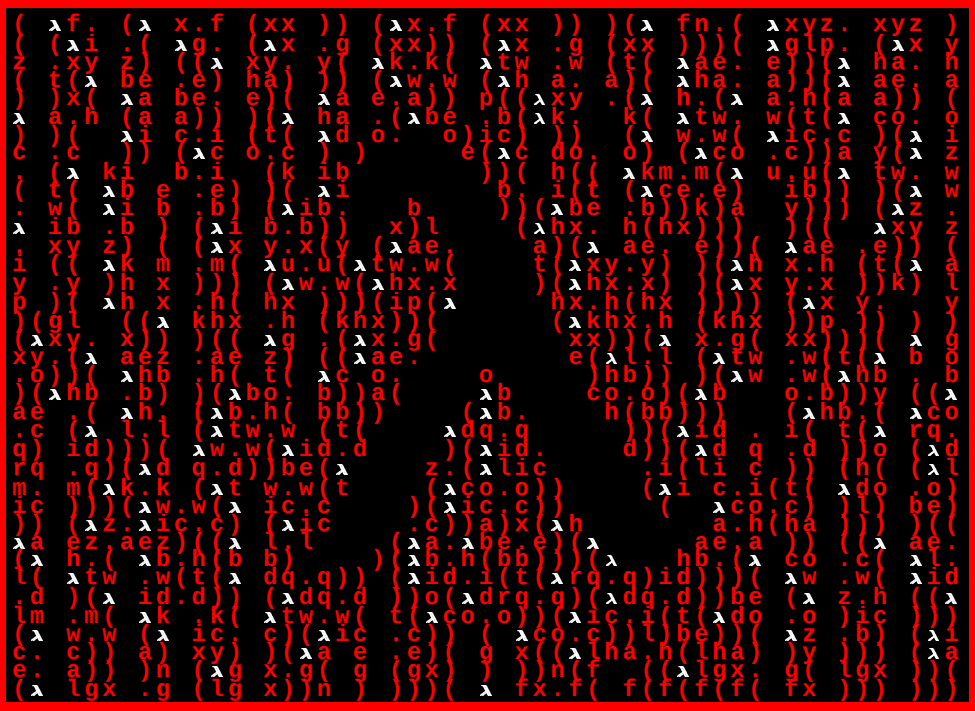
Documents d'habilitation



Approches langages pour la conception et la mise en œuvre de programmes

par Pascal FRADET

IRISA + IFSIC

HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut de Formation Supérieure
en Informatique et en Communication

par

Pascal Fradet

Approches langages pour la conception et la mise en œuvre de programmes

soutenue le 10 novembre 2000 devant le jury composé de

M.	Jean-Pierre Banâtre	Président
MM.	Pierre Lescanne	Rapporteur
	Michel Mauny	Rapporteur
	David A. Schmidt	Rapporteur
MM.	Paul Le Guernic	Examinateur
	Daniel Le Métayer	Examinateur

Illustration de couverture : *Goldbach in Church* [53 x 37 Matrix]

Lambda-expression décomposant successivement chaque entier pair en une somme de deux nombres premiers. L'hypothétique (et très improbable) forme normale serait le plus petit entier de Church invalidant la conjecture de Goldbach.

Avant-propos

Ce mémoire retrace une grande partie des activités de recherche que j'ai menées depuis ma thèse. La plupart de mes travaux peuvent être vus comme des approches langages aux problèmes qu'ils abordent. C'est sous cet angle que j'ai choisi d'unifier et d'orienter la présentation. Certaines publications qui ne rentraient pas clairement sous ce qualificatif sont justes citées. D'autres travaux (analyse de pointeurs, preuves de correction d'optimisations pour Java Card, moteur d'analyse) auxquels j'ai participé sont passés sous silence.

Le style de présentation se veut plus intuitif que technique. Pour des raisons d'uniformité et de lisibilité, j'ai changé le cadre de présentation de travaux anciens (section 2.8 et 3.1). Cependant, je n'ai pas refait tous les développements techniques (transformations, preuves, etc.) dans le nouveau cadre et la présentation n'est sans doute pas aussi rigoureuse qu'elle le devrait pour ces deux sections. D'autres points nécessitaient une description précise et détaillée que j'ai tâché de fournir (section 2.2 et 2.3). Pour les sujets abordés par plusieurs publications successives, je me suis contenté de présenter les résultats les plus récents et/ou aboutis (*e.g.* chapitre 2). Les principales publications sont lesquelles est basé ce mémoire sont [169][173] pour le chapitre 2, [172][174] pour le chapitre 3 et [177][170][179] pour les chapitres 4, 5 et 6 respectivement. La bibliographie et les références personnelles sont présentées séparément en fin de document.

Comme il se doit, chaque chapitre s'efforce de motiver et de justifier les travaux qu'il présente. Pourtant, même si les arguments développés sont justes, ils ne racontent pas toute l'histoire. Le choix d'un sujet de recherche se fait le plus souvent par rapport au passé, à des rencontres, des lectures, des critères extra-scientifiques. À la fin de chaque partie, j'ai trouvé intéressant de situer les travaux présentés dans leur contexte "historique". Suivant son profil et ses intérêts, le lecteur pourra toujours sauter ces deux pages … ou ne lire qu'elles.

La majorité de mes recherches sont le fruit d'efforts collectifs. En particulier, les chapitres 2 (exceptée la section 2.8), 4, 5 et 6 sont basés sur des travaux effectués en collaboration avec, respectivement, Rémi Douence, Daniel Le Métayer, Julien Mallet et Thomas Colcombet. Que Rémi, Daniel, Julien et Thomas, ainsi que tous ceux avec lesquels j'ai travaillé sur d'autres sujets, trouvent ici l'expression de ma reconnaissance.

Merci à :

- Jean-Pierre Banâtre, qui s'est soucié de l'état d'avancement de mon habilitation durant de longues années et qui m'a fait le plaisir de présider le jury,
- Pierre Lescanne, Michel Mauny et David Schmidt, qui se sont intéressé à mes recherches et ont accepté la tâche de rapporteur. J'ai été particulièrement sensible aux commentaires et suggestions de Pierre Lescanne sur le document, et au fait que Michel Mauny et David Schmidt aient fait le voyage de Manhattan (New York et Kansas, respectivement) pour assister à la soutenance,
- Paul Le Guernic, pour sa participation au jury et son regard "réactif et synchrone" sur ces travaux,
- Daniel Le Métayer, pour sa participation au jury et pour tout le chemin parcouru ensemble.

Merci enfin aux membres passés et présents du projet Lande pour les coups de main, la bonne humeur et les discussions de cafet.

Table des matières

CHAPITRE 3 Optimisations et typage

PARTIE II LANGAGES DÉDIÉS ET DISCIPLINES DE PROGRAMMATION

CHAPITRE 4 Types graphe

CHAPITRE 5 Langage dédié au parallélisme massif

CHAPITRE 1 *Introduction*

Les recherches présentées dans ce document ont, ce que l'on peut appeler, une *approche langage* des problèmes. C'est à dire, une approche qui s'exprime, soit *dans* un langage de programmation, soit *par* un langage de programmation.

1.1 Dans le langage

Dans ce premier type d'approche, le langage de programmation est l'unique cadre de travail. Les solutions sont exprimées dans le langage même sans faire appel à un formalisme éloigné. Un exemple de cette approche est l'optimisation par transformation de programmes [53][12][155][29]. La déforestation est une optimisation des programmes fonctionnels qui fut tout d'abord exprimée par une traduction dans un langage impératif [153] avant d'être présentée comme une transformation de source à source [155]. L'avantage évident de la transformation sur la traduction est de rester dans le langage et ainsi de laisser la porte ouverte à d'autres transformations sur le programme. Ce type d'approche a également l'avantage de la simplicité. Le langage de programmation fournit un cadre unifié pour exprimer le problème, le résoudre et appliquer la solution. En général, cela facilite grandement les preuves de correction. De plus, les outils basés sur une approche langage restent proches des connaissances du programmeur.

Ces avantages supposent tout de même que le langage soit un formalisme agréable et puissant. Les langages noyaux (*e.g.* le λ-calcul, le π-calcul, CCS, etc.) constituent des cadres de travail idéaux. La première partie de ce mémoire (chapitres 2 et 3), intitulée "*Dans le cadre fonctionnel*", présente nos travaux autour de cette idée. La compilation et diverses optimisations de programmes fonctionnels sont traitées dans le cadre du λ-calcul, noyau des langages fonctionnels.

Dans le chapitre 2, nous montrons comment le processus de compilation peut s'exprimer dans le langage lui-même par transformation de programme. Il est connu

depuis longtemps que la gestion de l'environnement des programmes fonctionnels peut se décrire par transformation de programmes *via* un algorithme d'abstraction et des combinateurs. Nous poussons cette idée jusqu'au bout en décrivant tout le processus de compilation comme une suite de transformations de programme. Nous introduisons un cadre formel unifié qui nous permet de décrire, prouver, comparer et classifier une douzaine de mises en œuvre de langages fonctionnels. Si la plus grande part de l'étude se concentre sur les implémentations classiques, nous modélisons également plusieurs extensions (mises en œuvre hybrides, réduction sous les lambdas).

Le chapitre 3 décrit deux optimisations de l'implémentation des langages fonctionnels. La première optimisation, qui vise à autoriser des mises à jour en place, est basée sur une analyse syntaxique de programme. Si les analyses sémantiques (*e.g.* par interprétation abstraite) ont l'avantage de la précision, les analyses syntaxiques sont, elles, moins coûteuses. Elles restent également plus proches des connaissances du programmeur qui peut ainsi garder la maîtrise des outils à sa disposition. L'interaction avec l'utilisateur, nécessaire lorsque la propriété d'intérêt ne peut être montrée par l'analyseur, est souvent plus facile à concevoir qu'avec une analyse sémantique. La deuxième optimisation prend la forme d'un glaneur de cellules (GC) qui utilise les informations de type pour récupérer plus de mémoire qu'un GC standard. Ce travail se rattache à une approche langage dans le sens où il repose sur le type des expressions sources et que la technique se décrit et se justifie dans le cadre fonctionnel.

1.2 Par le langage

Pour les approches qui s'expriment *par* un langage de programmation, il s'agit de prévenir le problème ou d'assurer une propriété *via* l'utilisation d'un langage (ou d'une discipline de programmation). Le système de types de ML est exemplaire de cette approche. La discipline de programmation, imposée par le système de types et assistée par l'inféreur de types, évite de nombreuses erreurs de programmation. Les langages dédiés à un domaine d'application particulier sont une autre incarnation de cette idée. Le but visé est d'assurer des propriétés fonctionnelles ou non fonctionnelles (*e.g.* efficacité) cruciales pour le domaine. Le langage, à travers des restrictions syntaxiques fortes, assure les propriétés par construction ou les rend décidables. Ces approches, intellectuellement séduisantes ("mieux vaut prévenir que guérir"), sont souvent techniquement supérieures (plus efficaces et moins coûteuses)

aux approches *a posteriori* (*i.e.* par analyse de programmes). Toutefois, ces avantages se payent par un changement des habitudes de programmation. La seconde partie (chapitres 4, 5 et 6), intitulée "*Langages dédiés et disciplines de programmation*", présente trois études autour de cette idée.

Dans le chapitre 4, nous introduisons une nouvelle notion de type (les types graphe) et une discipline de programmation associée. Les types graphes permettent de définir précisément le partage des structures de données à pointeurs. On peut, par exemple, définir les types "liste circulaire", "liste doublement chaînée", "arbre binaire avec feuilles chaînées", etc. La discipline de programmation associée impose que ces structures soient exclusivement accédées et modifiées par des réécritures de graphe. Nous proposons un algorithme de vérification qui permet d'assurer qu'un type graphe est préservé par les réécritures du programme. Les types et réécritures de graphe s'intègrent naturellement dans un langage comme C. Le système de type permet alors de détecter de nombreuses erreurs de programmation et la manipulation de pointeurs devient plus sûre.

Le chapitre 5 présente un langage dédié à la programmation de machines massivement parallèles. La particularité du langage est de permettre une analyse des coûts de communication et de calcul à la fois précise et symbolique (les tailles effectives des données n'ont pas à être connues). Une telle analyse de coût est centrale à plusieurs choix de compilation. Elle nous permet en particulier de choisir automatiquement la meilleure distribution de données parmi un ensemble de distributions standards. L'ensemble des restrictions syntaxiques du langage source définit une discipline de programmation qui assure portabilité et prédictibilité du coût. Cette étude est de plus originale d'un point de vue méthodologique puisqu'elle intègre des travaux développés dans trois domaines de recherche différents : la parallélisation de Fortran, les langages à patrons et la programmation fonctionnelle.

Nous décrivons au chapitre 6 une technique permettant d'imposer automatiquement des propriétés à des programmes. La propriété est déclarée séparément du programme et un outil automatique, appelé tisseur, impose la propriété par transformation de programme. Le programme tissé est équivalent au programme source sauf qu'il produit une exception s'il tente de violer la propriété. Le tisseur se doit d'insérer le minimum d'instructions et de tests dans le programme pour assurer la propriété. Une application particulièrement intéressante de cette technique est la sécurisation de code mobile à la réception. Dans ce cas, le programme est l'applet téléchargée et la propriété à imposer est la politique de sécurité locale. Ce style de

"programmation par propriétés", proche de la programmation par aspects, rend les programmes plus déclaratifs, plus simples à écrire, à comprendre et à maintenir.

Le chapitre 7 tire le bilan des travaux présentés et conclut le mémoire en dégageant quelques perspectives de recherche.

Partie I

Dans le cadre fonctionnel

CHAPITRE 2

Compilation par transformation de programme

Un des sujets les plus étudiés concernant les langages fonctionnels est leur mise en œuvre. Depuis le célèbre article de Landin [95], il y a plus de 35 ans, une pléthore de nouvelles machines abstraites ou techniques de compilation a été proposée. La liste des machines abstraites fonctionnelles comprend (entre autres) la SECD [95], la Fam [22], la Cam [36], la CMCM [101], la Tim [48], la Zam [96], la G-machine [80] et la machine de Krivine [37]. D'autres implémentations ne sont pas décrites par une machine abstraite mais par des collections de transformations ou de techniques de compilation. Citons par exemple, les compilateurs basés sur la transformation CPS [6][167][93]. De plus, de nombreux articles proposent des optimisations adaptées à une machine ou une approche spécifique [7][19][85]. A la vue de cette myriade de travaux différents, certaines questions s'imposent. Quels sont les choix fondamentaux ? Quels sont leurs avantages respectifs ? Quels sont précisément les points communs et différences entre deux compilateurs donnés ? L'optimisation conçue pour telle machine abstraite ne pourrait-elle pas être adaptée pour telle autre machine ? On trouve comparativement très peu de travaux consacrés à ces questions. Malgré le besoin évident, il y n'a eu, à notre connaissance, aucune approche globale visant à décrire, classifier et comparer les mises en œuvre de langages fonctionnels.

Nous proposons ici une approche langage qui pallie cette lacune. Le processus de compilation est exprimé dans un cadre commun comme une succession de transformations de programme. Ce cadre est constitué d'une collection de langages intermédiaires sous-ensembles du λ-calcul. Une implémentation est décrite comme une série de transformations $\Lambda \xrightarrow{T_1} \Lambda_1 \xrightarrow{T_2} \ldots \xrightarrow{T_n} \Lambda_n$ chacune compilant une tâche en convertissant une expression d'un langage intermédiaire Λ_i dans le suivant. Le dernier langage Λ_n est composé d'expressions fonctionnelles particulièrement simples (combinateurs + flot de contrôle explicite) qui peuvent être vues comme du code machine.

Un des avantages de cette approche est de décomposer et de structurer le processus de compilation. On peut, par exemple, montrer que deux implémentations apparemment très différentes partagent un choix de mise en œuvre. L'approche facilite également les preuves de correction et les comparaisons. La correction de chaque étape peut être montrée indépendamment et se résume à la preuve d'une transformation dans le cadre fonctionnel. Des comparaisons formelles peuvent s'envisager sous la forme d'étude de complexité des transformations.

Dans ce mémoire, nous commençons par introduire le cadre formel sur lequel est basée l'approche (section 2.1). Nous décrivons ensuite son application :

- à la *description*, en détaillant une mise en œuvre particulière, la Cam, en section 2.2,
- aux *preuves*, en décrivant la preuve de correction d'une étape de la Cam en section 2.3,
- aux *comparaisons*, en étudiant deux choix de compilation de la stratégie d'évaluation en section 2.4,
- à la *classification*, en passant en revue en section 2.5 les différents choix de compilation choisis par une douzaine de mises en œuvre standards,
- à l'*optimisation*, en exprimant des optimisations locales et globales classiques comme des transformations de programme (section 2.6),
- à la *conception*, en proposant un nouveau schéma de compilation de la stratégie d'évaluation (section 2.7),
- à la *réduction sous les lambdas*, en montrant comment la réduction de tête peut se compiler dans ce cadre (section 2.8).

Nous concluons par un rapide survol des travaux existants (section 2.9) et indiquons quelques extensions possibles à cette étude (section 2.10).

2.1 Cadre formel

Nous ne considérons que des λ-expressions pures [11] et notre langage source Λ est défini par la grammaire

$$E ::= x \mid \lambda x.E \mid E_1\ E_2$$

La plupart des choix fondamentaux de mise en œuvre peuvent se décrire avec ce

langage. Chaque étape de compilation est représentée par une transformation d'un langage intermédiaire dans un autre, plus proche d'un code machine. Ici, nous modélisons un compilateur par une séquence de transformations partant de Λ et impliquant trois langages intermédiaires (sous-ensembles du λ-calcul) :

$$\Lambda \xrightarrow{V} \Lambda_s \xrightarrow{A} \Lambda_e \xrightarrow{S} \Lambda_k$$

Les transformations *V, A* et *S* représentent respectivement la compilation du schéma d'évaluation (*i.e.* la recherche du prochain rédex), la compilation de la β-réduction (*i.e.* la gestion de l'environnement) et la mise en œuvre des ruptures de séquence (*i.e.* les appels et retours d'évaluation de fermetures).

2.1.1 Langages intermédiaires

Le premier langage intermédiaire (Λ_s) restreint l'utilisation de l'application et explicite le schéma d'évaluation. Le langage Λ_s est défini comme une restriction du λ-calcul équipé des trois combinateurs "**;**", "**store**$_s$", et "**fetch**$_s$".

$$\Lambda_s \qquad E ::= x \mid E_1 \,;\, E_2 \mid \mathbf{store}_s\, E \mid \mathbf{fetch}_s\,(\lambda x.E) \qquad x \in \mathit{Vars}$$

On peut remarquer que la syntaxe de Λ_s ne comporte pas d'applications générales. L'application est toujours associée à un combinateur. Intuitivement, le combinateur infixe "**;**" est un opérateur de séquencement et E_1 **;** E_2 se lit "évaluer E_1 puis évaluer E_2". La paire de combinateurs **store**$_s$ et **fetch**$_s$ spécifie un composant *s* qui mémorise les résultats intermédiaires comme, par exemple, une pile de données. L'expression **store**$_s$ *E* se lit "rendre *E* en résultat dans *s*" et **fetch**$_s$ ($\lambda x.E$) "lier à *x* le dernier résultat intermédiaire stocké dans *s* avant d'évaluer *E*". Nous verrons que la propriété clé de Λ_s est que le choix du rédex n'a plus d'impact sémantique. Cette propriété explicite le fait que transformer une expression de Λ en une expression de Λ_s revient à compiler la stratégie d'évaluation.

Le second langage (Λ_e) restreint l'utilisation des variables et explicite la gestion de l'environnement à l'aide de combinateurs.

$$\Lambda_e \qquad E ::= v \mid E_1 \,;\, E_2 \mid \mathbf{store}_s\, E \mid \mathbf{fetch}_s\,(\lambda v.E)$$
$$\mid \mathbf{store}_e\, E \mid \mathbf{fetch}_e\,(\lambda v.E) \qquad v \in \{f,x,e\}$$

La gestion de l'environnement est encodée grâce à la nouvelle paire de combinateurs **store**$_e$ et **fetch**$_e$. Ils se comportent exactement comme **store**$_s$ et **fetch**$_s$; ils spécifient juste une structure de données *e* (au moins conceptuellement) différente

comme, par exemple, une pile d'environnements. Dans Λ_e, les variables ne sont utilisées que pour définir des combinateurs de manipulation des environnements.

Le troisième langage (Λ_k) code les transferts de contrôle à l'aide d'instructions d'appel et de retour de fonction. Sa syntaxe est :

$$\Lambda_k \quad E ::= v \mid E_1 \,;\, E_2 \mid \mathbf{store}_s\, E \mid \mathbf{fetch}_s\, (\lambda v.E)$$
$$\mid \mathbf{store}_e\, E \mid \mathbf{fetch}_e\, (\lambda v.E)$$
$$\mid \mathbf{store}_k\, E \mid \mathbf{fetch}_k\, (\lambda v.E) \qquad v \in \{f,x,e\}$$

Comparé à Λ_e, le langage Λ_k interdit certaines séquences de code qui correspondent à des appels de fonctions (*e.g.* x **;** E). La paire de combinateurs $\mathbf{store}_k$ et $\mathbf{fetch}_k$ spécifie un composant (*e.g.* une pile de retour) pour enregistrer les adresses de retour. Ce dernier langage peut être vu comme du code machine.

2.1.2 Règles de conversion

Les langages Λ_s, Λ_e et Λ_k sont des sous-ensembles du λ-calcul et la définition de la substitution ainsi que la notion de variable libre ou liée restent identiques. On peut donner diverses définitions aux combinateurs $\mathbf{store}_i$ et $\mathbf{fetch}_i$ (*cf.* section 2.1.3). Nous n'optons pas pour une définition précise ici mais nous imposons que leurs définitions satisfassent l'équivalent de la β- et η-conversion :

$$(\beta_i) \quad (\mathbf{store}_i\, F) \,;\, \mathbf{fetch}_i\, (\lambda x.E) = E[F/x] \qquad i \in \{s,e,k\}$$

$$(\eta_i) \quad \mathbf{fetch}_i\, (\lambda x.\mathbf{store}_i\, x \,;\, E) = E \quad \textit{si } x \textit{ n'est pas libre dans } E \qquad i \in \{s,e,k\}$$

Nous imposons également l'associativité du combinateur "**;**". Cette propriété est spécialement utile pour transformer les programmes.

$$\text{(assoc)} \qquad (E_1 \,;\, E_2) \,;\, E_3 = E_1 \,;\, (E_2 \,;\, E_3)$$

Nous considérons une seule règle de réduction correspondant à la β-réduction :

$$\mathbf{store}_i\, F \,;\, \mathbf{fetch}_i\, (\lambda x.E) \bullet\!\!\rightarrow E[F/x] \qquad i \in \{s,e,k\}$$

Comme toutes les mises en œuvre classiques, nous ne traitons que les réductions faibles. Les expressions sous les $\mathbf{fetch}_i$ ou les $\mathbf{store}_i$ ne sont pas réductibles et, sauf indication contraire, nous utilisons par la suite rédex (resp. réduction, forme normale) pour rédex faible (resp. réduction faible, forme normale faible).

Ce langage dispose d'une collection de lois algébriques utiles pour transformer le code fonctionnel (*e.g.* exprimer des optimisations) ou montrer la correction ou l'équivalence de transformations. Par exemple :

si x n'est pas libre dans F $\quad$ $\mathbf{fetch}_i(\lambda x.E)\ \mathbf{;}\ F = \mathbf{fetch}_i(\lambda x.(E\ \mathbf{;}\ F))$ $\quad$ (L1)

Une propriété importante de ces langages est que le choix du rédex n'est plus sémantiquement significatif. Il est facile de voir que, dans Λ_i, deux rédex sont forcément disjoints. De plus, comme tout rédex est nécessaire (une réécriture n'est pas à même de supprimer un rédex) toute stratégie est normalisante [92].

Propriété 2-1 *Dans Λ_s, Λ_e ou Λ_k, toute stratégie de réduction est normalisante*

Par la suite, par alléger l'écriture, nous écrivons $\lambda_i x.E$ pour $\mathbf{fetch}_i\ (\lambda x.E)$ et nous omettons souvent les parenthèses. Par exemple, nous écrivons $\mathbf{store}_s E\ \mathbf{;}\ \lambda_s x.F\ \mathbf{;}\ G$ au lieu de $(\mathbf{store}_s\ E)\ \mathbf{;}\ (\mathbf{fetch}_s(\lambda x.F)\ \mathbf{;}\ G)$. Nous utilisons le symbole "≡" pour dénoter l'égalité syntaxique et "=" pour les définitions et l'égalité sémantique. Nous nous permettons également d'utiliser des facilités syntaxiques comme les n-uplets $(x_1,\ldots,x_n)$ ou le filtrage $\lambda_i(x_1,\ldots,x_n).E$.

L'exemple suivant illustre la réduction dans ce type de langage :

$$\mathbf{store}_e\ E\ \mathbf{;}\ \mathbf{store}_s(\mathbf{store}_s\ F\ \mathbf{;}\ \lambda_s z.G)\ \mathbf{;}\ \lambda_s x.\lambda_e y.\mathbf{store}_s(\mathbf{store}_e\ y\ \mathbf{;}\ x)$$

$$\bullet\!\!\rightarrow\ \mathbf{store}_e\ E\ \mathbf{;}\ \lambda_e y.\mathbf{store}_s(\mathbf{store}_e\ y\ \mathbf{;}\ \mathbf{store}_s\ F\ \mathbf{;}\ \lambda_s z.G)$$

$$\bullet\!\!\rightarrow\ \mathbf{store}_s\ (\mathbf{store}_e\ E\ \mathbf{;}\ \mathbf{store}_s\ F\ \mathbf{;}\ \lambda_s z.G)$$

2.1.3 Instanciation

Cette hiérarchie de langages est une abstraction utile pour structurer et décrire les mises en œuvre. Il ne faut pas perdre de vue que chaque langage est en fait un sous-ensemble du λ-calcul. Un point notable est que nous n'avons pas à opter dès le départ pour une définition spécifique des combinateurs **;**, $\mathbf{store}_i$, et $\mathbf{fetch}_i$. Les définitions peuvent être choisies en dernier lieu. Afin de fixer les idées, nous donnons ici plusieurs choix possibles.

La définition la plus naturelle pour **;** est :

$$\mathbf{;} = \lambda a.\lambda b.\lambda c.a\ (b\ c)$$

c'est à dire E_1 **;** $E_2 = \lambda c.E_1\ (E_2\ c)$. La variable fraîche c peut être vue comme une continuation qui explicite le séquencement.

Chaque paire de combinateurs $\mathbf{store}_i$, et $\mathbf{fetch}_i$ spécifie un composant i (comme une pile, une liste, un arbre ou un vecteur) capable de mémoriser les résultats intermédiaires. Il est possible de garder ces composants séparés ou de fusionner certains d'entre eux.

Les définitions suivantes font le choix d'avoir des composants s (données), e (environnement) et k (contrôle) séparés.

$\mathbf{store}_s = \lambda n.\lambda c.\lambda s.\lambda e.\lambda k.c\ (s,n)\ e\ k$ $\qquad$ $\mathbf{fetch}_s = \lambda f.\lambda c.\lambda(s,n).\lambda e.\lambda k.f\ n\ c\ s\ e\ k$

$\mathbf{store}_e = \lambda n.\lambda c.\lambda s.\lambda e.\lambda k.c\ s\ (e,n)\ k$ $\qquad$ $\mathbf{fetch}_e = \lambda f.\lambda c.\lambda s.\lambda(e,n).\lambda k.f\ n\ c\ s\ e\ k$

$\mathbf{store}_k = \lambda n.\lambda c.\lambda s.\lambda e.\lambda k.c\ s\ e\ (k,n)$ $\qquad$ $\mathbf{fetch}_k = \lambda f.\lambda c.\lambda s.\lambda e.\lambda(k,n).f\ n\ c\ s\ e\ k$

Il est facile de vérifier que ces définitions respectent les propriétés (*assoc*), (β_i) et (η_i). La réduction (par β-réduction standard et ordre normal) des λ-expressions peut alors être vue comme des transitions d'état d'une machine abstraite à quatre composants (code, pile de données, pile d'environnement, pile de retour) :

$(E_1$ **;** $E_2)\ C\ S\ E\ K \rightarrow E_1\ (E_2\ C)\ S\ E\ K$

$\mathbf{store}_s\ N\ C\ S\ E\ K \rightarrow C\ (S,N)\ E\ K$ $\qquad$ $(\lambda_s x.X)\ C\ (S,N)\ E\ K \rightarrow X[N/x]\ C\ S\ E\ K$

$\mathbf{store}_e\ N\ C\ S\ E\ K \rightarrow C\ S\ (E,N)\ K$ $\qquad$ $(\lambda_e x.X)\ C\ S\ (E,N)\ K \rightarrow X[N/x]\ C\ S\ E\ K$

$\mathbf{store}_k\ N\ C\ S\ E\ K \rightarrow C\ S\ E\ (K,N)$ $\qquad$ $(\lambda_k x.X)\ C\ S\ E\ (K,N) \rightarrow X[N/x]\ C\ S\ E\ K$

Un code est un transformateur d'état et C joue le rôle d'une continuation. Sa réduction est initiée en l'appliquant à une continuation initiale (*e.g.* la fonction identité) et des composants données, environnement et contrôle initiaux (vides).

A l'autre bout du spectre des possibilités d'instanciation, tous les composants sont fusionnés. La machine abstraite sous-jacente ne contient alors que deux composants (le code et une pile de données-environnement-contrôle). Les définitions des combinateurs deviennent :

$$\mathbf{store}_s = \mathbf{store}_e = \mathbf{store}_k = \lambda n.\lambda c.\lambda z.c\ (z,n)$$

$$\mathbf{fetch}_s = \mathbf{fetch}_e = \mathbf{fetch}_k = \lambda f.\lambda c.\lambda(z,x).f\ x\ c\ z$$

et la réduction des expressions est de la forme

$$\mathbf{store}_i\ N\ C\ Z \rightarrow C\ (Z,N) \qquad \text{et} \qquad (\lambda_i x.X)\ C\ (Z,N) \rightarrow X[N/x]\ C\ Z \quad \text{pour } i \in \{s,e,k\}$$

En résumé, la composition des différentes transformations décrit un compilateur transformant du code source (λ-expressions) en du code machine (expressions combinatoires sans variable avec séquencement explicite). L'étape d'instanciation décrit l'état d'une machine abstraite et ses règles de transition associée. Soulignons que le terme "machine abstraite" ne doit pas suggérer une couche d'interprétation. La seule abstraction réside dans le fait qu'on ne se lie pas à une architecture ou un code machine spécifiques. À la fin du processus de compilation, on obtient du véritable code assembleur et les machines abstraites sont proches de machines réelles.

2.2 Décrire

Afin d'illustrer l'utilisation du cadre formel, nous décrivons ici les choix de compilation de la machine abstraite catégorique (Cam) [36]. La Cam est une machine abstraite à environnement, support des premières mises en œuvre du langage Caml. La conception de la Cam est basée sur les indices de de Bruijn (qui codent l'accès aux variables) et une description de la sémantique du λ-calcul à l'aide de combinateurs catégoriques (qui explicitent la construction de fermetures et la gestion des environnements). Au final, la Cam peut être vue comme une version simplifiée d'une des premières machines abstraites : la machine SECD.

Nous décrivons ici la Cam en suivant les étapes de notre cadre formel : compilation de la stratégie d'évaluation, compilation de la β-réduction et instanciation. Pour simplifier, nous omettons l'étape de compilation des transferts de contrôle.

2.2.1 Compilation de la stratégie d'évaluation

La Cam utilise la version gauche-à-droite de l'appel par valeur. Cette stratégie de réduction a plusieurs implémentations. Le modèle utilisé par la Cam considère les λ-abstractions comme des résultats et l'application d'une fonction à son argument est une opération explicite. Ce modèle, dit *eval-apply* dans la terminologie de [122], est naturel pour l'appel par valeur. Il en existe toutefois d'autres (*cf.* section 2.4).

La transformation compilant la version gauche-à-droite de l'appel par valeur en

modèle *eval-apply* est décrite en Figure 2-2.

$$
\begin{array}{lcl}
Va : \Lambda \rightarrow \Lambda_s & & \\
Va\ [\![x]\!] & = & \mathbf{store}_s\ x \\
Va\ [\![\lambda x.E]\!] & = & \mathbf{store}_s\ (\lambda_s x.Va\ [\![E]\!]) \\
Va\ [\![E_1\ E_2]\!] & = & Va\ [\![E_1]\!]\ \mathbf{;}\ Va\ [\![E_2]\!]\ \mathbf{;}\ \mathbf{app} \qquad \text{avec } \mathbf{app} = \lambda_s x.\lambda_s f.(\mathbf{store}_s\ x\ \mathbf{;}\ f)
\end{array}
$$

Figure 2-2 Compilation de l'appel par valeur avec applications (*Va*)

Les variables et les λ-abstractions dénotent des valeurs ; elles sont rendues en résultat. Pour une application $E_1\ E_2$, on évalue la fonction E_1, l'argument E_2 et le résultat de E_1 est appliqué au résultat de E_2.

Nous présentons maintenant un exemple de code produit par *Va* et détaillons sa réduction.

Exemple 2-3 Soit $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$, après simplifications locales (*cf.* section 2.6) on obtient :

$$
\begin{array}{lcl}
Va\ [\![E]\!] & \equiv & \mathbf{store}_s(\lambda_s z.\mathbf{store}_s\ z)\ \mathbf{;}\ (\lambda_s y.\mathbf{store}_s\ y)\ \mathbf{;}\ (\lambda_s x.\mathbf{store}_s\ x) \\
 & \bullet\!\!\rightarrow & \mathbf{store}_s(\lambda_s z.\mathbf{store}_s\ z)\ \mathbf{;}\ (\lambda_s x.\mathbf{store}_s\ x) \\
 & \bullet\!\!\rightarrow & \mathbf{store}_s(\lambda_s z.\mathbf{store}_s\ z) \equiv Va\ [\![\lambda z.z]\!]
\end{array}
$$

L'expression source a deux rédex, $(\lambda x.x)((\lambda y.y)(\lambda z.z))$ et $(\lambda y.y)(\lambda z.z)$, mais une stratégie d'appel par valeur choisit uniquement le deuxième. En revanche, $Va\ [\![E]\!]$ a uniquement (la version compilée de) $(\lambda y.y)(\lambda z.z)$ comme rédex. La réduction (illégale en appel par valeur) $E \rightarrow (\lambda y.y)(\lambda z.z)$ ne peut avoir lieu avec $Va\ [\![E]\!]$. Ceci illustre le fait que la stratégie d'évaluation a été compilée. ❒

2.2.2 Compilation de la β-réduction

La gestion de l'environnement utilisée par la Cam se décrit par une transformation proche de la sémantique dénotationnelle du λ-calcul [145]. La structure de l'environnement est un arbre de fermetures. Une fermeture est ajoutée à l'environnement en temps constant. En contrepartie, l'accès aux valeurs se fait en suivant une chaîne d'indirections. La complexité temps de l'accès à une variable est

O(n) où n est le nombre de λ_s depuis l'occurrence de la variable jusqu'à son λ_s liant (*i.e.* son nombre de de Bruijn). Ce mode de gestion des environnements en liste (permettant le partage d'environnements) est largement utilisé par les machines abstraites (*e.g.* la SECD [95], la machine de Krivine [37]).

La transformation *As* (Figure 2-4) est faite relativement à un environnement statique ρ enregistrant le nom des variables. Afin de simplifier la présentation, les indices des variables dans $(\ldots((\rho,x_i),x_{i-1})\ldots,x_0)$ dénotent leur rang dans l'environnement statique.

$$As : \Lambda_s \rightarrow env \rightarrow \Lambda_e$$

$$As[\![E_1 \,;\, E_2]\!]\,\rho = \mathbf{dupl}_e \,;\, As[\![E_1]\!]\,\rho \,;\, \mathbf{swap}_{se} \,;\, As[\![E_2]\!]\,\rho$$

$$As[\![\mathbf{store}_s\ E]\!]\,\rho = \mathbf{store}_s\ (As[\![E]\!]\,\rho) \,;\, \mathbf{mkclos}$$

$$As[\![\lambda_s x.E]\!]\,\rho = \mathbf{mkbind} \,;\, As[\![E]\!]\,(\rho,x)$$

$$As[\![x_i]\!]\,(\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{fst}^i \,;\, \mathbf{snd} \,;\, \mathbf{appclos}$$

Figure 2-4 Abstraction avec des environnements partagés (*As*)

As introduit sept nouveaux combinateurs pour exprimer la sauvegarde et la restauration des environnements (**dupl**$_e$, **swap**$_{se}$), la construction et l'ouverture des fermetures (**mkclos**, **appclos**), l'accès aux valeurs (**fst**, **snd**), et enfin l'ajout d'une fermeture à l'environnement (**mkbind**). Ils sont définis dans Λ_e par :

$$\mathbf{dupl}_e = \lambda_e e.\mathbf{store}_e\ e \,;\, \mathbf{store}_e\ e \qquad \mathbf{swap}_{se} = \lambda_s x.\ \lambda_e e.\mathbf{store}_s\ x \,;\, \mathbf{store}_e\ e$$

$$\mathbf{mkclos} = \lambda_s x.\ \lambda_e e.\mathbf{store}_s\ (x,e) \qquad \mathbf{appclos} = \lambda_s(x,e).\mathbf{store}_e\ e \,;\, x$$

$$\mathbf{fst} = \lambda_e(e,x).\mathbf{store}_e\ e \qquad \mathbf{snd} = \lambda_e(e,x).\mathbf{store}_s\ x$$

$$\mathbf{mkbind} = \lambda_e e.\ \lambda_s x.\mathbf{store}_e\ (e,x)$$

Pour évaluer une expression de la forme E_1 **;** E_2, l'environnement est dupliqué (**dupl**$_e$), E_1 est évaluée avec le premier environnement, le résultat de cette évaluation et le deuxième environnement sont intervertis (**swap**$_{se}$) pour que E_2 puisse s'évaluer. Notons que le combinateur **swap**$_{se}$ n'est vraiment utile que si les composants *s* et *e* sont représentés par la même structure de données. Ce choix apparaît dans l'étape d'instanciation décrite plus bas.

