Un système pour l'optimisation globale de programmes d'ordre supérieur par compilation abstraite séparée

Dominique Boucher
Marc Feeley

Rapport technique no. 992
Université de Montréal

Septembre 1995

Résumé

Le présent document a été rédigé dans le cadre de la deuxième partie de l'examen prédoctoral de Dominique Boucher sous la direction du professeur Marc Feeley. La proposition consiste en l'architecture d'un compilateur pour un langage fonctionnel d'ordre supérieur capable de réaliser des optimisations intermodules. Le système utilise une technique d'implantation des analyses statiques baptisée «compilation abstraite».

Table des matières

1 Motivations

La programmation *modulaire* est essentielle à l'écriture de systèmes informatiques de grande taille et ce, pour plusieurs raisons. D'abord, elle facilite la décomposition de ces systèmes en plus petites unités logiques correspondant à des sous-systèmes plus ou moins indépendants qui peuvent être développés et entretenus séparément et qui reflètent généralement la structure du problème à résoudre. Ces unités logiques, les modules, fournissent idéalement une interface qui permet de faire abstraction de la réalisation du module, permettant ainsi d'encapsuler des décisions de conception et de cacher les détails d'implantation. De plus, les modules peuvent être écrits de manière à être réutilisés dans d'autres systèmes.

Enfin, ces modules forment des unités distinctes pour la compilation, de sorte qu'une modification dans un module n'entraîne pas la recompilation du système complet (à condition que l'interface n'ait pas été modifié). Du point de vue du compilateur, il est généralement plus rapide de compiler (et optimiser) 100 modules de 1000 lignes chacun qu'un système complet comportant 100,000 lignes. En effet, la plupart des optimisations effectuées par le compilateur (comme par exemple les analyses statiques interprocédurales) sont typiquement implantées à l'aide d'algorithmes quadratiques ou cubiques, prohibant ainsi leur utilisation sur de trop gros programmes. Il faut alors traiter chaque module séparément, d'où le besoin de compilateurs supportant la *compilation séparée*. Dans le reste du texte, nous considérerons un cycle de développement divisé en deux phases. Dans un premier temps, tous les modules sont compilés séparément et une édition de liens crée l'exécutable. La deuxième phase est constitué d'un certain nombre d'itérations. Chaque itération consiste en des modifications à un module suivies de la recompilation du module et de l'édition de liens.

Évidemment, les avantages de la programmation modulaire ne vont pas sans créer des difficultés pour la compilation. D'abord, les possibilités d'optimisations à l'intérieur d'un module sont limitées par les informations recueillies dans ce module. On ne peut pas bénéficier pleinement des analyses effectuées dans les autres modules.

De plus, considérons un module M écrit et compilé lors du développement d'un programme P_1 et que l'on décide d'utiliser dans un programme P_2. Il est fort possible que les caractéristiques du programme P_2 auraient permis de compiler plus efficacement le module M. Par contre, recompiler en entier le module M peut s'avérer un gaspillage de ressources, surtout s'il est utilisé dans de nombreuses applications différentes.

1.1 Trois exemples

Nous allons illustrer les opportunités d'optimisation ratées par la compilation séparée à l'aide de trois programmes multi-modules écrits respectivement en C, Scheme et Dylan. Chacun des programmes a été compilé de deux façons : en compilant les modules séparément et en compilant tous les modules d'un seul coup. Dans chaque cas, la compilation séparée donne un programme bien moins performant que par une compilation globale. Les tests ont tous été effectués sur une machine DEC 3000 de 64 Mégaoctets de mémoire vive.

```
/* Le module main.c */
main ()
{
  long x1, x2, x3, x4, i ;

  x1 = x2 = x3 = x4 = 0 ;
  for (i = 1000000; i > 0 ; i--)
    proj9 (x1,
           proj4 (x1, x2, x3, x4),
           proj1 (x4),
           proj4 (x1, x2, x3, x4),
           proj1 (x4),
           proj4 (x1, x2, x3, x4),
           proj1 (x4),
           proj4 (x1, x2, x3, x4),
           proj1 (x4)) ;
}
```

```
/* Le module projections.c */
long proj1 (long x1)
{ return x1 ; }

long proj4 (x1, x2, x3, x4)
long x1, x2, x3, x4;
{ return x1 ; }

long proj9 (x1, x2, x3, x4, x5, x6, x7, x8, x9)
long x1, x2, x3, x4, x5, x6, x7, x8, x9;
{ return x1 ; }
```

FIG. 1 - *Le programme en C.*

Le programme C. Le programme C (figure 1) que nous avons testé est celui qui a donné les résultats les plus mauvais pour la compilation séparée. Ce programme, contrairement aux deux autres, ne fait pas de travail utile. Il fait néanmoins ressortir clairement les lacunes au niveau des optimisations engendrées par la compilation séparée.

Le programme est divisé en deux modules : `projections.c` qui contient des fonctions de projection et `main.c` qui contient la routine principale. Le programme effectue simplement un certain nombre de projections qu'il répète 1,000,000 de fois. Le programme a été compilé de deux manières différentes, d'abord en appelant `cc` sur chacun des modules et en faisant une édition de liens, puis en appelant `cc` sur les deux modules en même temps. Dans les deux cas, la compilation a été effectuée avec l'option `-O3`, permettant ainsi au compilateur de faire des optimisations globales au module et de l'intégration de fonctions. Voici les temps d'exécution du programme :

	Comp. séparée	Comp. globale
Temps d'exécution	0.540 sec.	0.017 sec.

On voit donc que la compilation globale permet d'obtenir un programme près de 31 fois plus rapide. Ceci s'explique par le fait que le compilateur a pu faire l'intégration des fonctions de projection, éliminant ainsi tous les appels de fonctions. En fait, le programme résultant n'est rien d'autre qu'une boucle de 1,000,000 à 0.

Le programme Scheme. Le deuxième exemple (figure 2) est un programme Scheme [9] qui ajoute 1 à tous les nombres premiers compris entre 1 et 32,000. Le module principal, `main`, se sert de quatre autres modules : `list` contient des fonctions d'ordre supérieur sur

```scheme
;;; Le module main.scm
(define (main argl)
  (let ((l (consec 1 32000)))
    (let ((l2 (filter prime? l)))
      (map1 (lambda (x) (my+  x 1)) l2))))
```

```scheme
;;; Le module list.scm
(define (filter p l)
  (if (null? l)
      '()
      (let ((x (car l)))
        (if (p x)
            (cons x (filter p (cdr l)))
            (filter p (cdr l))))))

(define (map1 f l)
  (if (null? l)
      '()
      (cons (f (car l)) (map1 f (cdr l)))))

(define (consec n m)
  (if (> n m)
      '()
      (cons n (consec (my+ n 1) m))))
```

```scheme
;;; Le module arith.scm
;;; opérateurs arithmétiques de taille fixe.
(define (my+ x y)  (+fx x y))
(define (my> x y)  (>fx x y))
(define (my<= x y) (<=fx x y))
(define (my= x y)  (=fx x y))
```

```scheme
;;; Le module number.scm
(define (prime? n)
  (let ((i 2))
    (while (lambda () (and (my<= (my* i i) n) (not (my= (modulo n i) 0))))
           (lambda () (set! i (my+ i 1)))
           (lambda () (if (my> (my* i i) n) #t #f)))))
```

```scheme
;;; Le module iter.scm
(define (while test body finally)
  (let loop ((t (test)))
    (if t (begin (body) (loop (test))) (finally))))
```

FIG. 2 - *Le programme Scheme.*

les listes, `number` contient des fonctions numériques diverses, `arith` redéfinit les opérateurs arithmétiques et `iter` contient une fonction d'ordre supérieur d'itération.

Si on compile chacun de ces modules séparément, il est clair que les appels `(p x)` dans la fonction `filter` et `(f (car l))` dans la fonction `map1` devront être des appels calculés[1], puisqu'on ne peut savoir, à la compilation, que `p` sera lié à `prime?` et `f` à `(lambda (x) (my+ x 1))`.

De plus, on constate que les fonctions `map1` et `filter` retournent toutes deux des listes et que le deuxième argument à `filter` dans la fonction `main` est une liste. Sachant ces informations à la compilation, il serait possible d'éliminer plusieurs tests de types dans `map1` et `filter`. Aussi, une analyse globale permettrait de détecter que l'argument de `prime?` est toujours un entier, le compilateur pouvant ainsi optimiser les appels à `sqrt` et `modulo` en

1. Un appel calculé est un appel pour lequel on ne connaît pas, à la compilation, la fonction précise qui sera appelée. Le mécanisme général (et plus coûteux) d'application d'une fermeture doit alors être utilisé.

éliminant les tests de types. Malheureusement, la compilation séparée ne permet pas de faire de telles optimisations intermodules.

Il est donc clair que la compilation séparée de Scheme fait passer les compilateurs à côté d'un grand nombre d'optimisations potentielles. Des annotations de types à l'intérieur de chacun des modules permettraient probablement de faire quelques unes de ces optimisations, mais certainement pas celles liées à l'allocation des fermetures. De plus, le compilateur obéirait aveuglément au programmeur, ne pouvant s'assurer de l'exactitude des annotations.

Pour donner une meilleure idée de l'avantage d'une compilation globale, nous avons procédé comme pour le programme en C. Nous avons d'abord compilé le programme module par module et ensuite il a été compilé en mettant tous les modules dans un seul fichier. Le compilateur `bigloo` [32], qui produit du C, a été utilisé avec l'option d'optimisation `-O3`. Les résultats suivant donnent les temps d'exécution du programme, montrant bien les pertes de performance (un facteur de 2) causées par la compilation séparée :

	Comp. séparée	Comp. globale
Temps d'exécution	3.58 sec.	1.85 sec.

Le programme Dylan. Le dernier exemple (figure 3) est un fragment d'un hypothétique compilateur écrit en Dylan [33], un langage à objets avec héritage multiple à base de fonctions génériques et dont le système de types est dynamique. Le programme comporte deux modules : `collections` qui implante la librairie standard des collections et `compiler` qui implante le compilateur en question.

