Université de Montréal

Analyse et optimisation globales de modules compilés séparément

par

Dominique Boucher

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiae Doctor (Ph.D.)
en informatique

Août 1999

À Martine, Guillaume et Florence

SOMMAIRE

La présente thèse décrit une architecture de compilateur basée sur une technique d'implantation des analyses par interprétation abstraite connue sous le nom de *compilation abstraite*. Cette technique, inspirée des équations de l'évaluation partielle, permet d'obtenir des analyses statiques dont la couche interprétative est pratiquement éliminée. Dans une étude de cas détaillée, nous montrons différentes approches à l'application de cette technique à partir d'une analyse de flot de contrôle pour Scheme, un langage impératif à noyau fonctionnel.

À partir de cette idée, nous développons une architecture de compilateur effectuant l'analyse et l'optimisation globales de programmes multimodules à l'édition des liens. Cette architecture est intéressante à plusieurs niveaux. Entre autres, elle rend possible l'analyse de librairies fournies par des tiers sans recours au code source. De plus, les optimisations peuvent être effectuées sur des représentations de haut niveau du programme source, exposant ainsi les constructions de base du langage, ce qu'une analyse du code machine ne rendrait pas possible. Ceci est particulièrement important pour les langages modernes de haut niveau.

Nous présentons aussi une implantation partielle de cette architecture, une extension à Gambit-C, un compilateur Scheme vers C. Le système en question, GOLD, infère automatiquement les annotations spécifiques au compilateur Gambit-C. Grâce à ce système, il est possible d'écrire un programme multimodules en Scheme standard en ne subissant que de légères pertes de performances dans la plupart des cas.

ABSTRACT

This thesis presents a new compiler architecture based on an technique for the implementation of static analyses developed in the abstract interpretation framework. This technique, called *abstract compilation*, has been inspired by the equations of partial evaluation. It speeds up the analyses by an order of magnitude by almost eliminating the interpretation overhead. In a detailed case study, we show several approaches to the use of this technique on a first-order control-flow analysis for Scheme, an imperative language with a functional subset.

Based on this idea, we present a new architecture for compilers supporting the global analysis and optimization of multimodule programs at link time. This architecture allows the analysis of third party libraries without having to rely on source code. Morever, the link-time optimizations can be performed on a high-level representation of the source program, thus exposing constructs at the source language level that would not be available with link-time analysis of object code. This is crucial for the optimization of high-level, modern languages.

We finally present a partial implementation of this architecture, namely an extension to the Gambit-C system, a Scheme to C compiler. This extension, called GOLD, automatically infers Gambit-C specific annotations, allowing Gambit-C to produce better C code. With this system it is possible to write multimodule Scheme programs without having to use system-specific annotations, and with only small performance penalties in most cases.

TABLE DES MATIÈRES

LISTE DES FIGURES

REMERCIEMENTS

Il me faut avant tout remercier Marc Feeley, mon directeur de thèse. Sans lui, cette thèse n'aurait pas vu le jour. Sa très grande rigueur scientifique et sa grande expertise ont toujours été et seront pour moi sources d'inspiration. Marc sait toujours cerner l'essentiel d'un problème et proposer des solutions astucieuses. Ses conseils ont été cruciaux pour l'accomplissement de mon travail. Mais je le remercie surtout pour sa patience. Il a su me montrer l'importance de mes travaux lors de mes trop nombreux moments de doute.

Je tiens à remercier Olin Shivers. C'est à la lecture de sa thèse de doctorat, à l'été 1994, que me sont venues la plupart des idées qui sont ici développées.

Je dois souligner le soutien financier du Conseil de Recherche en Sciences Naturelles et en Génie du Canada, sans lequel je n'aurais pu me consacrer aussi entièrement à mes études doctorales. Je remercie aussi la direction de Locus Dialogue, mon employeur, qui m'a procuré le soutien nécessaire dans les derniers mois de recherche et de rédaction.

Merci aussi à mes parents. Ils ont toujours cru en moi et m'ont fourni le soutien nécessaire pour entreprendre de si longues études.

Enfin, j'aimerais décerner une mention toute spéciale à Martine, mon épouse. Elle a enduré sans broncher toutes mes manies, mes (trop) nombreuses soirées passées devant l'ordinateur. Mais surtout, elle m'a encouragé comme personne. Mille fois merci!

Chapitre 1

MA THÈSE

L'analyse statique globale de programmes multimodules peut être implantée efficacement grâce à une nouvelle architecture de compilateur basée sur la compilation abstraite.

La traduction d'un programme informatique, écrit dans un langage hautement synthétique plus près de la langue du programmeur que de la machine, en un équivalent binaire efficace et sémantiquement équivalent n'est pas une tâche aussi aisée qu'il en paraît au premier abord. Les compilateurs modernes doivent pour cela utiliser un vaste attirail d'analyses de code et d'optimisations afin de produire des exécutables relativement performants. Or les analyses déployées par ces compilateurs sont généralement effectuées sur une partie du code à la fois, introduisant par le fait même des approximations plus conservatrices aux propriétés calculées, et résultant en bout de ligne en de moins bonnes optimisations.