Les variables sont liées aux fermetures stockées dans les environnements. Avec les règles initiales, $As\,[\![\mathbf{store}_s\ x_i]\!]$ construirait encore une nouvelle fermeture. Cet emboîtement inutile est évité grâce à la règle :

$As\,[\![\mathbf{store}_s\ x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{fst}^i\ \mathbf{;}\ \mathbf{snd}$

Dans le cas de la Cam, il est utile d'introduire un combinateur représentant la version abstraite de **app** :

$$\mathbf{appclos}_L = \lambda_s x.\lambda_s(f,e).\mathbf{store}_s\ x\ \mathbf{;}\ \mathbf{store}_e\ e\ \mathbf{;}\ f$$

On pose $Cam\ [\![E]\!]\ \rho = As\ (Va\ [\![E]\!])\ \rho$ et on obtient :

$Cam : \Lambda \rightarrow env \rightarrow \Lambda_e$

$Cam\ [\![x_i]\!]\ (\ldots((\rho,x_i),x_{i-1})\ldots,x_0) = \mathbf{fst}^i\ \mathbf{;}\ \mathbf{snd}$

$Cam\ [\![\lambda x.E]\!]\ \rho = \mathbf{store}_s\ (\mathbf{mkbind}\ \mathbf{;}\ Cam\ [\![E]\!]\ (\rho,x))\ \mathbf{;}\ \mathbf{mkclos}$

$Cam\ [\![E_1\ E_2]\!]\ \rho = \mathbf{dupl}_e\ \mathbf{;}\ Cam\ [\![E_1]\!]\ \rho\ \mathbf{;}\ \mathbf{swap}_{se}\ \mathbf{;}\ Cam\ [\![E_2]\!]\ \rho\ \mathbf{;}\ \mathbf{appclos}_L$

Figure 2-5 Schéma de compilation de la Cam (*As* o *Va*)

Les combinateurs $\mathbf{store}_s$ et **mkclos** apparaissent toujours ensemble dans le même type d'expression. On peut les remplacer par un nouveau combinateur $\mathbf{cur}_s$ défini par :

$$\mathbf{store}_s\ E\ \mathbf{;}\ \mathbf{mkclos} = \mathbf{cur}_s\ E = \lambda_e e.\mathbf{store}_s\ (E,e)$$

Exemple 2-6 Reprenons l'exemple précédent $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$, après simplifications on obtient :

$$CAM\ [\![E]\!] = \mathbf{dupl}_e\ \mathbf{;}\ \mathbf{cur}_s\ C_1\ \mathbf{;}\ \mathbf{swap}_{se}\ \mathbf{;}\ \mathbf{dupl}_e\ \mathbf{;}\ \mathbf{cur}_s\ C_1\ \mathbf{;}$$

$$\mathbf{swap}_{se}\ \mathbf{;}\ \mathbf{cur}_s\ C_1\ \mathbf{;}\ \mathbf{appclos}_L\ \mathbf{;}\ \mathbf{appclos}_L$$

avec $C_1 \equiv \mathbf{mkbind}\ \mathbf{;}\ \mathbf{snd}$

On se rapproche d'un véritable code machine. L'expression est composée de séquences de combinateurs. De plus, chaque combinateur s'exprime comme quelques instructions assembleur. ❒

Il manque la gestion des transferts de contrôle. La réduction d'expressions de la

forme $\mathbf{appclos}_L$ **;** E implique la réduction d'une fermeture suivie d'un retour à la réduction de E. Ceci est généralement implémenté en utilisant une pile d'adresses de retour. De nombreuses descriptions de mises en œuvre laissent cette étape implicite ; nous faisons de même ici.

2.2.3 Instanciation

La composition des transformations décrit un compilateur. En choisissant une définition pour les combinateurs nous obtenons les règles de la machine associée. Nous choisissons de garder les composants s et e séparés. Une instanciation possible est :

$$\mathbf{;} = \lambda a.\lambda b.\lambda c.a\ (b\ c)$$

$$\mathbf{store}_s = \lambda n.\lambda c.\lambda e.\lambda s.c\ e\ (s,n) \qquad \mathbf{fetch}_s = \lambda f.\lambda c.\lambda e.\lambda(s,n).f\ n\ c\ e\ s$$

$$\mathbf{store}_e = \lambda n.\lambda c.\lambda e.\lambda s.c\ (e,n)\ s \qquad \mathbf{fetch}_e = \lambda f.\lambda c.\lambda(e,n).\lambda s.f\ n\ c\ e\ s$$

Les définitions des combinateurs suivent :

$$\mathbf{appclos}_L = \lambda_s n.\lambda_s(f,e).\ \mathbf{store}_s\ n\ \mathbf{;}\ \mathbf{store}_e\ e\ \mathbf{;}\ f = \lambda c.\lambda z.\lambda(s,(n,(f,e))).f\ c\ (z,e)\ (s,n)$$

$$\mathbf{dupl}_e = \lambda_e e.\mathbf{store}_e\ e\ \mathbf{;}\ \mathbf{store}_e\ e = \lambda c.\lambda(z,e).\lambda s.c\ ((z,e),e)\ s$$

$$\mathbf{cur}_s\ F = \lambda_e e.\mathbf{store}_s\ (F,e) = \lambda c.\lambda(z,e).\lambda s.c\ ((z,(F,e))\ s$$

$$\mathbf{mkbind} = \lambda_e e.\lambda_s x.\mathbf{store}_e\ (e,x) = \lambda c.\lambda(z,e).\lambda(s,n).c\ (z,(e,n))\ s$$

$$\mathbf{fst} = \lambda_e(e,x).\mathbf{store}_e\ e = \lambda c.\lambda(e,x).\lambda s.c\ e\ s$$

$$\mathbf{snd} = \lambda_e(e,x).\mathbf{store}_s\ x = \lambda c.\lambda(e,x).\lambda s.c\ e\ (s,x)$$

$$\mathbf{swap}_{se} = \lambda_s n.\lambda_e e.\mathbf{store}_s\ n\ \mathbf{;}\ \mathbf{store}_e\ e = \lambda c.\lambda(z,e).\lambda(s,n).c\ (z,e)\ (s,n)$$

Comme les composants s et e sont disjoints, $\mathbf{swap}_{se}$ est l'identité. Il peut être retiré du code.

Si ces combinateurs sont vus comme les instructions d'une machine abstraite alors leurs définitions doivent être interprétées comme les transitions d'états suivantes :

appclos$_L$	C	$(Z,(F,E))$	S	$\rightarrow$	$(F\ C)$	(Z,E)	S
mkbind	C	(Z,E)	(S,X)	$\rightarrow$	C	$(Z,(E,X))$	S
dupl$_e$	C	(Z,E)	S	$\rightarrow$	C	$((Z,E),E)$	S
cur$_s$ F	C	(Z,E)	S	$\rightarrow$	C	$(Z,(F,E))$	S
fst	C	(E,X)	S	$\rightarrow$	C	E	S
snd	C	(E,X)	S	$\rightarrow$	C	E	(S,X)

On retrouve presque exactement la description de la Cam proposée par Hannan dans [64] (Figure 4, p. 139). Les combinateurs **dupl**$_e$, **cur**$_s$, **fst** et **snd** correspondent respectivement aux opérations *push*, *lam*', *cdr* et *car* de Hannan. La seule différence vient du fait que nous construisons des fermetures dont la première opération est de mettre leur argument dans l'environnement (**mkbind**). La Cam fait cette opération à l'appel de la fermeture. En décalant le combinateur **mkbind** jusqu'à l'endroit où les fermetures sont évaluées et en le fusionnant avec **appclos**$_L$, nous obtenons

$$\textbf{mkbind ; appclos}_L = \lambda_s x.\lambda_s(f,e).\textbf{store}_s\ x\ \textbf{; store}_e\ e\ \textbf{; mkbind ;}\ f$$

qui est exactement l'opération *ap* de [64].

Comparée à la description originale de la Cam dans [36], les combinateurs **dupl**$_e$, **cur**$_s$, **fst** et **snd** correspondent respectivement aux opérations *push*, *Cur*, *fst* et *snd*. La séquence **mkbind ; appclos**$_L$ correspond à la séquence **Cons;App**. Cependant, si la Cam originale a deux composants, le premier (le terme) regroupe les fermetures et l'environnement courant et le second est la pile d'environnements. La conséquence est que le combinateur **swap**$_{se}$ reste nécessaire pour charger l'environnement dans le terme. Avec une étape d'instanciation fusionnant piles de données et d'environnements, on obtiendrait une machine effectuant les mêmes opérations que la Cam originale mais sur un unique composant regroupant le terme et la pile.

Rappelons que notre but premier est de décrire les principaux choix de compilation afin de classifier et comparer les implémentations. Nos mises en œuvre sont décrites en composant des transformations conçues indépendamment les unes des autres. Il n'est donc pas étonnant que la simple composition des transformations ne modélise pas à la lettre les implémentations existantes. Les différences sont le plus

souvent superficielles et il suffit de transformer/spécialiser la composition de transformations pour obtenir l'implémentation réelle.

2.3 Prouver

Le fait d'exprimer le processus de compilation comme des transformations syntaxiques facilite les preuves de correction. La correction de chaque étape est montrée indépendamment et se résume à la preuve d'une transformation dans le cadre fonctionnel. Nous montrons ici que la transformation *Va* compile effectivement l'appel par valeur.

L'appel par valeur est décrit par la sémantique naturelle suivante (où V et N dénotent des formes normales) :

$$\frac{E_1 \xrightarrow[cbv]{} \lambda x.F \qquad E_2 \xrightarrow[cbv]{} V \qquad F[V/x] \xrightarrow[cbv]{} N}{E_1\, E_2 \xrightarrow[cbv]{} N}$$

La Propriété 2-7 établit que la réduction des expressions transformées ($\bullet\!\!\xrightarrow{*}$) simule la réduction en appel par valeur ($\xrightarrow[cbv]{}$) des λ-expressions sources.

Propriété 2-7 $\forall E$ *fermée, N forme normale,* $E \xrightarrow[cbv]{} N \Leftrightarrow Va\,[\![E]\!] \bullet\!\!\xrightarrow{*} Va\,[\![N]\!]$

Deux lemmes sont nécessaires :

Lemme 2-8 *Si* $Va\,[\![F]\!] \equiv \mathbf{store}_s\ F'$ *alors* $Va\,[\![E[F/x]]\!] \equiv Va\,[\![E]\!]\ [F'/x]$

Ce lemme se montre simplement par induction sur la structure de l'expression E.

Lemme 2-9 *La forme normale d'une expression fermée* $Va\,[\![E]\!]$ *est de la forme* $Va\,[\![\lambda x.F]\!]$.

Le lemme est trivial si $Va\,[\![E]\!]$ est une forme normale. Si $Va\,[\![E]\!]$ est réductible, il est facile de montrer (par induction structurelle) qu'elle contient au moins une sous-expression réductible de la forme :

$Va\,[\![F]\!] \equiv Va\ [\![\lambda x.F_1]\!]$ **;** $Va\ [\![\lambda y.F_2]\!]$ **; app**

qui se réduit en 3 étapes de réduction en une expression de la forme $Va\,[\![F']\!]$.

$$Va\ [\![\lambda x.F_1]\!]\ ;\ Va\ [\![\lambda y.F_2]\!]\ ;\ \mathbf{app} \quad \bullet\!\!\xrightarrow{*} \quad Va\ [\![F_1]\!]\,[\lambda_s y.Va\ [\![F_2]\!]/x]$$

$$\equiv \quad Va\ [\![F_1[\lambda_s y.F_2/x]]\!] \qquad \text{(Lemme 2-8)}$$

Une expression compilée se réduit en une autre expression compilée (*i.e.* de la forme $Va\ [\![.]\!]$) . Par induction et par le fait que toutes les stratégies sont normalisantes, ceci implique que les formes normales sont de la forme $Va\ [\![X]\!]$ et les formes normales fermées de la forme $Va\ [\![\lambda x.F]\!]$.

La preuve de la Propriété 2-7 est sur la structure des arbres de réduction.

Cas de base.

($\Rightarrow$) Si l'expression E n'est pas réductible, elle est de la forme $\lambda x.F$ (E est fermée) et donc $Va\ [\![\lambda x.F]\!] \equiv \mathbf{store}_s\ (\lambda_s x.Va\ [\![F]\!])$ qui n'est pas réductible.

($\Leftarrow$) Si $Va\ [\![E]\!]$ n'est pas réductible alors E est de la forme $\lambda x.F$ qui n'est pas réductible.

Induction.

($\Rightarrow$) E est réductible, c'est à dire, $E \equiv E_1\ E_2$ et

$$E_1 \xrightarrow{cbv} \lambda x.F, \qquad E_2 \xrightarrow{cbv} V \qquad \text{et} \qquad F[V/x] \xrightarrow{cbv} N$$

Par hypothèse d'induction, on a

$$Va\ [\![E_1]\!] \bullet\!\!\xrightarrow{*} Va\ [\![\lambda x.F]\!], \qquad Va\ [\![E_2]\!] \bullet\!\!\xrightarrow{*} Va\ [\![V]\!] \quad \text{et} \quad Va\ [\![F[V/x]]\!] \bullet\!\!\xrightarrow{*} Va\ [\![N]\!]$$

Puisque V est une forme normale fermée, $Va\ [\![V]\!]$ est de la forme $\mathbf{store}_s\ V'$. On a donc :

$$\begin{aligned}
Va\ [\![E_1\ E_2]\!] \equiv\ & Va\ [\![E_1]\!]\ ;\ Va\ [\![E_2]\!]\ ;\ \mathbf{app} \\
\bullet\!\!\xrightarrow{*}\ & \mathbf{store}_s(\lambda_s x.Va\ [\![F]\!])\ ;\ \mathbf{store}_s\ V'\ ;\ \mathbf{app} \\
\bullet\!\!\xrightarrow{*}\ & Va\ [\![F]\!]\ [V'/x] && \text{(définition de } \mathbf{app} \text{ et } \beta\text{-réduction)} \\
\equiv\ & Va\ [\![F[V/x]\ ]\!] && \text{(Lemme 2-8)} \\
\bullet\!\!\xrightarrow{*}\ & Va\ [\![N]\!]
\end{aligned}$$

($\Leftarrow$) $Va\ [\![E]\!]$ est réductible, c'est à dire $E \equiv E_1\ E_2$ et

$Va\ [\![E_1\ E_2]\!] \equiv\ Va\ [\![E_1]\!]\ ;\ Va\ [\![E_2]\!]\ ;\ \mathbf{app}\ \bullet\!\!\rightarrow^{*}\ Va\ [\![N]\!]$

Les sous-expressions $Va\ [\![E_1]\!]$ et $Va\ [\![E_2]\!]$ doivent avoir des formes normales :

$Va\ [\![E_1]\!]\ \bullet\!\!\rightarrow^{*}\ Va\ [\![\lambda x.F]\!]$ (Lemme 2-9)

$Va\ [\![E_2]\!]\ \bullet\!\!\rightarrow^{*}\ Va\ [\![V]\!]$ avec $V \equiv \lambda y.F'$ (Lemme 2-9)

D'où $Va\ [\![E_1\ E_2]\!]\ \bullet\!\!\rightarrow^{*}\ Va\ [\![\lambda x.F]\!]\ ;\ Va\ [\![V]\!]\ ;\ \mathbf{app}$

$\bullet\!\!\rightarrow^{*}\ Va\ [\![F]\!]\ [\lambda_s y.Va\ [\![F']\!]\ /x]$ (définition de **app** et β-réduction)

$\equiv\ \ Va\ [\![F[V/x]\ ]\!]$ (Lemme 2-8)

$\bullet\!\!\rightarrow^{*}\ Va\ [\![N]\!]$

Par hypothèse d'induction, on a

$E_1\ \xrightarrow{cbv}\ \lambda x.F$ $\qquad$ $E_2\ \xrightarrow{cbv}\ V$ $\qquad$ et $\qquad$ $F[V/x]\ \xrightarrow{cbv}\ N$

d'où $E\ \xrightarrow{cbv}\ N$

Ceci conclut la preuve de correction de la transformation *Va*. Notons que la preuve est à la fois simple, standard et indépendante des autres étapes de compilation.

2.4 Comparer

Un des intérêts d'un cadre unique est de pouvoir comparer les différents choix de compilation. Nous comparons ici deux choix de compilation du schéma d'évaluation. Nous avons vu en section 2.2 le modèle *eval-apply* qui évalue la fonction et l'argument avant de les appliquer. Un autre choix pour compiler l'appel par valeur est d'évaluer l'argument et d'appliquer directement la fonction non évaluée. Ce modèle, appelé *push-enter*, est tout à fait naturel pour l'appel par nom où une fonction n'est évaluée que quand elle est appliquée. En appel par valeur, une fonction peut aussi être évaluée en tant qu'argument. Dans ce cas, elle n'est pas appliquée et doit être rendue en résultat. Il faut donc un moyen à la fonction de distinguer ces deux cas, c'est à dire de déterminer si elle dispose de son argument ou non. On utilise pour se faire des marques.

La marque ε est supposée être une valeur distinguable des autres et les fonctions

$\lambda x.E$ sont transformées en expressions de la forme $\mathbf{grab}_s\ (\lambda_s x.E')$. Le combinateur $\mathbf{grab}_s$ teste la présence d'une marque et à deux règles de réduction.

(grab_1) $\qquad \mathbf{store}_s\ \varepsilon\ ;\ \mathbf{grab}_s\ F \bullet\!\!\rightarrow \mathbf{store}_s\ F$

(grab_2) $\qquad \mathbf{store}_s\ V\ ;\ \mathbf{grab}_s\ F \bullet\!\!\rightarrow \mathbf{store}_s\ V\ ;\ F \qquad (V \not\equiv \varepsilon)$

Si une marque est présente (règle grab_1), il n'y a pas d'argument disponible et la fonction est rendue en résultat (une fermeture est construite). Sinon, l'argument est présent (règle grab_2) et la fonction s'applique. Cette technique évite de construire certaines fermetures au prix de tests dynamiques. Le combinateur $\mathbf{grab}_s$ et la marque ε pourraient se définir en Λ_s. En pratique, $\mathbf{grab}_s$ est implémenté à l'aide d'une conditionnelle qui teste la présence d'une marque. La transformation de l'appel par valeur droite-à-gauche avec marques est décrite dans la Figure 2-10.

$$
\begin{array}{l}
Vm : \Lambda \rightarrow \Lambda_s \\
Vm\,[\![x]\!] \quad = \quad \mathbf{grab}_s\ x \\
Vm\,[\![\lambda x.E]\!] \quad = \quad \mathbf{grab}_s\ (\lambda_s x.\ Vm\,[\![E]\!]) \\
Vm\,[\![E_1\ E_2]\!] \quad = \quad \mathbf{store}_s\ \varepsilon\ ;\ Vm\,[\![E_2]\!]\ ;\ Vm\,[\![E_1]\!]
\end{array}
$$

Figure 2-10 Compilation de l'appel par valeur droite-à-gauche avec marques (*Vm*)

Exemple 2-11 Considérons de nouveau l'expression $E \equiv (\lambda x.x)((\lambda y.y)(\lambda z.z))$; après simplifications, on obtient :

$$
\begin{aligned}
Vm\,[\![E]\!] \equiv\ & \mathbf{store}_s\ \varepsilon\ ;\ \mathbf{store}_s(\lambda_s z.\mathbf{grab}_s\ z)\ ;\ (\lambda_s y.\mathbf{grab}_s\ y)\ ;\ (\lambda_s x.\mathbf{grab}_s\ x) \\
\bullet\!\!\rightarrow\ & \mathbf{store}_s\ \varepsilon\ ;\ \mathbf{grab}_s\ (\lambda_s z.\mathbf{grab}_s\ z)\ ;\ (\lambda_s x.\mathbf{grab}_s\ x) \\
\bullet\!\!\rightarrow\ & \mathbf{store}_s\ (\lambda_s z.\mathbf{grab}_s\ z)\ ;\ (\lambda_s x.\mathbf{grab}_s\ x) \\
\bullet\!\!\rightarrow\ & \mathbf{grab}_s\ (\lambda_s z.\mathbf{grab}_s\ z) \equiv Vm\,[\![\lambda z.z]\!]
\end{aligned}
$$

❒

Un ordre d'évaluation de gauche à droite n'a pas grand sens avec cette option. L'idée des marques est d'éviter la construction de fermeture en testant si l'argument est présent. L'argument doit donc être évalué avant la fonction.

Nous donnons maintenant quelques éléments de comparaison des codes produits

par les transformations *Va* et *Vm*. Les expressions produites par *Vm* construisent moins de fermetures que le code correspondant produit par *Va* et une marque peut se coder par un bit dans une pile de bits parallèlement à la pile de d'arguments. La transformation *Vm* produit un code vraisemblablement moins demandeur d'espace mémoire en moyenne. En ce qui concerne l'efficacité, la taille des expressions compilées donne une bonne approximation du surcoût de gestion de l'ordre d'évaluation introduit par les transformations. Il est facile de montrer que dans tous les cas l'expansion de code est linéaire en fonction de la taille de l'expression source. Plus précisément,

$$\text{si } \mathit{Taille}\,[\![E]\!] = n \quad \text{alors } \mathit{Taille}\,(V\,[\![E]\!]) < 3\,n \quad \text{pour } V = \mathit{Va} \text{ ou } \mathit{Vm}$$

Une comparaison plus précise est possible en associant un coût aux combinateurs. Si on note *store* pour le coût d'empiler une variable ou une marque, *clos* pour le coût d'une construction de fermeture (*i.e.* $\mathbf{store}_s\ E$), *app* et *grab* pour le coût des combinateurs correspondants, et enfin n_l et n_v pour le nombre de λ-abstractions et d'occurrences de variables dans l'expression source, on a :

$$\mathit{Coût}\,(\mathit{Va}\,[\![E]\!]) = n_l\,\mathit{clos} + n_v\,\mathit{store} + (n_v\text{-}1)\mathit{app}$$

et

$$\mathit{Coût}\,(\mathit{Vm}\,[\![E]\!]) = (n_l + n_v)\,\mathit{grab} + (n_v\text{-}1)\mathit{store}$$

L'avantage de *Vm* sur *Va* est de remplacer parfois une construction de fermeture et un **app** par un test et un **app**. Si *clos* est comparable au coût d'un test (par exemple, quand rendre une fermeture se résume à la construction d'une paire comme dans la transformation *As* en section 2.2) *Vm* produira un moins bon code que *Va*. Si la construction de fermeture n'est pas en temps constant *Vm* peut être arbitrairement meilleur que *Va*. En fait, la complexité du code change pour des programmes pathologiques. En pratique, la situation n'est pas si claire et le gain éventuel apporté par *Vm* dépend de la probabilité de présence d'une marque et du coût moyen d'une construction de fermeture par rapport à un test.

Cet exercice de comparaison illustre le fait que notre cadre permet d'étudier des complexités théoriques, de trouver des exemples pathologiques, ou encore d'argumenter formellement sur les avantages et inconvénients des techniques de compilation. Il va sans dire que ce style de comparaison ne se substitue pas aux expérimentations, toujours utiles pour prendre en compte des aspects plus complexes comme, par exemple, les interactions avec le cache ou le GC.

2.5 Classifier

Nous avons présenté deux choix particuliers pour la compilation du schéma d'évaluation et un choix pour la gestion des environnements. Bien d'autres options existent et elles peuvent se décrire dans notre modèle. Cela ouvre la voie à une taxonomie des implémentations existantes.

2.5.1 Compilation de la stratégie d'évaluation

Pour compiler la stratégie d'évaluation (que ce soit l'appel par valeur ou l'appel par nom) trois classes de modèles se distinguent:

- *le modèle eval-apply.* La Cam utilise la version gauche-à-droite du modèle *eval-apply* pour l'appel par valeur. D'autres implémentations comme la SECD [95] et SML-NJ [6] utilisent également une variante de ce modèle. Tabac utilise également ce modèle pour compiler l'appel par valeur (version droite-à-gauche) et l'appel par nom (*cf*. transformation *Na* en section 2.6).
- *le modèle push-enter.* La version stricte de la machine de Krivine décrite dans [96] utilise (une variante de) la transformation *Vm* (Figure 2-10, section 2.4). Pour l'appel par nom, c'est un choix naturel adopté par Tim [48], Clean [126], la spineless tagless G-machine [122] et la machine de Krivine [37].
- *le modèle interprété.* Les implémentations basées sur la réduction de graphe manipulent l'expression sous la forme d'un graphe qui est réécrit plus ou moins interprétativement. La description de ce modèle dans notre cadre se trouve dans [169][190]. Un des avantages de la réduction de graphe est qu'elle représente naturellement le partage nécessaire dans l'implémentation de l'appel par nécessité. Dans la pratique, ce modèle n'est pas utilisé pour mettre en œuvre l'appel par valeur (mais rien ne l'interdirait). La G-machine [80], la spineless G-machine [19], la machine SKI [150] optent pour ce modèle.

2.5.2 Compilation de la β-réduction

Pour la compilation de la β-réduction, trois grands choix existent:

- *les environnements en liste* (partagés). Une fermeture est créée en temps constant mais l'accès aux variables est en temps linéaire (il faut de suivre une chaîne de liens dans la liste). C'est le type de gestion d'environnement utilisé par la SECD, la Cam, la machine de Krivine (stricte ou non).

- *les environnements en vecteur* (copiés). L'accès aux variables est en temps constant mais ce schéma nécessite de faire des copies d'environnement. Un avantage annexe est d'éviter certaines fuites de mémoire en ne retenant dans les fermetures que la partie utile de l'environnement. De multiples variantes existent suivant le moment où sont faites les copies et la manipulation des environnements [189]. La G-machine, les variantes *spineless* et *tagless* de la G-machine, Clean, Tim, SML-NJ, Tabac (en version stricte et paresseuse) utilisent toutes des variantes de ce modèle.
- *les environnements répartis*. Ce modèle ne manipule pas à proprement parler d'environnement mais encode séparément chaque substitution. Les algorithmes d'abstraction basés sur des ensembles de combinateurs comme {**S**, **K**, **I**} ou {**S**, **K**, **I**, **B**, **C**, **S'**, **B'**, **C'**} sont de ce type [150][151]. Il faut toutefois noter que la plupart des réducteurs de graphe optimisés utilisent des schémas qui se rapprochent du modèle précédent (avec environnements copiés).

2.5.3 Compilation des transferts de contrôle

L'évaluation d'une fermeture implique deux ruptures de séquence : l'appel de la fermeture et le retour. Pour implémenter ces transferts de contrôle, on trouve deux options : utiliser une pile de retour ou le tas. La première solution est utilisée par la plupart des mises en œuvre. La seconde solution, sans pile, est adoptée par le compilateur SML-NJ qui utilise une transformation CPS préliminaire. Les adresses de retour sont représentées par des continuations qui sont mises en œuvre par des fermetures. D'autres compilateurs utilisent également une transformation CPS mais traitent à part les variables de continuations qui sont implémentées dans la pile de retour.

2.5.4 Mise à jour des fermetures

L'appel par nécessité peut être vu comme l'appel par nom équipé d'un mécanisme de mise à jour des fermetures après leur évaluation. La technique standard est la mise à jour par l'appelé (la fermeture elle-même). De cette façon, une fermeture est remplacée par sa forme normale une seule fois. Une mise à jour par l'appelant impliquerait des mises à jour à chaque appel. La version non-optimisée de la G-machine met à jour tout nœud d'application du graphe après évaluation.

Le lecteur pourra trouver dans [169] une taxonomie classifiant une douzaine de

compilateurs selon ces différents choix de mises en œuvre.

2.6 Optimiser

Si les choix de compilation se décrivent naturellement par des transformations de programme, il en est de même pour de nombreuses optimisations utilisées par les compilateurs. Nous donnons ici quelques exemples d'optimisations locales qui s'expriment par des règles de réécriture et un exemple d'optimisation globale prise en compte par la transformation de compilation de la stratégie d'évaluation.

2.6.1 Optimisations locales

Il est clairement inutile de rendre une fonction en résultat pour l'appliquer immédiatement après. Cette optimisation s'exprime par la loi suivante :

$$\mathbf{store}_s\ F\ ;\ Va\ [\![E]\!]\ ;\ \mathbf{app}\ =\ Va\ [\![E]\!]\ ;\ F$$

Pour illustrer ce style d'optimisation, prenons le cas courant d'une fonction appliquée à tous ses arguments $(\lambda x_1 \ldots \lambda x_n.E_0)\ E_1\ \ldots\ E_n$. On peut montrer que

$$Va\ [\![(\lambda x_1 \ldots \lambda x_n.E_0)\ E_1\ \ldots\ E_n]\!] = Va\,[\![E_n]\!]\,;\ldots;\ Va\,[\![E_1]\!]\ ;\ \lambda_s x_1 \ldots \lambda_s x_n.Va\,[\![E_0]\!]$$

Ceci évite la construction de n fermetures intermédiaires correspondant aux n fonctions unaires de l'expression $\lambda x_1 \ldots \lambda x_n.E_0$. La généralisation de cette optimisation constitue la phase de décurryfication présente dans de nombreuses mises en œuvre. Soulignons que, dans notre cadre, $\lambda_s x_1 \ldots \lambda_s x_n.E$ dénote une fonction qui est toujours appliquée à n arguments (autrement, il y aurait des $\mathbf{store}_s$ entre les λ_s).

De façon similaire, la transformation *As* s'optimise par la règle :

$$As\,[\![\lambda_s x.E]\!]\,\rho = \mathbf{pop}_{se}\ ;\ As\,[\![E]\!]\,\rho \qquad \textit{si } x \textit{ n'est pas libre dans } E$$

avec $\mathbf{pop}_{se} = \lambda_e e.\ \lambda_s x.\mathbf{store}_e\ e$

qui exprime le fait que, si une fonction n'utilise pas son argument, il est inutile de le mettre dans l'environnement.

2.6.2 Optimisations globales

L'implémentation des langages non stricts est souvent optimisée à l'aide d'une analyse de nécessité. Une analyse de nécessité vise à détecter les fonctions strictes

(*i.e.* qui ont dans tous les cas besoin de leur argument). L'argument d'une fonction stricte peut être évalué avant l'appel. L'optimisation associée n'est pas directement le changement d'ordre d'évaluation mais la réduction du nombre de constructions de fermetures que ce changement permet.

La transformation compilant l'appel par nom selon le modèle *eval-apply* est décrite Figure 2-12.

$$Na : \Lambda \rightarrow \Lambda_S$$

$$Na\ [\![x]\!] = x$$

$$Na\ [\![\lambda x.E]\!] = \mathbf{store}_S\ (\lambda_S x.Na\ [\![E]\!])$$

$$Na\ [\![E_1\ E_2]\!] = \mathbf{store}_S\ (Na\ [\![E_2]\!])\ \mathbf{;}\ Na\ [\![E_1]\!]\ \mathbf{;}\ \mathbf{app}_N \quad \text{avec } \mathbf{app}_N = \lambda_S f.f$$

Figure 2-12 Compilation de l'appel par nom avec applications (*Na*)

Une variable dénote une fermeture qui doit être évaluée lors de son utilisation. Les λ-abstractions sont toujours des formes normales rendues en résultat. L'évaluation d'une application consiste à construire une fermeture représentant l'argument non évalué puis à réduire la fonction avant de l'appliquer à la fermeture.

Supposons qu'une analyse de nécessité ait annoté le code par $\underline{E}_1\ E_2$ si E_1 est une fonction stricte et $\bar{x}$ si la variable x est déclarée par une λ-abstraction stricte. Cette information est prise en compte dans la compilation de la stratégie d'évaluation en ajoutant à *Na* les deux règles :

$$Na\ [\![\bar{x}]\!] = \mathbf{store}_S\ x$$

$$Na\ [\![\underline{E}_1\ E_2]\!] = Na\ [\![E_2]\!]\ \mathbf{;}\ Na\ [\![E_1]\!]\ \mathbf{;}\ \mathbf{app}$$

Les variables surlignées sont déjà évaluées. Elles n'ont pas à être emboîtées (*boxed*) dans une fermeture et sont rendues en résultat. Les arguments d'une fonction stricte ($\underline{E}_1$) sont évalués avant l'appel.

Par exemple, sans information de nécessité, l'expression $(\lambda x.x+1)\ 2$ est compilée en :

$$\mathbf{store}_S\ (\mathbf{store}_S\ 2)\ \mathbf{;}\ (\lambda_S x.x\ \mathbf{;}\ \mathbf{store}_S\ 1\ \mathbf{;}\ \mathbf{plus}_S)$$

Le code $\mathbf{store}_S\ 2$ est placé dans une fermeture qui est évaluée par l'appel x. Dans notre cadre, $(\mathbf{store}_S\ 2)$ est la représentation emboîtée de 2. Avec les annotations de

nécessité ($\underline{\lambda x.\bar{x}+1}$) 2 est compilée en

$$\mathbf{store}_s\ 2\ \mathbf{;}\ (\lambda_s x.\mathbf{store}_s\ x\ \mathbf{;}\ \mathbf{store}_s\ 1\ \mathbf{;}\ \mathbf{plus}_s)$$

et l'évaluation est identique à une réduction en appel par valeur (aucune fermeture n'est construite).

D'autres formes plus générales de déboîtement (*cf.* [97] ou [124]) et d'autres optimisations globales (comme le *let-floating* [125]) pourraient être également être décrites. Notons en passant que si les analyses sont souvent prouvées correctes, il est beaucoup plus rare de trouver la preuve de correction de l'optimisation associée. Le fait d'exprimer l'optimisation comme une transformation de programme permet d'envisager de telles preuves. C'est l'approche suivie par [20] qui montre la correction globale de l'optimisation précédente en fonction de la spécification d'une analyse de nécessité.

2.7 Concevoir

L'étude systématique du spectre des choix de compilation favorise également la conception de nouvelles techniques de mises en œuvre. On peut, par exemple, étudier des combinaisons de transformations (stratégie d'évaluation + gestion des environnements) qui n'existent pas dans les implémentations classiques. L'étude de la compilation de la β-réduction a également suggéré toute une famille de transformations modélisant de nouveaux types de gestion des environnements [189]. Une autre piste pour proposer de nouvelles techniques de mises en œuvre est l'étude de solutions hybrides.

Nous avons vu que, pour la plupart des étapes de compilation, il n'existe pas de meilleur choix. Chaque technique a ses avantages et inconvénients suivant l'expression à compiler. Il est naturel d'essayer de combiner les techniques afin de cumuler leurs avantages. Le cadre commun permet d'exprimer des hybridations complexes et de montrer leur correction.

Nous décrivons maintenant comment *Va* et *Vm* peuvent être combinées en une seule transformation. Le choix d'utiliser un mode *eval-apply* ou *push-enter* dépend du programme et des coûts des instructions de base (tests, application, construction de fermeture). Notre but ici n'est pas de concevoir une analyse de ces coûts mais de décrire comment la compilation pourrait profiter d'une telle information.

Supposons qu'une analyse statique ait produit un code annoté indiquant le

meilleur mode (*eval-apply* ou *push-enter*) pour chaque sous-expression sous la forme de types :

$$T ::= a \mid m \mid T_1 \overset{a/m}{\rightarrow} T_2$$

avec *a* (resp. *m*) pour le mode *eval-apply* (resp. *push-enter*) mode. Intuitivement, une fonction $E : \alpha \overset{\delta}{\rightarrow} \beta$ prend un argument évalué dans le mode α, le corps de la fonction est évalué dans le mode δ. Ce style d'annotation impose que chaque variable soit évaluée dans un mode fixé.

$$MixV[\![x^{\alpha}]\!] = \mathbf{X}_{\alpha}\, x$$

$$MixV[\![(\lambda x.E)^{\alpha \overset{\delta}{\rightarrow} \beta}]\!] = \mathbf{X}_{\delta}\, (\lambda_s x.MixV[\![E]\!])$$

$$MixV[\![E_1{}^{\alpha \overset{\delta}{\rightarrow} \beta}\, E_2{}^{\alpha}]\!] = \mathbf{Y}_{\alpha}\ ;\ MixV[\![E_2]\!]\ ;\ MixV[\![E_1]\!]\ ;\ \mathbf{Z}_{\delta}$$

avec $\mathbf{X}_a = \mathbf{store}_s$ $\quad \mathbf{Y}_a = \mathbf{Id}$ $\quad \mathbf{Z}_a = \mathbf{app}$

$\mathbf{X}_m = \mathbf{grab}_s$ $\quad \mathbf{Y}_m = \mathbf{store}_s\, \varepsilon$ $\quad \mathbf{Z}_m = \mathbf{Id}$

Figure 2-13 Compilation hybride de l'appel par valeur (version droite à gauche)

On suppose comme dans la section 2.4 que la marque ε est une valeur distinguable des autres. *MixV* (Figure 2-13) ajoute $\mathbf{store}_s\ \varepsilon$ avant l'évaluation d'un argument en mode *m* et **app** après l'évaluation d'une fonction en mode *a*. Les résultats sont retournés par $\mathbf{store}_s$ ou $\mathbf{grab}_s$ selon leur mode d'évaluation.

De la même façon, on peut exprimer une gestion des environnements mélangeant des représentations en liste (partagés) et en vecteur (copiés) [169]. Ce type de représentation hybride des environnements a également été proposé dans [143].

2.8 Réduire sous les lambdas

Les compilateurs de langages fonctionnels ne mettent en œuvre que la réduction faible (on ne réduit pas sous les lambdas). En certaines occasions il est utile, voire nécessaire, d'évaluer des formes normales de tête ou de véritables formes normales. En particulier, la réduction de tête ou forte est utilisée par :

- des transformations de programme comme l'évaluation partielle qui simplifient le corps des fonctions,

- la programmes logiques d'ordre supérieur comme λ-prolog [116] où l'unification implique la mise en forme normale (forte) des λ-termes,
- l'évaluation de structures de données codées par des λ-expressions,
- des stratégies d'évaluations essayant de partager au maximum les calculs.

Nous indiquons ici comment la compilation de la réduction de tête peut s'exprimer dans notre cadre (en fait Λ_s). Nous nous contentons de donner l'intuition. Une description précise et complète se trouve dans [173].