Si on effectue une compilation séparée des deux modules, les appels à `element` et à `element-setter` dans le module `compiler` passeront obligatoirement par la fonction d'aiguillage *(dispatcher)* [2]. Et ceci pour deux raisons. D'abord, la clause du module `collections` qui exporte ces deux fonctions génériques n'indique pas quelles sont les signatures des différentes méthodes qui leur sont attachées. D'autre part, le module `compiler` pourrait importer un autre module qui associerait d'autres méthodes à ces fonctions génériques.

Cette situation est malheureuse. La table de symboles, une table de hachage, est une structure à laquelle le compilateur fait abondamment référence. Si chaque accès à la table doit passer par le dispatcher, la vitesse du compilateur en sera grandement affectée. Considérant qu'un appel à la fonction d'aiguillage peut parfois prendre plusieurs centaines d'instructions machine [3] (en comparaison aux quelques instructions nécessaires à un appel calculé d'une fermeture Scheme), il serait nettement préférable de connaître à la compilation la méthode qui sera calculée par la fonction d'aiguillage.

Nous n'avons évidemment pas de temps d'exécution pour ce programme. D'une part, ce n'est qu'un fragment de programme. D'autre part, il n'existe pas à l'heure actuelle de

2. La fonction d'aiguillage est le mécanisme qui détermine, pour un site d'appel (ou envoi de message) donné, la méthode la plus appropriée parmi l'ensemble des méthodes applicables.

3. Il existe évidemment de nombreuses techniques pour réduire le coût de la fonction d'aiguillage : utilisation de caches [22], de tables d'indexation [2], etc. Mais ces techniques sont quand même plus coûteuses qu'un appel de fonction normal, elles impliquent dans certains cas une utilisation abusive de la mémoire et elles sont mal adaptées aux langages à héritage multiple à base de multi-méthodes.

```
// Le module collections.dyl
define module collections
  create <hash-table>, <list>, element, element-setter ;
end ;

define generic element (coll :: <collection>, key :: <object>) ;

define generic element-setter (v :: <object>, c :: <collection>, k :: <object>) ;

define method element (hasht :: <hash-table>, key :: <symbol>)
  // code pour trouver un symbole dans une table.
end element ;

define method element-setter (v :: <object>, hasht :: <hash-table>, key :: <symbol>)
  // code pour changer l'attribut d'un symbole dans une table.
end element-setter ;

define method element (l :: <list>, k :: <integer>)
  // code pour trouver le k-ième élément d'une liste.
end element ;

define method element-setter (v :: <object>, l :: <list>, k :: <integer>)
  // code pour modifier le k-ième élément d'une liste.
end element-setter ;
```

```
// Le module compiler.dyl
define module compiler
  use collections ;
end ;

define constant $hash-table :: <hash-table> = make (<hash-table>, size: 1023) ;

define method most-recent-context (sym :: <symbol>)
  head (element ($hash-table, sym)) ;
end ;

define method push-new-context (env :: <context>, sym :: <symbol>)
  element-setter (pair (env, element ($hash-table, sym)), $hash-table, sym) ;
end ;
```

FIG. 3 - *Le fragment de programme Dylan.*

compilateur optimisant pour Dylan, ce dernier étant un langage qui a été développé très récemment. Il n'en reste pas moins que c'est un langage beaucoup plus difficile à compiler efficacement que Scheme. En effet, Dylan intègre plusieurs concepts rendant sa compilation difficile. Il y a d'abord les multi-méthodes, c'est-à-dire des méthodes pour lesquelles l'aiguillage se fait sur plus d'un argument. Il y a aussi l'héritage multiple, les mots-clés comme objets de première classe et le système de types dynamiques. Tous ces mécanismes sont très coûteux s'ils ne sont pas adéquatement optimisés.

1.2 Un constat et la suite

Il est clair, à la lumière des trois exemples précédents, que la compilation séparée peut augmenter considérablement le temps d'exécution de certains programmes. Ceci est d'autant plus vrai pour les langages de programmation de haut-niveau (malgré ce que peuvent peut-être suggérer les exemples en C et en Scheme que nous avons donnés). Ces langages possèdent des concepts difficiles à implanter et, par conséquent, à optimiser. Pensons tout simplement aux fermetures des langages fonctionnels d'ordre supérieur, aux systèmes de types dynamiques, aux fonctions génériques des langages à objets, pour n'en nommer que quelques-uns.

Nous présenterons, dans les prochaines pages, une nouvelle approche à la compilation séparée qui permettra des optimisations entre modules. L'approche que nous proposons origine de l'étude des analyses de flot de contrôle des langages fonctionnels d'ordre supérieur basées sur les techniques d'analyse statique par interprétation abstraite. Notre recherche portera donc principalement sur l'analyse et l'optimisation intermodules de langages fonctionnels d'ordre supérieur. Malgré tout, l'idée de base est aussi bien adaptée aux langages impératifs qu'à objets ou à prototypes.

La suite de ce document est divisée en six parties. D'abord, à la deuxième section, nous présentons les techniques conventionnelles d'analyse de flot de contrôle. Nous y introduisons aussi une nouvelle technique d'implantation que nous avons développée et qui est à la base de l'approche que nous proposons pour la compilation séparée. À la troisième section, nous présentons une nouvelle organisation d'un système pour la compilation séparée et l'optimisation globale. La section suivante décrit de manière plus détaillée le compilateur proposé, les conséquences pratiques et les problèmes auxquels nous devrons faire face. La cinquième section résume brièvement les objectifs pratiques de notre recherche. Nous décrivons, à la sixième section, un certain nombre de travaux connexes au nôtre.

2 Analyse de flot de contrôle et compilation abstraite

Voyons maintenant quelques approches à l'analyse de flot de contrôle afin de bien mettre en évidence les difficultés inhérentes à l'analyse des langages d'ordre supérieur. Une attention particulière sera portée à la technique d'analyse par interprétation abstraite. Nous terminerons cette section par une description sommaire d'une technique d'implantation que nous avons développée et que nous désignons par le terme *compilation abstraite*.

2.1 L'analyse de flot de contrôle

Les optimisations qu'un compilateur peut effectuer sont principalement de deux types (deux bonnes références, à cet effet, sont [1] et [16]). Il y a d'abord les optimisations dites *locales* effectuées uniquement sur de très petites portions du programme à la fois. Parmi celles-ci, on retrouve l'élimination de sous-expressions communes, l'élimination de code mort, les optimisations «peephole» et bien d'autres. À l'autre extrême, il y a les optimisations dites *globales* qui doivent tenir compte des fils d'exécution possibles du programme. Ces optimisations comprennent la propagation de copies, la détection et le déplacement des calculs invariants d'une boucle, l'élimination de variables d'induction, pour n'en nommer que quelques unes.

Ces dernières optimisations requièrent des techniques d'analyse de programmes sophistiquées, on le comprendra aisément. Dans le cas des langages impératifs comme C, Pascal et Fortran, ces techniques d'analyse sont bien connues et maîtrisées depuis de nombreuses années déjà [1, 16, 19].

La technique la plus répandue consiste à découper le programme (ou la procédure) en *blocs de base*. Un bloc de base est constitué d'une suite d'opérations élémentaires [4] à effectuer séquentiellement et dont seule la dernière de ces opérations peut être un saut vers un autre bloc. L'ensemble des blocs d'un programme est organisé en un graphe, le graphe de flot de contrôle. Dans ce graphe, il existe un arc d'un bloc A vers un bloc B si et seulement si la dernière opération du bloc A est un saut conditionnel ou inconditionnel à la première opération du bloc B. La figure 4 donne le graphe de flot de contrôle d'un bout de programme donné à la figure 5.

À partir de ce graphe, il est possible de calculer des informations qui serviront aux différentes optimisations. Par exemple, on peut vouloir déterminer les variables «vivantes» à chaque point du programme, une variable étant vivante à un point A si sa valeur courante est utilisée par un autre point du programme B et qu'il existe un chemin du point A au point B dans le graphe.

Le calcul de ces informations s'effectue généralement par approximations successives d'un point fixe [5]. On calcule d'abord les informations locales à chaque bloc de base et ensuite on propage ces informations le long des arcs du graphe. On répète ce processus jusqu'à ce qu'on

4. Le terme *élémentaire* est intentionnellement vague. La définition d'une opération élémentaire dépend fortement du langage source et de la représentation intermédiaire utilisée par le compilateur.

5. D'autres techniques plus sophistiquées sont aussi possibles, mais elles requièrent souvent des propriétés particulières du graphe de flot de contrôle.

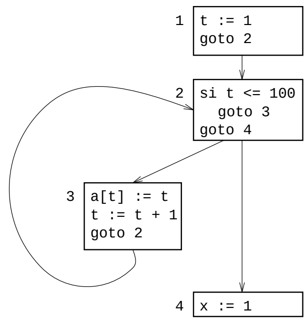

FIG. 4 - *Un graphe de flot de contrôle*

```
for (t = 1; t <= 100; t++)
  a[t] := t ;
x := 1 ;
```

FIG. 5 - *Une boucle en C*

ne puisse plus dériver d'informations nouvelles. La combinaison des informations provenant des prédécesseurs (ou successeurs) d'un bloc se fait à l'aide d'un opérateur dit de *confluence*. L'opérateur utilisé dépend du type d'analyse. Fischer et LeBlanc [16] donnent à ce sujet une taxonomie des différentes analyses possibles ainsi que les équations et opérateurs qui s'y rattachent.

