = i\ \mathsf{in}\ a & =_\mathcal{M} & a[x\backslash i] \\
\mathsf{let}x = \lambda y.a_1\ \mathsf{in}\ a_2 & =_\mathcal{M} & a_2[x\backslash \lambda y.a_1] \\
(\lambda\vec{x}.a)[b_1;...;j;...;b_n]) & =_\mathcal{M} & (\lambda\vec{x}|_i.a[x_i\backslash j])[b_1;...;;...;b_n]) \\
(\lambda\vec{x}.a)[b_1;...;y_i;...;b_n]) & =_\mathcal{M} & (\lambda\vec{x}|_i.a[x_i\backslash y_i])[b_1;...;;...;b_n]) \\
 & & si\ ord(type(x_i)) = 0 \\
\mathsf{rec}_f(\lambda\vec{x}.a) & =_\mathcal{M} & \lambda\vec{x}.a[f\backslash \mathsf{rec}_f(\lambda\vec{x}.a)]
\end{array}
$$

La contrainte sur la complexité du type pour la réduction de l'application sera expliquée plus loin. Cette égalité dit que les constructions sont invisibles pour une machine $\mathcal{M}$. Ici, seules les sous-expressions atteignables — leur profondeur est inférieure à $\mathcal{M}.h$ — sont réduites : l'égalité n'est pas appliquée dans tous les contextes. Ceci est dû à la règle de dépliage des récursions. En effet, sans l'utilisation de $\mathcal{M}$, nous pourrions avoir la séquence infinie suivante :
$F[x;y], ..., F[x;op(y)], ..., (\lambda y.F[x;op(y)])(op(y)), ...$
où $F = \mathsf{rec}_f(\lambda[x;y].f[x;op(y)])$.
Dans notre définition, les calculs vides sont éliminés lorsque c'est nécessaire.

Définition 9 (Sémantique opérationnelle) *La sémantique opérationnelle d'un programme sur une machine $\mathcal{M}$ est donnée par la relation $\Downarrow$. $a \Downarrow b$ signifie que a s'évalue en b. La relation $\Downarrow$ est construite à partir de la relation d'évaluation instantanée $\triangleright$. Les règles sont considérées modulo $=_{\mathcal{M}}$.*

$$\frac{\mathcal{M}(a) = \{t_1, ..., t_n\} \quad t_i \triangleright v_i \quad a[t_1 \backslash v_1]...[t_n \backslash v_n] \Downarrow b}{a \Downarrow b}$$

Les axiomes de $\Downarrow$ sont:

$$i \Downarrow i \quad \lambda\vec{x}.a \Downarrow \lambda\vec{x}.a \quad \mathsf{rec} f.(\lambda\vec{x}.a) \Downarrow \mathsf{rec} f.(\lambda\vec{x}.a)$$

La relation de réduction instantanée est telle que:

$$\frac{a \triangleright v_1 \quad b \triangleright v_2 \quad v_1 + v_2 = v}{a + b \triangleright v} \qquad \frac{a \triangleright \lambda\vec{x}.c \quad b_i \triangleright v_i}{a[b_1; ...; b_n] \triangleright c[x_1 \backslash v_1]..[x_n \backslash v_n]}$$

$$\frac{a \triangleright v_1 \quad b[x \backslash v_1] \triangleright v_2}{\mathsf{let} x = a \;\mathsf{in}\; b \triangleright v_2} \quad \frac{a \triangleright \mathsf{true} \quad b \triangleright d}{\mathsf{if}\; a \;\mathsf{then}\; b \;\mathsf{else}\; c \triangleright d} \quad \frac{a \triangleright \mathsf{false} \quad c \triangleright d}{\mathsf{if}\; a \;\mathsf{then}\; b \;\mathsf{else}\; c \triangleright d}$$

Les axiomes de $\triangleright$ sont:

$$i \triangleright i \quad \lambda\vec{x}.a \triangleright \lambda\vec{x}.a \quad \mathsf{rec} f.(\lambda\vec{x}.a) \triangleright \mathsf{rec} f.(\lambda\vec{x}.a)$$

Le temps d'exécution $|a|_{\mathcal{M}}$ de a est la hauteur de la preuve de $a \Downarrow v$.