L'idée est d'appliquer une expression de Λ_s à une continuation particulière afin que sa forme normale (forte ou de tête) soit évaluée par l'habituelle réduction faible. L'expression source E est transformée en une expression $Na\ [\![E]\!]$ de Λ_s (*Na* est la transformation *eval-apply* compilant l'appel par nom, *cf.* section 2.6). L'expression à réduire est appliquée à un entier qui sert à compter le nombre de lambdas et un combinateur **enter** :

$$\mathbf{store}_s(0)\ ;\ Na\ [\![E]\!]\ ;\ \mathbf{enter}$$

La réduction de ce style d'expressions se passe ainsi :

- $Na\ [\![E]\!]$ est réduite en forme normale faible (de la forme $\mathbf{store}_s(\lambda_s x.Na\ [\![F]\!])$)

$$\mathbf{store}_s(n)\ ;\ Na\ [\![E]\!]\ ;\ \mathbf{enter} \bullet\!\!\xrightarrow{*} \mathbf{store}_s(n)\ ;\ \mathbf{store}_s(\lambda_s x.Na\ [\![F]\!])\ ;\ \mathbf{enter}$$

- Le combinateur **enter** a une définition récursive telle que

$$\mathbf{store}_s(n)\ ;\ \mathbf{store}_s(E)\ ;\ \mathbf{enter} \bullet\!\!\rightarrow \mathbf{store}_s(n+1)\ ;\ \mathbf{store}_s(\mathbf{rebuild}_n)\ ;\ E\ ;\ \mathbf{enter}$$

La règle du combinateur **enter** incrémente le compteur et applique le combinateur **rebuild** à la forme normale faible trouvée. L'application de cette règle donne :

$$\bullet\!\!\xrightarrow{*} \mathbf{store}_s(n+1)\ ;\ \mathbf{store}_s(\mathbf{rebuild}_n)\ ;\ (\lambda_s x.Na\ [\![F]\!])\ ;\ \mathbf{enter}$$

- Le combinateur $\mathbf{rebuild}_n$ est substitué à x et la réduction du corps se poursuit

$$\bullet\!\!\xrightarrow{*} \mathbf{store}_s(\mathrm{n}+1)\ ;\ Na\ [\![\mathrm{F}]\!]\ [\mathbf{rebuild}_n/\mathrm{x}]\ ;\ \mathbf{enter}$$

et ainsi de suite jusqu'à ce que la forme normale de tête soit atteinte. Dans ce cas, une variable (en fait un combinateur $\mathbf{rebuild}_n$) se trouve en position de réduction. Le rôle de ce combinateur est de reconstruire l'expression, c'est à dire de réintroduire les λ-abstractions supprimées et de remplacer les combinateurs $\mathbf{rebuild}_n$ par les va-

riables correspondantes. Les définitions (en λ-calcul pur) de ces combinateurs sont données dans [173][187].

La réduction forte se compile de manière similaire. Un nouveau jeu de combinateurs est introduit qui poursuit la réduction au lieu de reconstruire dès que la variable de tête est atteinte. Si on peut parler de compilation dans le sens où il n'y a plus de recherche de rédex, la gestion d'un index comptant les lambdas est assez interprétative. Ceci est nécessaire dans le cas général où on ne peut pas connaître statiquement le nombre de lambdas de la forme normale d'une expression. Dans le cas typé (un système de types très lâche est suffisant pour fixer la fonctionnalité des expressions), ce comptage n'est pas nécessaire.

En restant dans le cadre du λ-calcul, nous n'avons pas à concevoir de machine abstraite spécialisée et cette technique autorise un compilateur standard à réduire sous les lambdas. Une façon triviale de faire est en partant d'une expression *E*, de produire l'expression $\mathbf{store}_s(0)$ **;** *Na* $[\![E]\!]$ **; enter**. Après instanciation, cette expression est une λ-expression qui peut être donnée à un compilateur standard. Cela permet de réutiliser toutes les techniques (analyses, optimisations) de compilation développées dans le cadre de la réduction faible. L'exécution (*i.e.* la réduction faible) du code machine obtenu simulera la réduction de tête de l'expression.

2.9 Travaux connexes

Une multitude de travaux autour de la compilation des langages fonctionnels peuvent être considérés comme connexes. Nous nous contentons de passer en revue les différents formalismes (λ-calcul, substitutions explicites, combinateurs, monades) utilisés pour décrire les implémentations de langages fonctionnels ainsi que les travaux comparant des implémentations spécifiques.

Nos premiers travaux sur le sujet avaient pour but d'exprimer complètement la compilation par transformation de programme. Les étapes étaient la compilation de la stratégie d'évaluation par une transformation CPS et la compilation de la β-réduction à l'aide de combinateurs qui pouvaient être vus comme des instructions d'une machine à pile. Le compilateur SML-NJ a également été décrit par des transformations de programme [6]. D'autres compilateurs utilisent la transformation CPS comme étape préliminaire [93][144]. Le codage de l'implémentation dans le λ-calcul conduit à des expressions fonctionnelles complexes (*e.g.* le séquencement est codé par des compositions de continuations). Les combinateurs **;**, $\mathbf{store}_i$ et $\mathbf{fetch}_i$ rendent

notre cadre plus abstrait et simplifient les expressions manipulées. Certains choix d'implémentation comme la représentation des composants de la machine abstraite n'interfèrent pas avec le reste de la compilation. Dans le λ-calcul pur, on doit choisir *a priori* si les données et les environnements résident dans le même composant ou pas.

Le λ-calcul de de Bruijn [41] utilise des indices au lieu de noms de variable. Un indice de de Bruijn peut être vu comme l'adresse d'une valeur dans l'environnement. Il a été utilisé en tant que langage intermédiaire pour expliciter l'accès aux variables par certaines mises en œuvre. Une collection de formalismes, les λ-calculs à substitutions explicites, sont bien adaptés à modéliser la gestion des environnements [1]. Ces calculs facilitent le raisonnement sur les substitutions et rendent explicites certains détails d'implémentation. Cependant, des choix importants comme la représentation des environnements (listes ou vecteurs) ne sont pas traités. Hardin *et al.* [65] introduisent un λ-calcul à substitutions explicites ($\lambda\sigma_w$) et décrivent plusieurs machines abstraites. Le but de ce travail est faire ressortir les points communs de techniques d'implémentation, pas de modéliser précisément les mises en œuvre existantes. Un autre calcul ($\lambda\sigma_w^a$) facilite la description du partage et les preuves de correction de mises à jour en place [13]. Les $\lambda\sigma_w^a$ -expressions restent assez éloignées d'un véritable code machine puisque, par exemple, le partage est représenté par des étiquettes et des réductions parallèles.

Les combinateurs ont été utilisés pour décrire la compilation de la β-réduction. Citons les combinateurs {**S**, **K**, **I**, **B**, **C**} [38][150][151] qui ont été utilisés par des réducteurs de graphe et les combinateurs catégoriques utilisés par des machines à environnement comme la Cam [36] et la machine de Krivine [37].

Même si les buts et points de départ sont différents, notre premier langage intermédiaire (Λ_s) est très proche du métalangage de Moggi [112]. En particulier, on peut interpréter la construction monadique **[***E***]** comme **store**$_s$ E et (**let** $x \Leftarrow E_1$ **in** E_2) comme E_1 **;** $\lambda_s x.E_2$. On retrouve alors les lois monadiques (let.β), (let.η) and (ass). Il n'est pas toujours facile d'exprimer des détails d'implémentation ou d'aller jusqu'au code machine avec des monades. De plus, un point clé de notre approche est de décrire la compilation comme une composition de transformations indépendantes. Le cadre monadique ne semble pas bien adapté puisque la composition de monades est délicate. Liang *et al.* [100] introduisent des monades paramétrisées complexes pour pouvoir décrire et composer des étapes de compilation. Le cadre monadique est plus une approche générale pour structurer les programmes fonction-

nels [159] alors que notre cadre a été conçu sur mesure pour décrire des mises en œuvre.

Peu de mises en œuvre de langage fonctionnel ont été comparées au-delà des mesures de performances. Quelques travaux ont exploré les relations entre deux machines spécifiques comme la CMCM et Tim [101] ou Tim et la G-machine [123]. Leur but est de montrer l'équivalence d'implémentations apparemment très différentes. La CMCM et Tim sont comparées en définissant des transformations entre les états des deux machines. La comparaison de Tim et de la G-machine est plus informelle mais souligne les relations entre machine à environnement et réducteur de graphe. Mentionnons également Asperti [8] qui présente la machine de Krivine et la Cam dans un cadre catégorique et Crégut [37] qui compare Tim et la machine de Krivine. Toutes ces comparaisons se concentrent sur des étapes de compilation ou machines particulières.

2.10 Conclusion et perspectives

Notre cadre nous a permis de couvrir une grande partie du spectre des implémentations (strict, paresseux, à environnement, par réduction de graphe) et nous avons dressé une classification incluant une douzaine de mises en œuvre standards. Nous avons comparé formellement (sous la forme de calcul de complexité) certains choix de compilation et montré dans certains cas, que deux implémentations apparemment très différentes partageaient un choix de mise en œuvre. Si la plus grande part de l'étude s'est concentrée sur les implémentations classiques et le λ-calcul pur, nous avons également modélisé plusieurs extensions : les opérateurs et structures de données, la récursion, des optimisations et la conception de mises en œuvre hybrides [43][169][189][190].

L'utilisation de transformations de programme s'est révélée particulièrement bien adaptée à la description du processus de compilation et des optimisations associées (décurryfication, déboîtement, optimisations locales). Le cadre de description nous semble posséder plusieurs avantages :

- Il repose sur des *bases formelles solides*. Chaque langage intermédiaire peut être vu soit comme un système formel avec ses propres règles de conversion, soit comme un sous-ensemble du λ-calcul. Les langages intermédiaires offrent une collection de lois et propriétés, la plus importante étant que toutes les stratégies de réduction sont normalisantes. Cela facilite les transformations de programme,

les preuves de correction et les comparaisons. Nous n'avons pas à introduire explicitement une machine abstraite et à montrer que les transitions d'états sont cohérentes avec la sémantique du langage (comme dans [102] ou [99]). Nos preuves reviennent à montrer indépendamment la correction de transformations de programme. Des études de complexité et comparaisons formelles peuvent être entreprises en calculant l'expansion de code engendrée par les transformations.

- Il est (relativement) *abstrait*. Nous voulions décrire complètement et précisément les implémentations. Les langages intermédiaires devaient donc se rapprocher d'un code machine au fur et à mesure de la compilation. Le cadre produit néanmoins des descriptions plus abstraites qu'en λ-calcul. La compilation du contrôle est, par exemple, plus abstraite qu'un codage en CPS [51][127][66]. Les combinateurs des langages intermédiaires et leurs règles de conversion encodent naturellement et précisément des notions bas niveau comme instructions, séquencement, piles, etc.
- Il est *modulaire*. Chaque transformation modélise une étape de compilation et est définie indépendamment des autres étapes. Les transformations décrivant différentes étapes se composent librement pour spécifier des compilateurs. Les transformations modélisant la même étape représentent différentes options qui peuvent ainsi être comparées.
- Il est *extensible*. Rien n'interdit l'ajout de nouveaux langages et transformations dans la séquence de compilation. Ceci serait utile pour décrire de nouvelles étapes de compilation comme l'allocation de registres.

Notre but principal était de structurer et de clarifier l'ensemble des possibilités d'implémentation de langages fonctionnels. Les compositions inédites de transformations, la dérivation de nouvelles techniques de gestion de l'environnement et les transformations hybrides montrent que notre approche n'est pas limitée à la description de l'existant mais qu'elle suggère aussi de nouvelles techniques d'implémentation. De nombreuses extensions sont envisageables :

- Il serait intéressant de concrétiser le cadre en un atelier de construction de compilateurs. Cela permettrait d'implémenter et d'expérimenter une grande variété d'implémentations juste en composant les transformations. On pourrait essayer de nouvelles combinaisons et évaluer les différents choix et optimisations en pratique. Un tel atelier serait aussi un outil pédagogique intéressant.

- Une étude systématique des optimisations et transformations de programme permettrait de clarifier leur impact suivant les choix de mise en œuvre. Le λ-lifting, par exemple, est une transformation controversée [80][104]. Intuitivement, le λ-lifting est bénéfique pour des implémentations basées sur des environnements en liste. Son effet est de raccourcir l'accès aux variables en effectuant des copies. Suivant le nombre d'accès aux variables le gain peut excéder le surcoût des copies. Cette question pourrait être étudiée précisément dans notre cadre. De plus, la preuve de correction des optimisations basées sur des analyses de programmes est un problème important trop souvent négligé [20]. L'expression de ces optimisations comme des transformations de programme devrait faciliter les choses.
- Une autre direction de recherche est l'étude de transformations hybrides (mélangeant plusieurs techniques). Nous avons indiqué une solution pour combiner les modèles *eval-apply* et *push-enter* en section 2.7. Nous avons également étudié l'association d'environnements en liste et en vecteur [169]. Les analyses associées et d'autres hybridations attendent d'être conçues. Sans l'aide d'un cadre formel simple, l'expression et la preuve de telles techniques seraient particulièrement difficiles.
- De nombreuses comparaisons formelles de transformations attendent d'être faites. Nous nous sommes contentés de comparer trois couples de transformations (*Va* et *Vm* en section 2.4, *Na* et *Nm* dans [190], *As* et *Ac1* dans [189]). Il est possible que le choix fait à une étape détermine le meilleur choix pour une autre étape. Ceci pourrait se démontrer en comparant des compositions de transformations.

Nous pensons que le travail déjà accompli montre que notre cadre est assez simple et expressif pour accueillir et traiter toutes ces extensions.

CHAPITRE 3 *Optimisations et typage*

Dans ce chapitre, nous décrivons deux optimisations de l'implémentation des programmes fonctionnels basées sur la syntaxe et le type des expressions. La première optimisation autorise l'utilisation de variables globales dans la mise en œuvre et la deuxième améliore la gestion mémoire.

Le chaîne de compilation décrite précédemment ne comporte pas de phase d'allocation de registres. Le passage des arguments et l'évaluation d'expression sollicitent uniquement la pile. L'utilisation naïve de registres est peu intéressante en raison des nombreux changements de contexte dus aux appels récursifs. Les registres apportent un véritable gain en performance lorsqu'ils implantent des variables globales (modifiables en place). La notion de variable globale n'existe pas dans un langage fonctionnel pur mais il reste possible d'en utiliser pour sa mise en œuvre. Pour ce faire, il faut préalablement analyser le programme afin d'assurer qu'une telle optimisation ne change pas la sémantique. Nous décrivons en section 3.1 une analyse de globalisation comme un ensemble de critères syntaxiques (sur la structure et le type des expressions). Cette approche langage est de complexité linéaire et indique en cas d'échec les sous-expressions fautives. Ce style d'information peut être utilisé par le compilateur ou l'utilisateur pour transformer le programme. Comparées aux analyses sémantiques, le principal inconvénient des analyses syntaxiques est leur manque de précision. Pour remédier (partiellement) à ce problème, nous définissons notre analyse sur les expressions de Λ_s (*cf.* section 2.1.1). L'analyse est plus précise car elle prend en compte l'ordre d'évaluation qui est explicite dans la syntaxe. De plus, cela permet de définir l'analyse de globalisation pour toute stratégie d'évaluation séquentielle (*cf.* section 3.1.6).

Un des problèmes majeurs des langages fonctionnels est leur consommation prohibitive de mémoire. En particulier, de nombreux programmes fonctionnels comportent des "fuites de mémoire" : des références sont conservées sur des structures devenues inutiles qui ne sont pas récupérées par les glaneurs de cellules (GC) classi-

ques. De tels programmes allouent énormément d'espace ou épuisent la mémoire avant de pouvoir rendre leur résultat. Nous décrivons en section 3.2, une méthode pour récupérer plus de mémoire que les GC traditionnels. Cette technique est basée sur le type polymorphe des expressions et le théorème de paramétricité. Par exemple, la paramétricité formalise le fait qu'une fonction de type $List\ \alpha \rightarrow int$ n'a pas besoin des éléments de sa liste argument pour produire son résultat. Le GC étendu utilise ces informations de type pour détecter et récupérer des (parties de) structures inutiles toujours référencées. Cette technique de GC se rattache à une approche langage dans le sens où elle repose sur le type des expressions. Elle résout plusieurs sortes de fuites de mémoire très communes et s'applique à tout type de langage fonctionnel fortement typé (d'ordre supérieur, strict ou paresseux, pur ou à effets de bord) et de GC (*stop&copy*, *mark&sweep*, etc.).

3.1 Analyse syntaxique de globalisation

L'analyse de globalisation a pour but de détecter si une structure de donnée peut être implémentée par une variable globale (*i.e.* être mise à jour en place). Cette information a plusieurs applications :

- *L'allocation de registres*. Un paramètre de fonction globalisable peut être mis en œuvre à l'aide d'un registre. Un appel récursif terminal avec un nouvel argument sera implémenté par une mise à jour du registre qui n'a pas à être sauvegardé dans la pile.
- *La mise en œuvre de tableaux*. Les structures de données contiguës comme les tableaux sont très utilisées pour leur temps d'accès constant. L'implémentation fonctionnelle naïve des tableaux est terriblement inefficace. En effet, pour conserver la propriété de transparence référentielle, un tableau est copié avant chaque mise à jour (car l'ancienne version du tableau est peut-être toujours référencée). Une analyse de globalisation permet d'éviter les copies de tableau inutiles.
- *La génération de compilateur à partir d'une sémantique dénotationnelle*. L'idée ici est de produire du code en évaluant partiellement les équations sémantiques appliquées à un programme. Une sémantique dénotationnelle représente l'état (la mémoire) comme un argument des équations sémantiques [139][145]. L'évalua-

tion partielle directe produirait du code effectuant une copie de l'état à chaque modification. Pour espérer dériver des compilateurs réalistes, il faut préalablement s'assurer que l'état est modifiable en place.

Deux approches ont été considérées pour l'analyse de globalisation : l'approche syntaxique [138] et l'approche sémantique (basée sur une interprétation abstraite) [15][44][72][142]. L'interprétation abstraite donne des résultats plus précis au prix d'une complexité exponentielle. Les analyses syntaxiques sont moins précises mais ont deux avantages :

- ce sont des analyses peu coûteuses (typiquement en temps linéaire),
- leurs résultats sont maîtrisables par le programmeur.

En cas d'erreur, l'analyse permet d'exhiber les sous-expressions fautives (qui ne respectent pas les critères syntaxiques ou de typage). Cette information peut être utilisée pour corriger le programme, manuellement ou automatiquement (section 3.1.5).

Nous décrivons dans les sections suivantes une analyse syntaxique de globalisation prenant en compte l'ordre supérieur, la plupart des stratégies séquentielles d'évaluation, tout en étant de complexité linéaire et d'une précision suffisante. Ce compromis est atteint en travaillant sur le langage Λ_s (*cf.* section 2.1)[*]. L'ordre d'évaluation est explicite dans la syntaxe des expressions de Λ_s. L'analyse de globalisation devient à la fois plus simple et plus précise sur ce style d'expressions.

Nous commençons par donner l'idée du problème et de notre solution en section 3.1.1. L'analyse est présentée sous la forme de critères syntaxiques en section 3.1.2 et la transformation associée (explicitant l'utilisation de registres) est donnée en section 3.1.3. Son application à l'allocation de registres est décrite en section 3.1.4. Nous avons suggéré qu'un avantage de l'analyse est de retourner de l'information exploitable en cas d'échec. Cette idée est illustrée en section 3.1.5 où l'on montre comment certaines expressions violant les critères peuvent être transformées en expressions équivalentes les respectant. En composant l'analyse et la compilation d'une stratégie d'évaluation (*e.g.* la transformation *Va* de la section 2.2.1) il est pos-

[*] A l'origine, ce travail (analyse, transformations, etc.) fut exprimé sur des expressions en CPS. Pour des raisons d'homogénéité, nous avons choisi de le reformuler dans le cadre présenté au chapitre 2. Nous n'avons toutefois pas refait tout le traitement formel (*e.g.* les preuves de correction) dans ce cadre. La présentation qui suit s'attache avant tout à donner l'intuition de l'approche.

sible de dériver des analyses sur les λ-expressions standards. Nous décrivons l'analyse obtenue pour l'appel par valeur en section 3.1.6. Nous terminons par un rapide survol des travaux apparentés en section 3.1.7.

3.1.1 Préliminaires

Afin de donner l'idée du problème et de notre solution, considérons l'expression suivante :

$$f\ (update\ a\ i\ p)\ (access\ a\ i)$$

La fonction *f* prend la valeur du *i*ème élément du tableau *a* et le tableau *a* avec le *i*ème élément remplacé par *p*. Une implémentation naïve doit copier le tableau avant la mise à jour. Le tableau peut-il être représenté par une variable globale et mis à jour en place ? En supposant une sémantique par valeur pour notre exemple, deux cas se présentent :

- la mise à jour (*update*) est faite avant l'accès (*access*). Dans ce cas, le tableau n'est pas globalisable et une copie est nécessaire,
- l'accès est effectué avant la mise à jour et le tableau peut être modifié en place.

Ceci est facilement détecté sur les expressions de Λ_s où la stratégie d'évaluation a été explicitée dans la syntaxe.

Remarquons tout d'abord que c'est à la fois *a* et *update a i p* que nous cherchons à représenter par une même variable globale. En général, l'entité à globaliser est une séquence de valeurs. On utilise les types pour exprimer la propriété de globalisation. On dit qu'un type ρ est globalisable dans l'expression *E* si toutes ses instances créées lors de l'évaluation de *E* peuvent se représenter par une unique variable globale. Dans le reste de la section, nous utilisons ρ pour dénoter le type candidat à la globalisation. On suppose que ρ est un type basique (non fonctionnel).

Comme l'analyse de globalisation repose sur les types, nous introduisons une version typée de Λ_s en Figure 3-1. Les restrictions imposées par le système de type portent sur la combinaison des résultats et des fonctions. Le typage impose que dans une composition E_1 **;** E_2, E_1 dénote un résultat (*i.e.* est de type $R\sigma$, R étant un constructeur de type) et E_2 dénote une fonction. Les transformations présentées au chapitre précédent produisent des expressions bien typées. De la même façon que l'on peut instancier les combinateurs **;**, $\mathbf{store}_s$ et $\mathbf{fetch}_s$ pour retrouver des expressions en CPS, on peut interpréter R et $\rightarrow_s$ pour retrouver les types des expressions en CPS

(*e.g.* $\mathsf{R}\sigma = (\sigma \to Ans) \to Ans$).

$$\frac{\Gamma \vdash E : \sigma}{\Gamma \vdash \mathbf{store}_s\ E : \mathsf{R}\sigma} \qquad \frac{\Gamma \cup \{x{:}\sigma\} \vdash E : \tau}{\Gamma \vdash \mathbf{fetch}_s\ (\lambda x.E) : \sigma \to_s \tau}$$

$$\frac{\Gamma \vdash E_1 : \mathsf{R}\sigma \quad \Gamma \vdash E_2 : \sigma \to_s \tau}{\Gamma \vdash E_1\ ;\ E_2 : \tau} \qquad \frac{}{\Gamma \cup \{x{:}\sigma\} \vdash E : \tau}$$

Figure 3-1 Sous ensemble typé de Λ_s

En compilant l'exemple précédent selon une stratégie d'appel par valeur de gauche à droite (*Va* en Figure 2-2), on obtient :

$$\mathbf{store}_s\ f\ ;\ \mathbf{store}_s\ p\ ;\ \mathbf{store}_s\ i\ ;\ \mathbf{store}_s\ a^{\rho}\ ;\ update_s^{\rho \to_s int \to_s int \to_s R\rho}\ ;$$

$$\mathbf{store}_s\ i\ ;\ \mathbf{store}_s\ a^{\rho}\ ;\ access_s^{\rho \to_s int \to_s Rint}\ ;\ \mathbf{app}\ ;\ \mathbf{app}$$

Un opérateur op_s (*e.g.* $update_s$, $access_s$) de Λ_s a la même sémantique que l'opérateur correspondant *op* dans le langage source mais prend ses arguments et rend son résultat dans le composant *s*. Un opérateur créant une nouvelle instance de type ρ (comme *update*) a un type de la forme $\tau \to_s \ldots \to_s \mathsf{R}\rho$. On voit clairement sur l'exemple qu'une variable libre de type ρ apparaît dans la continuation de $update_s$. Le type ρ n'est pas implémentable par une variable globale puisqu'elle serait mise à jour par $update_s$ alors que l'ancienne valeur est toujours référencée par une variable libre.

Avec une stratégie droite à gauche, on obtient après simplifications :

$$\lambda_s f.\lambda_s a^{\rho}.\lambda_s i.\lambda_s p.\ \mathbf{store}_s\ i\ ;\ \mathbf{store}_s\ a^{\rho}\ ;\ access^{\rho \to_s int \to_s Rint}\ ;$$

$$\mathbf{store}_s\ p\ ;\ \mathbf{store}_s\ i\ ;\ \mathbf{store}_s\ a^{\rho}\ ;\ update_s^{\rho \to_s int \to_s int \to_s R\rho}\ ;\ f$$

Ici, aucune variable libre de type ρ n'apparaît dans la continuation d'un modificateur et nous verrons plus loin que ρ est bien globalisable dans cette expression.

3.1.2 Critères syntaxiques

Les critères syntaxiques sont exprimés sous la forme d'une formule logique $\Phi_\rho(E)$ vraie si le type ρ est globalisable dans *E*. Φ_ρ est définie récursivement sur la

structure des expressions en Figure 3-3. Le prédicat auxiliaire $\Theta_\rho(E)$ est vrai si de plus l'expression E ne crée aucune nouvelle instance de type ρ. Dans certaines terminologies, on dit que Φ_ρ caractérise les écrivains et Θ_ρ les lecteurs.

Définition 3-2 *Le type* ρ *est dit globalisable dans l'expression E si* $\Phi_\rho(E)$.

$$\Phi_\rho(\mathbf{store}_s\ x) = \Phi_\rho(x^\sigma) = \Phi_\rho(\mathrm{op}^\sigma) = \mathit{tt}$$

$$\Phi_\rho(\lambda_s x.E) = \Phi_\rho(E)$$

$$\Phi_\rho(\mathbf{store}_s\ E^\sigma) = \mathit{nfv}_\rho(E) \wedge \mathit{si}\ \sigma \not\equiv \tau \rightarrow_s \ldots \rightarrow_s \mathsf{R}\rho\ \mathit{alors}\ \Theta_\rho(E)\ \mathit{sinon}\ \Phi_\rho(E)$$

$$\Phi_\rho((E_1\ ;\ E_2)^\sigma) = \Theta_\rho(E_1) \wedge \Phi_\rho(E_2) \vee \mathit{nfv}_\rho(E_2) \wedge \mathit{nt}_\rho(\sigma) \wedge \Phi_\rho(E_1) \wedge \Phi_\rho(E_2)$$

$$\Theta_\rho(\mathbf{store}_s\ x) = \mathit{tt}$$

$$\Theta_\rho(x^\sigma) = \Theta_\rho(\mathrm{op}^\sigma) = \sigma \not\equiv \tau \rightarrow_s \ldots \rightarrow_s \mathsf{R}\rho$$

$$\Theta_\rho(\lambda_s x.E) = \Theta_\rho(E)$$

$$\Theta_\rho(\mathbf{store}_s\ E^\sigma) = \mathit{nfv}_\rho(E) \wedge \mathit{si}\ \sigma \not\equiv \tau \rightarrow_s \ldots \rightarrow_s \mathsf{R}\rho\ \mathit{alors}\ \Theta_\rho(E)\ \mathit{sinon}\ \Phi_\rho(E)$$

$$\Theta_\rho(E_1\ ;\ E_2) = \Theta_\rho(E_1) \wedge \Theta_\rho(E_2)$$

Figure 3-3 Critères de globalisation

Une nouvelle instance de type $\mathsf{R}\rho$ peut être créée par un opérateur (*e.g.* update_s^ρ $^{\rightarrow s\ \mathit{int} \rightarrow s\ \mathit{int}\ \rightarrow s\ \mathsf{R}\rho}$), une constante (*e.g.* $\mathbf{store}_s\ \mathrm{K}^\rho$) ou l'évaluation d'une fermeture. Intuitivement, la création d'une nouvelle instance ne peut se faire que s'il n'y a plus de références à d'autres instances. Ceci se détecte syntaxiquement en vérifiant les deux points suivants :

- Il n'y a pas de variables libres de type ρ dans la continuation. Dans le cas contraire, on ne peut éviter une copie puisque la suite du calcul fait référence *via* une variable libre à une ancienne instance. On utilise le prédicat $\mathit{nfv}_\rho(E)$ qui est vrai si E ne contient pas de variables libres de type τ.

- Aucune valeur de type ρ n'est présente dans la pile (le composant *s*) de l'expression courante. Ceci constituerait également une référence à une ancienne instance. Ce point est vérifié sur le type de l'expression $(E_1 \,;\, E_2)^\tau$ par le prédicat $nt_\rho(\tau)$ défini par : $nt_\rho(\tau) = si\ \tau \equiv \tau_1 \to \tau_2\ alors\ \tau_1 \not\equiv \rho \wedge nt_\beta(\tau_2)\ sinon\ \mathbf{tt}$.

Ces deux critères expliquent la règle pour la séquence. Le type ρ est globalisable dans $(E_1 \,;\, E_2)^\tau$ si le type ρ est globalisable dans chaque sous-expression et dans le cas où E_1 crée de nouvelles instances, ni la continuation, ni la pile ne doivent contenir de références à d'autres instances de ρ ($nfv_\rho(E_2) \wedge nt_\rho(\tau)$). Par exemple, l'expression

$$\lambda_s x^\rho.\ \mathbf{store}_s\ x^\rho \,;\, succ^{\rho \to_s \mathsf{R}\rho} \,;\, \mathbf{store}_s\ x^\rho \,;\, succ^{\rho \to_s \mathsf{R}\rho} \,;\, E$$

ne respecte pas le critère, puisque *succ* est un écrivain (il crée $(x+1)^\rho$) et une variable libre de type ρ apparaît dans la continuation. Par contre, l'expression

$$\lambda_s x^\rho.\ \mathbf{store}_s\ x^\rho \,;\, chr^{\rho \to_s \mathsf{R}char} \,;\, \mathbf{store}_s\ x^\rho \,;\, succ^{\rho \to_s \mathsf{R}\rho} \,;\, E$$

respecte le critère (en supposant *E* fermée) puisque *chr* est un lecteur.

Les fermetures ($\mathbf{store}_s\ E^\sigma$) posent deux problèmes spécifiques :

- Il n'est en général pas possible de savoir quand la fermeture sera appliquée. On impose qu'aucune variable libre de type ρ n'apparaisse dans les fermetures.
- On ne sait pas *a priori* si une variable *x* sera liée à un écrivain ou un lecteur. Par exemple, dans une expression de la forme $x^\rho \,;\, E$, l'évaluation de *x* correspond à un appel de fermeture. Les critères à imposer sur la continuation *E* diffèrent si *x* est liée à une fermeture créant de nouvelles instances ou non. Une solution possible serait de considérer que toute fermeture peut potentiellement créer de nouvelles instances et d'imposer aux expressions de la forme $x^\rho \,;\, E$ que $nfv_\rho(E) \wedge nt_\rho(E) \wedge \Phi_\rho(E)$. Une autre solution serait d'imposer que les fermetures soient des lecteurs et le critère pour une expression $x^\rho \,;\, E$ se réduit à $\Phi_\rho(E)$. La solution que nous prenons est de considérer qu'une fermeture F^σ rendant en résultat un élément de type ρ (*i.e.* $\sigma \equiv \tau \to_s \ldots \to_s \mathsf{R}\rho$) est un écrivain (et on impose $\Phi_\rho(F)$) et sinon un lecteur (et on impose $\Theta_\rho(F)$). Cette solution s'exprime par les règles $\Phi_\rho(\mathbf{store}_s\ E^\sigma)$, $\Theta_\rho(\mathbf{store}_s\ E^\sigma)$ et $\Theta_\rho(x^\sigma)$.

Exemple 3-4 On peut vérifier que le type ρ est globalisable dans l'expression

$E \equiv \mathbf{store}_s\ i \,;\, \mathbf{store}_s\ a \,;\, access_s^{\rho \to_s int \to_s \mathsf{R}int} \,;$

$$\mathbf{store}_s\ p\ \mathbf{;}\ \mathbf{store}_s\ i\ \mathbf{;}\ \mathbf{store}_s\ a^{\rho}\ \mathbf{;}\ update_s^{\rho\ \to s\ int \to s\ int\ \to s\ R\rho}\ \mathbf{;}\ f$$

En effet :

$$\begin{aligned}
\Phi_\rho(E) &= \Phi_\rho(\mathbf{store}_s\ i\ \mathbf{;}\ \ldots) \\
&= \Theta_\rho(\mathbf{store}_s\ i) \wedge \Phi_\rho(\mathbf{store}_s\ a\ \mathbf{;}\ \ldots) \\
&= \mathit{tt} \wedge \Phi_\rho(\mathbf{store}_s\ a\ \mathbf{;}\ \ldots) \\
&= \ldots \\
&= \Phi_\rho(update_s^{\rho\ \to s\ int \to s\ int\ \to s\ R\rho}\ \mathbf{;}\ f) \\
&= nfv_\rho(f) \wedge nt_\rho(^{\rho\ \to s\ int \to s\ int\ \to s\ R\rho}) \wedge \Phi_\rho(update_s) \wedge \Phi_\rho(f) \\
&= \mathit{tt}
\end{aligned}$$

❒

Il existe d'autres solutions plus précises pour le traitement des fermetures. Une analyse sémantique détectant un sur-ensemble des fermetures pouvant être liées à chaque variable (comme l'analyse de [142]) apporterait clairement un gain en précision. Mais cette analyse est potentiellement coûteuse. Une solution syntaxique plus précise que celle que nous avons présenté serait de construire pour le programme l'ensemble des types des fermetures écrivains M (*i.e.* $\sigma \in M$ ssi il existe une fermeture F de type σ telle que $\Phi_\rho(F) \wedge \neg\Theta_\rho(F)$). Le critère $\Phi_\rho(\mathbf{store}_s\ E^{\sigma})$ serait alors : $nfv_\rho(E) \wedge$ *si* $\nu \in M$ *alors* $\Theta_\rho(E)$ *sinon* $\Phi_\rho(E)$.

3.1.3 Transformation associée

La transformation de globalisation s'exprime naturellement dans notre cadre. Comme nous l'avons déjà fait pour modéliser les piles d'environnements ou de retour, il suffit d'introduire un nouveau composant *r via* deux nouveaux combinateurs $\mathbf{store}_r$ et $\mathbf{fetch}_r$ pour représenter un registre.

Si une expression E vérifie les critères (*i.e.* $\Phi_\rho(E)$) alors elle est transformée en remplaçant toutes les expressions $\mathbf{store}_s\ V$ avec V de type ρ par $\mathbf{store}_r\ V$ et les expressions $\lambda_s x^{\rho}.E$ (*i.e.* $\mathbf{fetch}_s(\lambda x.E)$ avec x de type ρ) par $\lambda_r x.E$ (*i.e.* $\mathbf{fetch}_r\ (\lambda x.E)$). La définition des opérateurs prenant ou produisant une instance de type ρ doit également être changée. Par exemple, l'opérateur $access_s^{\rho\ \to s\ int \to s\ Rint}$ devient $access_r$ avec la règle de réduction :

$$\mathbf{store}_s\ i\ ;\ \mathbf{store}_r\ a\ ;\ access_r \bullet\!\!\rightarrow \mathbf{store}_s\ a[i]$$

Le composant *r* est particulier puisqu'il est sensé représenter un registre. En d'autres termes, la définition en λ-calcul de ses primitives d'accès sont de la forme :

$$\mathbf{store}_r = \lambda n.\lambda c.\lambda s.\lambda r.c\ s\ n \qquad \mathbf{fetch}_r = \lambda f.\lambda c.\lambda s.\lambda r.f\ r\ c\ s\ r$$

Contrairement aux définitions déjà vues (*e.g.* en section 2.1.3), $\mathbf{store}_r$ n'empile pas son argument mais remplace (écrase) le composant par son argument. Ces définitions ne sont pas valides en général. Ce sont les critères syntaxiques (Φ_ρ) qui les rendent correctes.

Exemple 3-5 : En utilisant le nouveau composant *r*, notre exemple se réécrit en

$$\mathbf{store}_s\ i\ ;\ \mathbf{store}_r\ a\ ;\ access_r\ ;\ \mathbf{store}_s\ p\ ;\ \mathbf{store}_s\ i\ ;\ \mathbf{store}_r\ a\ ;\ update_r\ ;\ f$$

En supposant que *a*, *i* et *p* soient liés à [4,3,2], 3 et 1, il se réduit comme suit:

$$\mathbf{store}_s\ 3\ ;\mathbf{store}_r\ [4,3,2]\ ;\ access_r\ ;\ \mathbf{store}_s\ 1\ ;\ \mathbf{store}_s\ 3\ ;\ \mathbf{store}_r[4,3,2]\ ;\ update_r\ ;\ f$$

$\bullet\!\!\xrightarrow{*}$ $\mathbf{store}_s\ 2\ ;\ \mathbf{store}_s\ 1\ ;\ \mathbf{store}_s\ 3\ ;\ \mathbf{store}_r\ [4,3,2]\ ;\ update_r\ ;\ f$ {accès du 3ème élément}

$\bullet\!\!\xrightarrow{*}$ $\mathbf{store}_s\ 2\ ;\ \mathbf{store}_r\ [4,3,1]\ ;\ f$ {mise à jour en place du 3ème élément} ❐

Les critères syntaxiques et la transformation associée s'étendent au traitement de plusieurs types $\rho_1,...\rho_i$. Le composant *r* devient un vecteur de registres modifiables en place.

3.1.4 Application

Nous décrivons maintenant comment l'analyse peut être mise à profit pour allouer les arguments de fonctions récursives dans des registres. Le typage standard n'est pas bien adapté à cette application. En effet, si deux paramètres partagent le même type (*e.g. int*) l'analyse vérifie qu'ils peuvent être représentés par un même registre (un cas très improbable). Une solution simple consiste à annoter les types de telle façon que les paramètres d'une fonction récursive aient des types différents. Une fonction récursive à *n* arguments est initialement typée par $\alpha 1 \rightarrow \beta 2 \ldots \rightarrow \gamma n \rightarrow \delta$. L'inférence de type doit être légèrement modifiée pour prendre en compte cette notion de type annoté. Un type annoté τi et sa version non annotée τ s'unifient en τi (*e.g.* $int1 \rightarrow int$ et $int \rightarrow int2$ s'unifient en $int1 \rightarrow int2$).

Certaines fonctions (*e.g.* qui permutent leurs arguments dans l'appel récursif) posent problème. Par exemple, le typage de la fonction :

$$\textbf{rec}\ f = \lambda x^{\,int1}.\lambda y^{\,int2}.\ \ldots\ f\ (sub\ y\ 1)\ x\ \ldots$$

conduit à un échec (*int*1 et *int*2 ne s'unifient pas). On introduit une famille d'opérateurs de coercion *Id* fonctionnellement équivalents à l'identité mais permettant de passer d'un type annoté à un autre. En utilisant ces opérateurs, l'expression précédente est transformée en l'expression typable :

$$\textbf{rec}\ f = \lambda x^{\,int1}.\lambda y^{\,int2}.\ \ldots f\ (sub^{int2 \to int \to int1}\ y\ 1)\ (Id^{int1 \to int2}\ x)\ \ldots$$

Ce style de transformation peut se faire automatiquement lors de l'inférence à chaque fois que deux types dont les annotations diffèrent doivent être unifiés.