Heureusement, la vitesse grandissante des ordinateurs contemporains, la baisse marquée du prix de la mémoire primaire ainsi que leur plus grande capacité de stockage permettent d'envisager aujourd'hui la mise en oeuvre de techniques de compilation de plus en plus gourmandes en mémoire et en temps de calcul, et impossible à envisager il y a quelques années seulement. Parmi ces techniques, on retrouve l'analyse et l'optimisation globales à travers la barrière des modules. La présente thèse propose une approche originale à ce problème. Il s'agit d'une architecture de compilateur supportant à la fois la compilation séparée et les analyses et optimisations au travers la barrière des modules, dont nous avons exploré les possibilités et les limites. Parmi les

contributions apportées par le présent travail, mentionnons :

- Une étude détaillée d'une technique d'implantation des analyses statiques connue sous le nom de *compilation abstraite*, présentée pour la première fois dans [17] et redécouverte indépendamment par l'auteur [8]. Cette technique permet d'accélérer considérablement le temps d'analyse de chaque module, mais surtout le temps pris par l'analyse globale. J'y discute aussi différentes façons de mettre en oeuvre la compilation abstraite et les contextes favorisant ce ce choix d'implantation.

- Une architecture pour les compilateurs supportant à la fois la compilation séparée et les analyses et optimisations intermodules. Cette architecture est basée sur une technique permettant l'implantation efficace des analyses statiques, la *compilation abstraite*. L'idée de base est de générer, pour chaque module d'un programme, un *programme d'analyse* qui, lorsqu'exécuté, calcule l'analyse du module correspondant. L'analyse globale intermodule d'une application est alors réalisée en combinant tous les programmes d'analyse des modules de l'application et d'exécuter le programme résultant sur les données partielles recueillies lors des analyses intramodules.

- Une première implantation partielle de cette nouvelle architecture et des tests démontrant les avantages de la compilation abstraite et des optimisations intermodules de haut niveau.

1.1 Structure de la thèse

La suite du présent ouvrage est divisée de la manière suivante :

- Les concepts de base ainsi que la technique de compilation abstraite sont ensuite décrits aux chapitres 2 et 3. Quelques résultats expérimentaux y sont aussi présentés, montrant les avantages de cette technique d'analyse.

- Le chapitre 4 présente la nouvelle architecture de compilateurs que je propose ainsi que les forces qui contraignent cette architecture.

- Le chapitre 5 décrit une implantation partielle de l'architecture proposée. Il s'agit d'une extension à un compilateur Scheme existant, Gambit-C, développé par Marc Feeley à l'Université de Montréal.

- Une évaluation détaillée de l'implantation est ensuite présentée au chapitre suivant. Un certain nombre de mesures y sont décrites, montrant les avantages de l'architecture proposée.

- Finalement, je mets en perspective l'implantation en la comparant à d'autres systèmes.

1.2 Motivation: la programmation modulaire

À notre époque, les systèmes informatiques comprenant plusieurs centaines de milliers, voire même plusieurs millions de lignes de code sont monnaie courante. Pour la réalisation de projets d'une telle envergure, la programmation dite *modulaire*, prise au sens large du terme comme je le décrirai plus loin, est essentielle. Il en va ainsi pour plusieurs raisons :

Encapsulation. 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 sous-jacente 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.

Réutilisation. S'ils sont conçus de manière adéquate, certains modules d'un système peuvent être récupérés et réutilisés dans d'autres systèmes. Il devient alors possible de créer des bibliothèques de procédures, de fonctions, de classes, etc. De telles bibliothèques peuvent même être développées et vendues à des tiers partis.

Gestion de projets De plus, puisque ces systèmes sont rarement l'oeuvre d'un seul programmeur, la décomposition d'un système en divers modules ou sous-systèmes facilite la répartition du travail entre les divers membres d'une même équipe de développeurs.

Compilation. Enfin, ces modules forment des unités distinctes pour la compilation, de sorte qu'une modification dans un module n'entraîne pas nécessairement la recompilation du système complet. 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 en temps et en espace en pire cas, 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 la présente thèse, le terme **module** dénotera une unité distincte pour la compilation séparée. Dans la plupart des cas, cela correspond à un fichier contenant le code source à compiler et non pas nécessairement à un module du langage source. (En Dylan, par exemple, il est possible de définir plusieurs modules Dylan dans un seul fichier.)

Je supposerai aussi un **cycle de développement** typique, 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 constituée d'un certain nombre d'itérations. Chaque itération consiste en des modifications à un ou plusieurs modules, suivi de la recompilation de ces modules et d'une nouvelle édition de liens (ces modifications peuvent aussi comporter l'ajout ou le retrait de modules). C'est le cycle de développement le plus couramment rencontré pour les langages fortement typés statiquement. Dans le cas des langages typés dynamiquement, comme Common Lisp et Scheme, il est souvent possible de compiler séparément un module et de le charger dynamiquement dans un programme en cours d'exécution. Ce type d'environnement rend encore plus difficile l'analyse d'un module puisque ce dernier peut dépendre de fonctionnalités qui n'existent même pas lors de sa compilation. Je me restreindrai donc au premier type d'environnement, soit celui des langages ne permettant pas la modification dynamique du code, via le chargement dynamique.

Malheureusement, la programmation modulaire n'est pas une panacée à tous points de vue, elle vient aussi avec son lot de problèmes.