Pour simplifier, nous ne donnons que la règle d'évaluation de l'addition. À chaque étape, la machine sélectionne un sous-ensemble des sous-termes prêts, les évalue en parallèle (relation de réduction instantanée $\triangleright$) puis continue avec le reste du programme. On peut noter ici que la contrainte sur $\mathcal{M}$ va garantir que jamais plus d'un flot de contrôle ne sera suivi (l'ensemble des sous-termes sélectionnés contenant au plus une application, une seule beta-réduction au plus est effectuée par $\triangleright$).

4.3. Sémantique des dépendances

Dans le second exemple introductif, nous avons utilisé des transformations de programmes telles que le renommage ou le dépliage. On peut donc se demander quelles sont les transformations qui doivent être intégrées à un compilateur compactant. Dans le cadre classique, seules les transformations préservant les *dépendances* sont autorisées. On dit qu'il y a une dépendance entre deux instructions si l'une modifie une valeur lue par une autre. Ces dépendances sont décomposées en *vraies* dépendances — ou *dépendances de données* — lorsque la seconde instruction lit la valeur produite par la première, et les *fausses* dépendances — *anti-dépendances* et *dépendances de sortie* — lorsque la seconde modifie une valeur lue ou modifiée par la première [CYT 87].

Parce que $\mathcal{F}$ est un langage fonctionnel, les dépendances au sens habituel en tant que liens *définition-utilisation* sont directement données dans le texte du programme par les variables et la composition des calculs. Ainsi, dans $\mathsf{let}\, x = a \,\mathsf{in}\, b$, x dépend de a — et de tous ses sous-termes — et chaque sous-terme de b contenant x en dépend. La fermeture transitive de cette relation exprime la notion classique de graphe statique de dépendances. Il est donc inutile de définir la notion de graphe de dépendance d'un terme de $\mathcal{F}$. Cependant, nous devons accepter certaines transformations syntaxiques.

Nous préférons donc définir les dépendances par une équivalence entre "programmes ayant les mêmes dépendances". Cette équivalence est appelée *équivalence structurelle.* On peut alors se demander comment cette équivalence interagit avec la β-réduction[9]. D'un côté, deux programmes peuvent être extensionnellement (beta) équivalents et il est clairement impossible (dans un compilateur) d'autoriser tous les types de β-réductions. De plus, nous aurions alors des problèmes de cohérence puisque $1 + 2$ n'est pas équivalent à 3 alors qu'une traduction en λ-calcul pur les rendraient équivalents. D'un autre côté, nous ne pouvons limiter l'équivalence à l'égalité syntaxique. Il est raisonnable d'autoriser le déplacement d'instructions, le dépliage, la duplication[10] qui sont les transformations nécessaires minimales pour la compaction. Le choix adopté ici est de se limiter aux transformations de premier ordre : seules les β-réductions où le type de l'argument est simple (la complexité est 0) seront autorisées.

Définition 10 (Équivalence structurelle)*Soit a et b, deux termes de $\mathcal{F}$. On dit que a et b sont structurellement équivalents, noté $a \sim b$, lorsque $lim_{n\to\infty} a^n = lim_{n\to\infty} b^n$ où a^n est défini par $a \to a^2 \to \ldots \to a^n$ et par la réécriture $\to$ appliquée à tous les contextes. $\to$ est définie par :*