Après translation dans Λ_s, analyse et introduction du vecteur de registres, l'instanciation des combinateurs est de la forme

$$\textbf{store}_{ri} = \lambda n.\lambda c.\lambda s.\lambda(r_1,\ldots,r_i,\ldots,r_n).c\ s\ (r_1,\ldots,n,\ldots,r_n)$$

$$\textbf{fetch}_{ri} = \lambda f.\lambda c.\lambda s.(r_1,\ldots,r_i,\ldots,r_n).f\ r_i\ c\ s\ (r_1,\ldots,r_i,\ldots,r_n)$$

$$\textbf{Id}_{\rho i,\rho j} = \lambda c.\lambda s.\lambda(r_1,\ldots,r_i,\ldots,r_j,\ldots,r_n).c\ s\ (r_1,\ldots,r_i,\ldots,r_i,\ldots,r_n)$$

Le combinateur $\textbf{Id}_{\rho i,\rho j}$ est l'analogue dans Λ_s de $Id^{\rho i \to \rho j}$ et correspond à l'affectation $R_j := R_i$.

Comme mentionné dans l'introduction, l'analyse a d'autres applications. L'application de l'analyse à la dérivation de compilateurs dirigée par la sémantique est décrite dans [172].

3.1.5 Transformations globalisantes

Un cas d'échec courant survient quand les arguments ne sont pas évalués dans le bon ordre. Pourtant, l'ordre d'évaluation des arguments d'une fonction stricte n'est pas sémantiquement significatif. Une idée naturelle est de réordonner l'évaluation des arguments afin que les critères soient respectés. Plus spécifiquement, lorsque l'échec provient de l'existence de variables libres de type ρ dans la continuation d'un écrivain, on peut essayer de retarder l'évaluation de l'écrivain jusqu'à ce que son résultat soit vraiment nécessaire.

Reprenons l'exemple de la section 3.1.1,

store$_s$ *f* **; store**$_s$ *p* **; store**$_s$ *i* **; store**$_s$ *a* **;** *update*$_s$ **; store**$_s$ *i* **; store**$_s$ *a* **;** *access*$_s$**;app;app**

qui ne respecte pas les critères (*update*$_s$ est effectué avant *access*$_s$). La règle de transformation

$$E^{R\alpha} \text{ ; } F^{R\beta} \text{ ; } G = F \text{ ; } \lambda_s k.(E \text{ ; } \textbf{store}_s\ k \text{ ; } G) \qquad \textit{avec k une variable fraîche}$$

permet de réordonner les calculs dans Λ_s et en particulier de réécrire l'exemple en :

store$_s$ *f* **; store**$_s$ *i* **; store**$_s$ *a* **;** *access*$_s$ **;**

$$\lambda_s k.(\textbf{store}_s\ p \text{ ; } \textbf{store}_s\ i \text{ ; } \textbf{store}_s\ a \text{ ; } \textit{update}_s \text{ ; } \textbf{store}_s\ k \text{ ; app ; app})$$

opérationnellement équivalente mais qui respecte les critères. Une solution plus sophistiquée serait de combiner l'analyse avec la compilation de la stratégie d'évaluation afin de choisir le meilleur ordre d'évaluation pour la globalisation.

3.1.6 Dérivation d'analyses

A partir des critères sur Λ_s, il est possible de dériver des critères sur des λ-expressions standards (de Λ). L'idée est de composer l'analyse avec une transformation *T* compilant la stratégie d'évaluation. En simplifiant $\Phi_\rho \circ T$ on obtient des critères sur les expressions sources pour la stratégie d'évaluation considérée. La Figure 3-6. présente les critères obtenus en considérant la transformation *Va* (compilation de l'appel par valeur de la section 2.2.1).

$\Phi_\rho \circ Va\ (E)$ ssi:

- $E \equiv$ v ou op
- $E \equiv \lambda v^{\tau 1}.F^{\tau 2} \wedge \Phi_\rho \circ Va\ (F) \wedge$

  $\tau 1 \equiv \rho \Rightarrow$ toutes les variables libres de type ρ de *F* sont v^ρ

  $\tau_1 \not\equiv \rho \Rightarrow$ *F* ne contient pas d'expression active de type ρ

  $\tau_2 \not\equiv \rho \Rightarrow$ *E* ne contient aucune expression active de type ρ différente d'un identificateur
- $E \equiv E_1{}^{\tau 1 \to \tau 2}\ E_2{}^{\tau 1} \wedge \Phi_\rho \circ Va\ (E_1) \wedge \Phi_\rho \circ Va\ (E_2) \wedge$

  $fv_\rho(E_2) \Rightarrow E_2$ ne contient aucune expression active de type ρ différente d'un identificateur

Figure 3-6 Critères pour le λ−calcul selon l'appel par valeur gauche-à-droite (Δ_ρ o *Va*)

Nous avons choisi d'exprimer certains critères en langue naturelle. Par exemple, on peut montrer que $\Theta_\rho \circ V(E)$ est équivalente à :

$\Phi_\rho \circ V(E) \wedge$ (*E ne contient aucune expression active de type* ρ *différente d'un identificateur*)

où une expression est dite active dans *E* si elle n'est pas sous une λ-abstraction de *E*.

Il est également possible d'obtenir les critères associés à l'appel par nom. En pratique, ces critères seraient peu utiles. Avec l'appel par nom, la plupart des arguments de fonctions sont dans des fermetures et nos critères n'autorisent pas les variables libres de type ρ d'apparaître dans les fermetures. Par contre, les critères associés à une transformation comme *Na* qui intègre des informations de nécessité (*cf.* section 2.6) pourraient être utiles.

3.1.7 Travaux connexes

L'analyse la plus proche (en fait, l'inspiratrice) de la notre est celle de Schmidt [138] qui considère un λ-calcul strict. Elle s'exprime de façon semblable comme des critères sur la syntaxe et les types des expressions. Nos critères combinés avec la transformation *Va* (Figure 3-6) sont plus précis que ceux de Schmidt. En fait, ce serait le cas pour toute transformation compilant une stratégie d'appel par valeur. La principale raison est que les critères de Schmidt sont valides quelle que soit la version (séquentielle ou parallèle) de l'appel par valeur et ne tirent pas profit d'un ordre d'évaluation particulier.

La plupart des autres analyses sont basées sur des interprétations abstraites. Kastens et Schmidt [86] construisent une grammaire de lectures/écritures pour analyser un langage procédural. Bloss [15] part d'une sémantique de chemin pour analyser un langage paresseux du premier ordre. Draghicescu et Purushhothaman [44] traitent le même type de langage mais prennent en compte des informations de nécessité. Ces deux dernières analyses sont en temps exponentiel. L'analyse de Sestoft [142] utilise une analyse de fermetures pour construire une grammaire de lectures/écritures de programmes fonctionnels stricts d'ordre supérieur. Cette analyse a ensuite été étendue pour essayer de prendre en compte les variables capturées dans les fermetures [58]. L'idée, mentionnée en section 3.1.5, de choisir l'ordre d'évaluation des arguments lors de l'analyse a été explorée dans [137].

Un autre type d'approche relève davantage de la discipline de programmation

que de l'analyse de programme. Dans [62] et [157], le langage est étendu par des nouvelles constructions exprimant la séquence et la mise à jour en place. Un système de type, inspiré de la logique linéaire, assure que la transparence référentielle est préservée. Les monades [158] sont une autre technique qui ne nécessite pas de nouveau système de type. Les mises à jour en place sont valides par construction. L'écriture de programmes en style monadique impose au programmeur d'expliciter l'ordre d'évaluation et l'utilisation mémoire. Selon le point de vue, cela peut être un avantage ou un inconvénient.

3.2 Glaneur de cellules étendu

Nous présentons maintenant une technique pour libérer plus de mémoire que les GC traditionnels. Elle est conçue comme une extension des GC *stop&copy* ou *mark&sweep* [32][165]. Contrairement aux autres, notre GC est capable de détecter et libérer des portions de structures devenues inutiles mais toujours référencées. La méthode repose sur la propriété de *paramétricité* [134][156][2] dont jouit, par exemple, le système de types polymorphes d'Hindley/Milner [106]. La paramétricité donne indirectement des informations sur l'utilité des arguments des fonctions polymorphes. Par exemple, la fonction *length : List* $\alpha \rightarrow$ *Int* peut s'évaluer indépendamment des éléments de sa liste argument. Cette propriété n'est rien d'autre qu'une instance des "*theorems for free*" de Wadler [156]. Elle est valide pour toutes les fonctions de ce type et permet à un GC de récupérer les éléments de la liste.

La mise en œuvre de cette idée nécessite d'attacher des annotations de type aux adresses de retour et aux fermetures. Le GC explore la pile (de données et de retour) en utilisant les annotations de type et une procédure d'unification. Ces techniques de base sont présentées en section 3.2.1.

Nous avons suggéré que le GC pouvait libérer les éléments d'une liste argument de *length*. Qu'advient-il si cette liste est partagée par une autre fonction ayant besoin de la structure complète ? Le problème des structures partagées est traité en section 3.2.2.

Clairement, le système de Hindley/Milner ne fut pas conçu comme une analyse d'utilité et il perd de l'information qui pourrait être exploitée par le GC. La section 3.2.3 suggère deux extensions de l'algorithme de typage afin d'inférer des types plus adaptés (*i.e.* moins instanciés).

De nombreux programmes fonctionnels comportent des fuites de mémoire qu'un

GC standard ne peut traiter. Nous montrons dans la section 3.2.4 comment notre technique “colmate” plusieurs formes de fuites. Un rapide compte-rendu d’expérimentations est présenté en section 3.2.5.

Pour simplifier la présentation, on suppose disposer d’un GC de type *stop&copy* et d’une mise en œuvre manipulant une unique pile (données + retour) et des fermetures en vecteur. Les mêmes idées s’adapteraient à d’autres choix d’implémentation.

3.2.1 Glaner plus de cellules

Nous commençons par présenter la méthode de base sans prendre en compte les problèmes de partage traités dans la section suivante.

L’idée

Les structures de données accessibles ont leurs racines dans la pile d’exécution. La pile est une suite de contextes (*activation records*) composés de variables locales et d’une adresse de retour. Pour tracer les structures référencées, le GC parcourt la pile du contexte le plus ancien au plus récent. A chaque adresse de retour est associée une annotation de type représentant l’utilité des données du contexte. Le GC parcourt et recopie les structures du contexte en fonction de leur type.

Plus précisément, une adresse de retour peut être vue comme une continuation de la forme $\lambda x_1. \ldots \lambda x_n.\lambda r.\ E$. Les x_i représentent les variables locales dans la pile et r le résultat de l’appel de fonction correspondant à cette adresse de retour. Un contexte peut être vu comme une fermeture $(\lambda x_1. \ldots \lambda x_n.\lambda r.\ E)\ X_1 \ldots X_n$. Pour chaque appel, le type de la continuation est associé à l’adresse de retour. Par exemple, pour *length* (*append* [[1];[2]] [[3];[4]]) la continuation de l’appel à *append* est $\lambda r.$*length* r qui a le type *List* $\alpha \rightarrow$ *Int*.

Le GC explore la pile en unifiant le type de la continuation courante avec le type du résultat du contexte suivant. Quand le GC a tracé le contexte

$$(\lambda x_1 \ldots \lambda x_n.\lambda r.\ E)^{\sigma 1 \rightarrow \ldots \rightarrow \sigma n \rightarrow \sigma r \rightarrow \sigma)}\ X_1 \ldots X_n$$

il analyse le contexte suivant

$$(\lambda y_1 \ldots \lambda y_m.\lambda r.\ E)^{\tau 1 \rightarrow \ldots \rightarrow \tau m \rightarrow \tau r \rightarrow \tau)}\ Y_1 \ldots Y_m$$

en unifiant le type τ de ce contexte avec le type σ_r attendu par le contexte précédent. Cette unification est nécessaire car les annotations sont les types les plus généraux

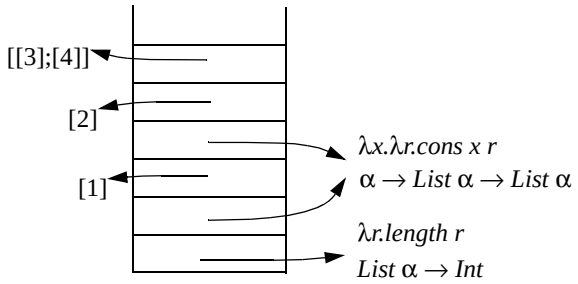
(instanciés au minimum). L'unification durant le GC peut être vu comme une analyse dynamique d'utilité spécialisée à la pile courante.

Exemple 3-7 Considérons le programme suivant et son évaluation en appel par valeur.

let rec *append* l_1 l_2 = **case** l_1 **in**
nil : l_2
cons x xs: *cons x* (*append xs* l_2)
in *length (append* [[1];[2]] [[3];[4]])

append : List $\alpha \rightarrow$*List* $\alpha \rightarrow$*List* α *length : List* $\alpha \rightarrow$*Int*

Si le GC est appelé au début du premier appel à *cons* la pile est de la forme :



Le contexte le plus ancien contient seulement une adresse de retour avec le type associé *List* $\alpha \rightarrow$*Int*. Le GC analyse l'adresse de retour suivante qui a le type $\alpha \rightarrow$*List* $\alpha \rightarrow$*List* α (l'unification avec le type attendu par $\lambda r.$*length r* ne le change pas). L'argument ([1]) dans ce contexte a le type α et peut être libéré. Le GC passe au dernier contexte dont l'adresse de retour a le type $\alpha \rightarrow$*List* $\alpha \rightarrow$*List* α. Le GC libère la première variable ([2]) et les éléments de la seconde. Toutes les sous-listes ([1], [2],[3],[4]) sont libérées et la mémoire ne contient plus que deux cellules *cons* et un *nil*. ❒

Formalisation

Nous formalisons maintenant le GC dans le cadre fonctionnel. Une pile de contexte est représentée comme une application d'expressions fonctionnelles closes :

$(\lambda x_1 \ldots \lambda x_m.\lambda r.\ E)\ X_1 \ldots X_m\ ((\lambda y_1 \ldots \lambda y_n.\lambda r.F)\ Y_1 \ldots Y_n\ (\ldots((\lambda z_1 \ldots \lambda z_p.G)\ Z_1 \ldots Z_p)\ldots))$

Les fonctions $(\lambda x_1.\ \ldots)$, $(\lambda y_1.\ \ldots)$ représentent les adresses de retour et la dernière-

re fonction (λz_1. …) est celle qui a appelé le GC. Les arguments X_i, Y_j, Z_k représentent les racines de la mémoire accessible.

Les objets manipulés par le programme (dans la pile ou la mémoire) sont décrits par la grammaire suivante :

$$S ::= V \mid S_1.S_2 \mid nil \mid f\, S_1 \ldots S_n$$

où V représente les valeurs de base (entiers, booléens, ...), $E_1.E_2$ et nil des listes et $f\, S_1 \ldots S_n$ des fermetures (f étant la fonction ou le code). Tout ce qui suit s'étend naturellement à des types définis par l'utilisateur.

Les types de ces objets sont définis par la grammaire :

$$T ::= V \mid B \mid T \rightarrow T \mid List\ T \qquad\qquad P ::= B \mid List\ P$$

où V représente les variables de type et B les types de base (*Int*, *Bool*,...). P peut être vu comme les types imprimables et on suppose que le type d'un programme (*i.e.* de la pile) appartient à P. On utilise les conventions suivantes :

$$\alpha,\beta,\chi,\delta,\varepsilon \in V \qquad b \in B \qquad \tau,\sigma \in T \qquad \pi \in P$$

La pile de l'Exemple 3-7 est représentée par l'expression de type *Int*

$$f_1(f_2\ [1]\ (f_2\ [2]\ \ [[3];[4]])) \qquad \text{avec} \quad f_1 \equiv \lambda r.length\ r \quad \text{et} \quad f_2 \equiv \lambda x.\lambda r.cons\ x\ r$$

Wadler a montré dans [156] comment on pouvait utiliser la paramétricité pour dériver des théorèmes à partir de types. Par exemple, le fait que la fonction f de type $List\ \alpha \rightarrow Int$ n'ait pas besoin des éléments de la liste est formalisé par le théorème associé au type $List\ \alpha \rightarrow Int$. Pour toute fonction a de type $T \rightarrow T'$,

$$f_{T'} = f_T \circ (map\ a)$$

où f_T dénote l'instance de f pour le type T. Ces théorèmes sont valides dans le λ-calcul polymorphique pur quelque soit la fonction a. Pour les langages de programmation (intégrant un opérateur de point-fixe), les théorèmes ne sont valides que pour toute fonction a stricte. Dans notre cas, on formalise la libération de données par la fonction stricte $\lambda x.\bot$. Le théorème associé au type $List\ \alpha \rightarrow Int$ implique entre autres que

$$f = f \circ (map\ (\lambda x.\bot))$$

ce qui formalise le fait que la récupération des éléments de la liste est valide (*i.e.* n'a pas d'impact sur *f*).

Soit *stack* une pile de type monomorphe π, le processus de récupération de mémoire se décrit comme (gc_π *stack*) avec

(gc1) $gc_\alpha(E) = \bot$

(gc2) $gc_\tau(f\ X_1 \ldots X_n) = f\ (gc_{\tau 1}\ X_1)\ \ldots\ (gc_{\tau n}\ X_n)$ avec $\vdash f{:}\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$

(gc3) $gc_b(V) = V$

(gc4) $gc_{\text{List}\,\tau}(E.F) = gc_\tau\ E\ .\ gc_{\text{List}\,\tau}\ F$

(gc5) $gc_{\text{List}\,\tau}(nil) = nil$

Afin de poursuivre le processus (gc_τ) à l'intérieur d'une fermeture, le type courant τ doit être le type de la fermeture (règle (gc2)). C'est exactement le même impératif que lors du passage d'un contexte à l'autre et cela implique une unification. Dans la pratique, une annotation de type est associée à chaque fonction apparaissant dans une fermeture. Quand une fermeture est rencontrée, gc_τ prend l'annotation de type associée à la fonction $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_{n+1} \rightarrow \ldots \rightarrow \tau_p$; du nombre de variables *n* de la fermeture on déduit le type du résultat, $\tau_{n+1} \rightarrow \ldots \rightarrow \tau_p$ qui est unifié avec τ.

Exemple 3-8 Reprenons l'exemple précédent

stack $\equiv f_1(f_2\ [1]\ (f_2\ [2]\ [[3];[4]]))$

avec $f_1 \equiv (\lambda r.length\ r) : List\ \alpha \rightarrow Int$ et $f_2 \equiv \lambda x.\lambda r.cons\ x\ r : \alpha \rightarrow List\ \alpha \rightarrow List\ \alpha$

$gc_{\text{Int}}\ stack = f_1\ (gc_{\text{List}\,\alpha}\ (f_2\ [1]\ (f_2\ [2]\ [[3];[4]])))$ $\vdash f_1 : List\ \alpha \rightarrow Int$ (gc2)

$= f_1\ (f_2\ (gc_\alpha[1])\ (gc_{\text{List}\,\alpha}(f_2\ [2]\ [[3];[4]])))$ $\vdash f_2{:}\alpha \rightarrow List\ \alpha \rightarrow List\ \alpha$ (gc2)

$= f_1\ (f_2 \perp (f_2\ (gc_\alpha[2])\ (gc_{\text{List}\,\alpha}[[3];[4]])))$ (gc1), (gc2)

$= f_1\ (f_2 \perp (f_2 \perp ((gc_\alpha[3]).(gc_{\text{List}\,\alpha}[[4]]))))$ (gc1),(gc4)

$= f_1\ (f_2 \perp (f_2 \perp [\perp;\perp]))$ (gc1),(gc4),(gc5)

Le GC a récupéré tous les éléments de liste ❒

Opérateurs de comparaison polymorphique

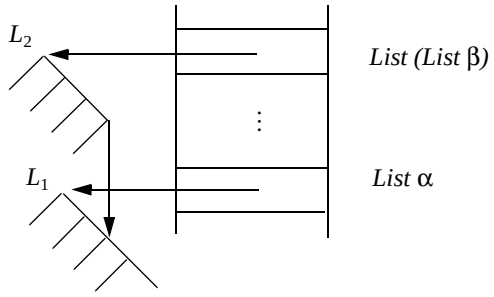
La fonction *member* semble invalider l'approche : elle a le type $\alpha \rightarrow List\ \alpha \rightarrow Bool$ mais a besoin des éléments de la liste. Ce problème provient des opérateurs de comparaison polymorphiques qui demandent un traitement particulier. Si, dans l'exemple, nous remplacions la fonction $\lambda r.length\ r$ par

$$\lambda r.\textbf{if}\ member\ (hd\ r)\ (tl\ r)\ \textbf{then}\ 1\ \textbf{else}\ 0$$

qui a le même type ($List\ \alpha \rightarrow Int$), le GC ($gc_\tau$) serait incorrect. Le polymorphisme particulier de *member* (appelé parfois polymorphisme *ad-hoc*) ne peut être défini dans le λ-calcul polymorphique pur. La propriété de paramétricité n'est pas valide pour ces opérateurs. Nous donnons aux opérateurs de comparaison polymorphique le type $\mathsf{T} \rightarrow \mathsf{T} \rightarrow Bool$. Le type T est un type spécial qui indique que la structure complète doit être gardée. L'unification de tout type (non fonctionnel) avec T donne T. Le type inféré pour $\lambda r.$**if** *member* (*hd r*) (*tl r*) **then** 1 **else** 0 est maintenant $\mathsf{T} \rightarrow Int$ et la liste argument sera conservée par le GC.

3.2.2 Structures de données partagées

Le problème posé par les structures partagées peut s'illustrer par le schéma suivant :



Quand la première liste L_1 est tracée avec le type $List\ \alpha$, seule sa structure est copiée. Le GC trace ensuite L_2 avec le type $List\ (List\ \beta)$ qui indique qu'il faut copier la structure de la liste *et* des sous-listes. Quand le GC arrive sur la liste partagée, il ne peut supposer (comme un GC *stop&copy* standard) qu'elle a été copiée et doit la reparcourir. Cette implémentation entraînerait de multiples reparcours des structures. Le problème des structures partagées peut se résoudre avec deux traversées complètes de la mémoire accessible.

Le premier parcours annote chaque cellule par son compteur de références. En

fait, il remplace chaque cellule partagée (*i.e.* dont le compteur de référence est au moins 2) par un pointeur vers une liste (qu'on appellera liste d'attente) qui contient la cellule, son compteur de référence et un type (initialement une variable de type α). Dans le cas du schéma précédent, la cellule partagée (c) est remplacée par un pointeur vers un élément de la liste d'attente (c, 2, α)

Le second parcours récupère la mémoire selon les types. Chaque pointeur de la pile et les structures non partagées sont suivis et copiés comme précédemment. Dans le cas d'une cellule partagée (*i.e.* d'un pointeur vers la liste d'attente), le compteur de référence est décrémenté, le type de la liste est unifié avec le type courant et le parcours reprend sur une autre structure. L'unification de type permet de calculer l'utilité des structures partagées pour le programme complet. Dans notre exemple, L_1 est tracée jusqu'au nœud partagé qui pointe ensuite vers (c, 1, *List* α). La liste L_2 est ensuite suivie jusqu'au même nœud qui pointe ensuite vers (c, 0, *List*(*List* α)).

Quand toutes les racines ont été suivies autant que possible, le GC doit tracer les structures pointées à partir de la liste d'attente (*i.e.* les structures partagées). En supposant l'absence de cycle, il existe obligatoirement une cellule dont le compteur de référence est nul. Son type a été instancié au maximum et il représente l'utilité de la structure pour le programme complet. Le GC trace la structure pointée par la cellule avec ce type, et ainsi de suite, jusqu'à ce que la liste d'attente soit vide. Dans notre exemple, les parties non partagées de L_1 et L_2 sont copiées selon les types *List* α et *List* (*List* α) respectivement. La liste d'attente contient à ce moment (c, 0, *List* (*List* α)) et la structure partagée est copiée selon le type *List* (*List* α). Lorsqu'il existe des cycles, le programme termine avec une liste d'attente non vide (aucun compteur de référence n'est nul). Dans ce cas, les structures pointées de cette liste sont copiées totalement (*i.e.* selon le type $\top$). D'autres solutions existent mais elles sont plus coûteuses.

3.2.3 Extensions

Dans de nombreux cas, le typage standard ne donne pas l'information d'utilité la plus précise. Nous envisageons deux extensions concernant le type des constructeurs et les types définis.

Constructeurs

Le type de *cons* ($\alpha \rightarrow$*List* $\alpha \rightarrow$*List* α) semble indiquer que ce constructeur a be-

soin de la structure de son second argument. Cependant, les arguments d'un *cons* n'ont pas à être récupérés si le contexte ne le demande pas. De même, $\lambda xs.cons\ 1\ xs$ et $\lambda x.cons\ x\ [1;2]$ ont des types monomorphes (*List Int*→*List Int* and *Int*→*List Int*) alors que leur argument pourrait être (partiellement) récupéré si la fonction englobante ne demandait que la structure de la liste résultat.

Nous évitons ces approximations en équipant les types de contraintes. On donne, par exemple, à *cons* le type $\alpha\rightarrow\beta\rightarrow\delta$ avec l'ensemble de contraintes $\{List\ \alpha \gg \delta, \beta \gg \delta\}$ où la relation "»" se lit "est plus instancié que". Les contraintes permettent de retarder l'unification. Le type et l'ensemble de contraintes sont inférés indépendamment pour chaque fonction représentant une adresse de retour.

Exemple 3-9 Considérons la fonction

rec *app* $l_1\ l_2$ = **case** l_1 **in**
nil : l_2
cons x xs : **if** x=0 **then** *app xs* l_2
else *cons x* (*app xs* l_2)

qui concatène deux listes en filtrant les zéros de la première. Le nouveau type de cette fonction est *List Int*→β→δ avec la contrainte {β»δ}. Si, durant le GC, la pile est :

$$(\lambda r.length\ r)\ ((\lambda l_1.\lambda l_2.app\ l_1\ l_2)\ [0;1]\ [2;3])$$

alors, comme le contexte ($\lambda r.length\ r$) n'a besoin que de la structure du résultat, les types δ et *List* α sont unifiés. Afin de satisfaire les contraintes, β s'unifie avec *List* γ et les éléments de la seconde liste peuvent être récupérés. Le typage standard de *app* (*List Int*→*List Int*→*List Int*) aurait conduit le GC à conserver complètement les deux listes. ❒

Types définis

Le type de la fonction *hd* (*List* α→α) semble indiquer que *hd* a besoin de la structure de sa liste argument, alors qu'elle n'a besoin de la première cellule *cons*. Cette imprécision vient de l'expression même des types qui identifie sous un même nom de nombreuses variantes (*e.g. List* α dénote à la fois une liste vide, une liste à un élément, etc.).

Au lieu d'utiliser des types prédéfinis, nous inférons des types récursifs. Par

exemple, nous n'inférons plus *List* α mais (μ*t. nil* + *cons* α *t*). L'unification classique est remplacée par l'unification d'arbres rationnels [73] (les deux ont la même complexité). Cette extension est proche de [109] dont le but est de faire l'inférence sans les déclarations de types.

Exemple 3-10 Considérons la fonction suivante qui somme les éléments en position paire dans la liste argument :

```
rec  f l = case l in
           nil       :  0
           cons x xs :  case xs in
                        nil       : 0
                        cons y ys : y + f ys
```

Le nouveau type inféré pour *f* est (μ*t. nil* + *cons* α (*nil* + *cons Int t*))→*Int* et les éléments en position impaire dans la liste peuvent être récupérés. Le typage standard (*List Int* → *Int*) aurait conduit le GC à conserver complètement les deux listes. ❒

Ces deux extensions sont facilitées par le fait que l'on considère des programmes typables. Le typage standard reste une approximation valide et offre une solution de repli si l'inférence devient trop coûteuse.

3.2.4 Application aux fuites de mémoire

De nombreux programmes fonctionnels consomment bien plus de mémoire que ce à quoi on pourrait s'attendre. Ce phénomène, appelé fuite de mémoire, apparaît lorsque des structures devenues inutiles sont toujours pointées [74][121]. Un GC standard conserve ces objets et il est du ressort du programmeur de débusquer et de "colmater" (en restructurant son programme) ces fuites. Nous passons en revue ici trois formes courantes de fuites de mémoire qui sont facilement traitées par notre technique.

- *Fonctions récursives*. Dans la plupart des implémentations, un appel récursif empile un nouveau contexte sur la pile sans toucher à l'ancien contexte qui sera utilisé au retour de l'appel. Ceci crée une fuite de mémoire si la continuation n'utilise pas tous les arguments du contexte. Par exemple, le code généré pour **rec** *f x l* = ... **else** *x* + *f* (*x*-1) (*tl l*) ne devrait pas garder l'argument *l* dans la pile durant l'appel récursif (la continuation n'a besoin que de *x*). Une solution prise par certaines implémentations est d'écraser dynamiquement les arguments deve-

nus inutiles dans la pile (technique appelée *blackholing*). En fait, le type de la continuation fournit une information suffisante pour éviter la fuite. Dans l'exemple précédent, la continuation $\lambda x.\lambda l.\lambda r.x+r$ a le type $Int \rightarrow \alpha \rightarrow Int \rightarrow Int$ et la partie inutile de *l* sera récupérée si le GC est déclenché pendant un appel récursif à *f*.

- *Mises à jour des fermetures*. Un problème similaire se pose lors de l'évaluation de fermetures dans les langages paresseux. Afin de réaliser la mise à jour après son évaluation, un pointeur est conservé sur la fermeture. La même technique d'écrasement est parfois utilisée pour prévenir cette forme de fuites [83]. Ici, il suffit de donner à l'opérateur de mise à jour *updt* le type $\mathbf{U} \rightarrow \beta \rightarrow \beta$; **U** étant un type spécial associé avec le pointeur sur la fermeture. Le GC se contente de remplacer une structure de type **U** par une fermeture vide. Cette fermeture sert uniquement à réserver la place nécessaire pour la mise à jour.
- *N-uplets.* Une autre classe de fuites a été décrite par Hughes [74]. Le résultat de fonctions paresseuses retournant des paires (ou plus généralement des n-uplets) apparaît souvent dans des expressions comme (*fst r*) ou (*snd r*). Hughes a montré que certaines de ces fonctions comportaient des fuites quelle que soit la manière dont elles étaient exprimées. Le type de *fst* (resp. *snd*) étant $(\alpha,\beta) \rightarrow \alpha$ (resp. $(\alpha,\beta) \rightarrow \beta$), notre GC récupérera le second (resp. premier) argument. Une autre solution, proposée par Wadler [154], est d'intégrer au GC des règles de simplifications comme $fst(x,y)=x$ et $snd(x,y)=y$.

Notons que ces différentes formes de fuites se traitent sans les extensions présentées en section 3.2.3. Contrairement au *blackholing*, la méthode n'implique aucun surcoût lors de l'exécution normale du programme et bien d'autres formes de fuites (*e.g.* sur les listes) peuvent être traitées.

3.2.5 Implémentation

Cette technique est applicable à tout langage fortement typé et peut prendre en compte différent schémas de GC. Son implémentation soulève plusieurs questions :

- *Les annotations de types*. Nous avons vu en section 3.2.1 que le GC doit disposer d'informations/types de bas niveau (*e.g.* les mises à jour, la structure de la pile ou des fermetures). Ceci implique, soit un compilateur propageant l'information de type jusqu'au code machine (comme, par exemple, [114][115]), soit de refaire une inférence de type sur le code produit.

- *La consommation mémoire*. Le GC se déclenchant en situation de pénurie, il est important qu'il consomme peu de mémoire. La liste d'attente, les substitutions ou même des types de grande taille, demandent de la mémoire. Une solution pragmatique est de fixer des tailles limites. En cas de débordement, on peut toujours abandonner les annotations de types, les unifications, ou la liste d'attente et recopier complètement certaines structures.
- *L'efficacité*. La technique n'implique aucun surcoût lors de l'exécution normale du programme. Par contre, il est clair que le GC est plus coûteux qu'un GC standard. La structure est parcourue deux fois et le processus implique des unifications. D'un autre coté, récupérer plus de mémoire peut simplifier les GC suivants voire permettre à certains programmes de terminer. L'utilisation la plus raisonnable semble être d'utiliser un GC standard la plupart du temps ; l'extension est utilisée lorsque l'occupation mémoire excède un certain pourcentage.

Nous avons mis en œuvre un prototype dans le compilateur Tabac. Nous avons choisi de refaire une inférence de type sur le code fonctionnel et d'étendre un GC de style *stop&copy*. Le prototype implémenté ne considère qu'un noyau de langage fonctionnel et n'intègre pas les extensions de la section 3.2.3. La Figure 3-11 rassemble les résultats obtenus sur quelques petits programmes. On a mesuré la taille mémoire minimale (avec GC étendu ou non) pour que le programme termine normalement.

	mirror	*queen*	*fft*	*compress*
GC *standard*	468 Kb	564 Kb	57 Kb	1,280 Kb
GC *étendu*	352 Kb	404 Kb	42 Kb	792 Kb
Gain	25 %	28 %	26 %	38 %

Figure 3-11 Quelques résultats

Il existe également des programmes (*e.g. qsort*) pour lesquels la technique ne gagne pratiquement rien et, au contraire, d'autres où elle change l'ordre de grandeur de la complexité mémoire. Le prototype (sans aucune optimisation) est en moyenne 3 à 4 fois plus lent qu'un GC standard. Dans tous les cas, différentes versions (*e.g.* sans extensions de typage ou traitant uniquement certaines formes de fuites) peuvent être fournies selon le temps que l'on est disposé à payer.

3.2.6 Travaux connexes

La technique présentée est très proche de celle utilisée par les GC sans étiquettes (*tag free garbage collection*)[5][18][56][57]. Les langages typés n'ont pas besoin d'étiqueter leurs données pour s'exécuter (ils n'ont pas à vérifier dynamiquement le type des opérandes). Pourtant la plupart des implémentations étiquettent les données pour que le GC puisse distinguer un pointeur d'un entier. Une idée est de remplacer les étiquettes dans la mémoire par des informations de type dans le code. Le GC parcourt la pile à l'aide de ces informations et, dans le cas des langages polymorphes, utilise l'unification. Il existe toutefois des cas où le GC ne peut reconstruire les types et où un étiquetage explicite est nécessaire [4][149]. L'objectif est de reconstruire totalement les types pour parcourir la structure complète. Notre but est opposé : il s'agit d'instancier au minimum les types afin de ne conserver que les structures nécessaires.

Diverses analyses de programmes ont été proposées pour détecter statiquement la durée de vie des objets alloués en mémoire [75][84]. En général, ces analyses sont basées sur la notion d'accessibilité et ne peuvent récupérer plus qu'un GC traditionnel. Le but est ici de compiler la gestion mémoire. Rien n'empêcherait de concevoir des analyses sophistiquées pour détecter des parties inutiles des structures accessibles. Nous pensons que les extensions présentées en section 3.2.3 sont un bon compromis entre coût et précision.

La correction de notre approche a été justifiée de façon assez abstraite et sans prendre en compte le partage. Une approche formelle devrait considérer un vrai code assembleur typé, une spécification complète du GC et représenter la mémoire et le partage. Plusieurs formalisations dans ce style ont depuis été proposées [113][71].

3.3 Conclusion

L'analyse de globalisation et le GC étendu exploitent uniquement la syntaxe et le typage des programmes. Comparée à des analyses sémantiques, ce style d'approche a un moindre coût, facilite l'expression de l'optimisation (comme une transformation pour la globalisation et comme une fonction pour le GC) et offre un retour plus exploitable (par l'utilisateur ou par un transformateur).

Deux points nous semblent devoir être soulignés :

- La compilation de la stratégie d'évaluation par transformation de programme est une technique intéressante pour améliorer la précision et la généralité des analyses. De nombreuses propriétés de programmes dépendent de l'ordre d'évaluation. Compiler la stratégie en la rendant explicite dans la syntaxe permet de séparer les problèmes. En analysant ce type d'expression, la stratégie d'évaluation n'a plus à être prise en compte. L'analyse est plus simple et est implicitement définie pour tout type d'ordre d'évaluation séquentiel. Ceci suppose toutefois que la stratégie s'exprime simplement dans le cade de travail. Le langage Λ_S nous semble être un meilleur cadre que les expressions CPS dont les arguments d'ordre supérieur (les continuations) complexifient les analyses.
- Les algorithmes d'inférence de type à la Hindley/Milner sont un outil intéressant d'analyse. Baker montre comment utiliser une inférence de type à la ML pour déduire des informations de partage [10]. Hall [63] et Prost [129] font de même pour respectivement, optimiser la représentation des listes et détecter du code mort. Dans ce chapitre, nous avons également montré d'autres utilisations du typage. L'expérience montre que ce style d'algorithme est efficace et conduit à des analyses modulaires. Il faut toutefois se méfier de la tentation naturelle d'enrichir le langage de types (*e.g.* avec les notions d'intersection, de sous-typage, etc.) pour obtenir des analyses plus précises. Les algorithmes deviennent rapidement beaucoup moins naturels et impliquent le plus souvent la résolution d'ensembles de contraintes complexes. On peut alors s'interroger sur les avantages par rapport aux analyses par itération de point fixe.

Pour conclure sur une note plus générale, on s'aperçoit vite que des optimisations puissantes et automatiques ne sont pas sans danger. Si l'impact de ces optimisations est fort (comme, par exemple, la mise à jour en place des tableaux ou la suppression de fuites de mémoire) il est important que le programmeur puisse en garder le contrôle. En effet, un changement de code apparemment mineur peut empêcher une optimisation importante et rendre le comportement du programme inacceptable. Avec le recul, il semble préférable que certaines propriétés soit assurées par le programmeur plutôt que détectées par analyse (*e.g.* utiliser les monades pour garantir les mises à jour en place). Ces approches peuvent être qualifiées de disciplines de programmation ; c'est la matière de la partie suivante.

Historique

Les premiers travaux sur la compilation par transformation de programme datent de ma période de thèse (1986-1988). La réduction de graphe était alors la technique à la mode pour la mise en œuvre des langages fonctionnels. L'algorithme d'abstraction utilisé dans cette approche pouvait être vue comme une compilation de la β*-réduction par transformation de programme. D'un autre coté, la réduction du graphe restait interprétative. L'idée de compiler également la recherche de redex par transformation est venue naturellement. Plusieurs versions de cette transformation ont finalement conduit à la nème redécouverte de la transformation* CPS*. L'objectif de décrire le processus de compilation jusqu'au code machine s'est alors imposé. Les combinateurs classiques (des transformateurs de graphe) ont été remplacés par des combinateurs qui pouvaient être vus comme de l'assembleur (des transformateurs de pile). En ajoutant quelques étapes supplémentaires, nous avons obtenu un compilateur complet exprimé comme une composition de transformations de programme. Les preuves de correction étaient assez simples et on pouvait exprimer et justifier de nombreuses optimisations* ([166][167][171][184]).