Dans le cas des langages d'ordre supérieur, l'analyse est beaucoup plus difficile à effectuer. Même si les graphes de flot de contrôle sont souvent beaucoup plus simples (les boucles étant habituellement exprimées à l'aide de fonctions récursives), des analyses interprocédurales deviennent nécessaires pour calculer les informations qui permettront d'optimiser le programme. Cela tient au fait qu'il n'est pas simple de déterminer statiquement (c'est-à-dire à la compilation) quelle fonction sera appelée à un site d'appel `f(x)`, surtout si la variable `f` peut être mutée et que de nouvelles fonctions peuvent être créées à l'exécution. La détermination du graphe de flot de contrôle devient donc un problème de flot de données, qui ne se résout, ironiquement, qu'à l'aide du graphe de flot de contrôle lui-même! Nous verrons que les techniques d'analyse par interprétation abstraite, que nous introduirons bientôt, permettent de résoudre ce problème.

On voit donc pourquoi les langages d'ordre supérieur, ceux qui permettent de passer des fonctions en argument, de les retourner comme résultat d'un appel de fonction ou de les emmagasiner dans des structures de données, sont beaucoup plus difficiles à compiler efficacement.

Un des problèmes majeurs de compilation de ces langages réside dans la représentation des fonctions à l'exécution et du coût d'un appel de fonction. Comme les appels calculés (non-optimisés) sont souvent plus coûteux qu'une simple instruction de branchement, il convient d'optimiser ces derniers au maximum. Cela peut se faire au moyen d'une analyse appelée *analyse de représentation des fermetures.*[6] Cette analyse consiste à trouver, pour chaque site d'appel $f(x_1, \ldots, x_n)$, toutes les fermetures qui pourraient être liées à f en cours d'exécution.

Plusieurs compilateurs effectuent cette analyse de manière ad hoc. Le compilateur Scheme ORBIT [23] effectue une analyse intraprocédurale du programme pour déterminer localement si les fermetures doivent être allouées dans le tas, sur la pile, dans les registres ou si elle ne doivent pas être allouées du tout. MIT-Scheme utilise une analyse semblable [30] qui permet même de compiler sous forme itérative des récursions exprimées à l'aide du combinateur de point-fixe Y. D'autres compilateurs, comme par exemple SML/NJ [4], ne font pas l'analyse des fermetures car ils prennent le parti d'allouer tous les objets dans le tas même si leur étendue est dynamique. Cette approche a le désavantage de consommer plus rapidement la mémoire puisque tous les blocs d'activation (entre autres) sont alloués dans le tas. Les glanages de cellules (GC) sont donc plus fréquents, déplaçant ainsi le problème de performance vers le GC.

Depuis quelques années toutefois, on a assisté à l'émergence de travaux qui ont permis de formaliser ces analyses, dans le but avoué de fournir des analyses prouvées sémantiquement correctes. Parmi ceux-ci, on retrouve les travaux de Steckler [37]. Son analyse des fermetures est calculée en étiquettant chaque noeud de l'arbre d'analyse par une série de contraintes et en résolvant ensuite le système de contraintes ainsi obtenu.

2.2 L'analyse par interprétation abstraite

Mais les techniques d'analyse qui semblent avoir le plus d'impact sont celles développées grâce à la technique de l'*interprétation abstraite*. Cette technique, décrite pour la première fois par Cousot et Cousot [11], offre un cadre théorique conceptuellement simple pour le développement d'analyses statiques. L'idée de base est simple. Prenons un programme p écrit dans un langage L dont la sémantique est donnée par $\mathcal{S}$. $\mathcal{S}(\pi, \rho)$ est donc la valeur retournée par le programme π lorsqu'il est exécuté dans un environnement initial ρ. On peut voir cette valeur comme une propriété du programme π. Si on veut calculer une autre propriété du programme π, il suffit de fournir une sémantique non-standard $\mathcal{S}^*$ pour L. La propriété du programme π sera donc donnée par $\mathcal{S}^*(\pi, \rho^*)$, où ρ^* est un environnement «abstrait» calculé à partir de ρ. Évidemment, il faut que la sémantique $\mathcal{S}^*$ soit reliée d'une certaine manière à $\mathcal{S}$, ce que je ne décrirai pas ici.

Plusieurs analyses statiques pour les langages fonctionnels d'ordre supérieur ont été développées dans ce cadre théorique. Mycroft [27] a proposé une analyse de «nécessité» (*strictness analysis*). Cette analyse permet de déterminer les arguments qu'une fonction doit nécessairement évaluer. Hudak [20] utilise l'interprétation abstraite pour une analyse du comptage de référence permettant un GC à la compilation et diverses autres optimisations.

L'analyse de représentation des fermetures a, elle aussi, été formalisée à l'aide de l'interprétation abstraite. Les travaux les plus connus sont ceux de Shivers [34, 35] et de Ayers [5].

6. Une *fermeture* est la représentation à l'exécution d'une fonction. C'est un couple composé du code de la fonction et d'un environnement capturant les liaisons variable/valeur effectives lors de la création de la fermeture.

Shivers décrit une analyse très générale ainsi que deux cas particuliers, la 0CFA et la 1CFA, des analyses de flot de contrôle d'ordre 0 et 1 respectivement. La deuxième est une analyse beaucoup plus fine, mais également d'une complexité accrue. Shivers propose aussi un certain nombre d'optimisations pour le langage Scheme basées sur les résultats de l'analyse 0CFA. De son côté, Ayers propose une analyse (et plusieurs variantes) très proche de la 0CFA mais dont les preuves d'exactitude sont plus simples. Il utilise les *connexions de Galois* pour établir les liens entre $\mathcal{S}$ et $\mathcal{S}^*$. Il propose aussi plusieurs algorithmes pour l'implantation efficace de ses analyses et diverses optimisations.

De son côté, Serrano [31] a montré l'utilité de telles analyses pour la compilation des langages fonctionnels d'ordre supérieur. Il a implanté la 0CFA de Shivers ainsi qu'une analyse de types basée sur la 0CFA dans un compilateur Scheme et ML produisant du C [32]. Ses résultats prouvent la valeur des analyses de fermeture et montre aussi leur coût, qui est souvent élevé. Par contre, son implantation n'effectue pas les optimisations au niveau de l'analyse comme celles que propose Ayers. Parmi ces optimisations, on retrouve la détection des sites d'appel initiaux, c'est-à-dire ceux qui contribuent au résultat de l'analyse uniquement lors la première itération.

2.3 La compilation abstraite, un aboutissement logique

L'attrait pour la technique d'interprétation abstraite ne cesse d'augmenter depuis plusieurs années. Par contre, peu de compilateurs implantent de telles analyses; à notre connaissance, il n'y a que `bigloo`, le compilateur Scheme/ML de Serrano [32]. Mon sentiment est que peu d'efforts ont été consacrés à leur implantation efficace (elles sont d'ailleurs souvent implantées par de simples interprètes). Nous allons donc maintenant décrire une technique d'implantation plus efficace que nous avons développée et qui a inspiré la nouvelle approche pour la compilation séparée que nous présenterons à la prochaine section. Cette nouvelle technique ne permet pas de réduire la complexité des algorithmes mais elle permet de nombreuses optimisations qui réduisent tout de même de façon appréciable la «constante cachée».

Comme je nous l'avons montré, calculer une analyse $\mathcal{A}$ sur un programme π revient à interpréter π en utilisant une sémantique différente de la sémantique standard du langage, cette sémantique non-standard $\mathcal{S}^*$ variant selon l'analyse à effectuer. Or, de l'interprétation à la compilation, il n'y a qu'un pas à franchir. Nous avons montré qu'il est en effet possible de compiler l'analyse statique [6], processus que nous nommons *compilation abstraite*. Le principe est de générer, à partir d'un programme π, un programme π' qui effectue, lorsqu'il est exécuté, l'analyse statique du programme π.

Plus formellement, supposons qu'on soit intéressé à calculer une analyse statique $\mathcal{S}^*$. Comme on l'a vu, $\mathcal{S}^*$ peut être vue comme une fonction de deux arguments : le premier est π, le programme à analyser, et l'autre est ρ^*, un environnement abstrait initial qui dépend de l'analyse. Le résultat de l'analyse, $\mathcal{S}^*(\pi, \rho^*)$, est un environnement abstrait qui contient l'information désirée. La compilation abstraite de π, $\mathcal{C}^*(\pi)$, est donc une fonction d'un argument telle que :

$$\mathcal{C}^*(\pi)(\rho^*) = \mathcal{S}^*(\pi, \rho^*).$$

Nous appellerons *programme d'analyse de* π le programme $\mathcal{C}^*(\pi)$.

Il est aussi possible d'exprimer $\mathcal{C}^*$ à l'aide des équations de l'évaluation partielle [10] (ce que l'on nomme les projections de Futamura). Soit p un programme prenant en entrée une donnée d que l'on peut décomposer en deux parties : une partie connue d_1 et et une partie inconnue d_2. Un évaluateur partiel *Mix* est un programme prenant en entrée le programme p et la donnée d_1 et qui produit en sortie un programme *résiduel* p_{d_1} d'un seul argument tel que

$$[Mix(\langle p, d_1\rangle)](d_2) = p_{d_1}(d_2) = p(\langle d_1, d_2\rangle).$$

Ce qui est intéressant, c'est qu'on peut appliquer l'évaluateur partiel sur lui-même :

$$p_{\text{gen}} = Mix(\langle Mix, p\rangle).$$

Le programme résiduel p_{gen} est donc un évaluateur partiel spécialisé pour la spécialisation de p. Lorsqu'il est exécuté sur une donnée d_1, il produit le programme résiduel p_{d_1}. Dans le cas qui nous intéresse, il suffit de remplacer p par $\mathcal{S}^*$ et on obtient

$$\mathcal{C}^* = Mix(\langle Mix, \mathcal{S}^*\rangle).$$

Dans sa thèse, Andersen [3] utilise des idées semblables dans le cadre de l'évaluation partielle du langage ANSI C. Le compilateur C qu'il propose est en fait un générateur d'*extensions génératrices :* à partir d'un programme p, il produit un programme p' chargé de spécialiser p et de générer le code approprié à cette spécialisation.