D'abord, les possibilités d'optimisations à l'intérieur d'un module sont ordinairement limitées par les informations recueillies à l'intérieur de ce module et ne peuvent bénéficier pleinement des analyses statiques 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 entièrement le module M peut s'avérer un gaspillage de ressources, surtout s'il est utilisé dans de nombreuses applications différentes. Pensons simplement à une bibliothèque de procédures comportant des centaines de milliers de lignes. D'autant plus que le code source de cette bibliothèque n'est probablement pas disponible, provenant d'une tierce partie.

Évidemment, un bon système de modules permet de récupérer une certaine partie de l'information qui ne peut être déduite lors de l'analyse du module. Par exemple, les *interfaces* de Modula-3 et les *signatures* de SML permettent d'exporter la

signature complète d'une fonction ou le type d'une variable. Les modules qui importent ces interfaces ou signatures peuvent donc bénéficier d'informations de typage supplémentaires lors de leur compilation. (Ce qui n'est pas le cas avec les *packages* de Common Lisp, qui permettent uniquement d'exporter ou d'importer des symboles, qu'ils dénotent des fonctions ou des variables.)

Malgré tout, ces informations ne sont généralement pas suffisantes pour effectuer toutes les optimisations payantes. Le cas de SML est éloquent. Étant un langage à noyau fonctionnel d'ordre supérieur, il est possible de créer une fermeture[1] dans un module A et d'appliquer cette fermeture dans le corps d'une fonction définie dans un autre module B. Malheureusement, seule la signature de la fermeture est connue grâce à la signature du module A (SML est un langage fortement typé statiquement). Il est alors impossible, lors de l'optimisation du module B, de compiler l'application de la fermeture en un branchement direct au corps de la fermeture ou mieux, de faire l'intégration de cette fonction. Une indirection sera nécessaire: il faudra passer par le pointeur de code contenu dans la fermeture.

La programmation orientée-objet, elle aussi, favorisant l'abstraction et la généralisation, entraîne généralement des pertes de performance, en contrepartie d'une plus grande flexibilité. Contrairement aux principes de la programmation structurée, qui dicte généralement une approche du haut vers le bas (*top-down*) faisant dépendre les modules de plus haut niveau des modules de bas niveau, la conception orientée-objet tend à renverser le sens des dépendances, dictant une approche du bas vers le haut (*bottom-up*). Ce sont les modules (ou classes) de bas niveau, plus spécifiques, plus susceptibles d'être modifiés, qui doivent dépendre des modules (classes) de haut niveau, plus abstraits et moins sujets à subir des modifications. Il en résulte des systèmes moins fragiles au changement. Mais le prix à payer est qu'il est souvent

[1] Une *fermeture* est la représentation d'une fonction à l'exécution, comprenant généralement un pointeur vers le code de cette fonction et l'ensemble de ses variables libres. Plusieurs variantes sont possibles quant à leur représentation.

```
/* 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;  }
```

Figure 1.1. Le programme en C.

nécessaire d'introduire des niveaux d'indirection afin d'obtenir des modèles plus faciles à entretenir, plus facilement extensibles.

1.2.1 Deux exemples

Je vais illustrer les opportunités d'optimisation ratées par la compilation séparée à l'aide de deux programmes multi-modules écrits respectivement en C et Scheme. Les deux programmes ont été compilés de deux façons : en compilant les modules séparément et en compilant tous les modules d'un seul coup dans un seul fichier. 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 programme C. Le programme C (figure 1.1) que j'ai 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 1.2) est un programme Scheme [11] 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 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 raisonnable de croire que les appels `(p x)` dans la fonction `filter` et `(f (car l))` dans la fonction `map1` devront être des appels calculés[2], puisqu'on ne peut savoir, à la compilation, que `p` sera lié à `prime?` et `f` à `(lambda (x) (my+ x 1))`.

[2] Un appel calculé est un appel pour lequel on ne connaît pas, à la compilation, la fonction précise

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

```
;;;  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))))
```

```
;;;  Le module arith.scm
;;;  Les 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))
```

```
;;;  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)))))
```

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

Figure 1.2. Le programme Scheme.

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

---

qui sera appelée. Le mécanisme général (et plus coûteux) d'application d'une fermeture doit alors être utilisé.

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, j'ai procédé comme pour le programme en C. J'ai 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` [49], qui produit du C, a été utilisé avec les options d'optimisation `-O3 -unsafe -farithmetic`. 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.

Chapitre 2

LA COMPILATION ABSTRAITE

Dans ce chapitre, je décris la technique de compilation abstraite, quelques approches pour son implémentation ainsi que des résultats expérimentaux montrant l'avantage de cette technique. Bien que cette technique soit applicable à tout type d'analyse statique, elle trouve ses racines dans l'analyse par interprétation abstraite. J'introduirai donc au préalable quelques notions de base d'interprétation abstraite et l'évaluation partielle qui permettent de formaliser le concept de compilation abstraite.

2.1 Analyse statique

Le processus de compilation d'un programme π écrit dans un langage de programmation L peut être vu comme une suite de traductions vers des langages intermédiaires de plus en plus près du langage de la machine, et possédant chacun des caractéristiques propres. Certains compilateurs, comme par exemple la plupart des compilateurs Java, produisent uniquement du code pour une machine virtuelle. D'autres produisent du code C qui sera ensuite compilé vers du code machine par un compilateur C indépendant. Enfin, certains produisent directement du code machine.