$$
\begin{array}{rcl}
\mathsf{let}\, x = a_1 \,\mathsf{in}\, a_2 & \to & a_2[x \backslash a_1] \\
\mathsf{if}\, a \,\mathsf{then}\, [b_1; \ldots; b_n] \,\mathsf{else}\, [c_1; \ldots; c_n] & \to & [\mathsf{if}\, a \,\mathsf{then}\, b_1 \,\mathsf{else}\, c_1; \ldots; \mathsf{if}\, a \,\mathsf{then}\, b_n \,\mathsf{else}\, c_n] \\
\mathsf{if}\,\mathsf{true}\,\mathsf{then}\, a_1 \,\mathsf{else}\, a_2 & \to & a_1 \\
\mathsf{if}\,\mathsf{false}\,\mathsf{then}\, a_1 \,\mathsf{else}\, a_2 & \to & a_2 \\
(\lambda[x_1; \overset{x_i}{\overset{\vee}{\ldots}}; x_n].a)[b_1; \overset{b_i}{\overset{\vee}{\ldots}}; b_n] & \to & (\lambda[x_1; \overset{x_i}{\overset{\wedge}{\ldots}}; x_n].a[x_i \backslash b_i])[b_1; \overset{b_i}{\overset{\wedge}{\ldots}}; b_n] \\
 & & si\ ord(type(b_i)) = 0 \\
\mathsf{rec}_f(a) & \to & a[f \backslash \mathsf{rec}_f(a)]
\end{array}
$$

La première règle permet la duplication d'un calcul. La deuxième suit le même principe pour la conditionnelle. Les deux règles suivantes sont des simplifications. La règle suivante est la β-réduction lorsque le type de l'argument b_i est simple.

[9] la β-réduction est la règle suivante $(\lambda x.a_1)a_2 \to_\beta a_1[x \backslash a_2]$.

[10] La duplication est utile lorsqu'il est moins coûteux de recalculer une valeur plutôt que de la communiquer.

4.4. Compaction

Nous avons vu dans les exemples que la compaction pouvait être obtenue par la sélection et la remontée de sous-termes prêts. Nous allons d'abord définir le déplacement d'instructions par deux règles simples puis définir comment sélectionner un sous-ensemble de sous-termes prêts.

Définition 11 (Move-up) *Les déplacements d'instructions sont définis par la relation $a \xrightarrow{t} b$ où t est prêt dans a, tels que :*

(MOVE) $a \xrightarrow{t} \mathsf{let} x = t \mathsf{~in~} a[t \backslash x]$ *si* $x \not\prec a$

(TEST) $a \xrightarrow{t} \mathsf{if~} t \mathsf{~then~} a[t\backslash\mathsf{true}] \mathsf{~else~} a[t\backslash\mathsf{false}]$ *si* ($\mathsf{if~} t \mathsf{~then~} a_1 \mathsf{~else~} b_1$) *est un sous-terme de* a

Les déplacements d'instructions sont de simples β-réductions inverses. Maintenant, pour sélectionner un sous-ensemble des sous-termes prêts pouvant être exécutés en parallèle sur une machine, le procédé de compaction a besoin d'une *stratégie.* Par exemple, la stratégie classique de *List-scheduling* [GAR 79] sélectionne les sous-termes prêts appartenant au plus long chemin de dépendances.

Définition 12 (Stratégie) *Une stratégie $\mathcal{S}$ est une fonction $\mathcal{S}$ entre termes et ensemble de termes telle que $\mathcal{S}(a) = \{t_1, ..., t_n\}$ avec $n \leq 1$ et où les t_i sont des sous-termes disjoints et stricts de a tels que $t_i \prec a$.*

Une stratégie n'est pas très différente d'une machine si ce n'est que — pour l'instant — il n'y a pas de contrainte (en particulier de type contrôle) sur la position des sous-termes prêts dans le terme. Pour cette raison, nous pouvons manipuler $=_{\mathcal{S}}$ en remplaçant $\mathcal{M}$ par $\mathcal{S}$.

Étudions maintenant la compaction. Elle est très proche de la sémantique opérationnelle : à chaque étape, certains calculs sont sélectionnés et remontés dans le programme. Le reste du programme est alors ensuite compacté.