$\mathcal{C}^*$ n'est donc rien de plus qu'une version currifiée de $\mathcal{S}^*$. En pratique, toutefois, il est préférable d'implanter le compilateur abstrait sans intermédiaire. Nous avons montré qu'il est ainsi possible d'optimiser le code généré par $\mathcal{C}^*$ pour obtenir un programme d'analyse plus rapide, semblable à la méthode itérative proposée par Ayers [5].

Évidemment, l'implantation de $\mathcal{C}^*$ pourrait aussi se faire à l'aide de $\mathcal{S}^*$ et *Mix* à partir de l'équation précédente. L'implantation d'un générateur de programmes d'analyse efficace pourrait être un objectif intéressant, mais à long terme seulement. En effet, un tel générateur se butterait certainement aux mêmes problèmes que les générateurs de compilateurs. De plus, il est plus avantageux de l'écrire directement pour permettre des optimisations (comme l'η-réduction) impossibles à réaliser avec les techniques actuelles d'évaluation partielle.

L'analyse de représentation des fermetures (0CFA) est la première analyse que nous aonvs implantée par compilation abstraite. Elle utilisait une technique d'implantation s'apparentant beaucoup à la technique de génération de code à l'aide de fermetures de Feeley et Lapalme [13]. Elle a permis d'accélérer le temps d'analyse par un facteur approchant 4 dans la plupart des cas (les programmes analysés comportaient entre 400 et 1600 lignes chacun).

Dans le cadre de notre recherche, nous comptons générer directement du code machine à l'exécution : le compilateur génère le programme d'analyse et l'exécute immédiatement. Ceci permettra d'accélérer davantage le temps d'analyse. Cette technique, la génération de code à l'exécution, semble d'ailleurs devenir de plus en plus populaire dans le domaine de la compilation. Keppel et al. [21] expliquent les avantages de cette approche à l'optimisation de programmes. Leone et Lee [24, 25] développent des analyses et des techniques permettant la génération de code à l'exécution et font le lien avec l'évaluation partielle. Cette approche se retrouve aussi dans la technique d'implantation des fermetures de Feeley et Lapalme [14]. Dans leur approche, les fermetures sont représentées par du code machine plutôt que par une structure de données particulière.

En pratique, la compilation abstraite permet d'accélérer effectivement le temps d'analyse, mais cette accélération est annulée par le coût de la génération de code. Ainsi, l'approche n'a pas d'intérêt si l'analyse ne sert qu'une seule fois. Or nous sommes convaincus que la compilation séparée peut grandement bénéficier de cette technique. En effet, son avantage est que l'analyse peut être compilée vers du code machine une seule fois et réutilisée à maintes reprises par la suite (à chaque «`make`» du programme). De cette façon, on peut effectuer l'analyse du module localement et conserver sur fichier à la fois le résultat de l'analyse locale et le code compilé du programme d'analyse.

Ainsi, l'analyse globale d'une application consisterait à raffiner les informations obtenues localement à l'aide des informations des autres modules, permettant de réduire, dans certains cas, le nombre d'itérations nécessaires lors de l'analyse globale. En effet, si l'interface de chacun des modules encapsule adéquatement les opérations qui y sont effectuées, seule une quantité limitée d'informations devra être propagée d'un module à un autre. Il est clair, toutefois, que si le langage ne possède pas de déclarations d'exportation ou d'importation, le programmeur devra adopter un style de programmation qui limitera l'accès aux fonctions et variables locales du module.

3 La compilation séparée: nouvelle approche

Cette section décrit l'approche à la compilation séparée que nous proposons et qui permettra de meilleures optimisations intermodules. Évidemment, la compilation séparée comporte de nombreux enjeux. Il y a, par exemple, la cohérence entre l'interface d'un module et son implantation [12]. Nous ne nous intéresserons pas à ces problèmes. Nous désirons nous concentrer uniquement sur les aspects concernant les optimisations intermodules. L'ordre de recompilation des modules, par exemple, pourra être prise en charge par l'utilisateur (ce qui peut se faire à l'aide d'un programme comme `make`).

3.1 L'organisation traditionnelle

L'organisation traditionnelle d'un compilateur supportant la compilation séparée ressemble habituellement à celle de la figure 6.

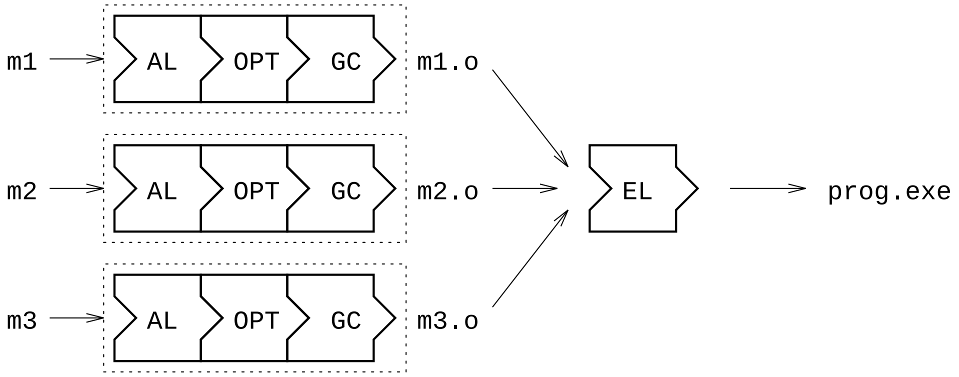

FIG. 6 - *Organisation traditionnelle d'une compilation séparée.*

Une première phase de compilation génère un fichier «objet» pour chacun des modules[7] `m1`, `m2` et `m3`. Chacune de ces compilations est faite séparément et indépendamment des autres. Le résultat est un ensemble de fichiers `.o` (par exemple), un pour chacun des modules. Ces fichiers contiennent généralement une table de symboles, des informations de relocalisation, des informations pour le déverminage et, évidemment, le code object.

La deuxième phase, l'édition de liens, rassemble tous ces fichiers `.o` et génère le programme exécutable. Pour cela, elle combine les différentes sections de données et de code des modules et elle utilise les informations de relocalisation pour corriger les adresses contenues dans les différentes instructions. De plus, les adresses des variables et fonctions importées par les modules sont calculées et insérées aux endroits appropriés.

7. Nous ne faisons ici aucune distinction entre l'interface d'un module et son implantation comme c'est le cas de plusieurs langages. Parmi ceux-ci, il y a Modula-2, Oberon et Dylan.

Les différentes étapes de la compilation, de l'analyse syntaxique à la génération de code en passant par les analyses de flot de données, les optimisations intra- et interprocédurales et l'allocation des registres, sont toutes effectuées dans la première phase. Si le code source d'un des modules est modifié, seul ce module doit être recompilé et l'édition de liens doit être effectuée de nouveau.

Un modèle de coût très rudimentaire pour la compilation séparée est le suivant : soit $M = \{m_1, \ldots, m_n\}$ l'ensemble des modules d'une application et $L = \{l_1, \ldots, l_k\}$ l'ensemble des librairies utilisées, le «`make`» initial a un coût donné par

$$\text{coût}_{\text{all}} = \underbrace{\left(\sum_{m \in M} c_1(\text{taille}(m))\right)}_{\text{compilation}} + \underbrace{\left(\sum_{m \in M \cup L} c_2(\text{taille}(m))\right)}_{\text{édition de liens}} \tag{1}$$

et les «`make`» subséquents (après modification d'un module m_i) ont un coût donné par la formule

$$\text{coût}_{m_i} = c_1(\text{taille}(m_i)) + \left(\sum_{m \in M \cup L} c_2(\text{taille}(m))\right) \tag{2}$$

où $c_1(n)$ et $c_2(n)$ sont des fonctions qui donnent respectivement le temps de compilation et le temps d'édition de liens d'un module de taille n. Typiquement, pour un compilateur optimisant, $c_1(n) \in O(n^3)$ et $c_2 \in O(n)$.

Certains compilateurs, comme par exemple `gcc`, permettent de compiler simultanément un ensemble de fichiers. Dans ce cas, un seul fichier `.o` est généré et les analyses de flot sont effectuées sur tous les fichiers. Mais on ne peut plus vraiment parler de compilation séparée puisqu'une modification dans un des fichiers entraîne la recompilation de l'ensemble au complet. L'application est traitée comme un seul gros fichier.

3.2 Une nouvelle approche

L'approche que nous suggèrons bouleverse cette organisation, afin d'obtenir de meilleures optimisations globales et, en bout de ligne, en des programmes plus performants. Cette approche utilise la technique d'analyse statique par compilation abstraite que nous avons décrite précédemment.

L'approche par compilation abstraite est illustrée à la figure 7. La première phase de compilation est remplacée par une phase d'analyse statique et d'une compilation abstraite de chacun des modules. Pour chaque module `m`, un fichier `m.cfa` est généré, contenant les informations locales recueillies par l'analyse statique du module ainsi qu'un programme permettant de propager ces informations aux autres modules et de recueillir les informations provenant des modules importés par `m` (le programme d'analyse de `m`). Un fichier `m.ir` est aussi généré. Il contient une représentation intermédiaire optimisée du module `m`. Nous discuterons plus loin de cette représentation.

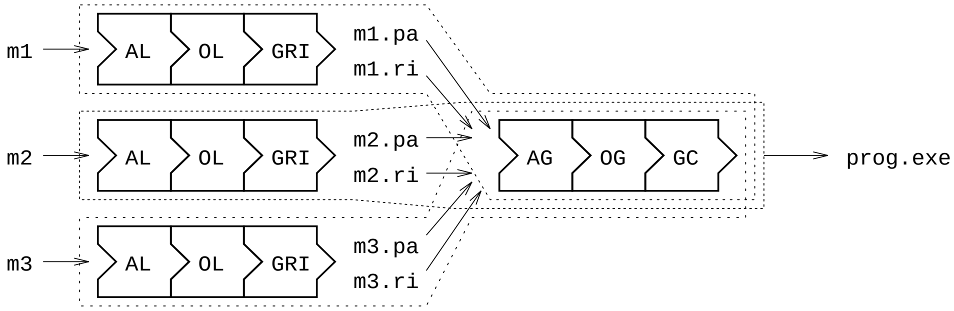

FIG. 7 - *Organisation proposée.*

La deuxième phase procède en deux étapes :

1. Elle lit d'abord tous les programmes d'analyse des modules d'une application (les fichiers `.cfa`) et effectue l'analyse globale par itérations successives, jusqu'à l'obtention d'un point fixe. Une itération correspond à l'exécution de tous les programmes d'analyse des différents modules.