L'intérêt d'utiliser plusieurs langages intermédiaires au sein d'un même compilateur est double. D'abord, chaque langage intermédiaire est généralement conçu pour faire apparaître certaines caractéristiques du programme source ou pour en simplifier la compilation. Par exemple, la plupart des compilateurs Scheme effectuent des transformations de code afin de normaliser (i.e. de simplifier) le code source : une phase d'élimination des affectations à des variables en les remplaçant par des effets de bord dans des structures, le remplacement des expressions `(let* ...)` par

une cascade d'expressions (`let ...`), la conversion CPS (que je décrirai plus loin), etc. D'autre part, certaines optimisations sont plus faciles à effectuer sur certaines représentations intermédiaires que sur d'autres. Par exemple, le *scheduling* des instructions peut difficilement s'effectuer sur une représentation de haut niveau dans laquelle les instructions machine ne sont pas explicitées.

À partir de ces langages intermédiaires, le compilateur doit effectuer des *analyses statiques*[1] en vue de la phase d'optimisation. Ces analyses calculent des invariants du programme indépendants de toute donnée extérieure au programme et qui sont vrais peu importe le chemin d'exécution du programme.

Les langages impératifs. 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, 26, 33]. 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[2] 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 flux de données. 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 2.2 donne le graphe de flux de données d'un bout de programme donné à la figure 2.1.

Les optimisations mises en oeuvre par les compilateurs de langages impératifs sont principalement de deux types. Il y a d'abord les optimisations dites *locales* qui sont ef-

[1] Une analyse est dite *statique* si elle est effectuée à la compilation, en contraste avec des analyses *dynamiques* qui sont effectuées lors de l'exécution du programme.

[2] 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.

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

Figure 2.1. Boucle en C

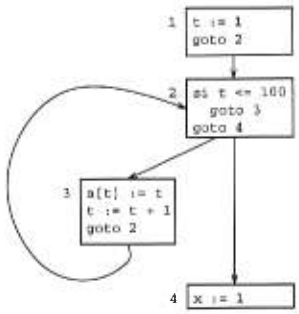

Figure 2.2. Graphe de flux de données

fectuées en ne considérant que les énoncés d'un seul bloc de base à la fois. Parmi celles-ci, on retrouve l'élimination de sous-expressions communes, les optimisations "peephole" et bien d'autres. Dans tous les autres cas, on parle d'optimisations *globales*. Ces optimisations comprennent la propagation de copies, la détection et le déplacement des calculs invariants d'une boucle, l'élimination de variables d'induction, l'élimination de code mort, l'allocation de registre, pour n'en nommer que quelques unes.

À partir du graphe de flux de données, 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 peut être utilisée par un autre point B du programme (et qu'il existe un chemin du point A au point B). Cette information est en général une propriété dynamique puisqu'il est indécidable s'il existe un chemin de A à B pour au moins une exécution du programme.

Le calcul de ces informations s'effectue généralement par approximations successives d'un point fixe[3]. On calcule d'abord les informations locales à chaque bloc de

[3] D'autres techniques plus sophistiquées sont aussi possibles, comme l'analyse par intervalles et l'analyse T_1 - T_2, mais elles requièrent souvent des propriétés particulières du graphe de flux de

base et ensuite on propage ces informations le long des arcs du graphe. On répète ce processus jusqu'à ce qu'on 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 [26] donnent à ce sujet une taxonomie des différentes analyses possibles ainsi que les équations et opérateurs qui s'y rattachent.

Les langages d'ordre supérieur. Dans le cas des langages d'ordre supérieur comme SML, Lisp ou Haskell, l'analyse est beaucoup plus difficile à calculer. Même si les graphes de flux de contrôle sont souvent beaucoup plus simples, les boucles étant habituellement exprimées à l'aide de fonctions récursives dans le cas des langages fonctionnels et les appels de fonctions étant très fréquents, des analyses globales 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 quelle fonction sera appelée à un site d'appel `f(x)`, de surcroit 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 j'introduirai 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

---

données.

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.*[4] 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 [38] 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 [45] 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 [2], 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.

Les langages à classe/objet/prototype. Ces langages, bien que souvent plus près des langages impératifs quant à leur syntaxe, ne sont pas aisés à optimiser. Bien que la plupart des langages à objet ne permettent pas de créer des fonctions de première classe, ils offrent des défis semblables à ceux des langages d'ordre supérieur en raison de caractéristiques comme l'héritage et le polymorphisme. Comme je l'ai montré au chapitre précédent, la détermination de la classe exacte du receveur d'un message, permettant de décider quelle méthode exactement doit être appelée, est une propriété dynamique, c'est-à-dire qu'elle dépend de l'exécution du programme.

[4] Une *fermeture* est la représentation à l'exécution d'une fonction dans les langages fonctionnels. 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.

Et c'est sans compter les langages comme CLOS ou Dylan qui permettent de définir des multi-méthodes, c'est-à-dire des méthodes dont la sélection se fait en considérant tous les paramètres plutôt que le seul receveur du message. Il s'agit d'une généralisation du concept de méthode. Dans ces langages, l'aiguillage des méthodes se fait à l'exécution et implique en général des coûts considérables, puisqu'il faut d'abord trouver les méthodes applicables, les classer en ordre décroissant de spécialisation par rapport aux paramètres actuels et ensuite choisir la plus spécialisée, le tout à l'exécution.