Définition 13 (Compaction) *Soit ρ un renommage. Les termes sont considérés modulo $=_{\mathcal{S}}$. La compaction générale est définie par le prédicat $e \models_{\mathcal{S}} a \Rightarrow_l b$ donné ci-dessus, signifiant que dans l'environnement e, le terme a est compacté en b en utilisant la stratégie $\mathcal{S}$ et en remontant les sous-termes prêts sélectionnés de l. Un environnement e vérifie :*

$$e ::= [f_1 \backslash \lambda \vec{v_1}.a_1, ..., f_n \backslash \lambda \vec{v_n}.a_n]$$

C'est un ensemble d'associations noms – expressions. Il sert à mémoriser les termes rencontrés au cours de la compaction.

La règle (SELECT) sélectionne un sous-ensemble de sous-termes prêts de a. Elle mémorise le terme courant a dans l'environnement e et compacte a avec les sous-termes prêts sélectionnés. Le programme résultant peut être récursif. Bien entendu, si la compaction de a n'utilise pas f, alors le terme $(\mathsf{rec}_f(\lambda v.b))v$

$$\text{(SELECT)}\quad \frac{\mathcal{S}(a) = l \quad FV(a) = [x_1; ...; x_n] \quad e[f\backslash\lambda[x_1; ...; x_n].a] \models_{\mathcal{S}} a \Rightarrow_l b}{e \models_{\mathcal{S}} a \Rightarrow (\mathsf{rec}_f(\lambda[x_1; ...; x_n].b))[x_1; ...; x_n]}$$

$$\text{(EQUIV)}\quad \frac{b = \rho(a) \quad \vec{v}' = \rho(\vec{v})}{e[f\backslash\lambda\vec{v}.a] \models_{\mathcal{S}} b \Rightarrow f(\vec{v}')} \qquad \text{(LAMBDA)}\quad \frac{e \models_{\mathcal{S}} a \Rightarrow a'}{e \models_{\mathcal{S}} \lambda\vec{x}.a \Rightarrow \lambda\vec{x}.a'}$$

$$\text{(MOVE)}\quad \frac{a \xrightarrow{t} \mathsf{let} x = t \;\mathsf{in}\; a_1 \quad e \models_{\mathcal{S}} a_1 \Rightarrow_l b_1}{e \models_{\mathcal{S}} a \Rightarrow_{\{t\}\cup l} \mathsf{let} x = t \;\mathsf{in}\; b_1} \qquad \text{(EMPTY)}\quad \frac{t \not\to a \quad e \models_{\mathcal{S}} a \Rightarrow_l b}{e \models_{\mathcal{S}} a \Rightarrow_{t\cup l} b}$$

$$\text{(PRIM)}\quad e \models_{\mathcal{S}} op(\vec{x}) \Rightarrow op(\vec{x}) \qquad \text{(CONST)}\quad e \models_{\mathcal{S}} i \Rightarrow i$$

$$\text{(APP)}\quad e \models_{\mathcal{S}} y(\vec{x}) \Rightarrow y(\vec{x}) \qquad \text{(VAR)}\quad e \models_{\mathcal{S}} x \Rightarrow x$$

$$\text{(TEST)}\quad \frac{a \xrightarrow{t} \mathsf{if}\; t \;\mathsf{then}\; a_1 \;\mathsf{else}\; a_2 \quad e \models_{\mathcal{S}} a_1 \Rightarrow_l b_1 \quad e \models_{\mathcal{S}} a_2 \Rightarrow_l b_2}{e \models_{\mathcal{S}} a \Rightarrow_{\{t\}\cup l} \mathsf{if}\; t \;\mathsf{then}\; b_1 \;\mathsf{else}\; b_2}$$

se simplifie en b. La règle (EQUIV) permet d'arrêter la compaction lorsque le terme courant est équivalent par un renommage ρ à un terme déjà rencontré. Dans ce cas, la compaction du terme courant est un appel de fonction. La règle (EMPTY) enlève de la liste un sous-terme sélectionné qui n'est pas prêt dans a. La règle (LAMBDA) est une règle compositionnelle. Les règles (PRIM), (APP), (CONST) et (VAR) sont les axiomes. La règle (MOVE) remonte un sous-terme prêt sélectionné et la règle (TEST) remonte un test.