2. Les fichiers `.ir` des différents modules sont lus et le compilateur génère alors le code pour chacun, en utilisant les informations obtenues à l'étape précédente pour effectuer les optimisations intermodules. Nous décrirons plus loin les optimisations que nous envisageons.

La première étape n'est ni plus ni moins qu'une édition de liens des programmes d'analyse suivie de leur exécution. On pourrait donc imaginer l'application des mêmes techniques à l'optimisation des fichiers `.cfa`, produisant ainsi des programmes d'analyse pour des programmes d'analyse pour ... etc. Heureusement, le programme d'analyse généré par la première phase du compilateur possède une structure trop simple pour être un bon candidat aux optimisations globales.

Cette organisation est intéressante à plusieurs points de vue. Tout d'abord, elle permet d'utiliser les techniques d'analyse de flot de contrôle dans un cadre plus général, soit celui des analyses globales intermodules. Et ce, à un coût relativement faible. En effet, l'analyse locale à chaque module est effectuée une seule fois, seules les informations provenant des autres modules étant calculées à chaque compilation (lors de la phase 2). De plus, elle permet de développer des librairies de fonctions dont le code sera adapté aux besoins particuliers de chaque programme. Les librairies pourront donc être de très haut niveau, sans pour autant compromettre les performances des systèmes les utilisant.

Évidemment, le coût de la deuxième phase de compilation risque d'être plus élevé qu'avec l'organisation traditionnelle. Il faut effectuer l'analyse intermodule, optimiser et générer du code machine pour tous les modules à chaque fois qu'un seul est modifié. Le modèle de coût est maintenant donné par

$$\text{coût}'_{\text{all}} = \left(\sum_{m\in M} c'_1(\text{taille}(m))\right) + c'_2(\sum_{m\in M\cup L}\text{taille}(m)) \tag{3}$$

pour la compilation initiale et

$$\text{coût}'_{m_i} = c'_1(\text{taille}(m_i)) + c'_2(\sum_{m\in M\cup L}\text{taille}(m)) \tag{4}$$

pour les compilations subséquentes et $c'_1(n), c'_2(n) \in O(n^3)$ à cause de la phase d'analyse globale. Dans l'optique d'un compilateur de hautes performances, ce coût pourrait être prohibitif. Dans ce cas, une approche hybride est envisageable. Par exemple, des groupes de modules dépendants entre eux pourraient être compilés simultanément, une modification dans l'un entraînant seulement la recompilation du groupe auquel il appartient.

On peut évidemment prétendre que cette approche n'est pas réellement de la compilation séparée. Cette position est défendable si on limite le terme «compilation» à la génération de code objet. Nous considèrons toutefois que le processus de compilation est trop complexe pour le restreindre à cette seule définition. Une compilation nécessite généralement plusieurs phases, utilisant chacune leurs propres représentations intermédiaires (toutes ces représentations dépendent fortement du langage à compiler et des stratégies de compilation). Le code machine n'est donc qu'une représentation parmi tant d'autres. En ce sens, il est justifié de parler ici de compilation séparée.

Cette nouvelle approche à la compilation séparée semble intéressante et prometteuse, sur papier du moins. Il importe donc de montrer qu'elle est aussi intéressante en pratique. C'est ce que nous comptons montrer par nos travaux.

4 Le compilateur proposé

Notre projet consiste à montrer les bénéfices et les coûts réels de l'approche à la compilation séparée que nous suggèrons. Pour ce faire, nous compte intégrer à un compilateur existant la technique de compilation abstraite séparée. Le compilateur utilisé sera `gambit` [15], le compilateur Scheme de Marc Feeley.

Le choix de Scheme comme langage source et de `gambit` comme compilateur n'est pas fortuit. La principale raison du choix de Scheme réside dans l'existence d'analyses de flot de contrôle par interprétation abstraite, soit celles de Shivers et Ayers. De plus, Scheme est certainement plus difficile à compiler efficacement que C ou Pascal, surtout séparément, d'où l'importance de techniques d'analyse plus poussées. Bien que le langage Dylan soit encore plus complexe à compiler que Scheme et que les opportunités pour l'optimisation globale sont plus nombreuses, il n'existe pas d'analyses statiques par interprétation abstraite pour Dylan. Il n'existe pas non plus de compilateur pour Dylan qui aurait pu servir de base à mon système.

Le choix de `gambit` est motivé par un certain nombre de facteurs. Tout d'abord, c'est un des meilleurs compilateurs Scheme disponibles présentement. Malheureusement, il ne fait ni analyse de flot de contrôle ni optimisations globales et il en bénéficierait certainement. Ensuite, il génère du code natif pour plusieurs architectures : MC68020, DEC Alpha et Sparc. Des tests sous plusieurs architectures seront donc possibles, permettant de mieux évaluer le système final. Aussi, il n'y a pas le coût d'une compilation vers C suivie d'un appel au compilateur C du système sous-jacent. Nous pourrons donc évaluer les coûts réels de la compilation abstraite séparée. Enfin, `gambit` est en continuel développement, contrairement à d'autres compilateurs comme ORBIT, par exemple. Mon système sera donc basé sur un compilateur qui est à la fine pointe des recherches actuelles.

Évidemment, cette nouvelle organisation pour la compilation séparée pose de nombreux défis. D'abord, il faut repenser la structure même du compilateur puisque certaines optimisations sont faites localement et d'autres globalement. Ensuite, il faut des représentations intermédiaires adéquates, que ce soit pour le code abstrait généré ou pour le programme partiellement optimisé. De plus, il importe de cibler adéquatement les optimisations globales qui seront implantées. Ces divers thèmes sont discutés dans les sections qui suivent.

4.1 L'architecture du compilateur

L'architecture du compilateur que nous proposons sera sensiblement différente de l'architecture traditionnelle. La figure 8 montre les diverses phases d'un compilateur traditionnel [1, 16]. En pratique, il n'est pas rare de voir plusieurs phases réunies en une seule. Par exemple, il est souvent possible de faire une partie de l'analyse sémantique lors de la construction de l'arbre d'analyse abstrait, c'est-à-dire pendant les analyses syntaxique et lexicale. Il arrive aussi fréquemment que les différentes phases changent la représentation intermédiaire du programme. Typiquement, l'analyseur syntaxique produit un arbre d'analyse abstrait (AST); l'analyseur sémantique convertit ensuite cet arbre en un graphe de flot de contrôle; après les optimisations, le générateur de code produit enfin du code machine.

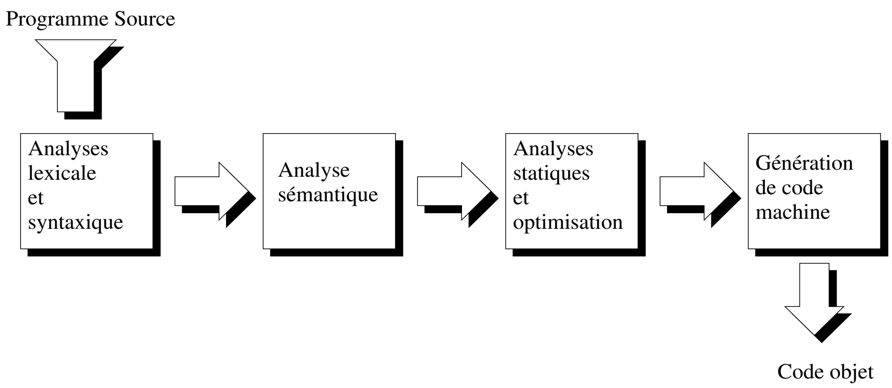

FIG. 8 - *Les phases d'un compilateur traditionnel*

Le compilateur que nous proposons opérera en deux temps. D'abord, il procédera comme un compilateur traditionnel jusqu'à la phase d'analyse statique et d'optimisation. Mais plutôt que de passer ensuite par la phase de génération de code, le compilateur produira deux fichiers : le programme d'analyse (PA) et une représentation intermédiaire optimisée localement (RI) du module. Dans un deuxième temps, le compilateur utilisera tous les programmes d'analyses des différents modules de l'application à compiler pour effectuer l'analyse globale. À l'aide des informations ainsi recueillies, il pourra générer le code optimisé de chacun des modules à partir de leur représentation intermédiaire.

4.2 Les optimisations

4.2.1 Les optimisations intramodules.

Avant de décider des optimisations intermodules à accomplir, il nous faudra déterminer celles qui seront effectuées localement. (Notons que nous nous intéresserons seulement aux optimisations liées au contrôle et non à la représentation des objets.) Évidemment, ces optimisations devront être plus conservatrices que les optimisations globales puisque le compilateur n'a à ce moment qu'une approximation locale du flot de données global. Il serait bon, à notre avis, de mettre au point une analyse qui permette de déterminer localement les points du programme dont les informations de flot ne dépendent pas des modules externes. Considérons par exemple le fragment de programme Scheme suivant :

```
(let ((discr (- (* b b) (* 4 a c))))
  (if (< discr 0)
      ...))
```

Si on suppose que les fonctions `-`, `*` et `<` sont celles de la librairies, `discr` sera lié à un nombre. De plus, si on sait à la compilation que `a`, `b` et `c` sont liés à des entiers, `discr` sera également un entier et on pourra éliminer tous les tests de types. Il est donc possible d'optimiser localement les appels si aucune de ces variables ne dépend de l'interaction avec les autres modules.