On comprend donc l'importance d'analyses de haut niveau permettant d'optimiser ces langages. L'analyse la plus importante est certainement l'*analyse de la hiérarchie de classes* (ou de type). Cette analyse permet de déterminer, pour chaque site d'envoie de message (ou appel de méthode) S, l'ensemble des classes concrètes C dont le receveur du message peut être une instance à l'exécution. Cette analyse est similaire à l'analyse des fermetures des langages fonctionnels. Elle rend possible l'intégration des méthodes et l'optimisation des appels de méthodes en général.

2.1.1 L'analyse par interprétation abstraite

La technique d'analyse par *interprétation abstraite*, technique formalisée pour la première fois par Cousot et Cousot [13], offre un cadre théorique conceptuellement simple pour le développement d'analyses statiques. Son principal intérêt est de relier formellement le résultat de l'analyse convoitée à la sémantique du langage de programmation.

L'idée de base de l'interprétation abstraite est simple. Prenons un programme p écrit dans un langage L dont la sémantique opérationnelle est donnée par $\mathcal{S}$. Cette sémantique implémente un interprète pour le langage L. $\mathcal{S}(\pi, \rho)$ est donc la valeur retournée par le programme π lorsqu'il est exécuté dans un environnement initial ρ.

Supposons maintenant que l'on veuille calculer une (autre) propriété du programme π. Cela peut être fait au moyen d'une *sémantique abstraite non-standard*

$\hat{\mathcal{S}}$ pour L. La propriété du programme π sera donc donnée par $\hat{\mathcal{S}}(\pi, \hat{\rho})$, où $\hat{\rho}$ est un environnement "abstrait" calculé à partir de ρ. Cette nouvelle sémantique peut être calculée, tout comme dans le cas de $\mathcal{S}$, par une forme d'interprétation.

Une sémantique abstraite non-standard doit posséder certaines qualités afin d'être utile. Avant tout, il faut qu'elle soit calculable : on doit pouvoir prouver que le calcul de l'analyse au moyen de cette sémantique termine dans tous les cas. Un compilateur ne peut se permettre de boucler à l'infini lors de l'analyse d'un programme. Afin de satisfaire ce critère, il est parfois nécessaire de dériver cette sémantique en approximant une sémantique non-standard *exacte* $\mathcal{S}^*$. Ceci implique que l'analyse statique ainsi dérivée n'est pas exacte, mais *conservatrice*. Évidemment, il faut que la sémantique $\hat{\mathcal{S}}$ soit reliée d'une certaine manière à $\mathcal{S}^*$. Les détails techniques de cette relation débordent du cadre de la présente thèse.

Exemple. L'exemple classique illustrant la technique d'interprétation abstraite consiste à déterminer le signe d'une expression arithmétique quelconque. La sémantique standard $\mathcal{S}$ d'une expression E, notée $\mathcal{S}[\![E]\!]$, calcule la valeur de l'expression (notons au passage que la présence d'un environnement est ici superflue). La sémantique non-standard exacte, $\mathcal{S}^*[\![E]\!]$, a pour valeur un élément de l'ensemble $\{\oplus, \odot, \ominus\}$. $\mathcal{S}^*$ est calculée de la manière suivante :

$$\mathcal{S}^*[\![E]\!] = \begin{cases} \oplus, & \text{si } \mathcal{S}[\![E]\!] > 0 \\ \odot, & \text{si } \mathcal{S}[\![E]\!] = 0 \\ \ominus, & \text{si } \mathcal{S}[\![E]\!] < 0 \end{cases}$$

Par exemple,

$$\mathcal{S}[\![(3+0) \times 4]\!] = 12$$

et

$$\mathcal{S}^*[\![(3+0) \times 4]\!] = \oplus.$$

Supposons maintenant (à tort mais pour les besoins de l'exposé) que le calcul de $\mathcal{S}$ ne soit pas calculable dans tous les cas ou soit trop coûteux en temps de calcul (ou

en espace mémoire). Il faut alors trouver une sémantique non-standard $\hat{\mathcal{S}}$ calculable et approximant $\mathcal{S}^*$. Une possiblité est de calculer, pour une expression E, non pas son signe exact mais un ensemble de signes possibles. Bien entendu, $\hat{\mathcal{S}}$ doit satisfaire

$$\forall E \in \text{Expr} \qquad \mathcal{S}^*[\![E]\!] \in \hat{\mathcal{S}}[\![E]\!]$$

afin d'être conservatrice, c'est-à-dire que l'approximation peut être imprécise mais qu'elle doit toujours contenir la valeur cherchée. Dans le cas qui nous intéresse, on pourrait définir $\hat{\mathcal{S}}$ par

$$\begin{aligned}
\hat{\mathcal{S}}[\![k]\!] &= \{\mathcal{S}^*[\![k]\!]\} \\
\hat{\mathcal{S}}[\![(E)]\!] &= \hat{\mathcal{S}}[\![E]\!] \\
\hat{\mathcal{S}}[\![E_1 \times E_2]\!] &= \hat{\mathcal{S}}[\![E_1]\!] \,\hat{\times}\, \hat{\mathcal{S}}[\![E_2]\!] \\
\hat{\mathcal{S}}[\![E_1 + E_2]\!] &= \hat{\mathcal{S}}[\![E_1]\!] \,\hat{+}\, \hat{\mathcal{S}}[\![E_2]\!] \\
&\text{etc.}
\end{aligned}$$