Sans contraintes sur $\mathcal{S}$, le système construit ne termine clairement pas. Nous l'illustrons d'abord sur le cas d'une fonction récursive terminale. Considérons l'expression $F = \mathsf{rec}_f(\lambda[x;y].f[op_1(x);op_2(op_3(y))])$ et $F([x;y])$. La suite de termes à compacter est la suivante :

$$F[x;y], \ldots, F[x;op_2(y)], ..., F[x;op_2(op_3(y))], ..., F[x;op_2(op_3(op_2(y)))]...$$

$F[x;y]$ est déplié et les deux sous-termes prêts $op_1(x)$ et $op_3(y)$ sont substitués. Ensuite, le terme courant à compacter est $F[x;op_2(y)]$, etc. Ceci est un problème très classique des méthodes itératives pour la compaction de programmes [AIK 88] : le problème vient du fait que des termes de plus en plus profonds sont remontés. Dans le cas de fonctions récursives terminales, e.g. de boucles, nous allons voir que le problème de la compaction est plus simple que dans le cas général et des conditions simples et satisfaisantes sur $\mathcal{S}$ peuvent être adoptées. Nous reviendrons sur le cas des récursions générales à la fin de l'article.

Contrainte 1 (Récursions terminales) *Un terme a est terminal si tout sous-terme* $\mathsf{rec}_f(b)$ *de* a *est tel que* f *apparaît en position terminale, noté* $\mathcal{T}(f, b)$.

$\mathcal{T}(f, \mathsf{if}\, a\, \mathsf{then}\, a_1\, \mathsf{else}\, a_2)$ *si* $f \notin FV(a)$, $\mathcal{T}(f, a_1)$ *et* $\mathcal{T}(f, a_2)$

$\mathcal{T}(f, f(a))$ *si* $f \notin FV(a)$
$\mathcal{T}(f, \mathsf{let} x = a\, \mathsf{in}\, b)$ *si* $f \notin FV(a)$ *et* $\mathcal{T}(f, b)$
$\mathcal{T}(f, \lambda x.a)$ *si* $\mathcal{T}(f, a)$
$\mathcal{T}(f, a(b))$ *si* $\mathcal{T}(f, a)$ *et* $f \notin FV(b)$
$\mathcal{T}(f, a)$ *si* $f \notin FV(a)$ *pour les autres constructions*

La définition des fonctions récursives terminales dit que les appels récursifs à f ne sont jamais suivis d'autres calculs. Maintenant, nous pouvons définir une contrainte sur la stratégie permettant de garantir la terminaison de la compaction.

Contrainte 2 (Stratégie bornée) *Soit* n_a *le nombre de sous-termes prêts de* a. *Une stratégie* $\mathcal{S}$ *bornée par* k *est une stratégie telle que si* $n_a > k$ *alors pour tout* t, $t \in \mathcal{S}(a)$ *ssi* $t \in \mathcal{S}(b)$ *où* $a \rightarrow b$.

Une stratégie bornée limite le nombre de sous-termes prêts sélectionnables. L'implication $t \in \mathcal{S}(a) \Rightarrow t \in \mathcal{S}(b)$ est raisonnable : elle signifie, par exemple, que les instructions dans une boucle ont une priorité sur les opérations appartenant à l'itération suivante. L'autre contrainte est moins naturelle : elle signifie que même si un dépliage fait apparaître un nouveau sous-terme prêt il ne peut être sélectionné si la taille du terme est plus grande qu'une certaine limite. Cette condition n'est pas si surprenante. En effet, considérons un terme $F(a)$ où F désigne une fonction récursive. Un argument possible pour la terminaison peut être de limiter le dépliage de la récursion au cas où la taille de l'argument est bornée. On retrouve ce type d'argument, exprimé d'une manière différente, dans plusieurs algorithmes de pipeline logiciel des boucles [BOD 89, AIK 91].