La détermination de cette information a deux impacts directs. D'abord, elle permet de décider des optimisations à effectuer lors de la compilation séparée d'un module. De plus, ceci permettra de réduire la taille des programmes d'analyse et du même coup le temps de l'analyse globale. Nous envisageons donc de développer cette analyse et l'intégrer à notre système.

4.2.2 Les optimisations intermodules.

Plusieurs types d'optimisations intermodules sont envisageables. D'abord, toutes les optimisations directement rattachées aux analyses de flot de contrôle peuvent être effectuées. Entre autres, on peut faire l'optimisation de représentation des fermetures et l'élimination de tests de types. Les optimisations possibles dépendent surtout de l'analyse effectuée.

Aussi, il nous semble possible d'effectuer de l'intégration de fonctions *(function inlining)*. Cette optimisation est essentielle pour la compilation efficace de certaines opérations «fondamentales» de Scheme comme `car`, `cons`, `+`, etc. Richardson et Ganapathi [18] montrent d'ailleurs que l'intégration de fonctions permet de faire de meilleures optimisations intraprocédurales qu'en utilisant seulement les informations des analyses interprocédurales.

Pour pouvoir effectuer cette optimisation efficacement, une condition essentielle est que le graphe de dépendance dynamique des modules[8], tel que construit par la phase d'analyse globale, soit acyclique de manière à éviter de charger tous les fichiers `.ir` avant d'effectuer la génération de code. De cette façon, on peut conserver en mémoire la représentation intermédiaire de courtes fonctions intégrables et utiliser cette représentation lors de la compilation des modules qui importent cette fonction. Cela impose néanmoins une certaine forme à la représentation intermédiaire. Nous discuterons de cet aspect à la prochaine section.

4.3 Les représentations intermédiaires

La conception de représentations intermédiaires adéquates pour le programme d'analyse et le programme à compiler sera au coeur de notre recherche. Nous décrirons maintenant quelques idées concernant cet aspect du système. Évidemment, ces idées constituent seulement un premier élan de réflexion et risquent fort d'être modifiées dans le résultat final de nos travaux.

Nous illustrerons ces idées à l'aide d'un exemple d'un module écrit en Scheme. Le module en question se trouve à la figure 9. Le module implante deux fonctions qui sont exportées[9]. La première, `creer-gen-pseudo`, permet de créer un générateur de nombres pseudo-aléatoires

8. Dans ce graphe, il y a un arc du module A vers le module B s'il existe un site d'appel `(f ...)` de B où `f` peut être liée à une λ-expression définie dans le module A.

9. Étant donné que le langage Scheme ne possède pas de déclarations d'importation et d'exportation, nous prendrons comme convention de considérer que toutes les variable libres d'un module sont importées et que toutes les variables globales sont exportées.

```
(define creer-gen-pseudo
  (lambda₁ (germe)
    (lambda₂ ()
      (let ((tmp (modulo (* germe 171) 30307)))
        (set! germe tmp)
        (/ tmp 30307)))))

(define creer-gen-paires-pseudo
  (lambda₃ ()
    (let ((r1 (creer-gen-pseudo 1013))
          (r2 (creer-gen-pseudo 1021)))
      (lambda₄ () (cons (r1) (r2))))))
```

FIG. 9 - *Un module écrit en Scheme.*

compris entre 0 et 1 et la deuxième, `creer-gen-paires-pseudo`, permet de créer un générateur de paires de nombres pseudo-aléatoires. Ces deux fonctions sont d'ordre supérieur : elles retournent des fonctions comme résultat.

Les expressions `(lambda (...) ...)` ont été numérotées. Ceci permettra d'identifier plus facilement ces expressions dans le reste du texte : λ_i identifie l'expression (`lambda`$_i$ `(...) ...)`.

4.3.1 Le programme d'analyse.

Le fichier contenant le programme d'analyse sera divisé en quatre sections :

1. une table des symboles (variables) importés et exportés;
2. les informations de flot de données recueillies localement;
3. le code pour l'analyse de chacune des `lambda`-expressions du module (auxquelles le compilateur a ajouté une `lambda`-expression contenant les expressions *top-level* du module); et
4. un pointeur vers le code d'analyse de la `lambda`-expression contenant les expressions *top-level*.

Voyons ce que contiendront ces sections dans le cas du module de la figure 9. Tout d'abord, la table de symboles indiquera que les variables `creer-gen-pseudo` et `creer-gen-paires-pseudo` sont exportées et que les primitives `modulo`, `*`, `/` et `cons` sont importées et qu'elles ne sont pas modifiées par le module.

Les informations de flot calculées localement lors de la compilation séparée du module contiendront une approximation du type des valeurs possibles de chaque variable du module. Dans l'exemple, ces informations sont :

`creer-gen-pseudo:` ⟨fermeture⟩ $\in \{\lambda_1\}$
`germe:` ⟨entier⟩
`tmp:` ⟨entier⟩
`creer-gen-paires-pseudo:` ⟨fermeture⟩ $\in \{\lambda_3\}$
`r1:` ⟨fermeture⟩ $\in \{\lambda_2\}$
`r2:` ⟨fermeture⟩ $\in \{\lambda_2\}$

Le code pour l'analyse de chacune des `lambda`-expressions du module vient ensuite. Le programme d'analyse pour λ_1 et λ_3 pourrait ressembler à ceci :

$\lambda_1(v_1)$: $\quad \texttt{germe} = \texttt{germe} \sqcup v_1$
$\qquad$ **retourner**$(\{\lambda_2\})$

$\lambda_3()$: $\quad \texttt{r1} = \texttt{r1} \sqcup \mathit{AbstAppl}(\texttt{creer-gen-pseudo}, \langle\text{entier}\rangle)$
$\qquad \texttt{r2} = \texttt{r2} \sqcup \mathit{AbstAppl}(\texttt{creer-gen-pseudo}, \langle\text{entier}\rangle)$
$\qquad$ **retourner**$(\{\lambda_4\})$

où *AbstAppl* est une fonction fournie par le compilateur, qui implante l'application abstraite de fonction et propage les informations des sites d'appel aux λ-expressions susceptibles d'être appliqués à ces sites; $t_1 \sqcup t_2$ est le type le plus spécifique qui a t_1 et t_2 comme sous-types. Par exemple $\langle\text{entier}\rangle \sqcup \langle\text{réel}\rangle = \langle\text{réel}\rangle$, et $\langle\text{fermeture}\rangle \sqcup \langle\text{paire}\rangle = \langle\text{objet}\rangle$.

Pour λ_2 et λ_4, l'utilisation de primitives Scheme complique un peu la situation. On peut être tenté de supposer que les fonctions `*`, `/`, `modulo` et `cons` sont celles fournies par le système. C'est généralement le cas. On peut alors tirer profit des propriétés de ces fonctions (le type du résultat, par exemple) pour opérer de meilleures optimisations locales et, du même coup, réduire la quantité de travail à effectuer lors de l'analyse globale. Dans le cas du module qui nous intéresse, on obtiendrait :

$\lambda_2()$: $\quad$ **retourner**$(\langle\text{rationnel}\rangle)$

$\lambda_4()$: $\quad$ **retourner**$(\langle\text{paire}\rangle)$

Malheureusement, il est possible qu'un autre module redéfinisse ces fonctions (i.e. la valeur liée à une variable globale est une caractéristique globale du programme). Dans ce cas, le compilateur ne peut rien supposer sur ces fonctions, sinon il changerait la sémantique de Scheme. Il doit alors générer des programmes d'analyse semblables à ceux-ci :

$\lambda_2()$: $\quad t_1 = \mathit{AbstAppl}(\texttt{*}, \texttt{germe}, \langle\text{entier}\rangle)$
$\qquad t_2 = \mathit{AbstAppl}(\texttt{modulo}, t_1, \langle\text{entier}\rangle)$
$\qquad \texttt{tmp} = \texttt{tmp} \sqcup t_2$
$\qquad \texttt{germe} = \texttt{germe} \sqcup \texttt{tmp}$
$\qquad$ **retourner**$(\mathit{AbstAppl}(\texttt{/}, \texttt{tmp}, \langle\text{entier}\rangle))$

$\lambda_4()$: $\quad t_1 = \mathit{AbstAppl}(\texttt{r1})$

$t_2 = \mathit{AbstAppl}(\texttt{r2})$
$\mathbf{retourner}(\mathit{AbstAppl}(\texttt{cons}, t_1, t_2))$

Quelques compromis semblent toutefois possibles. Par exemple, le compilateur pourrait faire la supposition que les primitives utilisées sont celles du système et mettre au début du fichier d'analyse, des indications sur les primitives utilisées et modifiées. Si, avant de faire l'analyse globale, le compilateur découvre qu'un module a été compilé séparément et que les suppositions faites lors de cette compilation ne sont plus vérifiées, il pourrait recompiler le module et utiliser le nouveau programme d'analyse de ce module.

Une autre possibilité consisterait à générer, lors de la compilation séparée, deux (ou plus) programmes d'analyse différents et l'analyse globale utiliserait la version qui convient le mieux à l'application. Enfin, le programme d'analyse pourrait contenir des instructions exécutables seulement si les primitives sont redéfinies dans le module ou ailleurs.

Plusieurs possibilités sont envisageables pour la représentation du programme d'analyse (généré par la compilation abstraite). On peut d'abord penser à générer du code virtuel (ou *byte-code*). Les avantages sont nombreux : c'est facile à générer et c'est très portable. Par contre, cela rajoute une couche d'interprétation à l'exécution, ayant pour effet de ralentir l'analyse par un facteur constant.

Étant donné que le langage Scheme sera utilisé pour implanter le compilateur, une autre possibilité consiste à effectuer la compilation vers des fermetures (voir Feeley et Lapalme [13]). Les temps d'exécution seront certainement meilleurs qu'avec la solution précédente, mais le code sera difficile à emmagasiner sur disque. À l'autre extrême, la génération d'instructions machine est envisageable, gagnant ainsi en vitesse d'exécution. Malheureusement, la phase d'analyse globale nécessite une forme d'édition de liens qui deviendrait plus difficile à réaliser.