où $\hat{+}, \hat{*}$, etc., seraient définis adéquatement. Ainsi, on aurait

$$\begin{aligned}
\hat{\mathcal{S}}[\![(3+0)\times 4]\!] &= (\{\oplus\} \,\hat{+}\, \{\odot\}) \,\hat{\times}\, \{\oplus\} \\
&= (\{\oplus\}) \,\hat{\times}\, \{\oplus\} \\
&= \{\oplus\}
\end{aligned}$$

ce qui vérifie bien

$$\mathcal{S}^*[\![(3+0)\times 4]\!] = \oplus \in \hat{\mathcal{S}}[\![(3+0)\times 4]\!]$$

□

Plusieurs analyses statiques ont été développées dans ce cadre théorique, dont de nombreuses pour des langages fonctionnels d'ordre supérieur. Mycroft [41] a proposé une analyse de "nécessité" (*strictness analysis*) pour les langages fonctionnels à évaluation paresseuse. Cette analyse permet de déterminer les arguments qu'une

fonction doit nécessairement évaluer. Hudak [34] 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 [50, 51] et de Ayers [5]. 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.

Mentionnons aussi Serrano [48], qui a montré l'utilité de telles analyses pour la compilation des langages fonctionnels d'ordre supérieur dans le cas d'un compilateur réel. Il a implanté la 0CFA de Shivers ainsi qu'une analyse de types basée sur la 0CFA dans un compilateur Scheme et SML produisant du C [24, 49]. 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 de la première itération.

D'autres analyses par interprétation abstraite ont été développées pour des langages de programmation logique. Par exemple, Debray et Warren [17] décrivent une analyse permettant de déterminer les *modes* possibles des clauses de Horn constituant un programme Prolog, connue aussi sous le nom d'inférence automatique des modes. Puisque les programmes logiques n'ont pas de notion de variables d'entrée

ou de sortie, il importe de déterminer comment sont instanciés les arguments d'un prédicat afin de générer du code plus efficace.

Enfin, un certain nombre de travaux, dont [29], ont présenté des analyses *ensemblistes* basées sur la résolution de contraintes (en anglais, *set-based analysis*). Ces analyses opèrent en deux temps : d'abord, une phase de *dérivation* construit un ensemble de contraintes liant les différentes expressions du programme; la deuxième étape, de *résolution*, consiste à résoudre le système d'équations ainsi construit. Les analyses développées à l'aide de cette technique permettent en général de calculer les mêmes propriétés que les analyses par interprétation abstraite. Leur attrait premier est de permettre la visualisation des différentes étapes du calcul, ce qui en fait un outil très attrayant au plan pédagogique [28].

2.2 Compilation abstraite

Comme l'a fait remarqué Ayers dans [5], l'interprétation abstraite a souffert d'un problème d'image à ses débuts, principalement dû à des implantations inefficaces. En effet, les sémantiques abstraites non-standards ainsi développées étaient réalisées naïvement à l'aide d'interprètes. Pour des analyses requérant un grand nombre d'itérations, le coût associé à la couche d'interprétation peut être élevé.

C'est ici qu'entre en jeu la compilation abstraite, afin de réduire au strict minimum cette couche d'interprétation. Il suffit de transposer dans le contexte qui nous intéresse ici la relation qui existe entre un interprète $\mathcal{S}$ pour un langage L et un compilateur $\mathcal{C}$ pour ce même langage. À partir d'un interprète abstrait $\hat{\mathcal{S}}$, on dérive un compilateur $\hat{\mathcal{C}}$ qui produit, lorsqu'on lui donne un programme π, un *programme d'analyse* $\hat{\mathcal{C}}(\pi)$ satisfaisant l'équation

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

Autrement dit, ce dernier programme, lorsqu'exécuté dans un environnement $\hat{\rho}$, produit le même résultat que l'analyse du programme π par $\hat{\mathcal{S}}$ directement et ce, dans un

temps (on l'espère) plus court. Comme je le montrerai, l'évaluation partielle permet de formaliser le lien entre un interprète et son compilateur associé.

2.2.1 Compilation par évaluation partielle

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 :

$$[Mix(\langle p, d_1\rangle)](d_2) = p_{d_1}(d_2) = p(\langle d_1, d_2\rangle). \tag{2.1}$$

Une propriété intéressante est que l'évaluateur partiel peut être appliqué à lui-même :

$$p_{\text{gen}} = Mix(\langle Mix, p\rangle). \tag{2.2}$$

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}. Cette dernière équation fait partie de ce que l'on nomme communément les équations de Futamura [12] et elle est appelée l'extension génératrice de p.

En plus d'être utilisée comme technique d'optimisation de programme, l'évaluation partielle peut aussi être utilisée pour la dérivation automatique de compilateurs (du moins théoriquement). Considérons l'équation 2.2 lorsque p est un interprète pour un langage de programmation L, d_1 est un programme écrit dans le langage L et d_2 est un environnement initial dans lequel d_1 doit être exécuté. p_{gen} est donc un compilateur pour le langage L.