J'ai ensuite (1989-1990) effectué un séjour post-doctoral dans l'équipe de David Schmidt. Quelques années auparavant, celui-ci avait proposé des critères syntaxiques afin d'assurer la globalisation de l'état dans les descriptions dénotationnelles. L'approche était séduisante mais manquait de précision pour l'utilisation qui m'intéressait alors : l'allocation de registres. L'étude de ces critères et ma familiarité avec les expressions en CPS *ont donné l'analyse décrite en section 3.1.*

*De retour à Rennes (1991), j'ai concrétisé tous ces travaux dans un compilateur (*TABAC *pour* Transformation BAsed Compiler*). Il prenait en entrée un langage fonctionnel intermédiaire (*FLIC*) utilisé par plusieurs implémentations de langages fonctionnels.* TABAC *pouvait compiler des langages fonctionnels d'ordre supérieur, stricts ou paresseux, typés ou non typés. Lorsqu'il s'est agit d'implémenter le glaneur de cellules, j'ai étudié les* GC *sans étiquettes. Le rapprochement avec le bel article de Wadler* (theorems for free!) *s'est fait et a donné le travail décrit en section 3.2. Au même moment, je me demandais s'il était possible d'utiliser la conversion* CPS *pour compiler la réduction forte. Cet exercice, inspiré plus par la curiosité qu'autre chose, est décrit en section 2.8.*

L'article de TOPLAS *(1991) sur la compilation par transformation de programme* [167] *se concluait par ces phrases :* Another promise of our approach could be the expression, within the same framework, of various implementation techniques. This would provide a better understanding of the relations between implementations, which is very difficult to assess, in general. *Ce fut le sujet de thèse proposé à Rémi Douence* [43]. *Le point le plus délicat fut de mettre au point un cadre formel taillé sur mesure pour la description de mises en œuvre. Le gros du travail fut ensuite l'étude systématique des multiples implémentations de langages fonctionnels. Ces recherches, qui ont eu lieu dans les années 1994-1996, forment l'essentiel du chapitre 2.*

Partie II

Langages dédiés et disciplines de programmation

CHAPITRE 4 *Types graphe*

Les systèmes de types usuels ne représentent que grossièrement les structures de données manipulées par les programmes impératifs. Ils ne font, par exemple, aucune distinction entre une liste doublement chaînée et un arbre binaire. Le partage, inhérent dans les structures à pointeurs, n'est pas exprimé dans le type. De ce fait, les systèmes de types laissent échapper un bon nombre d'erreurs lors des manipulations explicites de pointeurs.

Nous proposons ici une solution à ce problème sous la forme d'une nouvelle notion de type (*les types graphe*) et d'une discipline de programmation associée (*les transformateurs*). Un type graphe est défini comme une grammaire de graphe et les structures de données impératives sont définies comme des instances de types graphe. Les structures sont manipulées exclusivement par des réécritures de graphes : les transformateurs. Les points forts de cette approche sont :

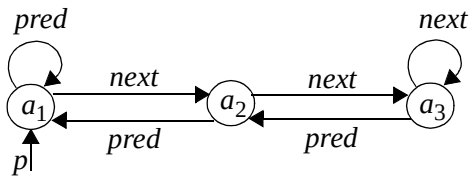
- *l'expressivité.* Les types graphe permettent de spécifier de nombreuses structures de données avec des relations de partage complexes. Nous donnons plusieurs exemples de structures de données classiques en section 4.1. Les transformateurs permettent également plus de flexibilité et d'expressivité qu'une approche basée sur des types abstraits de données avec une collection figée d'opérateurs.
- *la vérification statique de type*. On vérifie que chaque transformateur respecte l'invariant spécifié par le type graphe correspondant. C'est à dire que tout transformateur appliqué à une structure de type *T* doit produire une structure de type *T.* Cette vérification, qui fut la motivation première de ce travail, est décrite en section 4.2.
- *l'efficacité.* L'intérêt majeur des pointeurs est d'exprimer des algorithmes performants. Nous montrons en section 4.3 comment les types graphe et les transformateurs s'intègrent dans un langage avec manipulation explicite de pointeurs sans surcoût. Nous avons choisi ici le langage C mais d'autres choix seraient possibles.

La déclaration de types graphe et l'utilisation de transformateurs définissent une discipline de programmation dont le bénéfice est de prévenir certaines erreurs courantes de programmation. Ceci est à rapprocher de la discipline de programmation imposée par des systèmes de types à la ML. Dans notre cas, l'utilisation exclusive de types graphe n'est pas imposée. Le programmeur peut toujours utiliser explicitement des pointeurs pour décrire certaines structures de données (*e.g.* les structures non régulières). Aucune vérification statique ne sera faite sur la manipulation de ces structures.

4.1 Types graphe et transformateurs

4.1.1 Graphes et multi-ensembles

Un (hyper-)graphe est défini comme un multi-ensemble de n-uplets $R\ a_1 \ldots a_n$ où R est une relation n-aire et les a_i représentent des adresses. Par exemple, la liste doublement chaînée de 3 éléments avec un pointeur sur le premier élément



est définie comme le multi-ensemble:

$\Delta = \{\mathbf{p}\ a_1, \mathbf{pred}\ a_1\ a_1, \ \mathbf{next}\ a_1\ a_2, \ \mathbf{pred}\ a_2\ a_1\ , \mathbf{next}\ a_2\ a_3, \ \mathbf{pred}\ a_3\ a_2, \mathbf{next}\ a_3\ a_3\}$

Types graphe = grammaire de graphe

Un type graphe est une grammaire algébrique (*context-free*) générant de tels graphes [35][133]. Par exemple, le type graphe correspondant aux listes doublement chaînées peut s'écrire :

$$\begin{array}{lcl} \mathit{Doubly} & = & \mathbf{p}\ x, \mathbf{pred}\ x\ x, L\ x \\ L\ x & = & \mathbf{next}\ x\ y, \ \mathbf{pred}\ y\ x, L\ y \\ L\ x & = & \mathbf{next}\ x\ x \end{array}$$

Doubly et *L* sont les non terminaux ; *Doubly* est appelé l'*axiome* de la grammaire. La seconde règle indique qu'une liste $L\ x$ est composée de deux pointeurs (***next*** et

pred) entre *x* et une nouvelle adresse (fraîche) *y* suivie d'une liste *L y*. Il est simple de vérifier que le multi-ensemble Δ précédent peut être engendré par *Doubly.*

Nous définissons l'appartenance d'un graphe à une grammaire en considérant le système de réécriture dérivé d'une lecture de droite à gauche des règles de la grammaire. Par exemple, le système de réécriture associé à *Doubly* est :

$$\begin{array}{lll} \mathbf{p}\ x, \mathbf{pred}\ x\ x, L\ x & \rightarrow_{Doubly} & Doubly \\ \mathbf{next}\ x\ y, \mathbf{pred}\ y\ x, L\ y, X & \rightarrow_{Doubly} & L\ x,\ X \qquad y \notin X \\ \mathbf{next}\ x\ x, X & \rightarrow_{Doubly} & L\ x,\ X \end{array}$$

La variable *X* représente le reste du multi-ensemble (le contexte de la réduction). Les règles de réécriture sont globales (le terme gauche représente tout le multi-ensemble). La condition $y \notin X$ vient du fait que la variable *y* dans la seconde règle de la grammaire est fraîche.

Définition 4-1 *Soit T un type graphe d'axiome O et* $\rightarrow_O$ *le système de réécriture associé, un graphe (multi-ensemble) G est de type T ssi* $G \stackrel{*}{\rightarrow}_O \{O\}$.

Il est facile de vérifier que $\Delta \stackrel{*}{\rightarrow}_{Doubly} \{Doubly\}$. Par contre, le graphe :

$$\Delta' = \{\mathbf{p}\ a_1, \mathbf{pred}\ a_1\ a_1,\ \mathbf{next}\ a_1\ a_2,\ \mathbf{pred}\ a_2\ a_1, \mathbf{next}\ a_2\ a_1, \mathbf{pred}\ a_1\ a_2, \mathbf{next}\ a_1\ a_1\ \},$$

obtenu en fusionnant les nœuds a_3 et a_1 dans Δ, n'est pas de type *Doubly.* En appliquant la troisième règle de réécriture, on obtient

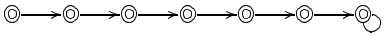
$$\{\mathbf{p}\ a_1, \mathbf{pred}\ a_1\ a_1, \mathbf{next}\ a_1\ a_2, \mathbf{pred}\ a_2\ a_1, \mathbf{next}\ a_2\ a_1, \mathbf{pred}\ a_1\ a_2, L\ a_1\}$$

La seconde règle de $\rightarrow_{Doubly}$ ne s'applique pas car la variable instanciant *y* (ici a_1) apparaît dans le reste du multi-ensemble.

La Figure 4-2 rassemble quelques exemples de types graphe spécifiant des structures impératives classiques. Les *skip lists* sont utilisées à la place des arbres équilibrés pour des insertions et destructions plus efficaces [130]. Les arbres rouge-noirs sont des arbres de recherche binaires dont les liens sont, soit "noirs", soit "rouges" [141]. Une propriété des arbres rouge-noirs est qu'aucun chemin de la racine à une feuille ne comporte deux liens rouges consécutifs. Cette propriété est exprimée dans le type graphe. Les arbres LCRS (*left-child-right-sibling trees*) sont utilisés pour modéliser des arbres avec un nombre de fils non borné [34]. Chaque nœud a un pointeur parent (***parent***), un pointeur vers son fils le plus à gauche (***leftc***) et un pointeur vers son frère droit (***rights***).

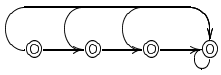
Simple lists:

$List = L\ x$
$L\ x = \mathbf{next}\ x\ y\ ,\ L\ y$
$L\ x = \mathbf{next}\ x\ x$



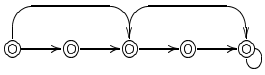
Lists with connections to the last element:

$Listlast = L\ x\ z$
$L\ x\ z = \mathbf{next}\ x\ y\ ,\ \mathbf{last}\ x\ z\ ,\ L\ y\ z$
$L\ x\ z = \mathbf{next}\ x\ z\ ,\ \mathbf{last}\ x\ z\ ,\ \mathbf{next}\ z\ z$



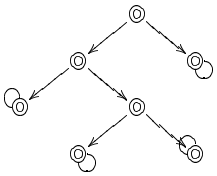
Skip lists of level 2:

$Skip = S\ x\ x$
$S\ x\ y = \mathbf{next}\ x\ z\ ,\ S\ z\ y$
$S\ x\ y = \mathbf{next}\ x\ z\ ,\ \mathbf{skip}\ y\ z\ ,\ S\ z\ z$
$S\ x\ y = \mathbf{next}\ x\ x\ ,\ \mathbf{skip}\ y\ x$



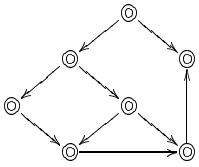
Binary trees:

$Bintree = B\ x$
$B\ x = \mathbf{left}\ x\ y\ ,\ \mathbf{right}\ x\ z\ ,\ B\ y\ ,\ B\ z$
$B\ x = \mathbf{leaf}\ x\ x$



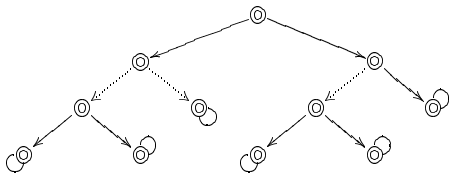
Binary trees with linked leaves:

$Binlink = L\ x\ y\ z$
$L\ x\ y\ z = \mathbf{left}\ x\ u\ ,\ L\ u\ y\ v\ ,\ R\ x\ v\ z$
$L\ x\ y\ z = \mathbf{left}\ x\ y\ ,\ R\ x\ y\ z$
$R\ x\ y\ z = \mathbf{right}\ x\ u\ ,\ \mathbf{next}\ y\ v\ ,\ L\ u\ v\ z$
$R\ x\ y\ z = \mathbf{right}\ x\ z\ ,\ \mathbf{next}\ y\ z$



Red-black trees:

$Redblack = L\ x$
$L\ x = \mathbf{leaf}\ x\ x$
$L\ x = \mathbf{leftb}\ x\ y\ ,\ R\ x\ ,\ L\ y$
$L\ x = \mathbf{leftr}\ x\ y\ ,\ R\ x\ ,\ B\ y$
$R\ x = \mathbf{rightb}\ x\ y\ ,\ L\ y$
$R\ x = \mathbf{rightr}\ x\ y\ ,\ B\ y$
$B\ x = \mathbf{leftb}\ x\ y\ ,\ \mathbf{rightb}\ x\ z\ ,\ L\ y\ ,\ L\ z$



Left-child, right-sibling trees:

$Lcrs = N\ x\ x$
$N\ x\ y = \mathbf{leftc}\ x\ z,\ \mathbf{parent}\ x\ y\ ,\ N\ z\ x\ ,\ L\ x\ y$
$N\ x\ y = \mathbf{leftc}\ x\ x,\ \mathbf{parent}\ x\ y\ ,\ L\ x\ y$
$L\ x\ y = \mathbf{rights}\ x\ z\ ,\ N\ z\ y\ ,\ L\ z\ y$
$L\ x\ y = \mathbf{rights}\ x\ x$

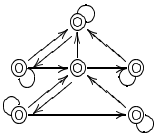


Figure 4-2 Exemples de types graphe

Les grammaires s'expliquent souvent facilement en donnant l'intuition de chaque non-terminal. Par exemple, dans la grammaire *Lcrs*, *N x y* dénote un arbre LCRS dont la racine est *x* et le parent *y*. *L x y* dénote une liste d'arbres LCRS dont le parent commun est *y* ; le premier arbre d'une liste *L x y* a comme racine *x*.

4.1.2 Transformateurs et réécritures de graphe

Un transformateur $C \Rightarrow A$ est une règle de réécriture de graphe qui remplace (dans le multi-ensemble) une instanciation de *C* (la *condition*) par une instanciation de *A* (*l'action*). Comme pour les règles de grammaire, les variables apparaissant uniquement dans le membre droit dénotent des adresses fraîches. Par exemple, le transformateur

P_1 : ***p*** *a*, ***next*** *a b*, ***pred*** *b a* $\Rightarrow$ ***p*** *a*, ***next*** *a a'*, ***pred*** *a' a*, ***next*** *a' b*, ***pred*** *b a'*

insère un élément entre le 1er et 2ème élément d'une liste *Doubly*. La variable fraîche *a'* dénote une nouvelle adresse (une allocation mémoire). Le transformateur

P_2 : ***next*** *a b*, ***pred*** *b a*, ***next*** *b c*, ***pred*** *c b* $\Rightarrow$ ***next*** *a c*, ***pred*** *c a*

enlève un élément d'une liste *Doubly*.

4.2 Vérification de type

La vérification de type revient à une preuve d'invariance : si un graphe *G* a le type graphe *T* et si *G* peut se réécrire en *G'* par le transformateur *P*, alors *G'* doit aussi être de type *T*. Formellement, soit *T* un type d'axiome *O* et $C \Rightarrow A$ un transformateur de graphes de type *T*, il faut montrer :

$$X + (\sigma\ C) \xrightarrow{*}_O \{O\} \quad \Rightarrow \quad X + (\sigma\ A) \xrightarrow{*}_O \{O\}$$

$(X + (\sigma\ C))$ dénote un multi-ensemble de type T dans lequel se trouve une instance $(\sigma\ C)$ de la condition *C* ; $(X + (\sigma\ A))$ dénote le même multi-ensemble après application de la règle $C \Rightarrow A$. On trouvera une description détaillée d'un algorithme de vérification dans [168]. Nous nous contentons ici de donner l'intuition.

Considérons la vérification du transformateur P_1 ci-dessus. La première étape est de trouver l'ensemble de tous les contextes *X* possibles tel que $X + (\sigma\ C) \xrightarrow{*}_O \{O\}$. Pour P_1 les contextes possibles sont représentés par l'arbre de réduction suivant :

p *a*, ***next*** *a b*, ***pred*** *b a*

$\downarrow$ *L b*

p *a*, *L a*

$\downarrow$ ***pred*** *a a*

Doubly

La racine de l'arbre est la partie gauche du transformateur (C = ***p*** *a*, ***next*** *a b*, ***pred*** *b a*) à vérifier. La seule façon de réduire C par $\rightarrow_{Doubly}$ est d'avoir *L b* dans le contexte. On représente cette réduction par un arc étiqueté par le contexte allant vers le terme réduit, ici C' = ***p*** *a*, *L a*. La seule possibilité pour réduire C' par $\rightarrow_{Doubly}$ est d'avoir ***pred*** *a a* dans le contexte. Le résultat de cette réduction est l'axiome (*Doubly*). La représentation de tous les contextes de réduction possibles est donc {*L b*, ***pred*** *a a*}. En d'autres termes, si le graphe auquel s'applique le transformateur est de type *Doubly* alors il est de la forme

$$\{L\ b, \textbf{pred}\ a\ a\} + \{\textbf{p}\ a, \textbf{next}\ a\ b,\ \textbf{pred}\ b\ a\}$$

L'ensemble {*L b*, ***pred*** *a a*} représente tous les X tels que

$$X + (\sigma\ C)\ \xrightarrow{*}_{Doubly} \{Doubly\}$$

La seconde étape consiste à vérifier que $X + (\sigma\ A)\ \xrightarrow{*}_{O} \{O\}$ pour tout contexte X trouvé. Pour notre exemple, il faut juste vérifier que

$$\textbf{p}\ a, \textbf{next}\ a\ a,\ \textbf{pred}\ a'\ a, \textbf{next}\ a'\ b,\ \textbf{pred}\ b\ a' + \{L\ b, \textbf{pred}\ a\ a\}\ \xrightarrow{*}_{RDoubly} Doubly$$

ce qui est direct. Notons que cette étape échouerait si nous avions oublié un pointeur (*e.g.* ***next*** *a' b*) ou interverti des adresses dans l'action.

Cet exemple est particulièrement simple. En général, l'ensemble des contextes possibles se représente par une grammaire de graphes. Formellement, l'ensemble des contextes est le langage quotient $L(O)/C = \{X \mid X+C \xrightarrow{*}_{O} \{O\}\}$ et la vérification revient à montrer que $L(O)/C \subseteq L(O)/A$. Les grammaires de graphe algébriques sont très expressives et le problème de l'inclusion de langage y est malheureusement indécidable. L'algorithme de vérification n'est donc pas complet. L'arbre de réduction exact peut être infini et des approximations sûres doivent être faites. Même si nous pensons qu'en pratique notre algorithme est capable de vérifier la plupart des trans-

formateurs, ce résultat théorique est déplaisant. Il est néanmoins possible de définir une sous-classe des grammaires de graphe et une sous-classe des transformateurs pour lesquelles il existe un algorithme de vérification complet. Sommairement, le système de réécriture sous-jacent doit être confluent et l'ensemble des contextes possibles du transformateur doit se représenter comme une collection finie de multi-ensembles. Ces restrictions ne sont pas toujours faciles à satisfaire. Elles fournissent toutefois un guide au programmeur pour réexprimer son programme lorsqu'un transformateur supposé correct est rejeté.

4.3 Intégration dans C

Nous présentons maintenant SHAPE-C, une extension de C qui intègre la notion de type graphe et de transformateur. La conception de SHAPE-C a été guidée par les impératifs suivants :

- les extensions doivent s'intégrer naturellement dans C,
- l'implémentation doit être efficace,
- l'algorithme de vérification de la section précédente doit s'appliquer.

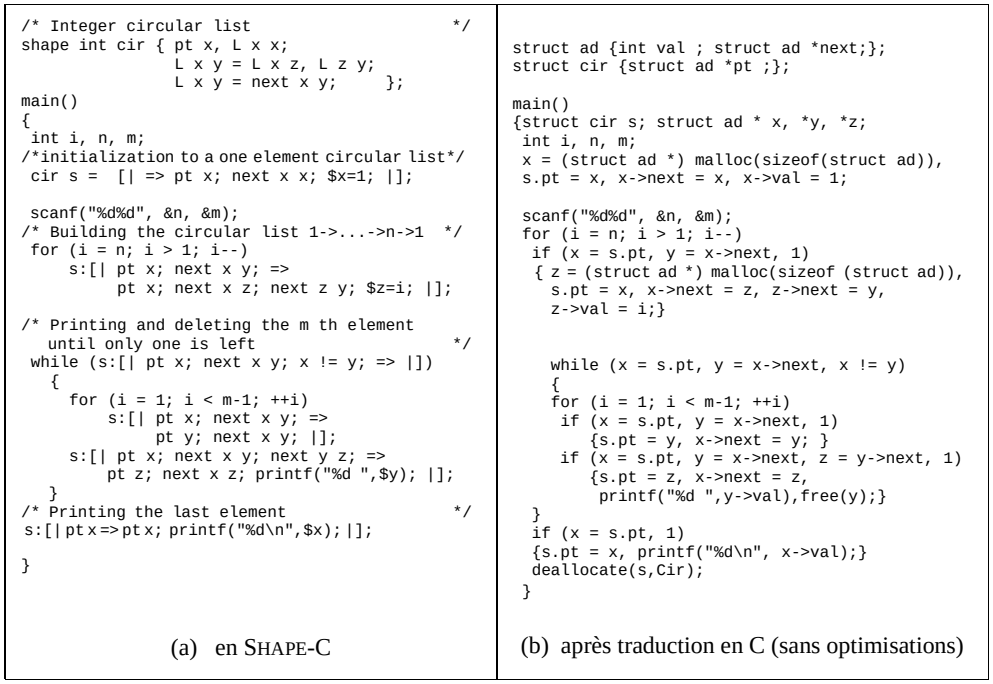
Nous présentons SHAPE-C à travers un exemple : le programme Josephus. Ce programme, tiré de [141], construit une liste circulaire de *n* entiers ; puis il parcourt la liste en passant *m*-1 éléments et détruisant le suivant, et ainsi de suite jusqu'à ce qu'il ne reste plus qu'un seul élément (qui pointe vers lui-même). La Figure 4-3 donne le programme Josephus en SHAPE-C et sa traduction en C.

4.3.3 Déclaration et représentation des types graphe

Le programme Josephus déclare un type graphe `cir` spécifiant les listes circulaires avec un pointeur `pt`. La déclaration d'un type graphe (appelé *shape*) est similaire aux grammaires algébriques de la section 4.1.

```
shape int cir { pt x, L x x;
                L x y = L x z, L z y;
                L x y = next x y;     };
```

Les variables de la grammaire sont interprétées comme des adresses. Elles possèdent une valeur dont le type doit être déclaré (ici `int`). Les valeurs peuvent être testées et modifiées dans le programme mais ne peuvent faire référence à des adresses. Elles n'ont aucun impact sur la vérification de type.



```
/* Integer circular list                          */
shape int cir { pt x, L x x;
                L x y = L x z, L z y;
                L x y = next x y;     };
main()
{
 int i, n, m;
/*initialization to a one element circular list*/
 cir s =  [| => pt x; next x x; $x=1; |];

 scanf("%d%d", &n, &m);
/* Building the circular list 1->....->n->1  */
 for (i = n; i > 1; i--)
     s:[| pt x; next x y; =>
          pt x; next x z; next z y; $z=i; |];

/* Printing and deleting the m th element
   until only one is left                        */
 while (s:[| pt x; next x y; x != y; => |])
   {
     for (i = 1; i < m-1; ++i)
         s:[| pt x; next x y; =>
              pt y; next x y; |];
     s:[| pt x; next x y; next y z; =>
         pt z; next x z; printf("%d ",$y); |];
   }
/* Printing the last element                      */
 s:[| pt x => pt x; printf("%d\n",$x); |];

}
```

(a) en SHAPE-C

```
struct ad {int val ; struct ad *next;};
struct cir {struct ad *pt ;};

main()
{struct cir s; struct ad * x, *y, *z;
 int i, n, m;
 x = (struct ad *) malloc(sizeof(struct ad)),
 s.pt = x, x->next = x, x->val = 1;

 scanf("%d%d", &n, &m);
 for (i = n; i > 1; i--)
  if (x = s.pt, y = x->next, 1)
  { z = (struct ad *) malloc(sizeof (struct ad)),
    s.pt = x, x->next = z, z->next = y,
    z->val = i;}


    while (x = s.pt, y = x->next, x != y)
    {
    for (i = 1; i < m-1; ++i)
     if (x = s.pt, y = x->next, 1)
        {s.pt = y, x->next = y; }
     if (x = s.pt, y = x->next, z = y->next, 1)
        {s.pt = z, x->next = z,
         printf("%d ",y->val),free(y);}
 }
  if (x = s.pt, 1)
  {s.pt = x, printf("%d\n", x->val);}
  deallocate(s,Cir);
 }
```

(b) après traduction en C (sans optimisations)

Figure 4-3 Le programme Josephus

Intuitivement, les relations unaires (ici `pt`) correspondent aux racines de la structure et les relations binaires (ici `next`) représentent des champs de type pointeur. Le type `cir` peut être traduit en C par :

```
struct ad  {int val ; struct ad *next;};
struct cir {struct ad *pt;};
```

Une adresse est représentée par une structure C (`struct ad`) avec un champ valeur (`val`) et autant de champs (de type pointeur vers `struct ad`) que le type graphe a de relations binaires (ici juste un). Le graphe lui-même est représenté par une structure (appelé *structure racine*) ayant autant de champs (de type `struct ad *`) que le type graphe a de relations unaires (ici, un pointeur `pt`). Par la suite, si *R* est une relation binaire, nous dirons que *x* (resp. *y*) est la *source* (resp. *destination*) dans *R x y.*

SHAPE-C utilise uniquement un sous-ensemble des types graphe qui correspond aux structures de données pointées des langages impératifs. Ce sous-ensemble est défini par les propriétés suivantes :

(S1) Les relations sont soit unaires, soit binaires.

(S2) Chaque relation unaire porte sur une unique adresse du graphe.

(S3) Les relations binaires sont des fonctions.

(S4) Le graphe complet peut être parcouru en partant des racines (relations unaires).

(S5) Une adresse est une source pour toutes les relations binaires.

Les quatre premières conditions correspondent directement aux propriétés des structures impératives. La dernière condition (S5) permet d'écarter le problème des pointeurs non initialisés. Cette condition, combinée avec (S2), assure que les racines et les pointeurs dans le graphe sont toujours valides. Les pointeurs nuls seront représentés par des éléments qui pointent vers eux-mêmes ; une technique courante en programmation C*. Excepté (S1) qui est purement syntaxique, les autres conditions sont vérifiées par une analyse simple (de type flot de donnée) de la grammaire.

4.3.4 Transformateurs

La réaction, notée `[| C => A |]`, est l'opération de base sur les graphes et correspond à un transformateur. Deux versions spécialisées d'une réaction sont également fournies : les *initialiseurs* `[| => A |]` dépourvus de condition et les *testeurs* `[| C => |]` dépourvus d'action.

Le programme Josephus déclare une variable locale `s` de type `cir` et l'initialise à une liste circulaire à un élément :

```
cir s = [| => pt x; next x x; $x = 1; |];
```

La valeur de l'adresse `x` est notée `$x` et est initialisée à 1. En général, les actions d'une réaction peuvent comporter des expressions C arbitraires sur les valeurs.

La boucle `for` construit une liste circulaire à *n* éléments à l'aide de la réaction

```
s:[| pt x; next x y; => pt x; next x z; next z y; $z=i; |];
```

La condition sélectionne l'adresse `x` pointée par `pt` et l'adresse suivante `y`. L'action insère une nouvelle adresse `z` et initialise sa valeur à `i`. L'interprétation des réactions comme des transformateurs est directe. La seule différence réside dans une

* Evidemment, cette technique de représentation n'empêche pas les erreurs. Par exemple, un parcours sans test de fin, qui s'arrête par une exception avec des pointeurs nuls, boucle avec des éléments pointant sur eux-mêmes.

nouvelle convention sur les noms de variables. Dans le contexte de la programmation, il nous a semblé judicieux de considérer que deux noms de variables différents pouvaient dénoter la même adresse. Par exemple, la réaction précédente dénote en fait deux transformateurs :

$$\textbf{\textit{pt}}\ x\ ,\ \textbf{\textit{next}}\ x\ y \Rightarrow\ \ldots \quad \text{et} \quad \textbf{\textit{pt}}\ x\ ,\ \textbf{\textit{next}}\ x\ x \Rightarrow\ \ldots$$

L'utilisateur peut expliciter la différence ou l'égalité de deux adresses à l'aide d'expressions de la forme `x == y` ou `x != y`. Par exemple, la boucle `while` décrit le retrait du *m*ème élément jusqu'à ce qu'il n'en reste qu'un. La condition est décrite par le testeur :

```
s:[| pt x; next x y; x != y; => |]
```

qui est fausse si `x` pointe vers lui-même.

La traduction en C des réactions nécessite tout d'abord de déclarer des variables afin de représenter les adresses manipulées par les réactions. Pour notre exemple, trois variables sont nécessaires :

```
struct ad *x, *y, *z;
```

Les conditions sont traduites en expressions booléennes avec effets de bord, comme

```
x = s.pt, y = x->next, x != y
```

pour le test de la boucle `while`. Les variables C dénotant les adresses sont initialisées avant d'effectuer les éventuelles comparaisons. Si aucun test n'est présent dans la condition, l'initialisation des adresses est suivie de `1` (*i.e.* "vrai" en C).

La traduction d'une action est faite d'affectation d'adresses et d'expressions C où les valeurs `$z` sont remplacées par le champ `val` de l'adresse pointée par `z`. Par exemple, la traduction de l'initialiseur de `s` est :

```
z = (struct ad *) malloc(sizeof (struct ad));
s.pt = x, x->next = z, z->next = y, z->val = i;
```

Cette implémentation efficace (après quelques optimisations locales simples) des réactions ne serait pas possible avec la définition générale des transformateurs. SHAPE-C impose les restrictions suivantes sur les transformateurs :

(R1) Deux variables peuvent dénoter la même adresse.

(R2) Dans une condition, une variable apparaît au plus une fois en tant que destination d'une relation.

(R3) Toute relation $\boldsymbol{f}_i\ x\ y$ dans une condition est précédée par une relation $\boldsymbol{f}_j\ z\ x$ ou $\boldsymbol{p}_j\ x$.

Les deux premières restrictions évitent de générer des tests. Par exemple, sans (R1) et (R2), la traduction de la condition ***next*** *x y*, ***next*** *y y* devrait inclure les tests `x!=y` et `y==y->next`. Dans SHAPE-C, le programmeur doit indiquer explicitement :

```
next x y; next y z; x!=y; y==z;
```

La dernière restriction (R3) permet de traduire une relation ***f*** *x y* en `y = x->f` car elle garantit que `x` a été initialisé. De plus, les conditions (S2) et (S5) sur les types graphe assurent que les déréférences faites dans la traduction sont valides.

4.3.5 Gestion mémoire

Nous avons traduit la déclaration d'instances de type graphe comme des déclarations de variables locales. En sortie de bloc, ces variables sont désallouées par la fonction `deallocate`. Cette fonction parcourt et libère la structure en partant de ses racines. La condition (S4) assure qu'un parcours complet est possible.

Un avantage de SHAPE-C est de rendre transparente la gestion mémoire des adresses d'une instance de type graphe. L'allocation est faite implicitement quand de nouvelles adresses apparaissent dans une action (comme dans la première boucle `for` de Josephus). De plus, une adresse qui apparaît comme une source dans une relation binaire de la condition et qui n'apparaît pas dans l'action peut être désallouée. En effet, les conditions (S2) et (S5) assurent que cette adresse n'apparaît plus dans aucune relation. Ce seul critère syntaxique est suffisant pour compiler totalement la gestion mémoire des graphes. Dans Josephus, ce cas est illustré par `y` dans la seconde réaction de la boucle `while`. La traduction explicite sa désallocation.

4.3.6 Interactions avec C

L'intégration des types graphe dans C est relativement intime. Par exemple, les valeurs des adresses d'une shape peuvent être de tout type C, les expressions C peuvent apparaître dans les réactions, le type "*pointeur sur shape*" est permis, etc. Les adresses des *shapes* ne doivent pas pouvoir être référencées par d'autres *shapes* ou

via des pointeurs C standards. Par construction, les adresses apparaissent uniquement comme des variables dans les relations ou dans les comparaisons des réactions. Seule une réaction peut les modifier. Il n'en reste pas moins qu'une utilisation indisciplinée de l'arithmétique de pointeurs ou des coercions (comme `(int *)` *intexp*) permettent d'enfreindre cette propriété. De telles pratiques sont risquées et fortement déconseillées ; SHAPE-C ne garantit rien dans ces cas.

Nous avons choisi de représenter une instance de type graphe par une structure de racines. Nous devons assurer une représentation unique du graphe et interdire la copie de ces structures. La restriction correspondante peut se formuler comme suit :

(C1) Le type `shape` est soumis aux mêmes restrictions que le type "*fonction retournant …*" en C.

En particulier, les shapes ne peuvent être affectés (sauf par les initialisateurs), ne peuvent être passés en paramètres ou rendus en résultat. Par contre, le programmeur peut utiliser des pointeurs sur `shape` pour les passer ou les rendre en résultat de fonctions.

Il est aussi essentiel d'assurer que les réactions sont des opérations atomiques. Une seconde restriction est nécessaire :

(C2) Les réactions imbriquées sur le même graphe sont interdites.

Une solution simple pour assurer cette restriction est d'interdire que les réactions contiennent des appels à des fonctions qui peuvent elles-même inclure des réactions.

4.3.7 Vérification de type

Avant la traduction en C, on vérifie que les réactions préservent le type graphe. Les valeurs et expressions n'ont aucun impact sur la forme du graphe. Les conditions et actions sont réduites aux relations et aux contraintes.

Pour une initialisation `T i = [| => A |]`, il suffit de vérifier que l'action A peut se réécrire dans l'axiome O, c'est à dire, $A \stackrel{*}{\rightarrow}_o \{O\}$.

En raison de nos conventions de nommage, une réaction se traduit en un ensemble de transformateurs correspondant à chaque possibilité d'(in)égalité entre variables (en accord avec les contraintes `x==y`, `x!=y` de la condition). La vérification d'une réaction revient à appliquer l'algorithme de la section 4.2 sur chaque transformateur de cet ensemble.

4.4 Travaux connexes

De nombreux travaux se sont préoccupé des propriétés de forme des structures de données. La plupart sont basés sur des analyses de programme comme des analyses d'alias ou de forme. Ces analyses utilisent différentes représentations abstraites de la mémoire (graphes *k*-limités, grammaires d'arbres, relations "pointe-vers", etc.) [26][42][176][54][82][94][136]. Ces approches, complètement automatiques, ont énormément de difficultés à inférer la forme précise de structures de données récursives.

D'autres travaux proposent des extensions linguistiques et une discipline de programmation associée. Le programmeur doit, par exemple, spécifier la forme de ses structures et les modifier de façon contrôlée. L'avantage de ces approches langages est de permettre la vérification de propriétés de forme précises.

La proposition la plus proche de la notre est sans doute celle de Klarlund et Schwartzbach autour des *graph types* [89]. Un *graph type* est défini comme un type récursif usuel (un arbre) complété de pointeurs auxiliaires pour exprimer le partage entre sous-termes. Ces pointeurs sont spécifiés par des expressions de routage [89] ou des formules logiques [90]. Quand la structure (l'arbre) est modifiée, les expressions de routage sont réévaluées et les pointeurs auxiliaires sont automatiquement mis à jour. Cette approche a deux inconvénients :

- Une opération basique sur un graphe peut impliquer un parcours de toute la structure pour remettre à jour les pointeurs auxiliaires. Ce coût (potentiellement de complexité linéaire) est caché au programmeur. Ceci est d'autant plus fâcheux que l'utilisation de pointeurs va souvent de pair avec le besoin de contrôler l'efficacité.
- La définition des types est assez peu naturelle. La spécification de la destination des pointeurs auxiliaires est souvent assez complexe. De plus, les pointeurs de l'arbre recouvrant et les pointeurs auxiliaires sont spécifiés par des techniques différentes. Il n'est pas satisfaisant de devoir distinguer un pointeur particulier dans une liste circulaire que ce soit du point de vue spécification ou implémentation.

D'autres approches, intermédiaires entre analyse et disciplines de programmation, sont basées sur des fragments de logique. Le langage de programmation est simplement équipé d'annotations permettant de décrire des propriétés de forme des structures de données. Ces annotations permettent au programmeur de spécifier des

pré et post-conditions ou des invariants de boucle. ADDS [68] associe des directions (*forward, backward*) aux pointeurs qui permettent de distinguer différentes structures de données et d'optimiser le programme. Ces annotations ne sont pas vérifiées. Les logiques utilisées par Jensen *et al.* [78] et Benedikt *et al.* [14] sont expressives et décidables. Malheureusement, l'algorithme de décision de [78] est de complexité non élémentaire et [14] se contente de montrer la décidabilité sans proposer d'algorithme réaliste.

4.5 Conclusion

De nombreux algorithmes utilisent des structures de données complexes qui sont invariantes lors de l'exécution [34][141]. Assurer ces invariants est une tâche sujette aux erreurs. Les types graphe permettent de décrire et de satisfaire ces invariants de façon relativement naturelle. Le formalisme des grammaires de graphe algébriques est très expressif même si la spécification de structures comme les grilles ou les arbres équilibrés reste hors de portée.

Nous avons développé un prototype pour expérimenter la programmation en SHAPE-C. Les programmes C obtenus sont efficaces après traduction et quelques optimisations locales standards. La complexité de l'algorithme de vérification est exponentielle mais en fonction de la taille de la grammaire et des transformateurs. En pratique, les structures de données se décrivent succinctement et l'algorithme est réaliste. Lors de nos expérimentations nous avons noté une certaine lourdeur de programmation résultant de l'utilisation exclusive de réactions. Le langage manque d'opérateurs de contrôle plus sophistiqués (*e.g.* `case`). Les structures devraient également pouvoir se parcourir (mais pas se modifier) *via* des pointeurs classiques. Ces extensions sont simples et ne demandent pas de modifier l'algorithme de vérification.

Enfin, notons que dans SHAPE-C, les graphes sont manipulés et vérifiés de façon totalement indépendante. Il serait utile de pouvoir combiner plusieurs graphes (éventuellement de types différents). Cette extension est plus profonde et constitue une piste de recherche à part entière.

CHAPITRE 5 *Langage dédié au parallélisme massif*

Un bon modèle de programmation parallèle doit être portable et avoir un coût prédictible. Les langages de programmation comme Fortran sont portables mais les estimations de coût sont le plus souvent très approximées. Une analyse de coût précise est particulièrement importante dans ce contexte puisque si la performance est le but premier du parallélisme massif, son impact sur la complexité des programmes est au plus une division par une constante. Un ordre de grandeur de la complexité ou une complexité maximale ne sont pas des mesures pertinentes pour guider une implémentation parallèle.