Cette contrainte n'est pas incompatible avec la stratégie classique de *List-scheduling*. En effet, cette stratégie sélectionne en priorité les sous-termes prêts qui sont le long du plus long chemin. La contrainte proposée ici signifie que seuls les chemins de taille bornée *a priori* sont considérés.

Proposition 1 (Terminaison) *Le système de compaction* $\models_{\mathcal{S}}$ *termine avec une stratégie bornée.*

Le résultat est basé sur le fait que le nombre de termes différents de taille bornée est fini. Au pire, après énumération de l'ensemble de ces termes, la compaction rencontre à nouveau le même terme. Cependant, comme pour la plupart des méthodes itératives, la terminaison peut être obtenu au prix d'une très forte complexité. Il reste donc à étudier des stratégies de dépliage (d'application des règles $\rightarrow$) permettant une convergence plus rapide. Ceci n'est pas abordé ici.

Proposition 2 (Correction)*La compaction préserve l'équivalence structurelle.*

Cette propriété vient immédiatemment du fait que les règles de déplacement sont autorisées par l'équivalence structurelle.

Nous devons prouver maintenant que ce système est utile, autrement dit qu'il diminue le temps d'exécution des programmes. Ceci ne peut être fait pour les machines générales parce que le problème de l'ordonnancement est un problème NP-difficile [GAR 79] et ne peut donc être obtenu par un algorithme glouton. Cependant, l'accélération peut être garantie pour les machines de type VLIW et une stratégie sélectionnant toujours au moins les sous-termes prêts sélectionnés par la machine.

Proposition 3 (Accélération)*Soit $\mathcal{M}$ une machine VLIW et $\mathcal{S}$ tel que pour tout a, il existe c tel que $\mathcal{M}(a) \subseteq \mathcal{S}(a) \subseteq \mathcal{M}(c)$. Alors si $\models_{\mathcal{S}} a \Rightarrow b$ alors $|a|_{\mathcal{M}} \geq |b|_{\mathcal{M}}$*

Le résultat s'applique aux machines VLIW parce que la profondeur des instructions dans le programme n'augmente jamais : une instruction sélectionnée par la machine dans le programme initial est encore sélectionnée dans le second (sauf si elle a déjà été exécutée). Le résultat se prouve par récurrence structurelle en utilisant la propriété suivante : si $t \in \mathcal{M}(a)$ et $a \xrightarrow{t} b$ alors $t \in \mathcal{M}(b)$.

Le programme produit par la compaction est une version pipeline du programme initial. En effet, le pipeline logiciel d'un programme récursif peut être vu comme une représentation finie du programme déplié infiniment et compacté. C'est un point fixe de la compaction : la compaction de la version dépliée produirait le même programme. Nous pouvons maintenant définir le pipeline logiciel à partir de la notion d'optimalité.

Définition 14 (Pipeline logiciel)*Soit a un terme de $\mathcal{F}$. Un programme a est optimal pour $\mathcal{S}$ si $\models_{\mathcal{S}} b \Rightarrow a$ où $a \rightarrow b$. Un programme a' est un pipeline logiciel de a par $\mathcal{S}$ lorsque $a \sim a'$ et a' est optimal.*

Proposition 4 (Optimalité)*$\models_{\mathcal{S}}$ construit un pipeline logiciel pour les récursions terminales.*

Le résultat est une conséquence de la contrainte sur la stratégie. Même si le programme est déplié, la stratégie sélectionnera les mêmes sous-termes. La compaction produira donc le même programme. Ce résultat est la version fonctionnelle du résultat de [AIK 91], qui est exprimé dans le cadre d'un langage de flots de contrôle.