Dans le cas qui nous intéresse, remplaçons donc p par $\hat{\mathcal{S}}$ dans l'équation 2.2, appliquons p_{gen} sur π et simplifions en utilisant l'équation 2.1 :

$$\begin{aligned} p_{\text{gen}}(\pi) &= [Mix(\langle Mix, \hat{\mathcal{S}}\rangle)](\pi) \\ &= Mix_{\hat{\mathcal{S}}}(\pi) && \text{par (2.1)} \\ &= Mix(\langle \hat{\mathcal{S}}, \pi\rangle) && \text{par (2.1).} \end{aligned}$$

Il n'est pas difficile de montrer que $p_{\text{gen}}(\pi)$ est précisement le résultat de la compilation abstraite de π. En effet, appliquons $p_{\text{gen}}(\pi)$ sur $\hat{\rho}$:

$$\begin{aligned}
[p_{\text{gen}}(\pi)](\hat{\rho}) &= [\mathit{Mix}(\langle \hat{\mathcal{S}}, \pi \rangle)](\hat{\rho}) \\
&= \hat{\mathcal{S}}_{\pi}(\hat{\rho}) && \text{par (2.1)} \\
&= \hat{\mathcal{S}}(\pi, \hat{\rho}) && \text{par (2.1).}
\end{aligned}$$

2.3 Une expérience avec Similix

Cette section décrit une expérience simple exploitant ces idées, expérience effectuée à l'aide du générateur d'extensions génératrices Similix [7], un évaluateur partiel pour un large sous-ensemble de Scheme destiné à la génération automatique de compilateurs. L'expérience en question avait pour but d'évaluer la difficulté de dériver automatiquement, à l'aide des techniques d'évaluation partielle, un compilateur abstrait à partir d'un interprète abstrait .

L'annexe A présente le code Scheme d'une réalisation très naïve d'une itération d'une analyse de flux de données vers l'avant (*forward flow analysis*) classique telle que décrite dans [1, 26]. Brièvement, à chaque bloc de base B d'un graphe de flux de données, on associe quatre ensembles: $\mathit{in}[B]$ contient les informations qui sont disponibles tout juste à l'entrée de B, $\mathit{out}[B]$ contient les informations qui quittent le bloc B, $\mathit{gen}[B]$ contient les informations qui sont "générées" par le bloc B et $\mathit{kill}[B]$ contient les informations qui sont "détruites" par B.

De ces ensembles, seuls $\mathit{gen}[B]$ et $\mathit{kill}[B]$ peuvent être calculés sans s'occuper de l'interaction des différents blocs de base. Par contre, $\mathit{in}[B]$ et $\mathit{out}[B]$ doivent être calculés lors de l'analyse de flux. Dans le cas d'une analyse vers l'avant, ces deux ensembles sont calculés par les équations

$$\begin{aligned}
\mathit{in}[B] &= \bigcup_{P \in \text{pred}(B)} \mathit{out}[P] \\
\mathit{out}[B] &= \mathit{gen}[B] \cup (\mathit{in}[b] - \mathit{kill}[B])
\end{aligned}$$

```
((loadt "dfa.adt")
 (define (dfa-0 sol_0)
   (my-vector-set! sol_0 0 (join '(1) (diff '() '(3))))
   (let* ((g_1 (my-vector-ref sol_0 0))
          (g_2 (my-vector-ref sol_0 3)))
     (my-vector-set! sol_0 1 (join '(2 3) (diff (join g_2 (join g_1 '()))
                                                '(1 5))))
     (let ((g_3 (my-vector-ref sol_0 1)))
       (my-vector-set! sol_0 2 (join '(4 5) (diff (join g_3 '()) '(2))))
       (let ((g_4 (my-vector-ref sol_0 2)))
         (my-vector-set! sol_0 3 (join '() (diff (join g_4 '()) '())))
         sol_0)))))
```

Figure 2.3. Programme d'analyse obtenu par évaluation partielle

Si on génère l'extension génératrice du programme de l'annexe A à l'aide de Similix pour obtenir un compilateur abstrait en utilisant la commande suivante :

```
> (cogen 'dfa '(static static static static dynamic) "dfa.sim")
```

et qu'on génère ensuite le programme d'analyse d'un graphe du flux de données semblable à celui de la figure 2.2 à l'aide des commandes suivantes (les trois premières lignes définissant le graphe de flux de données, les fonctions *gen* et *kill*) :

```
> (define adj  (vector '() '(0 3) '(1) '(2)))
> (define gen  (vector '(1) '(2 3) '(4 5) '()))
> (define kill (vector '(3) '(1 5) '(2)  '()))
> (pp (comp (list adj 4 gen kill '***)))
```

on obtient le résultat de la figure 2.3.

Outre quelques appels pouvant être évalués davantage (comme `(diff '() '(3))` à la ligne 3), il est intéressant de constater que ce programme d'analyse est optimal, au sens où la couche interprétation (dans ce cas-ci le parcours du graphe) a complètement été éliminée.

2.4 Constatations

Malgré les résultats que je viens de montrer, la dérivation de compilateurs abstraits en utilisant directement les techniques d'évaluation partielle n'est pas pratique. Il y a plusieurs raisons à cela :

1. En général, la structure du programme généré par le compilateur abstrait est souvent trop simple pour nécessiter toute la lourde machinerie des générateurs de compilateurs par évaluation partielle. En effet, j'ai observé que le processus de compilation abstraite consiste généralement en une transformation très simple de programme suivie d'une phase d'optimisation du programme abstrait.