Nous proposons de prendre une voie intermédiaire, à peine plus coûteuse lors du chargement du programme d'analyse. Elle consiste à générer un code virtuel sur disque et, lors de son chargement en mémoire, à générer du code machine à la volée. L'analyse sera donc très rapide et le fichier contenant le programme d'analyse sera très portable. Seul le générateur de code machine devra être réécrit pour chaque nouvelle plate-forme (ou bien on peut interpréter le code virtuel directement). Cette technique est inspirée des travaux de Franz [17] décrivant une méthode de compilation dans laquelle la génération de code se fait au chargement du programme.

4.3.2 Le programme source.

L'organisation du compilateur ainsi que les types d'optimisations intermodules ont un impact important sur la représentation intermédiaire des programmes à compiler, en mémoire et sur disque (les fichiers `.ir`). Tout d'abord, il faut garder trace des endroits susceptibles d'être optimisés lors de la phase d'optimisations globales. Il importe, par exemple, que le compilateur puisse détecter simplement les tests de types qui pourraient être éliminés et les sites d'appels calculés qui deviendraient connus statiquement après l'analyse globale.

Puisque les langages d'ordre supérieur possèdent certaines opérations de base n'ayant pas d'équivalent au niveau machine, il importe que la représentation intermédiaire du programme puisse exprimer de manière adéquate ces opérations. L'application d'une fermeture est un exemple; les tests de types en sont un autre. Mais par contre, il ne faut pas non plus qu'elle s'éloigne trop du niveau de la machine pour ne pas entraver inutilement certaines optimisations et ralentir la génération de code. L'allocation de registres, par exemple, devrait être effectuée à la phase d'optimisations globales si possible. La représentation intermédiaire doit donc posséder une certaine notion de registres.

Il ne semble pas satisfaisant de "rapiécer" une représentation intermédiaire existante en lui ajoutant quelques cas particuliers. Un langage de transfert de registres *(RTL)* pourrait être augmenté d'un certain nombre d'opérateurs, par exemple. Mais le tout manquerait certainement de cohérence. Cette approche nous apparaît donc trop simpliste et manquer de généralité.

Nous proposons donc de développer une représentation intermédiaire à partir des contraintes spécifiques posées par l'optimisation globale de programmes d'ordre supérieur. Voici un certain nombre de ces contraintes dont il faudra tenir compte :

Les appels génériques. Que ce soient des fermetures, des fonctions génériques ou des envois de messages, beaucoup de langages modernes possèdent une notion d'*appel générique :* un appel pour lequel on ne connaît pas statiquement le point du programme où il faut brancher. Ces langages doivent utiliser un mécanisme général qui détermine effectivement, à l'exécution, l'endroit où brancher.

Il apparaît important d'expliciter ces appels génériques dans la représentation intermédiaire car ils sont souvent la source de nombreuses optimisations.

Les langages typés dynamiquement. Ces langages sont difficiles à compiler efficacement car le compilateur doit inclure de nombreux tests de types dans le code généré. Les analyses statiques permettent souvent de déterminer statiquement le type des variables et donc d'éliminer ces tests. Pour simplifier la tâche du compilateur, il est important que ces tests de types soient explicités dans la représentation intermédiaire.

Les primitives. Généralement, il est préférable de faire l'intégration de certaines primitives du langage plutôt que de faire des appels à une librairie. Malheureusement, des langages comme Scheme freinent ce type d'optimisation car il est possible de lier les noms des primitives à d'autres objets (en Scheme, par exemple, il est sémantiquement correct d'écrire `(set! car 3)`). Il faut habituellement ajouter des annotations au programme afin d'indiquer au compilateur qu'il peut effectivement faire l'intégration. Évidemment, ces annotations ne sont en général pas portable et obligent le programmeur à s'assurer de leur exactitude, sans compter qu'elles changent la sémantique du langage.

Lors de la phase d'analyse globale, il sera possible de détecter les primitives qui sont susceptibles d'être modifiées. Il faudra donc que les primitives du langage puissent être exprimées adéquatement dans la représentation intermédiaire afin de faire l'intégration de celles qui ne sont pas modifiées, lors de l'optimisation globale.

Note 1 *La plupart des analyses de flot de contrôle de langages d'ordre supérieur qui ont été développées présupposent une transformation préalable des programmes sous forme CPS* (continuation-passing style)*. Cette transformation semble simplifier les algorithmes d'analyse ainsi que leur preuve d'exactitude. Malheureusement, elle introduit un nombre considérable de fermetures et il faut des optimisations très sophistiquées pour les éliminer. Nous prenons donc le parti de ne pas faire cette transformation et d'adapter les analyses à la représentation intermédiaire que nous développerons. C'est d'ailleurs ce que fait le compilateur* `bigloo`*.*

5 Objectifs de la recherche

Les analyses et optimisations intermodules telles que nous les aonvs décrites permettent d'ores et déjà d'envisager un éventail considérable de solutions (pour l'instant très partielles). Toutefois, le choix de solutions devra être guidé par les deux objectifs suivants :

1. Les analyses et optimisations intermodules devront évidemment se traduire par un code généré plus efficace, de manière à encourager davantage l'utilisation des fonctions d'ordre supérieur ainsi que l'écriture de librairies réutilisables.

2. Ces analyses et optimisations ne devront pas augmenter substantiellement le temps de génération de l'exécutable, c'est-à-dire que le temps d'exécution de la phase 2 soit du même ordre de grandeur que celui d'une édition de liens classique.

Afin de rencontrer ces deux objectifs, il nous faudra toujours garder à l'esprit le principe suivant lors du choix de solutions : il faut factoriser au maximum le travail à effectuer lors de la compilation séparée afin de minimiser la phase de compilation globale. En effet, si on se rapporte au modèle de coût du système que nous proposons (voir équations 3 et 4), on remarque que l'édition de liens est d'une complexité beaucoup plus grande qu'avec l'approche traditionnelle et qu'elle s'effectue sur tout le programme en même temps. Il faudra donc minimiser le coût de cette phase (la fonction c_2').

6 Travaux connexes

La recherche que nous proposons se situe à la rencontre de deux domaines de recherche déjà fort étudiés : l'analyse statique de langages d'ordre supérieur par interprétation abstraite et l'analyse statique et les optimisations interprocédurales et intermodules. Il n'existe pas à notre connaissance de travaux qui aient réuni ces deux domaines.

6.1 L'interprétation abstraite

Les travaux les plus reliés à notre recherche sont ceux de Shivers [34, 35], de Ayers [5] et de Serrano [31]. Shivers et Ayers donnent tous deux des algorithmes d'analyse de représentation des fermetures et de déduction de types pour le langage Scheme. Ils proposent aussi des optimisations qui utilisent l'information de ces analyses. Il faut noter que la thèse de Ayers met beaucoup plus d'emphase sur l'efficacité des algorithmes que celle de Shivers et que ses preuves d'exactitude sont beaucoup plus simples. Toutefois, aucun des deux n'a intégré ses analyses à un compilateur existant.

Serrano, quant à lui, prouve l'utilité des analyses par interprétation abstraite pour la compilation des langages fonctionnels. Son compilateur Scheme, `bigloo`, est à notre connaissance le seul compilateur du genre à utiliser des analyses par interprétation abstraite prouvées sémantiquement correctes. Dans [31], il décrit les analyses qu'il a implanté et les optimisations qui en découlent. Ses expérimentations montrent que le temps consacré à l'analyse est considérable (représentant parfois 90% du temps de compilation). De l'aveu même de l'auteur, ces performances sont principalement dues à l'implantation naïve des environnements abstraits.

Dans les trois cas, il n'est aucunement question d'appliquer ces analyses à la compilation séparée et aux optimisations intermodules.

Quant à la technique de compilation abstraite, elle a déjà été suggérée, mais sous une forme et pour un langage différents. Lin et Tan [26] montrent comment effectuer la compilation de l'analyse de flot de données pour des programmes Prolog. Mais comme leur objectif est de réutiliser au maximum la technologie des compilateurs Prolog actuels, c'est plutôt le code WAM généré par le compilateur Prolog qui est fourni à un interprète non-standard en vue de l'analyse. Leurs tests montrent des accélérations de l'ordre de 100 en moyenne en faveur de leur implantation.

Il faut toutefois mentionner que leur interprète est écrit en C et que les systèmes auxquels ils se comparent sont écrits en Prolog. De plus, ils expliquent leurs gains de performance par les deux facteurs suivants : l'élimination de la couche d'interprétation et l'efficacité des opérations sur des structures globales (en Prolog, ces opérations sont typiquement effectuées par les prédicats `assert` et `retract`, qui sont certainement beaucoup plus lents que des accès à un tableau global en C).

La méthode d'analyse proposée par Lin et Tan possède un désavantage par rapport à la technique que nous avons développée. Comme le code WAM utilisé pour l'analyse statique est le même que celui qui sera ultérieurement optimisé et interprété, aucune optimisation n'est effectuée pour réduire le temps d'analyse. Dans le système que nous proposons, plusieurs optimisations sont possibles, dont la détection des sites d'appels initiaux tels que décrits par Ayers [5].

6.2 Optimisations de code intermodules

De nombreux travaux ont été menés dans le domaine des optimisations intermodules, dont la quasi totalité pour des langages impératifs comme C, Fortran, Modula-2, etc. De ces travaux se dégagent deux approches. La première approche consiste à utiliser le code généré par la première phase de compilation séparée pour faire les optimisations au niveau du code objet. L'autre approche consiste à générer des fichiers contenant les informations recueillies lors des analyses locales à chacun des modules, dans un premier temps, et à utiliser ces informations lors de la génération de code dans un deuxième temps.