Le système proposé est un système glouton et n'est pas adapté au traitement des récursions générales. En effet, considérons l'expression très simple suivante $F = \mathsf{rec}_f(\lambda\vec{x}.A(\vec{x}, f(op(\vec{x}))))$ où A désigne une expression de F. Nous avons la liste suivante de termes à compacter : $F(\vec{x_1})$, $A(\vec{x_1}, F(\vec{x_2}))$, $A(\vec{x_1}, A(\vec{x_2}, F(\vec{x_3})))$, ... La compaction lit seulement le début du terme $(op(x_i))$. Ici, le terme grossit après l'appel récursif. Même si la compaction est arrêtée en utilisant des conditions sur $\mathcal{S}$, nous ne disposons pas de règle de repliage pour la production du pipeline logiciel. Ceci reste à étudier.

5. Conclusion et travaux futurs

Nous avons présenté une méthode pour compacter directement la partie de premier ordre de programmes fonctionnels. Pour les programmes où les fonctions apparaissent en position terminale, le pipeline logiciel peut être construit. La compaction est définie par un ensemble de transformations de programmes. Une sémantique opérationnelle a été proposée pour modéliser le comportement d'une machine parallèle à grain fin. Une notion de dépendances a été définie par une *équivalence structurelle* : elle définit les transformations légales nécessaires pour la parallélisation. Cet article peut être vu comme une formalisation de méthodes de compaction classiques mais aussi comme un début d'extension à une classe plus générale de programmes. En appliquant la compaction à des *termes* plutôt qu'à des représentations graphiques, le compilateur peut bénéficier d'informations sémantiques disponibles dans le programme source (pour l'analyse de dépendances, notamment). Parce que cette représentation est plus générale que la représentation sous forme de graphe de dépendances — celle-ci peut être obtenue par partage — elle peut être utilisée pour représenter les dépendances d'une large famille de langages.

Certaines extensions restent à faire. Tout d'abord, une meilleure stratégie peut améliorer la convergence puisque les méthodes itératives sont réputées lentes et ont tendance à produire de gros programmes. Ensuite, les caractéristiques impératives pourraient être traitées en modifiant légèrement la définition de *sous-termes prêts*. Il serait également intéressant (et utile) de tenir compte de propriétés sémantiques d'opérateurs. Ceci permettrait non seulement d'augmenter le parallélisme mais aussi d'augmenter le nombre de termes équivalents (et de réduire ainsi la taille du code produit). Ceci pourrait être fait par l'ajout d'autres règles de simplification (règles d'associativité ou de commutativité,...). C'est certainement pour ce type d'extension que la représentation fonctionnelle est meilleure que les représentations graphiques classiques car elle facilite les transformations de programmes.

Remerciements

Tous mes remerciements aux lecteurs attentifs Paul Caspi de VERIMAG, Thérèse Hardin et Bernard Serpette de l'INRIA de Rocquencourt. Je tiens également à remercier Laurie Hendren pour son acceuil au sein du laboratoire ACAPS de l'Université McGILL de Montréal. Le travail présenté ici a été financé par l'INRIA.

6. Bibliographie

[AIK 88]Alexander AIKEN. *Compaction-based Parallelization.* PhD thesis, Cornell University, 1988.

[FIS 84]J. A. FISHER and A. NICOLAU. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, C-33(11), November 1984.

[AIK 91]Alexander AIKEN and Alex NICOLAU. A realistic resource-constrained software pipelining algorithm. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advance in Languages and Compilers for Parallel Processing*, 1991.

[BOD 89]F. BODIN and F. CHAROT. Machine modeling and loop optimisation for horizontal microcode machines. Technical Report 1193, INRIA, Mars 1989.

[CYT 87]R. CYTRON and J. FERRANTE. What's in a name ? In *International Conference on Parallel Processing*, pages 19–27, August 1987.

[CYT 89]R. CYTRON, J. FERRANTE, B. K. ROSEN, M. N. WEGMAN, and F. K. ZADECK. An efficient method of computing static single assignement form. In *Conference on Principles of Programming Langages*, 1989.