L'approche que nous prônons est basée sur un langage dédié qui, en plus de la portabilité, garantit une analyse de coût précise, symbolique et automatique. Notre langage est de type fonctionnel, pur, strict, dépourvu de récursion et de conditionnelle générales. En contrepartie, le langage est doté d'une collection de fonctions d'ordre supérieur appelées indifféremment, *schémas de programmes*, *patrons* ou *squelettes* [33][39]. Les restrictions syntaxiques du langage imposent une discipline de programmation assurant des performances prédictibles sur la machine parallèle cible. Comme au chapitre 2, le processus de compilation est décrit comme une suite de transformations de programme. Chaque étape de compilation transforme un langage à patrons dans un autre, plus explicitement parallèle. La chaîne de compilation produit des programmes fonctionnels de type SPMD (*Single Program Multiple Data*) qui se traduisent directement en code parallèle.

Les contributions de ce travail sont à la fois d'ordre technique et méthodologique. D'un point de vue technique, nous identifions un ensemble de restrictions linguistiques assurant une analyse de coût précise et symbolique. L'analyse prend en compte à la fois le balancement de charge et les communications. Elle permet de déterminer la meilleure distribution globale parmi un ensemble de distributions standards. D'un point de vue méthodologique, cette étude est un exemple rare de fertilisation croisée entre trois domaines de recherche : la parallélisation de pro-

grammes Fortran, les langages de squelettes et la programmation fonctionnelle. Nous réutilisons en particulier des techniques polyédriques développées pour Fortran et des techniques d'analyse et de transformation de programme développées dans la communauté fonctionnelle.

Nous décrivons tout d'abord notre langage dédié et ses restrictions en section 5.1. La chaîne de compilation comprend une analyse de taille, une analyse de globalisation, une explicitation des communications et une transformation compilant la distribution des données. La section 5.2 décrit succinctement ce processus sur un exemple. La caractéristique du code produit est que son coût parallèle (calcul + communication) est évaluable statiquement. Nous décrivons le calcul de coût en section 5.3. Nous donnons quelques résultats d'expérimentations en section 5.4, comparons aux travaux connexes en section 5.5 et concluons en section 5.6.

5.1 Langage source

Le langage dédié L_1 est un langage fonctionnel pur, strict, du premier ordre, à récursion et de conditionnelle bridées, étendu d'une collection de fonctions d'ordre supérieur (les patrons). Les schémas de programmes et les structures de données du langage sont dédiés au calcul numérique. La structure de donnée de base est le vecteur qui peut être imbriqué pour modéliser des tableaux multi-dimensionnels.

La syntaxe de L_1 est décrite en Figure 5-1 (son système de type est présenté en section 5.2.1). La grammaire n'interdit pas les fonctions définies récursives. Contraindre l'utilisation de la récursion est indispensable pour rendre l'analyse de coût décidable. Cette restriction est imposée séparément en vérifiant que le graphe d'appel est acyclique. Un programme est une expression principale suivie de déclarations. Une déclaration est une définition de fonction ou la déclaration des entrées (*i.e.* les variables libres) du programme. La taille (numérique ou symbolique) des vecteurs d'entrée est portée par leur type. Une expression (non-terminal Exp_1) est une application, un n-uplet, une variable ou une constante. Les fonctions comportent des λ-abstractions unaires*, des opérateurs unaires, des appels de fonctions ou de patrons, ou l'itérateur ***iterfor***.

* Couplé au système de type, ceci restreint le langage au premier ordre. Ce n'est pas une restriction indispensable mais les fonctions d'ordre supérieur compliquent les analyses et posent des problèmes d'efficacité. Une solution pour lever partiellement cette restriction serait d'utiliser une transformation préliminaire d'élimination de l'ordre supérieur (*e.g.* [29]).

$Prog_1$	::=	Exp_1 *where* $Decl_1$
$Decl_1$	::=	$Decl_1$ $Decl_1$ | f = Fun_1 | x : $Type_1$
$Type_1$	::=	($Type_1$, … , $Type_1$) | *Vect* $LinF_1$ $Type_1$ | *Int* | *Float* | *Bool*
Exp_1	::=	Fun_1 Exp_1 | (Exp_1,…,Exp_1) | x | k
Fun_1	::=	$\lambda(x_1,\ldots,x_n).Exp_1$ | Op_1 | f | ***iterfor*** $LinF_1$ Fun_1
	|	$Calc_1$ | $Reorg_1$ | $Comm_1$ | $Mask_1$
Op_1	::=	+ | - | * | ***div*** | ***exp*** | ***log*** | ***cos*** | …
$Calc_1$	::=	***map*** Fun_1 | ***fold*** Exp_1 Op_1 | ***scan*** Exp_1 Op_1
$Reorg_1$	::=	***zip*** | ***unzip*** | ***makearray*** $LinF_1$
$Comm_1$	::=	***brdcast*** $LinF_1$ | ***transfer*** $LinF_1$ $LinF_1$ | ***rotate*** $LinF_1$ | ***scatter*** $LinF_1$
	|	***gather*** $LinF_1$ | ***allgather*** | ***allbrdcast***
$Mask_1$	::=	$\mathbf{poly}_n\ \lambda(x_1,\ldots,x_n).Ineq_1\ Fun_1\ Fun_1$
$LinF_1$	::=	$LinF_1 + LinF_1$ | $LinF_1 - LinF_1$ | k*$LinF_1$ | x | k
$Ineq_1$	::=	$Ineq_1 \wedge Ineq_1$ | $LinF_1 < LinF_1$ | $LinF_1 = LinF_1$
$x, x_1, \ldots, x_n \in$ VarIdent. $f \in$ FunIdent. VarIdent $\cap$ FunIdent = $\varnothing$. $k \in$ Constant.		

Figure 5-1 Le langage L_1

L'itérateur ***iterfor*** *n f a* est une boucle appliquant *n*+1 fois sa fonction *f* sur l'argument *a* (***iterfor*** *n f a* = *f*(*n*,…,*f*(1,*f*(0,*a*))…)) L'indice d'itération est accessible au corps de boucle (*i.e.* la fonction *f*).

Quatre classes de patrons manipulent des vecteurs :

- Les *patrons de calculs* sont les fonctions classiques ***map***, ***fold*** et ***scan***. Les expressions ***fold*** ou ***scan*** sont réduites en parallèle si leur opérateur est associatif, en séquentiel sinon. D'autres patrons de calculs pourraient être considérés.

- Les *patrons de réorganisation* incluent ***zip***, ***unzip*** et ***makearray***. Ils permettent de restructurer les vecteurs (*e.g.* ***zip*** transforme une paire de vecteurs en un vecteur de paires) ou créer de nouveaux vecteurs (***makearray*** *n e* crée un vecteur de *n* éléments *e*. D'autres patrons de réorganisation pourraient facilement être pris en compte.
- Les sept *patrons de communication* (***brdcast, transfer***, ***rotate***, ***gather, scatter, allgather, allbrdcast***) décrivent des mouvements de données dans les vecteurs. Lorsqu'ils sont appliqués à un vecteur représentant la machine parallèle (comme dans le langage cible L_6, *cf.* section 5.2.4), ces mouvements représenteront (et seront implémentés par) des communications. Par exemple, ***brdcast*** *e v* rend un vecteur dont les éléments sont égaux au *e*+1ème élément de *v* (*e.g.* ***brdcast*** *i* $[a_0;\ldots;a_i;\ldots;a_n] = [a_i;\ldots;a_i]$). Il modélise l'envoie d'une valeur sur tous les processeurs. Ces patrons ont été choisis car ils représentent des communications collectives standards, optimisées sur la plupart des machines parallèles.
- Les *patrons de masques* sont la seule forme de conditionnelle proposée par L_1. Ils sont notés $\mathbf{poly}_n$ où *n* est le nombre de dimensions de l'argument vecteur. $\mathbf{poly}_n$ *p* f_1 f_2 *v* applique la fonction f_1 aux éléments de *v* contenus dans le polytope de dimension *n* décrit par le prédicat *p* et la fonction f_2 aux éléments externes au polytope*. Notons que $\mathbf{poly}_1$ ($\lambda i.i{=}i$) *f Id* est équivalent à ***map*** *f*. Cette restriction des conditionnelles est cruciale puisqu'on ne pourrait donner un coût précis à des expressions conditionnelles générales mais seulement, un coût maximal ou minimal.

Afin de pouvoir réutiliser les travaux d'évaluation de coût basés sur le calcul de volume de polytope, les arguments des patrons de communication, de l'itérateur ***iterfor***, de ***makearray*** et des patrons de masque doivent être des expressions affines dépendant uniquement de tailles de vecteurs ou d'indices d'itérateurs. Ceci est assuré à la fois par la grammaire (non-terminal LinF_1) et le système de types (*cf.* section 5.2.1). Une conséquence de ces restrictions est que les tailles des vecteurs manipulés par le programme ne dépendent que de constantes ou de la taille des entrées. Ce type de programme est parfois appelé *shapely* [76]. La taille (symbolique) des vecteurs, si importante pour l'analyse de coût, y est évaluable statiquement.

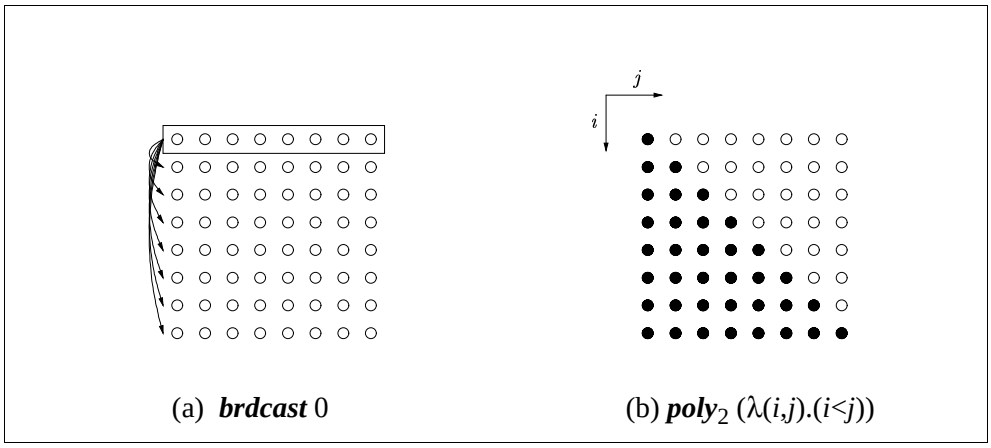
* Un *polytope* de dimension *n* est un polyèdre fini de dimension *n*. Un *polyèdre* de dimension *n* est un ensemble de points à *n* coordonnées vérifiant un ensemble d'inéquations affines.

5.2 Chaîne de compilation

Présenter la chaîne de compilation pour le langage complet serait beaucoup trop long ici. Le lecteur intéressé trouvera plus de détails dans [170] et une présentation complète dans [103]. Nous nous contentons de donner l'intuition des différentes étapes sur le petit programme suivant :

$$
\begin{array}{lll}
f\ m & \textbf{\textit{where}} & \\
 & m & : \ \textit{Vect}\ n\ (\textit{Vect}\ n\ (\textit{Float},\textit{Float})) \\
 & f & = \ \lambda m.(\textbf{\textit{poly}}_2\ (\lambda(i,j).(i<j))\ (+)\ (-)\ (\textbf{\textit{brdcast}}\ 0\ m),\ m)
\end{array}
$$

où m est une matrice de paires de flottants. Le patron ***brdcast*** 0 m rend une matrice constituée de copies de la première ligne de m (les mouvements de données sont montrés par la Figure 5-2-(a)). Puis, $\textbf{\textit{poly}}_2\ (\lambda(i,j).(i<j))\ (+)\ (-)$ somme les paires appartenant au triangle supérieur de la matrice (les points blancs de la Figure 5-2-(b)) et soustrait celles du triangle inférieur (les points noirs de la Figure 5-2-(b)). Le résultat est une paire composée de ces résultats et de la matrice initiale.



(a) ***brdcast*** 0 (b) $\textbf{\textit{poly}}_2\ (\lambda(i,j).(i<j))$

Figure 5-2 Mouvement de données (a) et patron de masque (b)

Le processus de compilation est décrit, dans le même esprit que le chapitre 2, comme une suite de transformations de programme :

$$L_1 \xrightarrow{G} L_2 \xrightarrow{E} L_3 \xrightarrow{A} L_4 \xrightarrow{D} L_5 \xrightarrow{O} L_6 \xrightarrow{T} \textit{Code}$$

Chaque flèche dénote la compilation d'une tâche en transformant les programmes d'un langage intermédiaire L_i à un autre L_{i+1}. Les étapes sont toutefois très différentes du chapitre 2 et le code généré n'est pas d'aussi bas niveau.

5.2.1 Inférence de type et de taille

La première étape est la vérification de type des programmes qui est également une analyse de taille. L'inférence de type a plusieurs objectifs. Elle vérifie que les programmes sont bien typés, elle assure que les manipulations de vecteurs sont bien définies (pas d'indice hors des bornes) et que certains arguments sont des expressions affines de tailles de vecteurs et d'indices d'itération. Enfin, elle calcule la taille (symbolique) de chaque vecteur du programme.

$$
\begin{array}{lll}
T & ::= & T_{Exp} \rightarrow T_{Exp} \\
T_{Exp} & ::= & (T_{Exp},\ldots,T_{Exp}) \mid \mathit{Vect}\ A\ T_{Exp} \mid \alpha \mid B \\
B & ::= & \mathit{Int} \mid \mathit{Float} \mid \mathit{Bool} \mid \mathit{Index}^A \mid \mathit{Size}^A \\
A & ::= & A + A \mid A - A \mid k*A \mid x \mid k
\end{array}
$$

$x \in$ VarTaille, $\alpha \in$ TypeTaille et $k \in$ Constante

Figure 5-3 Types taille

La syntaxe des types est décrite en Figure 5-3. Les types vecteurs comportent une expression affine (*A*) dénotant leur taille. Les types basiques sont, soit des scalaires (*Int*, *Float* ou *Bool*), soit des types *Size* ou *Index*. Intuitivement, une expression de type *Size* est une constante, la taille d'un vecteur d'entrée ou une expression affine de variables de type *Size*. Une expression de type *Index* est un indice d'itération, une expression de type *Size* ou une expression affine de variables de type *Index*. Les types *Size* et *Index* sont annotés par la valeur symbolique de l'expression (*A*). Par exemple, $\mathit{Size}^n \rightarrow \mathit{Vect}\ n\ \mathit{Int}$ est le type d'une fonction prenant une taille (de valeur symbolique *n*) et retournant un vecteur d'entiers de taille *n*.

Le système de type intègre la notion de sous-typage :

$$\mathit{Size}^a \subseteq \mathit{Index}^a \subseteq \mathit{Int} \subseteq \mathit{Float}$$

La relation $\mathit{Size}^a \subseteq \mathit{Index}^a$ indique entre autres que les indices de boucle peuvent dépendre de tailles mais la taille des vecteurs ne peut dépendre d'indices de boucle. Les règles d'inférence sont de la forme

$$C, \Gamma \vdash e : T, C_1$$

C'est à dire, *e* a le type *T* avec les contraintes de taille C_1 dans l'environnement Γ et les contraintes de sous-typage *C*. Si les contraintes C_1 sont satisfaites par la taille des paramètres d'entrée alors le typage garantit que l'exécution de *e* ne produit aucune erreur d'accès. La règle associée au ***brdcast*** est

$$\frac{C,\Gamma \vdash e : Index^{s1}, C_1}{C,\Gamma \vdash \textbf{\textit{brdcast}}\ e : Vect\ s\ \alpha \rightarrow Vect\ s\ \alpha,\ C_1 \cup \{0 \leq s_1 < s\}}$$

Elle impose que l'argument ait le type *Index* et introduit la contrainte ($0 \leq s_1 < s$) qui assure que cet indice est dans les bornes du second argument vecteur.

L'algorithme d'inférence repose sur les techniques standards pour traiter le sous-typage [110] et une bibliothèque de calcul polyédrique pour résoudre les contraintes [162]. En effet, les contraintes sont des inégalités affines dont les solutions sont les points d'un polyèdre convexe. Résoudre les contraintes revient à normaliser les inégalités et vérifier que le polyèdre résultant n'est pas vide. Toutes les expressions de vecteurs sont annotées par leur taille (obtenue par projection du polyèdre).

Sur notre exemple, (***poly***$_2$ ($\lambda(i,j).i<j$) (+) (-) (***brdcast*** 0 *m*), *m*) on obtient

m : *Vect n* (*Vect n* (*Float*,*Float*))

poly$_2$ ($\lambda(i,j).i<j$) (+) (-) :*Vect* s_1 (*Vect* s_2 (*Float*,*Float*)) $\rightarrow$ *Vect* s_1 (*Vect* s_2 *Float*)

brdcast 0 *m* :*Vect* s_3 (*Vect* s_4 (*Float*,*Float*))

avec l'ensemble de contraintes $\{s_1=s_3,\ s_3=n,\ s_2=s_4,\ s_4=n,\ 0 \leq s_2\}$.

En projetant sur la dimension de *n* on obtient $s_1 = n$, $s_2 = n$, $s_3 = n$, $s_4 = n$ et, par exemple, le patron de masque a le type

poly$_2$ ($\lambda(i,j).i<j$) (+) (-) : *Vect n* (*Vect n* (*Float*,*Float*)) $\rightarrow$ *Vect n* (*Vect n Float*)

5.2.2 Analyse de globalisation

Comme L_1 est un langage fonctionnel pur manipulant des tableaux, nous sommes confrontés au problème de la globalisation déjà abordé en section 3.1. Il nous faut montrer que chaque vecteur (d'entrée ou créé dynamiquement) peut être mis à jour en place. L'analyse de la section 3.1 ou le système de types linéaires de [Wadler90] pourraient être utilisés. Pour des raisons techniques (notamment, pouvoir prouver que la propriété de globalisation est préservée par les transformations successives), nous avons été amenés à concevoir une nouvelle analyse de globalisa-

tion [170][103]. Nous ne la décrivons pas ici.

Si tous les vecteurs ne peuvent être modifiés en place, le programme est transformé ($L_1 \xrightarrow{G} L_2$) en insérant des copies explicites de vecteurs. Ceci peut être fait automatiquement (l'analyse indique où insérer les copies) ou manuellement (le programmeur peut vouloir restructurer son programme pour insérer moins de copies). La globalisation est en première étape de la compilation afin de permettre au programmeur d'intervenir. Le langage L_2 est similaire à L_1. Il inclut deux opérateurs supplémentaires (***copy*** et ***dealloc***) et assure que tous les vecteurs sont globalisables.

La matrice de notre exemple n'est pas globalisable : elle est à la fois écrite et rendue en résultat. Une copie doit être insérée juste avant l'écriture (***brdcast***) sur m. On obtient le programme suivant :

$$(\textbf{\textit{poly}}_2\ (\lambda(i,j).i<j)\ (+)\ (-)\ (\textbf{\textit{brdcast}}\ 0\ (\textbf{\textit{copy}}\ m)),m)$$

La même analyse est utilisée pour garantir que les vecteurs sont, soit rendus en résultat, soit explicitement désalloués (par ***dealloc***). L'implémentation peut se passer de GC qui est compilé. La présence d'un GC aurait compromis l'évaluation des véritables coûts d'exécution.

5.2.3 Explicitation des communications

Intuitivement, afin d'exécuter une expression telle que $f(e,v) =$ ***map*** $(\lambda x.x+e)\ v$ en parallèle, la variable libre e doit être diffusée à chaque processeur avant d'appliquer la fonction. La transformation $L_2 \xrightarrow{E} L_3$ supprime ces variables libres et explicite ces communications. Dans L_3, aucune variable libre n'apparaît dans les arguments fonctionnels de ***map*** ou $\textbf{\textit{poly}}_n$. La transformation est proche du λ-lifting [80]. Par exemple, en supposant que le vecteur v soit de taille n, la fonction f précédente est transformée en

$$f'(e,v) = (\textbf{\textit{map}}\ (\lambda x.\lambda y.x+y) \circ \textbf{\textit{zip}}\ (\textbf{\textit{makearray}}\ n\ e))\ v$$

La variable libre e est éliminée en construisant (***makearray***) un vecteur de n éléments e. Ce vecteur est combiné avec v (***zip***) afin que chaque processeur (après distribution) ait une copie locale de e.

5.2.4 Distribution

Les transformations $L_3 \xrightarrow{A} L_4 \xrightarrow{D} L_5 \xrightarrow{O} L_6$ implémentent la distribution de

données. La première transformation *A* prépare la distribution proprement dite *D* dont le résultat est ensuite optimisé par *O*. La distribution considère un ensemble standard de distributions. Un vecteur peut être distribué cycliquement élément par élément, par bloc d'éléments ou alloué à un unique processeur. Pour une matrice (vecteur de vecteurs), ceci implique neuf distributions possibles (cyclique cyclique, bloc cyclique, cyclique de lignes, cyclique de colonnes, etc.). Les différents choix de distributions sont essayés à cette étape. Ils seront évalués par l'analyse de coût (*cf.* section 5.3) afin de choisir le plus efficace.

La première transformation est similaire à un algorithme d'abstraction. Les appels de fonctions sont dépliés et les λ-abstractions et les variables sont remplacées par un environnement global (pris et rendu en résultat par chaque expression) et des combinateurs d'accès.

La transformation d'un programme *F* selon une distribution *d* consiste à partir du programme équivalent

$$F \circ d^{-1} \circ d$$

et à propager la distribution inverse vers la gauche jusqu'à obtenir un programme de la forme

$$d'^{-1} \circ F' \circ d$$

Le programme transformé *F'* prend les entrées distribuées par *d*, et rend un résultat duquel on peut retrouver le résultat original par d'^{-1}. Après distribution les programmes manipulent un unique vecteur dont les éléments représentent les mémoires locales des processeurs. Ce sont des programmes SPMD composés de calculs parallèles et communications collectives de la forme

$$\ldots \textbf{\textit{pimap}}\ f \circ \mathit{Comm} \circ \textbf{\textit{pimap}}\ g \ \ldots$$

où ***pimap*** est la version de ***map*** sur le vecteur de processeur et qui, de plus, passe le numéro de processeur à sa fonction. On a également de nouvelles versions des patrons de communications (*e.g.* ***pbrdcast***) sur le vecteur de processeurs. Ils seront implémentés par des appels à des primitives de communications collectives.

Une distribution *d* est définie formellement comme une fonction de restructuration retournant un vecteur unique représentant la mémoire locale de chaque processeur. Nous donnons à cette structure de données le type *Vectproc*. Par exemple, la distribution ***bloc*** *p* découpe un vecteur en *p* blocs d'éléments contiguës qui sont alloués à chaque processeur (*e.g.* ***bloc*** 2 [1;2;3;4] = [[1;2];[3;4]]). Les distributions cy-

cliques et sur un unique processeur sont notées ***cyc*** et ***seq***. Les distributions se composent en utilisant les fonctions d'ordre supérieur ***dp*** et ***de*** pour traiter les paires et les vecteurs imbriqués. Par exemple, une distribution en bloc de colonnes s'écrit ***de seq*** (***bloc*** *p*).

Pour donner une idée de la transformation de distribution, la règle associée au patron $\textbf{\textit{poly}}_2$ est :

$D\,[\![\textbf{\textit{poly}}_2\ P\ F_1\ F_2 \circ \textbf{\textit{dei}}\ (\textbf{\textit{bloci}}\ p)\ \textbf{\textit{seqi}}]\!]$

$= \textbf{\textit{dei}}\ (\textbf{\textit{bloci}}\ p)\ \textbf{\textit{seqi}} \circ \textbf{\textit{pimap}}\ (\lambda(ip,v).\textbf{\textit{poly}}_2\ (P \circ \lambda(i_1,i_2).(ip*b+i_1,i_2))\ F_1\ F_2\ v)$

avec ***bloci*** *p* : *Vectproc p* (*Vect b* α) $\rightarrow$ *Vect n* α (***dei***, ***bloci*** et ***seqi*** sont les distributions inverses associées à ***de***, ***bloc*** et ***seq*** respectivement).

La règle pour $\textbf{\textit{poly}}_2$ modifie les coordonnées dans les inégalités décrivant le polytope. Par exemple, l'expression $\textbf{\textit{poly}}_2\ (\lambda(i,j).i<j)\ f\ g$ appliquée à une matrice 8×8 retourne la matrice de la Figure 5-4 où la fonction *f* a été appliquée aux points blancs et *g* aux points noirs. Après une distribution en blocs de lignes sur 4 processeurs, chaque processeur applique $\textbf{\textit{poly}}_2$ sur sa mémoire locale. La fonction *f* est appliquée sur les éléments (*i,j*) tels que 2**ip*+*i*<*j*, *ip* étant le numéro de processeur.

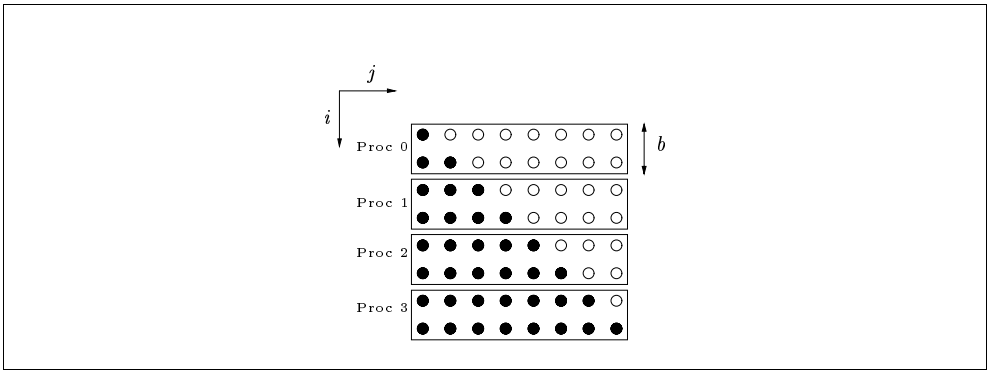


Figure 5-4 Impact de la distribution sur $\textbf{\textit{poly}}_2\ (\lambda(i,j).i<j)\ f\ g$

Cette règle repose sur l'information de taille calculée par l'inférence de type. Les inégalités après transformation définissent toujours un polytope. L'ensemble de distributions considéré assure cette propriété cruciale pour le calcul de coût (section 5.3). Après distribution en blocs de colonnes (***de seq*** (***bloc*** *p*)), l'exemple devient :

$\textbf{\textit{pimap}}\ \lambda(ip,(x,y)).(\textbf{\textit{poly}}_2\ (\lambda(i,j).i<ip*b+j)\ (+)\ (-)\ x,\ y)$
$\circ\ \textbf{\textit{pimap}}\ \lambda(ip,x).(\textbf{\textit{brdcast}}\ (0,(\textbf{\textit{copy}}\ x)),\ x)$

La transformation suivant la distribution ($L_5 \xrightarrow{O} L_6$) est une collection d'optimisations. Par exemple, les allocations (***copy*** ou ***makearray***) rendues inutiles par la distribution choisie, sont supprimées. Notre exemple précédent est transformé en

pimap λ(*ip*,(*x*,*y*)).(***poly***$_2$ (λ(*i*,*j*).*i*<*ip***b*+*j*) (+) (-) *x*,*y*)
∘ ***pimap*** λ(*ip*,*x*).(***makearray*** (*n*, ***lookup*** (0,*x*)), *x*)

où la copie de la matrice complète a été remplacée par une allocation de ligne par processeur.

5.2.5 Traduction en C

La transformation $L_6 \xrightarrow{T}$ *Code* est une traduction des programmes fonctionnels SPMD en du C avec des appels à la bibliothèque de communication MPI (*Message Passing Interface*). Ce choix garantit la portabilité du code qui est produit par le compilateur C de la machine parallèle. Les caractéristiques du code fonctionnel (premier ordre, strict, vecteurs globalisables) rendent la traduction directe. Le programme exécuté par tous les processeurs est composé des fonctions locales (arguments des ***pimap***), des boucles (***iterfor***), et d'appels à MPI (les sept patrons de communication sont des primitives de MPI).

5.3 Analyse de coût

En pratique, toutes les combinaisons de distributions des données d'entrées sont considérées et autant de programmes SPMD produits. Les coûts sont ensuite analysés et comparés. De nombreux programmes numériques n'ont qu'un petit nombre d'entrées (2 ou 3 matrices) et cette approche reste réaliste. Dans le cas contraire, il revient au programmeur de restreindre le nombre de choix de distributions.

Les restrictions du L_1, ainsi que le sous-ensemble fixé de distributions, assurent que le coût des programmes distribués s'exprime comme une somme de volumes de polytopes. Nous pouvons réutiliser les techniques de calcul de volume de polytopes développées dans la communauté Fortran [147][131][30].

Tout d'abord, les programmes sont abstraits en leur coût symbolique en termes d'inégalités, de sommes et de maxima. Par exemple, le coût de l'expression

pimap λ(*ip*,(*x*,*y*)).(***poly***$_2$ (λ(*i*,*j*).*i*<*ip***b*+*j*) (+) (-) *x*,*y*)

est :

$$\max|_{i_p=0}^{p-1}\left(\sum_{\substack{0\le j<b\wedge 0\le i<n\\ i<i_p b+j}} \alpha_+ + \sum_{\substack{0\le j<b\wedge 0\le i<n\\ (i_p b+j)\le i}} \alpha_-\right)$$

Le coût d'un ***pimap*** est le coût maximal sur l'un des *p* processeurs. Le coût d'un ***poly***$_2$ *P* f_1 f_2 est la somme des coûts de f_1 appliquée aux éléments de *P* (notée comme une somme généralisée décrivant le volume d'un polytope) et du coût de f_2 appliqué au complémentaire de *P*. Dans notre exemple, les coûts d'une addition et d'une soustraction sont notés α_+ et α_- ; ils dépendent de la machine considérée.

On peut ensuite appliquer les méthodes de calcul de volume de polytope. La méthode de Clauss [30] est exacte mais nécessite de connaître le nombre de processeurs. La méthode de Tawbi [147] est générale au prix d'approximations mineures (des arrondis de division entière). En utilisant cette deuxième technique, notre expression se réécrit en :

$$\max|_{i_p=0}^{p-1}(b^2(\alpha_+-\alpha_-)i_p + b^2(\alpha_+ + (p-1)\alpha_-))$$

La troisième étape est la suppression des maxima. Cela peut être fait sans approximation en calculant les zéros de la dérivée du polynôme dans le max. Dans notre cas, en supposant $\alpha_+ = \alpha_-$, l'expression se simplifie en $b^2p\alpha_+$.

La dernière étape est de comparer les coûts calculés pour les différentes distributions. Cela revient à déterminer quand le polynôme dénotant la différence de deux coûts est positif ou négatif. Des outils de calcul symbolique comme Maple [25] peuvent être utilisés dans ce but. Il se peut qu'une distribution soit toujours meilleure qu'une autre. En général, cela dépend de conditions sur la taille des données ou du nombre de processeurs (*e.g.* $Cost_1 > Cost_2$ ssi $n > p$). Maple rend alors des conditions symboliques qui guident le programmeur dans son choix. Une solution automatique est d'utiliser ces conditions comme des tests dynamiques pour choisir entre différentes versions (distributions) du programme.

5.4 Expérimentations

La chaîne de compilation a été partiellement implémentée à des fins d'expérimentation. Certaines étapes, complexes à mettre en œuvre comme les analyses de

globalisation et de coût, ont été réalisées manuellement sur les exemples.

Nous avons codé une demi-douzaine d'algorithmes numériques standards (factorisation LU, factorisation de Cholesky, itération de Jacobi, problème des n corps, …) que nous avons compilés et exécutés sur deux machines parallèles (une Intel Paragon XP/S et une Cray T3E).

Pour tous les exemples traités, la distribution choisie par le compilateur s'est avérée la meilleure en pratique. Les différences entre les coûts théoriques calculés et les coûts réels se sont avérés faibles (moins de 6% de différence). Les différences peuvent s'expliquer par le fait que le modèle de coût ne prend pas en compte la topologie de la machine cible ni le cache du processeur.

	LU (n=256)			LU (n=512)			Cholesky (n=1024)		
Proc.	Skel.	Seq.	HPF	Skel.	Seq.	HPF	Skel.	Seq.	HPF
1	2.14	1.73	2.16	14.77	13.61	15.36	67.45	53.17	65.40
2	1.34	×	1.38	8.43	×	8.67	35.54	×	35.97
4	0.93	×	0.95	5.25	×	5.41	21.35	×	20.65
8	0.76	×	0.77	3.23	×	3.38	14.81	×	13.10
16	0.66	×	0.67	2.97	×	3.06	11.55	×	9.53
32	0.62	×	0.61	2.57	×	2.67	9.91	×	7.83
	Householder (n=1024)			Jacobi (n=512)			N body (n=2048)		
Proc.	Skel.	Seq.	HPF	Skel.	Seq.	HPF	Skel.	Seq.	HPF
1	318.16	308.17	325.44	63.42	55.10	56.20	125.15	122.76	127.86
2	159.04	×	164.13	32.98	×	29.81	62.99	×	91.77
4	82.12	×	84.62	17.19	×	16.83	31.53	×	49.04
8	44.59	×	46.32	7.66	×	10.19	15.51	×	27.39
16	26.68	×	27.03	3.90	×	6.93	7.59	×	17.35
32	17.98	×	18.23	1.98	×	5.19	3.64	×	12.68

Figure 5-5 Temps (en sec.) pour L_1 (Skel.), C (Seq.) et HPF sur la Paragon.

Nous avons comparé l'exécution séquentielle de programmes L_1 avec les versions C correspondantes dans [128]. Nous avons également comparé notre implémentation parallèle avec High Performance Fortran (et une distribution manuelle). Les temps (*cf.* Figure 5-5) sont similaires et l'utilisation de patrons ne semble occasionner aucune perte pour ce style d'algorithme très régulier. Les différences en notre faveur pour Jacobi et Nbody sont vraisemblablement dues à la non reconnaissance de communications collectives par le compilateur HPF.

Nous avons également comparé notre code aux routines optimisées de ScaLA-

PACK avec les mêmes distributions. Même si la différence s'atténue quand le nombre de processeurs croit, notre code reste en moyenne 1,8 fois plus lent. Nous pensons que cette différence provient des primitives optimisées (BLAS) de calcul de matrices de ScaLAPACK. Une idée serait d'étendre L_1 avec de nouveaux patrons correspondant aux primitives BLAS.

5.5 Travaux connexes

D'autres langages dédiés au parallélisme de données sont basés sur des restrictions de la récursion. Le langage Alpha [163] est un langage fonctionnel du premier ordre où les arguments d'un appel récursif doivent être des expressions affines des paramètres de la fonction. Alpha, qui fut introduit pour exprimer des algorithmes systoliques, dispose d'analyses statiques très précises. Le langage Crystal [28] relâche la contrainte d'affinité pour gagner en expressivité mais les analyses statiques perdent en précision.

La communauté Fortran a étudié l'automatisation de la distribution de données [27][61]. Les coûts de communication et de calcul ne peuvent être correctement évalués pour le langage Fortran complet. Les techniques décrites dans [30], [131] et [147] se restreignent à un sous-ensemble de Fortran (boucles imbriquées et contraintes d'affinité) conduisant à un calcul de coût plus précis. Toutefois, les coûts de communication sont évalués en termes de communications point à point sans prendre en compte les routines de communications collectives de la machine [49].

Les patrons de Cole [33] ou de Darlington *et al.* [39] viennent chacun avec leur distribution (locale) optimale. En pratique, cela donne des implémentations inefficaces, redistribuant les données avant chaque appel de patron. Certains langages incluent des patrons de coordination et de distribution très expressifs [40][146]. Cette communauté a également défini des analyses de coût approximées pour des langages plus généraux que le notre [21][132][77]. Les coûts ne sont pas symboliques : les tailles des matrices et le nombre des processeurs sont supposés connus. Bratvold [16] et Michaelson *et al.* [105] se basent sur des jeux d'essais pour choisir la distribution. Ce style d'approche expérimentale pose problème lorsque la machine, le nombre de processeurs ou la taille des entrées changent. Signalons également l'analyse de Nitsche [118] qui vise à détecter les programmes *shapely* et Gorlatch *et al.* [60] qui décrivent des coûts de communications précis pour des combinaisons de ***scan*** et ***fold*** sur différentes topologies.

5.6 Conclusion

L'approche présentée a deux caractéristiques majeures :

- elle repose sur des restrictions syntaxiques qui autorisent une analyse de coût précise et le choix automatique de la distribution,
- elle intègre une sélection de techniques (typage, analyses statiques, transformation de programme, calcul de volume de polytope) originaires de diverses communautés.

Comme au chapitre 2, l'approche transformationnelle sépare et simplifie les preuves de correction. Pour chaque transformation $T : L_i \xrightarrow{T} L_{i+1}$, il faut montrer que les programmes transformés appartiennent bien à L_{i+1}, que la transformation respecte la sémantique des programmes et les propriétés de globalisation. La preuve de ces propriétés est la plupart du temps une simple induction sur la structure des expressions. Notons que si les transformations sont automatiques, l'utilisateur peut néanmoins interagir avec le compilateur pour modifier son code lors de l'étape de globalisation ou pour guider le choix de la meilleure distribution.

Nous pensons que de nombreux algorithmes de calculs numériques peuvent être codés dans notre noyau de langage. D'autres expérimentations seraient nécessaires afin de mieux évaluer l'expressivité du langage et l'efficacité de la compilation. Les algorithmes de l'ensemble de Cowinchan [164] semblent un bon point de départ. Les extensions suivantes méritent également qu'on s'y attarde :

- Nous avons pour l'instant considéré uniquement des machines de type MIMD (*Multiple Instructions Multiple Data*). Le processus de compilation et l'analyse de coût devraient s'adapter facilement à des machines de type SIMD (*Single Instruction Multiple Data*). Des expérimentations seraient utiles pour appuyer cette hypothèse et suggérer des optimisations spécifiques.
- Certains algorithmes demandent des redistributions en cours d'exécution. La technique développée pour la distribution se généralise à ce cas. Toutefois des interactions (de haut niveau) avec le programmeur seraient nécessaires afin éviter l'explosion du nombre de (types et d'endroits de) redistributions à considérer.
- L_1 gagnerait en expressivité avec des patrons de parallélisme de contrôle (*e.g.* diviser pour régner). Une telle extension pourrait s'appuyer sur des transformations du parallélisme de contrôle en parallélisme de données [69].