Les travaux de Wall [38] et Srivastava et Wall [36] ainsi que ceux de Chow [8] suivent la première approche. Wall [38] montre une technique d'allocation globale des registres à l'édition de liens. L'idée de base est de considérer l'assignation des variables à des registres comme une forme de relocalisation. Dans son système, la première phase de compilation génère du code objet exécutable tel quel, mais qui peut être optimisé. Pour cela, certaines instructions machine doivent être annotées pour indiquer à l'éditeur de liens de quelle manière les opérandes et les résultats de ces instructions sont reliées aux candidats à l'allocation globale de registres. L'éditeur de liens, une fois l'allocation de registres effectuée, peut réécrire les modules desquels il aura changé ou éliminé les instructions annotées.

Plus récemment, Srivastava et Wall [36] ont développé un système permettant des optimisation intermodules à l'édition de liens. Ce système, baptisé OM, effectue une analyse interprocédurale des variables ou registres vivants ainsi que plusieurs optimisations interprocédurales dont le déplacement du code invariant des boucles, l'élimination de code mort et le réordonnancement des procédures pour accroître la localité de traitement. Le système OM utilise directement le code objet généré par la première phase de compilation séparé, qu'il transforme dans un langage de transfert de registres (RTL). Une fois les analyses et optimisations effectuées, le programme est retraduit en code objet.

Chow [8] décrit lui aussi des analyses et des techniques pour l'allocation globale des registres et le placement des instructions de sauvegarde de registres. Dans son système, la première phase de compilation produit une représentation intermédiaire sur disque de chacun des modules. L'édition de liens utilise tous ces fichiers pour faire l'allocation globale des registres et la génération de code.

Les travaux de Odnert et Santhanam [28] tombent dans la deuxième catégorie. Dans cet article, il est question d'un système pour l'allocation globale (intermodule) des registres. Dans ce système, le processus de compilation est divisé en trois phases distinctes. La première phase de compilation génère deux fichiers pour chacun des modules : un fichier contenant une représentation intermédiaire du module et un *fichier de résumé* contenant les informations nécessaires pour la construction du graphe d'appel global et l'allocation globale des registres. Tous les fichiers de résumé de l'application sont ensuite lus par l'*analyseur de programme* qui produit une base de données contenant des directives pour l'allocation globale des registres. Ensuite, le compilateur est réexécuté sur chacun des fichiers contenant les représentations intermédiaires des modules. Il utilise la base de données lors de l'allocation des registres pour chaque module. L'éditeur de liens est ensuite exécuté sur tous les fichiers produits par la précédente phase.

Il y a aussi plusieurs travaux qui concernent l'optimisation (et la réoptimisation) globale incrémentale de programmes [29, 7]. Mais ils se concentrent surtout sur les problèmes de recompilation et de réoptimisation en présence de modifications dans un module. Burke et Torczon [7], par exemple, présentent un *test de recompilation* qui détermine les modules à recompiler suite à des modifications dans un module. Ce test se base sur les diverses contributions des informations locales aux informations globales de flot de données. De leur côté, Pollock et Soffa [29] montrent comment modifier localement le code d'un module qui aurait été globalement optimisé en réponse aux modifications d'un autre module. Ils présentent aussi un algorithme qui détermine si les optimisations qui ont déjà été effectuées deviennent invalides après ces modifications.

Comme on le voit, il n'y a donc pas vraiment eu de travaux qui traitent des optimisations intermodules propres aux langages fonctionnels d'ordre supérieur ou à objets, comme nous nous proposons de le faire. Évidemment, plusieurs de ceux que nous avons décrits partagent des idées communes avec la présentre proposition. Toutefois, l'originalité de notre démarche réside dans l'utilisation des techniques d'analyse par compilation abstraite pour l'optimisation intermodule de ces langages. Ces techniques impliquent le choix de nouvelles représentations intermédiaires, ce qui constituera le coeur de notre recherche.

Références

[1] Aho (A. V.), Sethi (R.) et Ullman (J. D.). – *Compilers: Principles, Techniques, and Tools.* – Reading, Massachussets, Addison Wesley, 1986.

[2] Amiel (E.), Gruber (O.) et Simon (E.). – Optimizing multi-method dispatch using compressed dispatch tables. *Proceedings OOPSLA'94, ACM SIGPLAN Notices*, vol. 29, n° 10, October 1994, pp. 244–258.

[3] Andersen (L. O.). – *Program Analysis and Specialization for the C Programming Language.* – Thèse de PhD, DIKU, University of CopenHagen, May 1994.

[4] Appel (A. W.). – *Compiling with continuations.* – Cambridge, Cambridge University Press, 1992.

[5] Ayers (A. E.). – *Abstract Analysis and Optimization of Scheme.* – Thèse de PhD, Massachussets Institute of Technology, September 1993.

[6] Boucher (D.) et Feeley (M.). – Abstract compilation: a new implementation paradigm for static analysis. *In : Proceedings of the 1996 International Conference on Compiler Construction.* – En préparation.

[7] Burke (M.) et Torczon (L.). – Interprocedural Optimization: Eliminating Unnecessary Recompilation. *ACM Transactions on Programming Languages and Systems*, vol. 15, n° 3, July 1993, pp. 367–399.

[8] Chow (F. C.). – Minimizing register usage penalty at procedure calls. *In : Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 85–94.

[9] Clinger (C.) et Rees (J.). – Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, vol. IV, n° 3, July–September 1991, pp. 1–55.

[10] Consel (C.) et Danvy (O.). – Tutorial Notes on Partial Evaluation. *In : Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pp. 493–501.

[11] Cousot (P.) et Cousot (R.). – Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximations of fixpoints. *In : Fourth Symposium on Principles of Programmings Languages*, pp. 238–252.

[12] Crelier (R.). – *Separate Compilation and Module Extension.* – Zurich, Thèse de PhD, Swiss Federal Institute of Technology, 1994.

[13] Feeley (M.) et Lapalme (G.). – Using closures for code generation. *Computer Languages*, vol. 12, n° 1, 1987, pp. 47–66.

[14] Feeley (M.) et Lapalme (G.). – Closure generation based on viewing `lambda` as epsilon plus compile. *Computer Languages*, vol. 17, n° 4, 1992, pp. 251–267.

[15] Feeley (M.) et Miller (J.). – A parallel virtual machine for efficient Scheme compilation. *In : Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.*

[16] Fischer (C. N.) et R. J. LeBlanc, Jr. – *Crafting A Compiler.* – Menlo Park, California, Benjamin Cummings, 1988.

[17] Franz (M. S.). – *Code-Generation On-the-Fly: A Key to Portable Software.* – Zurich, Thèse de PhD, Swiss Federal Institute of Technology, 1994.

[18] Ganapathi (M.) et Richardson (S.). – Interprocedural analysis vs. procedure integration. *Information Processing Letters*, vol. 32, 1989, pp. 137–142.

[19] Hecht (M. S.). – *Flow Analysis of Computer Programs.* – New-York, Elsevier North-Holland, 1977.

[20] Hudak (P.). – A semantic model of reference counting and its abstraction (detailed summary). *In : Conference of Lisp and Functional Programming*, pp. 351–363.

[21] Keppel (David), Eggers (Susan J.) et Henry (Robert R.). – *A case for runtime code generation.* – Rapport technique n° 91-11-04, Department of Computer Science and Enginnering, University of Washington, November 1991.

[22] Kiczales (G.) et Rodriguez (L.). – Efficient method dispatch in PCL. *In : Proceedings of the ACM Conference on Lisp and Functional Programming '90*, pp. 99–105.

[23] Kranz (D. A.). – ORBIT*: An Optimizing Compiler for Scheme.* – Thèse de PhD, Yale University, 1988.

[24] Leone (Mark) et Lee (Peter). – *Deferred Compilation: The Automation of Run-Time Code Generation.* – Rapport technique n° CMU-CS-93-225, Pittsburgh, PA 15212, Carnegie-Mellon, Department of Computer Science, December 1993.

[25] Leone (Mark) et Lee (Peter). – Lightweight Run-Time Code Generation. *In : Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* pp. 97–106. – Technical Report 94/9, Department of Computer Science, University of Melbourne.

[26] Lin (I-P.) et Tan (J.). – Compiling Dataflow Analysis of Logic Programs. *In : Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation*, pp. 106–115.

[27] Mycroft (A.). – *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* – Thèse de PhD, Univeristy of Edinburgh, 1981.

[28] Odnert (D.) et Santhanam (V.). – Register allocation across procedure and module boundaries. *In : Proceedings of the 1990 ACM Conference on Programming Language Design and Implementation*, pp. 28–39.

[29] Pollock (L. L.) et Soffa (M. L.). – Incremental Global Reoptimization of Programs. *ACM Transactions on Programming Languages and Systems*, vol. 14, n° 2, April 1992, pp. 173–200.

[30] Rozas (G. J.). – Taming the Y operator. *In : Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 226–234.

[31] Serrano (M.). – Control flow analysis: a compilation paradigm for functional language. *In : Proceedings of SAC 95.*

[32] Serrano (M.). – *Bigloo User's Manual.* – Rapport technique, Rocquencourt, Inria, March 1994.

[33] Shalit (A.). – *Dylan Interim Reference Manual.* – Apple Computer, Inc., June, 1994.

[34] Shivers (O.). – Control Flow Analysis in Scheme. *In : SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[35] Shivers (O.). – *Control-Flow Analysis of Higher-Order Languages.* – Thèse de PhD, Carnegie Mellon University, 1991.

[36] Srivastava (A.) et Wall (D. W.). – *A Practical System for Intermodule Code Optimization at Link-Time.* – Rapport technique n° 92/6, Palo Alto, California, DEC Western Research Laboratory, December 1992.

[37] Steckler (P. A.). – *Correct Higher-Order Program Transformations.* – Thèse de PhD, Northeastern University, 1994.

[38] Wall (D. W.). – *Global Register Allocation at Link Time.* – Rapport technique n° 86/3, Palo Alto, California, DEC Western Research Laboratory, October 1986.