[DIG 92]Digital. *Alpha Architecture Handbook.* Digital, 1992.

[DAM 82]L. DAMAS and R. MILNER. Principal type-schemes for functional programs. In *Conference on Principles of Programming Languages*, 1982.

[EBC 87]Kemal EBCIOGLU. A compilation technique for software pipelining of loops with conditional jumps. In *Annual Workshop on Microprogramming*, pages 69–79, December 1987.

[ELL 85]J. R. ELLIS. *Bulldog-A Compiler for VLIW Architectures.* MIT Press, 1985. Ph.D dissertation.

[FIS 84]J. A. FISHER, J. R. ELLIS, J. C. RUTTENBERG, and A. NICOLAU. Parallel processing: A smart compiler and a dumb machine. In *Symposium on Compiler construction.* SIGPLAN Notices, June 1984. Volume 19, Number 6.

[FIS 92]J. A. FISHER and S. M. FREUDENBERGER. Predicting conditional branch directions from previous runs of a program. In *Architectural Support for Programming Languages and Operating Systems*, 1992.

[FIS 81]J. A. FISHER. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7), July 1981.

[GAR 79]M. R. GAREY and D. S. JOHNSON. *Computers and Intractability - A guide to the Theory of NP-completeness.* Freeman, New-York, 1979.

[GUN 92]Carl A. GUNTER. *Semantics of programming languages: structures and techniques.* The MIT press, Cambridge, Mass., London, 1992.

[HEN 90]John L. HENNESSY and David A. PATTERSON. *Computer Architecture A Quantitive Approach.* Morgan Kaufmann Publishers Inc, 1990. second edition (1994).

[IBM 90]IBM. IBM Risc System/6000 Technology. Technical Report SA23-2619, IBM, 1990. Copies can be obtained from the local IBM Branch office.

[KAN 88]G. KANE. *Mips R2000 Risc Architecture.* Prentice Hall, 1988.

[LAM 88]Monica S. LAM. Software pipelining: An effective scheduling technique for VLIW machines. In *Conference on Programming Language, Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22-24 1988.

[LAM 92]M. S. LAM and R. P. WHILSON. Limits of control flow on parallelism. *ACM Sigarch. Computer Architecture News*, 20(4), 1992.

[MUL 87]Multiflow Computer Inc. Trace/300 series. Technical report, Multiflow Computer Inc, Brandford, Connecticut, 1987.

[NIC 81]A. NICOLAU and J. A. FISHER. Using an oracle to measure potential parallelism in instruction stream programs. In *14th Annual Workshop on Microprogramming*, pages 171–182, 1981.

[NIC 85]Alex NICOLAU. Percolation scheduling: A parallel compilation technique. Technical report, Cornell University, 1985.

[PLO 81]Gordon D. PLOTKIN. *A structural approach to operational semantics.* Daimi FN-19. University of Aarhus, Computer Science Department Aarhus University Ny Munkegade DK 8000 C Danemark, September 1981.

[RFI 93]B.R. RAU and J.A. FISHER. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[THE 92]K. B. THEOBALD, G. R. GAO, and L. J. HENDREN. On the linits of program parallelism and its smoothability. In *25th Annual International Symposium on Microarchitecture (MICRO 25)*, pages 10–19, Portland, Oregon, December 1-4 1992.

[TOU 84]R. F. TOUZEAU. A fortran compiler for the FPS-164 scientific computer. In *Symposium on Compiler Construction*, pages 48–57, June 1984.

Marc Pouzet a effectué une thèse au sein du projet PARA à l'INRIA de Rocquencourt sur la compaction des langages fonctionnels. À VERIMAG, il étudie une extension fonctionnelle du langage synchrone LUSTRE en vue d'une intégration dans un langage de la famille ML. Il effectue actuellement un séjour post-doctoral au sein du laboratoire ACAPS de l'Université MCGILL de Montréal.