- L_1 doit être vu comme un noyau de langage parallèle inclut dans un langage séquentiel général comme C. Seuls les appels à des programmes de L_1 sont exécutés en parallèle. A l'inverse, on peut envisager l'appel de fonctions séquentielles C à partir de L_1. Cette extension est simple si on suppose que les fonctions ne manipulent pas de vecteurs et que leur coût symbolique est connu.

Plus généralement, les schémas de programmes se sont révélés être une technique intéressante pour concevoir des langages spécialisés. Ils permettent de définir des langages de haut niveau assurant des propriétés importantes sans empêcher des extensions (par ajout de nouveaux patrons). Cette approche à la conception de langages dédiés nous semble mériter plus d'attention.

CHAPITRE 6 *Imposer des propriétés par transformation de programme*

L'implémentation de certaines propriétés confronte le programmeur à deux problèmes :

- *Un problème d'expression.* De nombreuses propriétés ne s'expriment pas naturellement *via* les mécanismes d'abstractions usuels (procédures, fonctions, objets). En conséquence, leur implémentation est dispersée dans tout le code. Les propriétés de sécurité ou de robustesse en sont deux illustrations : elles nécessitent d'insérer des tests dans tout le programme.
- *Un problème de correction.* La dérivation de programmes à partir de spécifications ainsi que les analyses statiques de programmes permettent de vérifier que certaines propriétés sont correctement implémentées. Toutefois, la dérivation formelle est un procédé très coûteux et peu utilisé alors que les analyses sont dédiées à des propriétés spécifiques et rejettent des programmes corrects.

Nous présentons ici une nouvelle approche, inspirée de la programmation par aspects [87], répondant aux problèmes d'expression et de correction pour une classe particulière de propriétés. L'expression de la propriété est faite déclarativement et séparément du programme. La correction est assurée par un transformateur de programme qui prend le programme et la propriété et produit automatiquement un programme "équivalent" respectant la propriété. Le programme transformé diffère pour les entrées pour lesquelles le programme source viole la propriété. Dans ce cas, le programme transformé produit une exception et s'arrête.

Nous considérons des propriétés de sûreté exprimées sur la trace d'exécution des programmes. L'application la plus évidente est de sécuriser du code mobile à la réception. De nombreuses politiques de sécurité (*e.g.* les modèles de contrôle d'accès *high-water-mark* ou de la muraille de Chine [17]) peuvent se spécifier comme des propriétés de traces. Le code mobile (souvent sous forme de *byte-code*) contient assez de structure et d'information pour être analysé et transformé. L'avantage d'une telle transformation à la volée (*Just In Time*) est qu'elle est peut être faite selon des

propriétés de sécurité locales et personnalisées. Pour la classe de propriétés considérée ici, c'est une alternative flexible et simple au PCC (*proof-carrying code*) [117].

Nous commençons par donner l'intuition des différentes étapes de la technique (section 6.1). Nous détaillons en section 6.2 les étapes indépendantes du langage de programmation ; cette partie générique est le cœur technique de l'approche. Nous indiquons quelques extensions en section 6.3, comparons aux travaux connexes en section 6.4 et concluons en section 6.5.

6.1 Survol

Nous partons d'un programme *P*, d'une propriété sur les traces *T* et produisons un programme transformé *Trans*[*P,T*] respectant *T*. Si la trace de (P,σ_0) (le programme source et un état initial) respecte la propriété alors (*Trans*[*P,T*],σ_0) donne les mêmes résultats. Sinon, (*Trans*[*P,T*],σ_0) produit une exception et s'arrête juste avant la violation de *T*. En fait, nous prenons également en entrée une fonction *E* reliant les instructions de *P* aux événements d'intérêts pour *T*. Cette fonction fait le lien entre le programme et la propriété exprimée sur ces événements.

```
P :  manager();                    E :   manager(*)     ->   m
     if(...) accountant();               accountant(*) ->   a
     if(...) {critical();                critical(*)   ->   c
              manager() ;}
     accountant();
     critical();
```

$$T : ((a^+\ m \mid m^+\ a)\ (a \mid m)^*\ c)^*\ (a \mid m)^*$$

```
Trans[P,T] :   state = 0;
               manager();
               if(...) { state=1; accountant();}
               if(...) { if (state == 0) {abort;}
                         critical();
                         manager();}
               accountant();
               critical();
```

Figure 6-1 Assurer une propriété par transformation de programme (exemple)

La Figure 6-1 présente un exemple simple où les événements d'intérêts `m`, `a`, `c` sont les appels aux procédures `manager`, `accountant` et `critical`. La propriété *T*, exprimée comme une expression régulière, impose que les événements `m` et `a` aient lieu avant chaque événement `c`. Informellement, c'est une politique de sécurité interdisant les actions critiques n'ayant pas obtenu l'accord de la direction et de la comptabilité. Le programme source *P* peut violer cette propriété si la première conditionnelle est fausse. Le programme transformé *Trans*[*P*,*T*] est identique à *T* si ce n'est qu'il comporte deux affectations et un test supplémentaires. Il respecte la propriété en ce sens qu'il s'arrête en erreur au lieu de violer la propriété.

Notre approche comprend sept phases :

a. *Codage de la propriété.* Nous considérons ici des *propriétés régulières de sûreté*[*]. Une telle propriété peut se caractériser par un ensemble régulier de traces finies d'exécutions interdites. Le langage de définition des propriétés peut être un formalisme logique (*e.g.* LTL [45], WS1S [91]) ou des expressions régulières. Le point important est que la propriété se code en un *automate à état fini*. Le langage reconnu par l'automate sera l'ensemble des traces partielles autorisées.

b. *Annotation du programme.* La deuxième étape est de repérer et d'annoter les événements dont parle la propriété dans le programme. Ces événements d'intérêts peuvent être des affectations à certaines variables, des appels de méthodes, des ouvertures de fichiers, etc. Les instructions sans rapport avec la propriété sont annotées par l'événement vide ★. Il est important que chaque instruction du programme soit associée avec un unique événement. C'est naturellement le cas quand les événements sont de nature syntaxique (*e.g.* la fonction *E* dans la Figure 6-1). Quand les propriétés font intervenir des événements sémantiques une même instruction peut dénoter différents événements. Par exemple, si la propriété fait intervenir l'événement "*x prend la valeur 0* ", l'affectation `x:=x-1` génère cet événement ou non suivant l'état de la mémoire. Pour prendre en compte les événements sémantiques, le programme doit être transformé. Pour l'exemple précédent, l'instruction `x:=x-1;` est transformée en `if x=1 then x:=x-1 else x:=x-1` et l'affectation du `then` est associée à l'événement "*x prend la valeur 0* ". Cette étape repose sur une analyse statique afin d'éviter d'introduire des tests inutiles.

[*] Les propriétés de vivacité ne peuvent, en général, être imposées dynamiquement [135][140].

c. *Abstraction du programme.* Le programme est abstrait en un graphe dont les nœuds dénotent les points de programmes et les arcs représentent les instructions. Cette phase permet de définir le reste de l'approche indépendamment du langage de programmation. Nous considérons par la suite des graphes de flot de contrôle produit par analyse statique. Les traces générées par la représentation abstraite doivent être un sur-ensemble des traces d'exécution réelles.

d. *Instrumentation directe.* L'étape suivante transforme le graphe afin d'exclure de son ensemble de traces partielles celles interdites par l'automate. Une idée naturelle serait de considérer le graphe comme un automate et d'exprimer la transformation comme un produit d'automate et des optimisations. Nous n'avons pas suivi cette approche pour deux raisons. Tout d'abord, la représentation doit rester proche du programme source puisque l'objectif est d'y insérer des instructions. Ensuite, si un graphe intra-procédural est similaire à un automate, ce n'est plus le cas d'un graphe de flot de contrôle inter-procédural. Nous avons choisi de représenter l'intégration de l'automate dans le graphe par un graphe instrumenté ou *I*-graphe. Un *I*-graphe est un graphe avec des structures (états et fonctions de transitions) lui permettant de simuler l'évolution d'un automate. On commence par produire un *I*-graphe naïf (l'instrumentation directe) qui est optimisé par les deux étapes suivantes.

e. *Minimisation.* L'instrumentation directe est spécialisée au programme. Cette phase consiste à calculer l'ensemble des états atteignables pour chaque point de programme (nœud du graphe) et à faire une transformation similaire à la minimisation d'automate. Cette étape évite, entre autres, d'insérer des tests inutiles dans le programme.

f. *Suppression des transitions.* Le but de cette phase est de supprimer les transitions inutiles du *I*-graphe. Elle repose sur plusieurs analyses statiques : choix des transitions à supprimer, renumérotation des états et calcul du nouvel *I*-graphe. Cette étape supprime des évolutions inutiles de l'état de l'automate sous-jacent (*i.e.* réduit le nombre d'affectations insérées).

g. *Concrétisation.* Cette phase, duale de l'abstraction, génère le programme correspondant au *I*-graphe optimisé. Ce graphe est resté proche du programme source : ses nœuds et arcs représentent toujours les points de programmes et les instructions. La concrétisation reproduit le programme original augmenté d'instructions

faisant évoluer et testant une variable globale (l'état de l'automate). La concrétisation doit assurer que l'introduction de cet état n'interfère pas avec l'exécution normale du programme.

6.2 Cœur générique de l'approche

Les phases (d), (e) et (f) sont indépendantes du langage de programmation considéré. Elles constituent le cœur générique de l'approche que nous décrivons dans cette section. Nous commençons par introduire les structures abstraites manipulées (automates, graphes, graphes instrumentés) puis décrivons les trois transformations.

6.2.1 Représentations abstraites

Propriétés = automates. Une propriété est représentée par un automate à état fini $A = (Q, q_0, A, \Sigma, \delta)$ où Q est un ensemble fini d'états, q_0 est l'état initial, A l'ensemble des états finaux (d'acceptation), Σ l'alphabet (les événements) et δ la fonction de transition. Toute trace partielle qui ne viole pas la propriété est acceptée par l'automate. En pratique, ceci implique que tous les états sont finaux excepté un unique état puits représentant la violation de la propriété.

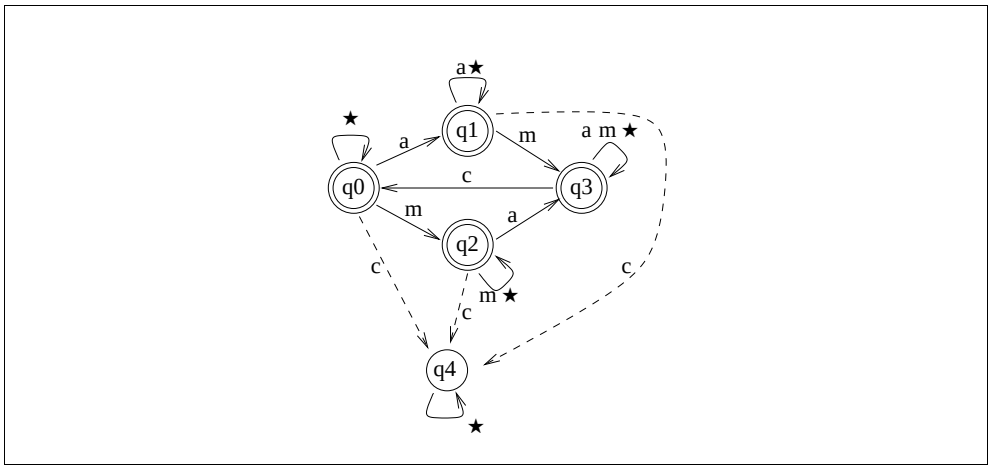


Figure 6-2 Exemple d'automate

L'ensemble des traces partielles vérifiant la propriété est défini par :

$$L(A) = \{w \in \Sigma^* \mid \delta(q_0,w) \in A\}$$

où, comme c'est l'usage, la fonction de transition δ a été étendue aux mots ($w \in \Sigma^*$). La Figure 6-2 présente l'automate correspondant à la propriété *T* de la Figure 6-1. La transition correspondant à l'événement vide $\star$ est ajoutée à chaque état. Quand la propriété est violée, l'automate passe (et reste) dans l'état puits q_4.

Programmes = graphes. Nous représentons un programme par un graphe $G = (V, v_0, E)$. Les nœuds (V) représentent les points de programme, la racine ($v_0 \in V$) dénote le premier point de programme, et les arcs ($E \subseteq V \times \Sigma \times \mathbb{N} \times V$) représentent les instructions générant les événements ($\in \Sigma$). L'arc (v, a, k, v'), noté $v \xrightarrow{a_k} v$', génère l'événement annoté a_k; l'entier k permet d'identifier sans ambiguïté l'instruction produisant l'événement. La notation est étendue aux chemins et on écrit $v \xrightarrow{w} v$' pour un chemin entre v et v' générant la chaîne d'événements annotés w (avec $w \in (\Sigma \times \mathbb{N})^*$). On note $\overline{w}$ la chaîne d'événements obtenue en supprimant les annotations entières de w.

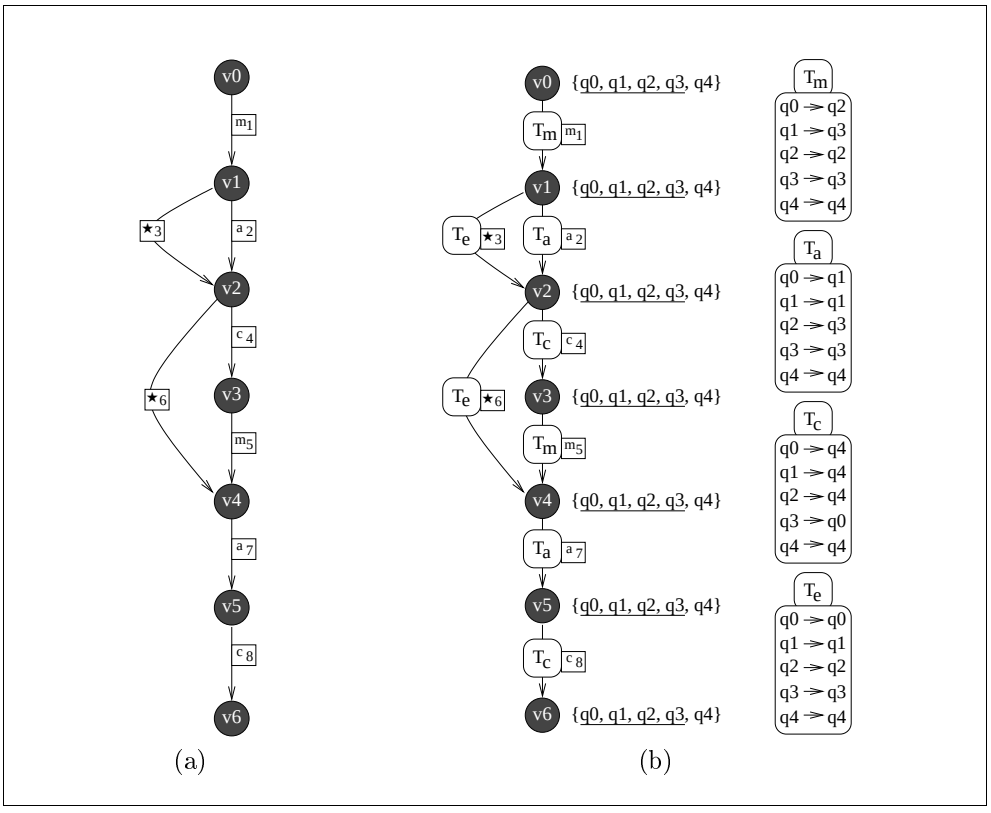


Figure 6-3 Graphe de flot de contrôle (a) et instrumentation directe (b)

L'ensemble des traces partielles générées par un graphe *G* est :

$$T(G) = \{w \in (\Sigma \times \mathbb{N})^* \mid \exists\, v \in V, v_0 \xrightarrow{w} v\}$$

La Figure 6-3 (a) présente le graphe dérivé du programme *P* de la Figure 6-2. Les arcs correspondant aux instructions non événementielles sont décorés par ★.

Programmes transformés = I-graphes. L'intégration d'un automate dans un graphe (partageant le même ensemble d'événements Σ) est représenté par un *graphe instrumenté* (ou *I-graphe*). Un *I*-graphe $I = (G, R, r_0, \gamma, S)$ est composé d'un graphe *G*, d'un ensemble d'états *R*, d'un état initial $r_0 \in R$, d'une collection d'états finaux $S = \{A_v \subseteq R \mid v \in V\}$ et d'une fonction de transition $\gamma : (\Sigma \times \mathbb{N}) \to R \to R$. La fonction de transition associée à l'événement annoté *e* est notée γ_e (au lieu de $\gamma(e)$).

L'instrumentation code les états et les transitions de l'automate. Chaque nœud du *I*-graphe est équipé d'un ensemble d'états finaux (d'acceptation) et chaque événement annoté vient avec une fonction de transition qui fait évoluer l'état en fonction de l'événement. L'ensemble des traces partielles d'un *I*-graphe *I* est défini par :

$$T(I) = \{w \in (\Sigma \times \mathbb{N})^* \mid \exists\, v \in V, v_0 \xrightarrow{w} v \wedge \gamma_w(r_0) \in A_v\}$$

où la fonction de transition γ a été étendue aux chaînes ($\gamma_{wv} = \gamma_v \circ \gamma_w$).

6.2.2 Instrumentation directe

L'instrumentation directe d'un graphe *G* par un automate *A* a les états de *A* comme ensemble d'états *R*, l'état initial de *A* comme état initial, les états finaux de *A* comme états finaux pour chaque nœud ($S = \{A_v = A \mid v \in V\}$) et associe à chaque arc la fonction de transition de *A* spécialisée à l'événement de l'arc ($\gamma = \lambda e.\lambda s.\delta(s,\bar{e})$). L'ensemble des traces partielles de l'instrumentation directe est tel que :

$$T(I) = \{w \in T(G) \mid \overline{w} \in L(A)\}$$

Cette propriété formalise le fait que l'instrumentation directe impose la propriété décrite par l'automate au graphe. La Figure 6-3 (b) présente le *I*-graphe obtenu pour le graphe et l'automate précédents. Les arcs sont décorés par une table de transition et les nœuds par un ensemble d'états où les états d'acceptation (finaux) sont soulignés.

Une concrétisation de ce *I*-graphe produirait un programme inefficace. Dans un cadre impératif, la concrétisation introduirait un test (l'état suivant est-il l'état puits?) et une affectation (la transition d'état) avant chaque instruction générant un événement. Nous décrivons maintenant comment transformer le *I*-graphe pour que

la concrétisation n'introduise que les tests et affectations indispensables.

6.2.3 Minimisation

Une première étape est de spécialiser l'instrumentation au programme. Nous calculons tout d'abord (par itération de point fixe) l'ensemble des états atteignables à chaque point de programme :

$$\forall\, v \in V \quad reach(v) = \{\gamma_w(r_0) \mid v_0 \xrightarrow{w} v\}$$

Détecter que l'état puits ne peut être atteint à un point de programme permet d'éviter un test.

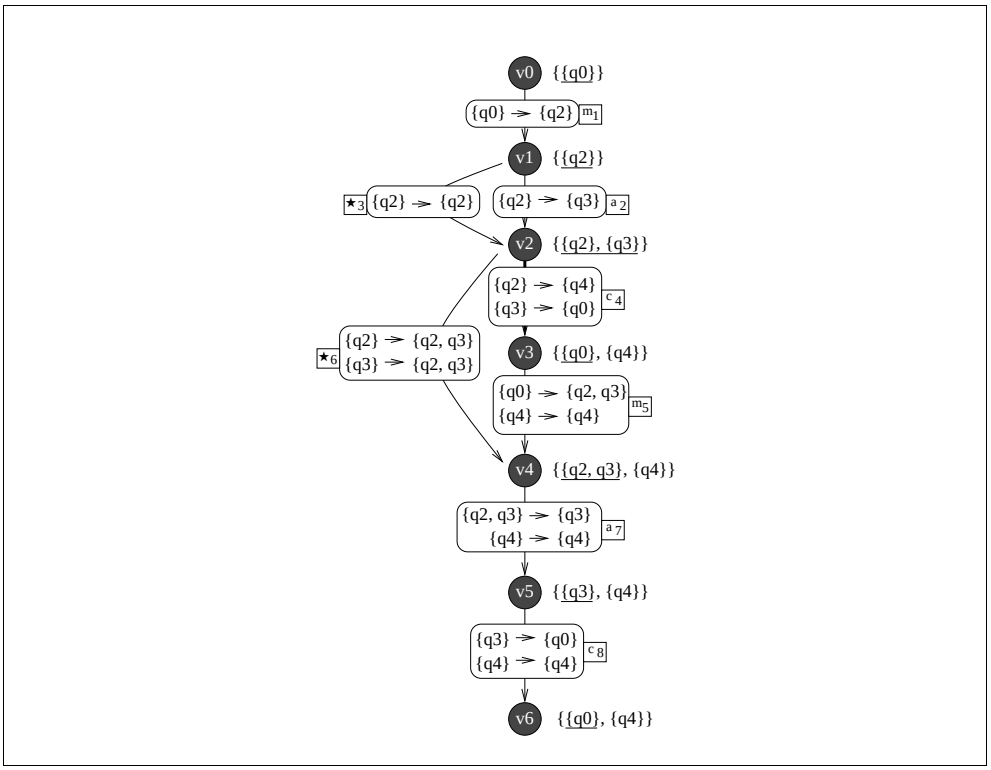


Figure 6-4 Après minimisation

Nous minimisons ensuite le *I*-graphe. L'idée est similaire à la minimisation d'automate : si le langage reconnu en partant de deux états est le même alors les états peuvent être fusionnés. Dans notre cas, ce langage est l'ensemble des traces partielles autorisées en partant du nœud *v* dans l'état *r*. Ce langage est noté $T(I\,[v,r])$ où $I\,[v,r]$ est le graphe instrumenté *I* avec comme nouvelle racine *v* et état initial *r*.

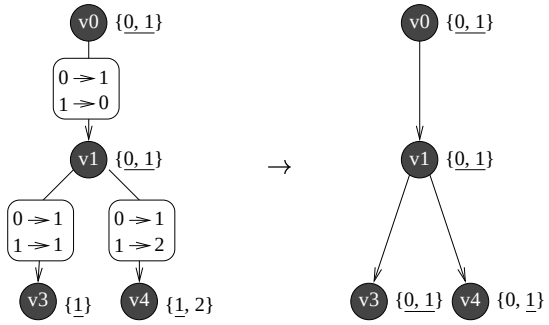
Deux états atteignables au nœud v sont considérés équivalents (et peuvent être fusionnés) si le langage généré en partant de v est le même pour les deux états.

Les états deviennent les classes d'équivalence des états atteignables, le nouvel état initial est la classe d'équivalence de l'ancien état initial, les nouveaux états finaux sont les classes d'équivalence des anciens états finaux et, pour chaque arc $v \xrightarrow{e} v'$, la nouvelle fonction de transition associe la classe d'équivalence de chaque état atteignable r au nœud v à la classe d'équivalence de l'état $\gamma_e(r)$ au nœud v'. La minimisation ne change pas la sémantique d'un I-graphe (l'ensemble des traces partielles est inchangé). L'intérêt principal de la minimisation est de permettre à la transformation suivante de produire une instrumentation optimale.

Le I-graphe minimisé pour notre exemple est présenté en Figure 6-4. Les états q_2 et q_3 ont été fusionnés au nœud v_4 (toutes les traces partielles partant de v_4 dans l'état q_2 ou q_3 sont valides). Ces états sont représentés par un état unique noté $\{q_2, q_3\}$.

6.2.4 Suppression des transitions

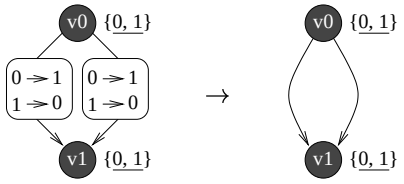
Les transitions représentent des évolutions d'état qui vont entraîner l'ajout de code dans le programme (*e.g.* des affectations dans un programme impératif). La suppression des transitions vise à remplacer autant que faire se peut les transitions par la transition identité. Les exemples suivants illustrent l'idée de cette transformation.



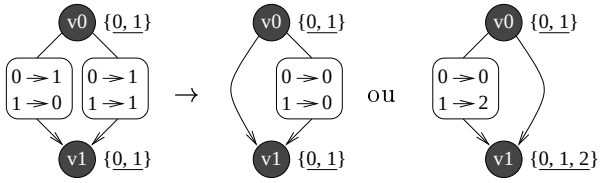
Toutes les transitions du graphe de gauche peuvent être remplacées par l'identité. Afin de garder le même langage de traces, les états d'acceptations doivent être changés (*cf.* par exemple, le nœud v_4). Des états doivent également être ajoutés aux nœuds. Par exemple, les états 0 et 1 au nœud v_3 dans le I-graphe transformé représentent l'état 1 dans le I-graphe de gauche. Toutes les transitions d'un arbre (graphe

sans partage) peuvent toujours être supprimées.

La même idée peut s'appliquer à d'autres formes de graphes comme le montre l'exemple suivant :



Les deux transitions sont identiques et il n'est pas nécessaire de distinguer les deux chemins. Les transitions peuvent être supprimées en même temps. Bien sur, certaines transitions restent indispensables. Dans le cas suivant :



on peut supprimer la transition de gauche ou celle de droite mais pas les deux.

En général, on ne peut supprimer toutes les transitions quand deux chemins du même nœud v vers le même nœud v' font évoluer le même état r vers deux états différents r_1 et r_2. Si on supprimait toutes les transitions, il n'y aurait aucune façon de distinguer r_1 de r_2 au nœud v'. Ce serait incorrect puisque les langages $T(I\ [v',r_1])$ et $T(I\ [v',r_2])$ peuvent être différent. En fait, après l'étape de minimisation, ces langages *doivent* être différents. C'est une des raisons d'être de la minimisation ; sans elle, on ne pourrait supprimer certaines transitions inutiles (*i.e.* quand $T(I\ [v',r_1]) = T(I\ [v',r_2])$).

Un *ensemble libre* est un ensemble d'arcs dont les transitions peuvent être supprimées (*i.e.* remplacées par l'identité). La suppression des transitions des arcs d'un ensemble libre ne change pas la sémantique des I-graphes.

Définition 6-5 *Un ensemble $F \subseteq E$ est dit libre pour $I = (G,R,r_0,\gamma,S)$ si :*

$$\forall\ v \xrightarrow{w1} v',\ v \xrightarrow{w2} v' \in F^*,\ \forall\ r \in reach(v),\ \gamma_{w1}(r) = \gamma_{w2}(r)$$

Une fois un ensemble libre choisi les transitions sont supprimées et les états re-

nommés. Le lecteur trouvera dans [179] une description formelle et détaillée de cette étape. Pour notre exemple, en prenant $\{m_1, \star_3, m_5, \star_6, a_7, c_8\}$ comme ensemble libre, on obtient le graphe de la Figure 6-6. Seule deux transitions restent contre six avant transformation.

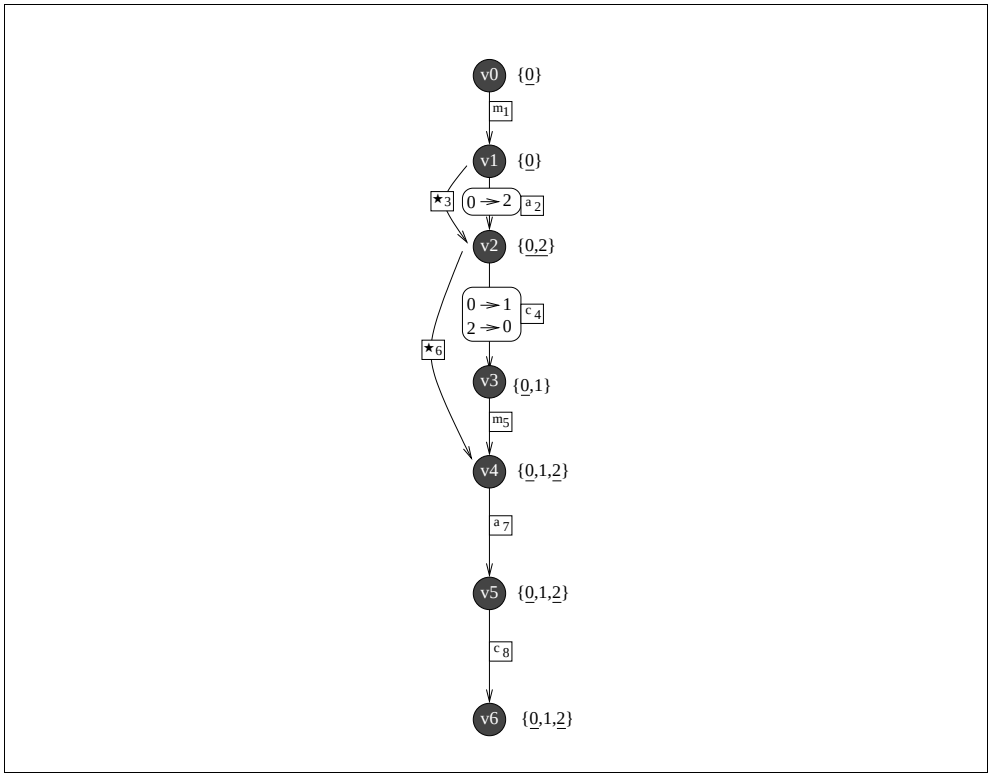


Figure 6-6 Après suppression des transitions

Nous avons souligné plus haut que la minimisation améliorait l'élimination des transitions. Elle amène en fait un résultat plus fort.

Propriété 6-7 *Soient les I-graphes I et I', si T(I) = T(I') alors* $\{e \mid \gamma'_e = Id\}$ *est un ensemble libre de la minimisation de I.*

Cette propriété est un résultat d'optimalité. Si *I'* est l'instrumentation optimale (*i.e.* elle maximise le nombre de transitions identité) équivalente à *I*, la Propriété 6-7 assure qu'elle peut être obtenue en choisissant un ensemble libre maximal pour la minimisation de *I*. Malheureusement, trouver un ensemble libre maximal est un problème NP-complet. Une heuristique possible est de prendre un arbre recouvrant arbitraire du graphe : l'absence de partage garantit que c'est un ensemble libre. On

essaie ensuite d'ajouter un arc en vérifiant que l'ensemble reste libre. Le processus s'arrête lorsque qu'aucun arc supplémentaire ne peut être ajouté. Pour un programme impératif structuré, l'arbre recouvrant seul assure qu'il suffit d'insérer une affectation à chaque instruction `if` et `while`.

6.3 Extensions

Nous avons jusqu'à présent travaillé sur un graphe de contrôle intra-procédural. L'extension au cas inter-procédural est esquissée en section 6.3.1. Il est également possible de considérer d'autres abstractions que des graphes de flot de contrôle. La section 6.3.2 indique l'intérêt et la manière de traiter des graphes d'appels.

6.3.1 Graphes inter-procéduraux

Modéliser les appels et retours de procédures par des arcs standards serait une approximation grossière* du flot de contrôle réel. Ce graphe comporterait de nombreux chemins impossibles (*e.g.* quand le chemin comporte un appel suivi d'un retour à un site d'appel différent). Nous représentons le flot de contrôle intra-procédural par une extension des graphes de flot de contrôle : les *graphes* CF (*context-free graphs*). Un graphe CF $G_{cf} = (V, v_0, E, Ret, C)$ est un graphe muni d'un nouvel ensemble de nœuds ($Ret \subseteq V$) représentant les retours de procédures et d'un nouvel ensemble d'arcs ($C \subseteq V \times V \times V$) représentant les appels. Un arc d'appel (v, f, v') signifie qu'un point de programme v appelle f et reprend au point v' après le retour de f. Les nœuds de retour n'ont pas d'arc sortant.

L'extension aux graphes CF revient à rendre les analyses sensibles au contexte des appels. La notion de trace est redéfinie en fonction de la pile d'appels. Par exemple, l'itération de point fixe calculant les états accessibles aux différents points de programmes, doit prendre en compte la nouvelle notion de trace. Un état est accessible à un point de programme s'il existe une trace l'atteignant en partant de l'état initial et d'une pile d'appels vide. La minimisation change profondément puisqu'il n'y a plus de relation d'équivalence entre états. On peut toujours appliquer l'algorithme précédent qui donne une minimisation partielle (insensible au contexte). La notion d'ensemble libre doit également être étendue pour prendre en compte

* Une analyse de flot de données sur ce type de graphe est dite insensible au contexte (*context insensitive*)

le contexte. En pratique et pour des programmes impératifs, la transformation ajoute au plus une affectation par instruction `if`, `while` et `call`.

6.3.2 Graphes d'appel et propriétés de pile

Le cœur de l'approche (décrit en section 6.2) s'applique telle quelle à d'autres abstractions. En particulier, notre technique peut être utilisée pour imposer des propriétés de sécurité définies comme des expressions régulières sur la pile d'appels. Ceci permet d'exprimer, entre autres, le modèle de sécurité de Java (JDK 1.2) basé sur l'inspection de pile. Une expression régulière sur un graphe de flot de contrôle ne pourrait pas exprimer ce style de politique.

Il suffit de changer la phase d'abstraction pour produire des graphes d'appels dont les chemins représentent l'ensemble des piles d'appels possibles. Le cœur technique (instrumentation directe, minimisation, suppression des transitions) ne change pas et optimise le graphe comme auparavant. La concrétisation doit par contre être adaptée. Il faut maintenant passer l'état de l'automate comme argument supplémentaire à chaque fonction. Cette technique, nommée *security passing style*, a été étudiée et proposée indépendamment dans [160][161].

6.4 Travaux connexes

On distingue trois types d'approches permettant d'imposer des propriétés : les approches dynamiques, statiques et mixtes.

- *Approches dynamiques.* Un processus concurrent surveille l'exécution du programme et le stoppe avant qu'il viole la propriété. Les moniteurs (par ex. VeriSoft [55] et Amos [31]) et les *security kernels* (comme les automates de sécurité de Schneider [140]) sont des représentants "système" de cette approche. Les représentants "langage" intègrent, sans optimisations, le moniteur en transformant le programme. Comme souligné dans la première partie, le fait de rester dans un même cadre (le langage de programmation) simplifie les preuves de correction. Comparé à l'approche système/moniteur, le code est plus portable et la sémantique du langage n'a pas à être étendue pour prendre en compte de nouveaux mécanismes de sécurité. Notre instrumentation directe (section 6.2.2) ainsi que le système Naccio [47] sont des approches langages dynamiques. Ces approches ne rejettent jamais de programme valide et peuvent imposer toute propriété de sûreté. L'inconvénient est le surcoût à l'exécution. La vérification dynamique n'est

pas spécialisée au programme. Pour faire un parallèle, une approche dynamique à la propriété de bon typage est l'implémentation du langage Scheme. Les vérifications de type dynamiques rendent l'approche flexible mais coûteuse.

- *Approches statiques.* Ici, la tâche d'imposer la propriété incombe au programmeur. Un analyseur statique est ensuite utilisé pour vérifier que la propriété est effectivement assurée. Citons, dans le domaine de la sécurité, les analyses statiques basées sur l'interprétation abstraite [111], le *model-checking* [79], ou les systèmes de types [67][152]. L'avantage principal de ce type d'approche est qu'il n'y a aucun coût à l'exécution. Cependant, une analyse peut rejeter des programmes valides. De plus, le problème d'expression de la propriété reste entier. Par exemple, dans un langage comme Java, les programmeurs doivent sécuriser leur code en insérant des appels à `AccessController.checkPermission` [59] à travers le programme. Les langages fortement typés peuvent être vus comme une approche statique pour assurer la propriété de bon typage. Le programmeur est ici guidé par la discipline de programmation imposée par le système de types.
- *Approches mixtes*. Une stratégie mixte ne rejette pas de programmes (elle ajoute des tests dynamiques) et s'efforce de minimiser le surcoût à l'exécution (en s'appuyant sur des analyses statiques). Notre approche appartient à cette famille. Le travail récent de Erlingsson et Schneider [46] s'attache également à spécialiser un automate de sécurité à un programme. Ils considèrent des automates plus généraux que les nôtres mais leurs optimisations sont plus simples et purement locales. En utilisant des analyses et transformations globales, nous supprimons plus de tests et d'évolutions d'état inutiles. De plus, en travaillant sur des abstractions de programmes, notre cadre est plus général que la restriction aux automates à état fini pourrait le faire penser (cf. section 6.3.2). Le *soft typing* [24] est une approche mixte à la propriété de bon typage. Au lieu de rejeter certains programmes non typables statiquement, des tests dynamiques sont ajoutés. Le but est similaire au notre : conserver un maximum de flexibilité à un coût minimal.

De nombreux autres travaux portent sur des propriétés de sécurité spécifiques. Citons, par exemple, les techniques qui ajoutent des tests dynamiques pour assurer des propriétés de typage/sécurité [119][98]. Même s'il se concentre sur une propriété de sécurité particulière, le travail de Wallach et Felten [161] partage plusieurs points communs avec le notre. Le modèle de sécurité de Java est spécifié par un automate à pile qui est implémenté par transformation de programme. Le code résultant est ensuite optimisé en utilisant une sorte d'élimination de code mort [160].

C'est une autre illustration du fait qu'une approche langage permet de spécialiser et d'optimiser les vérifications en fonction du programme.

6.5 Conclusion

La programmation par aspects (*Aspect-0riented Programming*) [87] fut une des sources d'inspiration de ce travail. Le but de la programmation par aspects est de séparer des aspects dont l'implémentation traverse tout le programme. L'intégration de l'aspect au programme est faite automatiquement par un transformateur de programme appelé tisseur. Même si la programmation par aspects est encore dépourvue de fondements théoriques, plusieurs cas d'études convaincants illustrent son potentiel [88]. Dans ces exemples, les aspects sont des annotations guidant des optimisations, décrivant la représentation des données ou la coordination de processus. Notre technique peut être vue comme un exemple de programmation par aspects où l'aspect est une propriété de trace. Cette séparation du programme et de la propriété conduit à des programmes plus faciles à développer et à maintenir. C'est particulièrement important dans le domaine de la sécurité. Il est difficile de prévoir toutes les attaques possibles et les programmes doivent pouvoir être changés rapidement pour faire face aux nouvelles menaces identifiées. De plus, considérer les aspects comme des propriétés permet de décrire et de contrôler précisément l'impact sémantique du tissage. C'est une autre caractéristique importante dans le contexte de la sécurité.

Les étapes présentées en section 6.2 ont été mises en œuvre en O'Caml. Des heuristiques simples ont été utilisées pour les étapes potentiellement coûteuses et la complexité globale du processus est linéaire en fonction de la taille du programme. L'application à la sécurisation d'applets à la réception est en cours de développement. Travailler sur le *byte-code* Java suppose quelques extensions. L'abstraction doit inclure une analyse de flot de contrôle pour produire des graphes précis en présence d'appels de méthodes virtuelles. On doit également prendre en compte le fait qu'une applet peut charger dynamiquement du nouveau code. Cette extension est plus délicate et il reste à voir si cela n'interdit pas certaines optimisations.

Cette étude suggère plusieurs pistes de recherches. Nous nous sommes limités à des transformations qui arrêtaient le programme quand une violation était détectée. Des traitements plus sophistiqués pourraient être considérés. Il faut toutefois garder l'impact sémantique des transformations sous contrôle. A plus long terme, il serait intéressant de concevoir un atelier générique de "programmation par propriétés" intégrant analyses et transformations de programme.

Historique

Les trois études présentées dans cette partie ont pour point commun une volonté d'explorer d'autres territoires que celui des langages fonctionnels stricto sensu.

Les types graphes découlent d'un travail en collaboration avec Daniel Le Métayer autour du formalisme Gamma (années 1995-1996). Le formalisme Gamma [9]*, proposé il y a près de 15 ans, permet une description abstraite des programmes dépourvue de séquentialité superflue. Notre objectif était de munir Gamma d'un système de types permettant de structurer les données et de contrôler la stratégie d'évaluation. On ne pouvait utiliser des types récursifs qui induisent un style de programmation séquentielle incompatible avec l'esprit de Gamma. L'extension, appelée* Gamma Structuré*, utilisait les grammaires algébriques de graphes pour décrire les structures de données* ([168][181][192])*. De façon indépendante, nous avions travaillé sur l'analyse de pointeurs dans les programmes C* ([176][191])*. Le but était de détecter automatiquement (a posteriori) certaines erreurs de pointeurs. L'application des idées de Gamma Structuré au langage C s'est imposée comme une approche langage (ou discipline de programmation) permettant de prévenir ce type d'erreurs. Il fut un moment évoqué l'approche a posteriori correspondante : concevoir une analyse de programme travaillant sur un domaine abstrait de grammaires de graphes. Cette piste s'annonçait trop complexe et ne fut pas poursuivie.*

Le but premier du travail sur le langage dédié au parallélisme (1995-1999) fut d'établir un lien entre nos thèmes de recherche et ceux d'une équipe voisine travaillant sur la synthèse d'architectures systoliques. Il était clair que ces deux domaines de recherche possédaient des points communs et étaient susceptibles d'enrichissement mutuel. Ce fut vers la même époque qu'une certaine pression privilégiant les recherches appliquées commença à se faire sentir. Il me semblait (il me semble toujours) que les langages dédiés était une façon intéressante de concilier applications et recherche académique. Le sujet de thèse proposé à Julien Mallet découle de ces motivations. Le chapitre 5 en décrit succinctement le résultat. Il s'est avéré qu'être à cheval sur différents domaines de recherche était une position assez délicate. Si, sur le principe, les études transversales sont clairement encouragées, elles rencontrent en pratique plus de difficultés que les autres.

Nous avons commencé à nous intéresser à la programmation par aspects à la fin 1997. Il était alors question d'établir une collaboration avec l'école de Mines de Nantes. Les racines "méta programmation objet" des aspects les intéressaient, alors que j'étais séduit par les slogans "separation of concerns" *et* "no smart compiler" *des premières présentations de Gregor Kiczales. Nous avons commencé par proposer un cadre générique* [182] *et son application à un exemple concret* [183]*. Un peu plus tard (début 1999), Tommy Thorn finissait sa thèse sur la vérification statique de propriétés de sécurité de programmes Java* [148]*. Le but était de vérifier statiquement qu'un programme (sécurisé manuellement) vérifiait bien une politique de sécurité. La rencontre des travaux préliminaires sur la programmation par aspects et de la thèse de T. Thorn a initié le travail décrit au chapitre 6. La politique de sécurité pouvait être vue comme un aspect de sécurité qu'il était possible de tisser. Il restait à trouver le moyen d'effectuer ce tissage aussi efficacement que possible. Ce fut le sujet de DEA proposé à Thomas Colcombet.*

CHAPITRE 7 *Conclusion*

7.1 Bilan

Tout domaine de recherche requiert un cadre de travail unifié permettant de décrire, prouver et comparer. Le chapitre 2 s'attache à démontrer que, pour le domaine de la compilation, le langage est souvent un cadre plus simple, plus adapté et plus précis que la sémantique. Certes, travailler sur des descriptions sémantiques est plus générique mais également très difficile. La génération de compilateurs à partir de la sémantique [81], par exemple, s'est heurté à cette complexité et n'a jamais vraiment abouti. Nous nous sommes focalisé sur les langages fonctionnels mais les principes de l'approche devraient pouvoir s'appliquer à d'autres paradigmes de programmation. Tout d'abord, le langage étudié doit pouvoir se représenter par un formalisme (ou calcul) simple et expressif. La compilation du langage peut ensuite se décrire et s'étudier comme des transformations dans le calcul. Si le λ-calcul [11] est *le* formalisme pour les langages fonctionnels, un calcul standard ne ressort pas aussi clairement pour les autres types de langages. On peut quand même citer les λ-calculs étendus avec l'affectation pour les langages impératifs (*e.g.* [50]), le calcul objet [3] et λ-Obj [52] pour les langages à objets, CCS [107] ou CSP [70] pour les langages parallèles, le π-calcul [108] ou les *ambients* [23] pour la mobilité.

Il ressort du chapitre 3 que la compilation de la stratégie d'évaluation et l'inférence de type sont des outils intéressants pour l'analyse de programmes. La transformation vers Λ_s permet de s'abstraire de la stratégie d'évaluation. En définissant une analyse de programme sur des expressions de Λ_s (ou en CPS), on profite naturellement de l'ordre d'évaluation rendu explicite. L'analyse gagne en généricité : elle est définie une seule fois pour toute stratégie. Quant à l'inférence de type, elle peut être utilisée pour déduire bien d'autres propriétés que celle de bon typage. L'inférence à la ML est une base simple et efficace qu'il est possible d'étendre. Il faut toutefois être conscient que l'inférence de types plus riches se

rapproche vite d'analyses par itération de point fixe.

Les analyses syntaxiques ont été justifiées par leur faible coût et par le fait qu'elles permettent à l'utilisateur de garder le contrôle. Ces avantages dépendent fortement de la propriété analysée. Si la propriété analysée a un impact faible sur le programme (*e.g.* l'allocation de registres) alors il est préférable d'optimiser le rapport précision/coût quel que soit le type d'analyse. Si l'impact peut changer l'ordre de grandeur de la complexité des programmes (*e.g.* la globalisation de tableaux), il est important que le programmeur en garde le contrôle. Dans ce cas, la meilleure approche est sans doute que la propriété soit assurée par construction, en utilisant des critères syntaxiques simples, un système de types et/ou une discipline de programmation.

Les disciplines de programmation au sens large (*i.e.* systèmes de types, langages dédiés, aspects, etc.) permettent d'assurer des propriétés qu'il serait impossible à un analyseur statique d'inférer. Le prix à payer est un changement de style de programmation et, parfois, une perte de flexibilité. Pour cette raison, certains considèrent les approches *a posteriori* comme plus pragmatiques et commercialement réalistes. Il nous semble que le prix des approches langages est acceptable s'il s'accompagne d'avantages bien identifiés (*e.g.* sous la forme d'outils). Par exemple, la description de grammaires se fait depuis longtemps à l'aide de langages dédiés. La génération automatique d'un analyseur syntaxique (par un outil comme Yacc) justifie largement cet effort. De même, choisir un langage fortement typé a un impact fort sur les habitudes de programmation. Si cette idée fait son chemin en dehors du monde académique, c'est grâce à l'inféreur de type, débogueur statique particulièrement utile.

Les trois études de la partie II associent toutes un outil à la discipline de programmation. Les types graphes permettent d'équiper un langage comme C d'un vérificateur de type beaucoup plus puissant. Le langage dédié au parallélisme garantit une analyse de coût précise et un choix automatique de la meilleure distribution. La programmation par propriétés vient avec un tisseur qui impose automatiquement des politiques de sécurité aux programmes.

7.2 Travaux en cours et perspectives

Les recherches autour des langages dédiés, en se plaçant aux confins de la théorie et de la pratique, sont particulièrement séduisantes. Le fait que l'objet d'étude soit spécialisé change la donne pour la programmation, l'analyse, la transformation

ou la compilation. La vérification de propriétés intéressantes devient souvent décidable. Il est donc envisageable de produire des outils puissants et réalistes pour ces langages. Actuellement, nous nous intéressons selon ces lignes à deux domaines de recherche : les architectures de logiciel et de la programmation par aspects. Ces deux domaines ont comme point commun de proposer de *nouvelles notions de modularité*. Les architectures de logiciel séparent la description d'un système en une collection de vues interdépendantes. La programmation par aspects sépare la description d'un programme en un composant principal et une collection d'aspects.

Architectures de logiciels

Le domaine des architectures de logiciel a connu un développement important ces dernières années. La tendance actuelle consiste à proposer des langages dédiés à la définition de l'architecture (ou organisation globale) de gros logiciels. Il faut ensuite fournir des outils pour décrire, tester ou analyser ces architectures. Nous avons ainsi étudié l'analyse de propriétés de coût et de sécurité sur des architectures afin de guider la mise en œuvre de services complexes [180].

Il s'avère en fait que les propriétés qu'on souhaite vérifier sont de natures très diverses et que chacune d'elle peut demander une présentation différente de l'organisation du logiciel. La définition d'une architecture comme un ensemble de vues complémentaires est une solution mais il est alors nécessaire de traiter le délicat problème des *relations* et de la *cohérence* de ces vues multiples.

Nous avons abordé cette question en utilisant une notation à la UML (graphes avec multiplicités) pour décrire les vues [178][194][120]. Nous avons considéré des attributs simples (de style annotation de type) et proposé une technique pour exprimer et vérifier des contraintes structurelles intra et inter-vues. Un exemple de contrainte inter-vue que l'on peut vouloir vérifier est que deux processus communiquant par variable partagée dans la vue fonctionnelle doivent être placés sur la même machine dans la vue physique. Le problème de la cohérence est d'autant plus complexe et important quand les vues portent des attributs plus riches (*e.g.* sécurité, comportement, etc.). Le problème devient très proche de la preuve de non-contradiction de spécifications. On peut, par exemple, vouloir montrer que l'ajout d'un lien de communication qui améliore la tolérance aux fautes, n'est pas néfaste pour la sécurité en introduisant un flux d'information illicite. Nous commençons à regarder ce type de problème. Au lieu de travailler sur des graphes avec multiplicités (*i.e.* spécifiant des familles d'architectures), nous considérons des vues

spécifiées par des graphes simples. Les descriptions sont certes moins génériques mais devraient nous permettre de décrire des relations inter-vues plus complexes et de vérifier des propriétés de cohérence sémantique.

Programmation par aspects

La programmation par aspects présente un logiciel comme un composant principal et une collection d'aspects (qui seraient autrement éparpillés à travers le code). Un outil, appelé tisseur, est chargé de produire automatiquement un programme intégrant les différents aspects au composant. Nous avons commencé à nous intéresser à cette approche en proposant un cadre générique de programmation par aspects [182]. Nous avons ensuite étudié l'utilisation de ce paradigme pour la production de logiciels robustes [183] et pour imposer des politiques de sécurité [179] (chapitre 6).

De manière plus fondamentale et prospective, un sujet important concernant les aspects est la conception d'un cadre formel permettant de raisonner sur leur impact sémantique. On ne peut ni restreindre les aspects à des optimisations de programmes (la majorité des aspects intéressants en serait exclue), ni autoriser tout type de transformation de programme (auquel cas on ne peut plus rien dire en général). Une voie intermédiaire est de n'autoriser que des impacts sémantiques contrôlés (comme l'interdiction de certaines exécutions dans les aspects de sécurité). Plus généralement, on peut restreindre l'impact sémantique d'un aspect à une sorte de raffinement de programme.

De façon idéale, la construction de programme dans un cadre aspect consisterait à écrire un programme composant de très haut niveau (*e.g.* le formalisme Gamma) et une collection d'aspects décrivant des choix de mise en œuvre (stratégie d'évaluation, représentation des données, gestion mémoire, optimisations, etc.). Le programme (bas niveau et efficace) intégrant ces différents choix serait produit automatiquement par tissage.

La devise des langages fonctionnels a longtemps été présentée comme "décrire le *quoi*, le compilateur se charge du *comment*". Cette devise visait à marquer la différence avec les programmes impératifs où le quoi et le comment sont enchevêtrés. Dans la pratique, les compilateurs de langages fonctionnels ont souvent du mal à réaliser le comment aussi efficacement qu'on le souhaiterait. Il est tentant de penser qu'une approche aspect pourrait "décrire le *quoi* dans le composant, le *comment* dans les aspects" et ainsi concilier haut niveau et efficacité.

Bibliographie

[1] M. Abadi, L. Cardelli, P. L. Curien et J.-J. Levy. Explicit Substitutions. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, p. 31–46, 1990.

[2] M. Abadi, L. Cardelli, P. L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121, p. 9-58, 1993.

[3] M. Abadi et L. Cardelli. *A Theory of Objects*, Springer-Verlag, New York, 1996.

[4] S. Aditya et A. Caro. Compiler-directed type reconstruction for polymorphic languages. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p. 74–82, 1993.

[5] A. W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2, p. 153–162, 1989.

[6] A. W. Appel. *Compiling with Continuations*. Cambridge University Press. 1992.

[7] G. Argo. Improving the three instruction machine. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p. 100–115, 1989.

[8] A. Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, 2(1), p. 23–59,1992.

[9] J.-P. Banâtre et D. Le Métayer. Programming by multiset transformation, *Communications of the ACM*, vol. 36–1, p. 98–111, January 1993.

[10] H. G. Baker. Unify and conquer (garbage, updating, aliasing, …) in functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, p. 218–226, 1990.

[11] H. P. Barendregt. *The lambda calculus. Its syntax and semantics*. North-Holland, 1981.

[12] F. Bellegarde. Rewriting systems on FP expressions to reduce the number of sequences yielded. *Science of Computer Programming*, 6, p. 11–34, 1986.

[13] Z. Benaissa, P. Lescanne et K. H. Rose. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Proc of PLILP'1996*, p. 393–407, 1996.

[14] M. Benedikt, T. Reps et M. Sagiv. A decidable logic for describing linked data structures. In *Proceedings of 8th European Symposium on Programming*, LNCS, vol. 1576, p. 2–19, 1999.

[15] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p. 26–38, ACM Press, 1989.

[16] T. Bratvold. A Skeleton-based parallelising compiler for ML. In *Proceedings of the International Workshop on Parallel Implementation of Functional Languages*, p. 23–33, 1993.

[17] D. F. Brewer et M. J. Nash. The Chinese wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, p. 206–214, May 1989.

[18] D. E. Britton. *Heap Storage Management for the Programming Language Pascal*. Master's Thesis, University of Arizona, 1975.

[19] G. Burn, S. L. Peyton Jones et J.D. Robson. The spineless G-machine. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, p. 244–258, 1988.

[20] G. Burn et D. Le Métayer. Proving the correctness of compiler optimisations based on a global analysis. *Journal of Functional Programming*, 6(1), p. 75–110, 1996.

[21] W. Cai et D. Skillicorn. Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Letters*, 5(2):179–190, 1995.

[22] L. Cardelli. Compiling a functional language. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, p. 208–217, 1984.

[23] L. Cardelli et A. D. Gordon. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)*, LNCS, vol. 1378, p. 140–155, 1998.

[24] R. Cartwright et M. Fagan. Soft typing. In *Proceedings of PLDI '91, Conference on Programming Language Design and Implementation*, p. 278–292, June 1991.

[25] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan et S. M. Watt. *Maple V Language Reference Manual*. Springer-Verlag, 1992.

[26] D. Chase, M. Wegman et F. Zadeck. Analysis of pointers and structures. In *Proceedings ACM Conference on Programming Language Design and Implementation*, vol. 25(6) of SIGPLAN Notices, p. 296–310, 1990.

[27] S. Chatterjee, J. R. Gilbert, R. Schreiber et S. Teng. Automatic array alignment in data-parallel program. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, p. 16–28, 1993.

[28] M. Chen, Y. Choo et J. Li. Crystal: theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers*, chapter 7, p. 255–308. ACM Press, 1991.

[29] W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, 1990.

[30] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proceedings of the ACM International Conference on Supercomputing*, p. 278–285, 1996.

[31] D. Cohen, M. S. Feather, K. Narayanaswamy et S. S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, p. 602–603, Springer, May 1997.

[32] J. Cohen. Garbage collection of linked data structures. *Computer Surveys*, vol. 13, 3, 1981.

[33] M. Cole. A skeletal approach to the exploitation of parallelism. In *Proceedings of CONPAR*, p. 667–675, Cambridge University Press, 1988.

[34] T. H. Cormen, C. E. Leiserson et R. L. Rivest. *Introduction to algorithms*, MIT Press, 1990.

[35] B. Courcelle. Graph rewriting: an algebraic and logic approach, *Handbook of Theoretical Computer Science, Chapter 5*, J. van Leeuwen (ed.), Elsevier Science Publishers, 1990.

[36] G. Cousineau, P.-L. Curien et M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2), p. 173–202, 1987.

[37] P. Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. Thèse de l'université de Paris VII, 1991.

[38] H.B. Curry et R. Feys. *Combinatory Logic*, vol. 1, North-Holland. 1958.

[39] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu et R. L. While. Parallel programming using skeleton functions. In *Proceedings of PARLE*, LNCS, vol. 694, p. 146–160, 1993.

[40] J. Darlington, Y. K Guo, H. W. To et Y. Jing. Skeletons for structured parallel composition. In *Proceedings of ACM Symposium on Principle and Practice of Parallel Programming*, pages 19–28, 1995.

[41] N. G. De Brujin. λ-calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to Church Rosser theorem. In *Indagationes mathematicae*, 34, p. 381–392, 1972.

[42] A. Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers, In *Proceedings ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM'95*, p. 226–229, 1995.

[43] R. Douence. *Décrire et comparer les mises en œuvre de langages fonctionnels*. Doctorat de l'université de Rennes I, 1996.

[44] M. Draghicescu et S. Purushothaman. A compositional analysis of evaluation order and its application. In *Proceedings of 1990 Conference on Lisp and Functional Programming*, ACM Press, p. 242–250, 1990.

[45] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, éditeur, *Handbook of Theoretical Computer Science: volume B*, p. 995–1072. Elsevier, 1990.

[46] U. Erlingsson et F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Sept. 1999.

[47] D. Evans et A. Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, p. 32–45, IEEE Computer Society Press, May 1999.

[48] J. Fairbairn et S. Wray. Tim: a simple, lazy abstract machine to execute supercombinators. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, LNCS, vol. 274, p. 34–45, 1987.

[49] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[50] M. Felleisen et D. P. Friedman. A Syntactic Theory of Sequential State. Tech Report, Computer Science Department, Indiana University, Oct., 1987.

[51] M. J. Fischer. Lambda-calculus schemata. In *Proceedings of the ACM Conference on Proving Properties about Programs*, Sigplan Notices, vol. 7(1), p. 104–109, 1972.

[52] K. Fisher, F. Honsell et J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, vol. 1(1), p. 3–37, 1994. (Preliminary version appeared in *Proceedings IEEE Symposium on Logic in Computer Science*, p. 26–38, 1993).

[53] Y. Futamura. Partial computation of programs. In *RIMS Symp. on Soft. Science and Eng.*, LNCS, vol. 147, p. 34–45, 1983.

[54] R. Ghiya et L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings ACM Principles of Programming Languages*, p. 1–15, 1996.

[55] P. Godefroid. Model checking for programming languages using VeriSoft. In *Conference record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, p. 174–186, Paris, 1997.

[56] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM SIGPLAN'91 Symposium on Programming Languages Design and Implementation*,p.165–176, 1991.

[57] B. Goldberg et M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, p. 53–65, 1992.

[58] C. K. Gomard et P. Sestoft. Globalization and live variables. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1991.

[59] L. Gong, M. Mueller, H. Prafullchandra et R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems Proceedings*, 1997, p. 103–112, 1997.

[60] S. Gorlatch, C. Wedler et C. Lengauer. Optimization rules for programming with collective operations. In *Proceedings of Symposium on Parallel and Distributed Processing*, p. 492–499, 1999.

[61] M. Gupta et P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[62] J. Guzmán et P. Hudak. Single-threaded polymorphic lambda-calculus. In *IEEE Symposium on Logic in Computer Science*, June 1990.

[63] C. Hall. Using Hindley-Milner type inference to optimise list representation. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, p. 162–172,1994.

[64] J. Hannan. Staging transformations for abstract machines. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 130–141, New Haven, June 1991.

[65] T. Hardin, L. Maranget et B. Pagano. Functional back-ends within the lambda-sigma calculus. In *Proceedings of International Conference on Functional Programming*, p. 25–33, 1996.

[66] J. Hatcliff et O. Danvy. A generic account of continuation-passing styles. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, p. 458–471, 1994.

[67] N. Heintze et J. G. Riecke. The SLam calculus: Programming with security and integrity. In *Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 365–377, Jan. 1998.

[68] L. J. Hendren, J. Hummel et A. Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs, in *Proceedings ACM Conference on Programming Language Design and Implementation*, p. 249–260, 1992.

[69] C. Herrmann et C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *Journal of Functional Programming*, 9(3):279–310, 1999.

[70] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[71] H. Hosoya et A. Yonezawa. Garbage collection via dynamic type inference – a formal treatment. In *Proceedings of the Second Workshop on Types in Compilation*, LNCS, vol. 1473, 1998.

[72] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of Conference on Lisp and Functional Programming*, ACM Press, p. 351–363, 1986.

[73] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2,…*ω. Thèse de doctorat d'état, Université de Paris VII, 1976.

[74] J. Hughes. *The design and implementation of programming languages.* D. Phil. Thesis, Oxford University, 1983.

[75] K. Inoue, H. Seki et H. Yagi. Analysis of functional programs to detect runtime garbage cells. *ACM Trans. on Programming Languages and Systems*, 10(4), p. 555–578,1988.

[76] B. Jay et P. Steckler. The functional imperative: shape! In *European Symposium on Programming*, LNCS, vol. 1381, p. 139–153, 1998.

[77] B. Jay. Costing parallel programs as a function of shapes. *Science of Computer Programming*, 37(1):207–224, 2000.

[78] J. Jensen, M. Jørgensen, M. Schwartzbach et N. Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, vol. 32, 5*, p. 226–234, 1997.

[79] T. Jensen, D. Le Métayer et T. Thorn. Verification of control flow based security policies. In *Proceedings of Symposium on Research in Security and Privacy*, p. 89–103, May 1999.

[80] T. Johnsson. *Compiling Lazy Functional Languages.* PhD Thesis, Chalmers University, 1987.

[81] N. D. Jones. (éditeur). *Semantics-Directed Compiler Generation*. LNCS, vol. 94, 1980.

[82] N. Jones et S. Muchnick, *Flow analysis and optimization of Lisp-like structures*, in Program Flow Analysis: Theory and Applications, New Jersey 1981, Prentice Hall, p. 102–131.

[83] R. Jones. Tail recursion without space leaks. *Journal of Functional Progr.*, 2 (1), p. 73–79, Jan. 1992.

[84] S. B. Jones et D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p.54–74, ACM Press, 1989.

[85] M. S. Joy, V. J. Rayward-Smith et F. W. Burton. Efficient combinator code. *Computer Languages*, 10(3), 1985.

[86] U. Kastens et M. Schmidt. Lifetime analysis for procedure parameters. In *ESOP 86*, LNCS, vol. 213, p.53–69, 1986.

[87] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier et J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1997.

[88] G. Kiczales *et al*. Aspect-oriented programming. Collection of Tech. Reports SPL-97-007 - 010, Xerox Palo Alto Research Center, 1997.

[89] N. Klarlund et M. Schwartzbach. Graph types. In *Proceedings of ACM Principles of Programming Languages*, p. 196–205, 1993.

[90] N. Klarlund et M. Schwartzbach. Graphs and decidable transductions based on edge constraints. in *Proceedings Trees in Algebra and Programming - CAAP'94*, Springer Verlag, LNCS, vol. 787, p. 187–201, 1994.

[91] N. Klarlund et M. Schwartzbach. A domain-specific language for regular sets of strings and trees. In *Proceedings of Domain Specific Languages*, p. 145–156, 1997.

[92] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*. vol. 2, p. 2–108, Oxford University Press, 1992.

[93] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin et N. Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of 1986 ACM SIGPLAN Symposium on Compiler Construction*, 219–233, 1986.

[94] W. Landi et B. Ryder. Pointer induced aliasing, a problem classification, In *Proceedings ACM Principles of Programming Languages*, p. 93–103, 1991.

[95] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4), p. 308–320, 1964.

[96] X. Leroy. The Zinc experiment: an economical implementation of the ML language. Inria Technical Report 117, 1990.

[97] X. Leroy. Unboxed objects and polymorphic typing. In *ACM Symposium on Principles of Programming Languages*, p. 177–188, 1992.

[98] X. Leroy et F. Rouaix. Security properties of typed applets. In *Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 391–403, 1998.

[99] D. Lester. *Combinator Graph Reduction: a congruence and its application*. PhD thesis, Oxford University, 1989.

[100] S. Liang, P. Hudak et M. P. Jones. Monad transformers and modular interpreters. In *ACM Symposium on Principles of Programming Languages 1995*, p. 333–343, 1995.

[101] R. D. Lins. Categorical multi-combinators. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, LNCS, vol. 274, p. 60–79, 1987.

[102] R. Lins, S. Thompson et S. L. Peyton Jones. On the equivalence between CMC and TIM. *Journal of Functional Programming*, 4(1), p. 47–63, 1992.

[103] J. Mallet. *Compilation d'un langage spécialisé pour machine massivement parallèle*. Doctorat de l'université de Rennes I, 1998.

[104] E. Meijer et R. Paterson. Down with lambda lifting. *Unpublished paper*. copies available at: erik@cs.kun.nl, 1991.

[105] G. Michaelson, N. Scaife, P. Bristow et P. King. Engineering a parallel compiler for SML. In *Proceedings of the International Workshop on Implementation of Functional Languages*, p. 213–226, 1998.

[106] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science 17*, p. 348–375, 1978.

[107] R. Milner. *A Calculus on Communicating Systems*, LNCS, vol. 92, Springer-Verlag, 1980.

[108] R. Milner. *Communicating and Mobile Systems: the π-Calculus*, Cambridge University Press, May 1999.

[109] P. Mishra et U. S. Reddy. Declaration-free type checking. In *Proceedings of the ACM Conference on Principles of Programming Languages*, p. 7–21, 1985.

[110] J. C. Mitchell. Type inference with simple sub-types. *Journal of Functional Programming*, 1(3):245–286, 1991.

[111] M. Mizuno et D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[112] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[113] G. Morrisett, M. Felleisen et R. Harper. Abstract models of memory management. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p. 66–76, 1995.

[114] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper et P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of Conference on Programming Language Design and Implementation*, p. 181–192, May 1996.

[115] G. Morrisett, D. Walker, K. Crary et N. Glew. From System F to Typed Assembly Language. In *Conference record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 85–97, Jan.1998.

[116] G. Nadathur et D. Miller. An overview of λProlog. In *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, p. 810–827, 1988.

[117] G. C. Necula. Proof-carrying code. In *Conference record of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 106–119, Jan. 1997.

[118] T. Nitsche. Shapeliness analysis of functional programs with algebraic data types. *Science of Computer Programming*, 37(1):225–252, 2000.

[119] J. Palsberg et P. Ørbæk. Trust in the lambda calculus. In *Proceedings of the 1995 Static Analysis Symposium*, LNCS, vol. 983, p. 314–330, 1995.

[120] M. Périn. *Spécifications graphiques multi-vues : formalisation et vérification de cohérence*. Doctorat de l'université de Rennes I, octobre 2000.

[121] S. L. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall, 1987.

[122] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[123] S. L. Peyton Jones et D. Lester. *Implementing functional languages, a tutorial*. Prentice Hall, 1992.

[124] S. L. Peyton Jones et J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, LNCS, vol. 523, p. 636–666, 1991.

[125] S. L. Peyton Jones, W. Partain et A. Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of International Conference on Functional Programming*, p. 1–12, 1996.

[126] R. Plasmeijer et M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[127] G. D. Plotkin: Call-by-name, call-by-value and the lambda-calculus. In *Theoretical Computer Science*, 1(2), p. 125–159, 1975.

[128] W. H. Press, S. A. Teukolsky, W. T. Vetterling et B. P. Flannery. *Numerical Recipes in FORTRAN. The Art of Scientific Computing*. Cambridge University Press, 1986.

[129] F. Prost. ML type inference for dead code analysis. Technical Report 97-09, ENS Lyon, 1997.

[130] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, vol. 33–6, p. 668–676, June 1990.

[131] W. Pugh. Counting solutions to presburger formulas: How and why. In *Proceedings of the Conference on Programming Language Design and Implementation*, p. 121–134, 1994.

[132] R. Rangaswami. *A Cost Analysis for a Higher-Order Parallel Programming Model*. PhD thesis, Edinburgh University, 1996.

[133] J.-C. Raoult et F. Voisin. Set-theoretic graph rewriting, In *Proceedings of International Workshop on Graph Transformations in Computer Science*, Springer Verlag, LNCS, vol. 776, p. 312–325, 1993.

[134] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *Proceedings of Information Processing 83*, p. 513–523, 1983.

[135] J. Rushby. Kernels for safety? In *Safe and Secure Computer Systems*, p. 310–320. Blackwell Scientific, 1987.

[136] M. Sagiv, T. Reps et R. Wilhelm. Solving shape-analysis problems in languages with destructive updating, In *Proceedings of ACM Principles of Programming Languages*, p. 16–31, 1996.

[137] A. Sastry, W. Clinger et Z. Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p. 266–275, 1993.

[138] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, vol. 7, 1985, p. 299–310.

[139] D. A. Schmidt. *Denotational Semantics, a Methodology for Language Development*. W.C. Brown Publishers, 1986.

[140] F. B. Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, 1998. (Revised version July 1999).

[141] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[142] P. Sestoft. Replacing function parameters by global variables. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p.39–53, 1989. (see also Tech. Report 88-7-2, University of Copenhagen, 1988.)

[143] Z. Shao et A. Appel. Space-efficient closure representations. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, p. 150–161,1994.

[144] G. L. Steele. Rabbit: a compiler for scheme. AI-TR-474, MIT, Cambridge, Mass., 1978.

[145] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass.,1977.

[146] M. Südholt. *The Transformational Derivation of Parallel Programs using Data-Distribution Algebras and Skeletons*. PhD thesis, Technische Universität Berlin, 1997.

[147] N. Tawbi. Estimation of nested loops execution time by integer arithmetic in convex polyhedra. In *Proceedings of International Symposium on Parallel Processing*, p. 217–223, 1994.

[148] T. Thorn. *Vérification de politiques de sécurité par analyse de programmes*. Doctorat de l'université de Rennes I, février 1999.

[149] A. Tolmach. Tag-free Garbage Collection Using Explicit Type Parameters, In *Proceedings of Conference on Lisp and Functional Programming, published as 1994 LISP Pointers*, 7(3):1-11, July-Sep. 1994.

[150] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience.*, 9, p. 31–49, 1979.

[151] D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44, p. 267–270, 1979.

[152] D. Volpano et G. Smith. A type-based approach to program security. In *Proceedings 7th TAPSOFT*, LNCS, vol. 1214, p. 607–621, 1997.

[153] P. Wadler. Listlessness is better than laziness. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, p. 45–52, 1984.

[154] P. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9), p. 595–608, 1987.

[155] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, LNCS, vol. 300, p. 344–358, 1988.

[156] P. Wadler. Theorems for free! In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, p. 347–359, 1989.

[157] P. Wadler. Linear types can change the world! In *IFIP Working Conference on Programming Concepts and Methods*, North Holland, 1990.

[158] P. Wadler. Comprehending monads. In *Proceedings of 1990 Conference on Lisp and Functional Programming, ACM Press*, p. 61–78, 1990.

[159] P. Wadler. The essence of functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages, ACM Press*, p. 1–14, 1992.

[160] D. S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Faculty of Princeton University, January 1999.

[161] D. S. Wallach et E. W. Felten. Understanding Java stack inspection. In *IEEE Symposium on Security and Privacy*, May 1998.

[162] D. Wilde. A library for polyhedral operations. *Publication Interne* 785, Irisa, 1993.

[163] D. Wilde. The ALPHA language. *Technical Report* 2295, Inria, 1994.

[164] G. Wilson. Assessing the usability of parallel programming systems: the Cowichan problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, 1994.

[165] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of Workshop on Memory Management*, LNCS, vol. 637, p. 1–42, 1992.

Liste des publications

THÈSE

[166] P. Fradet. *Compilation des langages fonctionnels par transformation de programmes.* Thèse de l'université de Rennes I, nov. 1988.

REVUES INTERNATIONALES AVEC COMITÉ DE LECTURE

[167] P. Fradet et D. Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*), 13(1), p. 21–51, 1991.

[168] P. Fradet et D. Le Métayer. Structured Gamma, *Science of Computer Programming* (*SCP*), 31(2–3), p. 263–289, 1998.

[169] R. Douence et P. Fradet. A systematic study of functional language implementations. *ACM Transactions on Programming Languages and Systems* (*TOPLAS*), 20(2), p. 344–387, 1998.

[170] P. Fradet et J. Mallet. Compilation of a Specialized Functional Language for Massively Parallel Computers, *Journal of Functional Programming*, (*JFP*), 2000.

CONFÉRENCES INTERNATIONALES AVEC COMITÉ DE LECTURE

[171] P. Fradet et D. Le Métayer. Compilation of λ-calculus into functional machine code. In *Proceedings **TAPSOFT'89***, LNCS, vol. 352, p. 155–166, 1989.

[172] P. Fradet. Syntactic detection of single-threading using continuations. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, ***FPCA'91***, LNCS, vol. 523, p. 241–258, Cambridge, MA, USA, août 1991.

[173] P. Fradet. Compilation of head and strong reduction. In *Proc of the 5th European Symposium on Programming*, ***ESOP'94***, LNCS, vol. 788, p. 211–224, Edinburgh, UK, avril 1994.

[174] P. Fradet. Collecting more garbage. In *Proceedings of ACM Conference on Lisp and Functional Programming*, ***LISP'94***, p. 24–33, Orlando, FL, USA, juin 1994.

[175] R. Douence et P. Fradet. Towards a taxonomy of functional language implementations. In *Proceedings of 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, ***PLILP'95***, LNCS, vol. 982, p. 34–45, Utrecht, the Netherlands, 1995.

[176] P. Fradet, R. Gaugne et D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *Proceedings European Symposium on Programming*, ***ESOP'96***, LNCS, vol. 1058, p. 125–140, Linkoping, Sweden, April 1996.

[177] P. Fradet et D Le Métayer. Shape types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, ***POPL'97***, p. 27–39, Paris, France, jan. 1997.

[178] P. Fradet, D. Le Métayer et M. Périn. Consistency checking for multiple view software architectures, In *Proceedings of the Joint European Software Engineering Conference and Symposium on Foundations of Software Engineering*, ***ESEC/FSE'99***, Software Engineering Notes 24 (6) ou LNCS vol. 1687 p. 410–428, 1999.

[179] T. Colcombet et P. Fradet. Enforcing trace properties by program transformation, In *Proceedings of the ACM Symposium on Principles of Programming Languages*, ***POPL'00***, ACM Press, p. 54–66, Boston, MA, jan. 2000.

[180] P. Fradet, V. Issarny et S. Rouvrais. Analyzing non-functional properties of mobile agents. In *Proceedings of Fundamental Approaches to Software Engineering*, ***FASE'00***, LNCS, mars 2000.

SÉMINAIRES INTERNATIONAUX AVEC COMITÉ DE LECTURE

[181] P. Fradet et D. Le Métayer. Type checking for a multiset rewriting language. In *Proceedings the LOMAPS workshop on Analysis and Verification of Multiple-agent Languages*, LNCS, vol. 1192, p. 126–140, 1996.

[182] P. Fradet et M. Südholt. Towards a generic framework for aspect-oriented programming, *Third Aspect Oriented Programming Workshop, Object-Oriented Technology. ECOOP'98 Workshop Reader*, LNCS, vol. 1543, p. 394–397, 1998.

[183] P. Fradet et M. Südholt. An aspect language for robust programming. In *International Workshop on Aspect-Oriented Programming*, ***AOP'99***, ECOOP, juin 1999.

CONFÉRENCES FRANÇAISES AVEC COMITÉ DE LECTURE

[184] P. Fradet et D. Le Métayer. Compilation de langages fonctionnels par transformation de programmes. In *Journées francophones des langages applicatifs*, ***JFLA'90***, Bigre No 69, p. 123–133, La Rochelle, jan. 1990.

[185] R. Douence et P. Fradet. Décrire et comparer les implantations de langages fonctionnels. In *Journées francophones des langages applicatifs*, ***JFLA'96***, Collection Inria Didactique, p. 183–203, Val-Morin, Québec, jan. 1996.

ARTICLE DANS UNE MONOGRAPHIE

[186] P. Fradet et D. Le Métayer. From lambda-calculus to machine code by program transformation. *Prospects for Functional Programming in Software Eng.*, Eds J.-P. Banâtre, S. B. Jones, D. Le Métayer. Springer Verlag, 1991.

RAPPORTS DE RECHERCHE

[187] P. Fradet et D. Le Métayer. Compilation of functional languages by program transformation. Inria research report 1040, mai 1989.

[188] P. Fradet. Compilation of head and strong reduction. (version étendue). Inria research report 2132, 1993.

[189] R. Douence et P. Fradet. A taxonomy of functional language implementations. Part I: call-by-value. Inria research report 2783, jan. 1996.

[190] R. Douence et P. Fradet. A taxonomy of functional language implementations. Part II: call-by-name, call-by-need, and graph reduction. Inria research report 3050, nov. 1996.

[191] P. Fradet, R. Gaugne et D. Le Métayer. An inference algorithm for the static verification of pointer manipulation. Technical Report 980, IRISA, 1996.

[192] P. Fradet et D. Le Métayer. Structured Gamma, Irisa Research Report PI-989, mars 1996.

[193] P. Fradet et J. Mallet. Compilation of a specialized functional language for massively parallel computers, Inria Tech. Report n° 3894, mars 2000.

[194] P. Fradet, D. Le Métayer et M. Périn. Consistency checking for multiple view software architectures (version étendue). Irisa research report 1249, 2000 (à paraitre).

DIVERS

[195] E. Denney, P. Fradet, C. Goire, T. Jensen et D. Le Métayer. Procédé de vérification de transformateurs de codes pour un système embarqué, notamment sur une carte à puce. juillet 1999, Brevet d'invention.