2. Ces optimisations simples et souvent payantes sont possibles sur le programme généré mais ne pourraient être effectuées par le compilateur abstrait ainsi dérivé.

3. Le compilateur abstrait serait certainement beaucoup plus lent qu'une version *ad hoc* étant donné qu'il lui faudra effectuer des analyses de moments de liaison afin de déterminer quelles sous-expressions du programme source peuvent être évaluées directement, évaluer ces dernières, générer le code abstrait, etc.

4. Le compilateur abstrait ne générerait qu'un programme source, qu'il faudra ensuite passer à un autre compilateur, ajoutant à l'inefficacité.

Le chapitre qui suit illustre ces différents problèmes dans le contexte de la réalisation d'un compilateur abstrait pour l'analyse des fermetures d'un langage fonctionnel.

Chapitre 3

ÉTUDE DE CAS – LA *0cfa*

Je montre ici l'application de la compilation abstraite à l'analyse des fermetures de niveau 0, la *0cfa*, telle que développée par Shivers. Comme je l'ai déjà mentionné, cette analyse est cruciale pour l'optimisation des langages fonctionnels. Cette analyse sera ici décrite pour une variante de Scheme que je nommerai CPS-Scheme. Les conclusions tirées de cette expérience sont à la base de l'architecture de compilateur présentée au prochain chapitre.

3.1 Analyse de fermetures par interprétation abstraite

La figure 3.1 donne la syntaxe abstraite de CPS-Scheme. Il s'agit d'un langage fonctionnel en forme CPS (pour *Continuation Passing Style*), c'est-à-dire que toutes les continuations sont explicites (tout programme en forme directe peut être CPS-converti). J'assumerai que tous les programmes écrits dans ce langage ont été préalablement α-convertis. La forme CPS est utilisée uniquement dans le but de simplifier la présentation de l'analyse. En effet, il est alors possible de traiter toutes les formes spéciales, comme `if`, au même niveau que toutes les autres fonctions primitives. De plus, tous les résultats intermédiaires, correspondant aux sous-expressions d'une expression composée, sont liés à des variables. Puisque seuls des λ-expressions, des variables ou des primitives peuvent apparaître en position opérateur d'un appel de fonction, le problème de la *0cfa* est équivalent à celui de trouver, pour chaque variable v apparaissant dans le programme, l'ensemble des λ-expressions dont la valeur peut être liée à v lors de l'exécution du programme.

La figure 3.2 donne la signature de chacune des fonctions de la sémantique abs-

$$
\begin{array}{lcl}
\Pi & \in & \mathit{Prog} \\
C & \in & \mathit{Call} \\
L & \in & \mathit{Lam} \\
F & \in & \mathit{Fun} \\
A & \in & \mathit{Arg} \\
V & \in & \mathit{Var} \\
K & \in & \mathit{Const} \\
P & \in & \mathit{Prim}\ \textit{(fonctions primitives:}\ \texttt{if}, \texttt{+}, \textit{etc.)}
\end{array}
$$

$$
\begin{array}{lcl}
\Pi & ::= & C \\
C & ::= & (F\ A_1\ \ldots\ A_n) \\
 & & \mid (\texttt{letrec}\ ((V_1\ L_1)\ \cdots\ (V_n\ L_n))\ C) \\
L & ::= & (\lambda\ (V_1\ \ldots\ V_n)\ C) \\
F & ::= & L \mid V \mid P \\
A & ::= & K \mid V \mid L
\end{array}
$$

Figure 3.1. La syntaxe abstraite de CPS-Scheme

$$
\begin{array}{rcl}
\textit{0cfa-program} & : & \mathit{Prog} \times \widehat{\mathit{Env}} \to \widehat{\mathit{Env}} \\
\textit{0cfa-call} & : & \mathit{Call} \times \widehat{\mathit{Env}} \to \widehat{\mathit{Env}} \\
\textit{0cfa-app} & : & \mathit{Fun} \times \mathit{Arg}^* \times \widehat{\mathit{Env}} \to \widehat{\mathit{Env}} \\
\textit{0cfa-abstract-app} & : & 2^{\mathit{Lam}} \times \mathit{Arg}^* \times \widehat{\mathit{Env}} \to \widehat{\mathit{Env}} \\
\textit{0cfa-args} & : & \mathit{Arg}^* \times \widehat{\mathit{Env}} \to \widehat{\mathit{Env}} \\
\textit{0cfa-prim} & : & \mathit{Prim} \times \mathit{Arg}^* \times \widehat{\mathit{Env}} \to \widehat{\mathit{Env}} \\
\mathit{lookup} & : & \mathit{Arg} \times \widehat{\mathit{Env}} \to \widehat{\mathit{Env}}
\end{array}
$$

$$
\widehat{\mathit{Env}} = \mathit{Var} \to 2^{\mathit{Lam}}
$$

Figure 3.2. Signature des fonctions sémantiques

$$\text{0cfa-program}(p, \sigma) = \text{0cfa-call}(p, \sigma)$$

$$\begin{aligned}
&\text{0cfa-call}([\![(f\ a_1 \ldots a_n)]\!], \sigma) = \\
&\qquad \text{0cfa-app}(f, \langle a_1, \ldots, a_n \rangle, \text{0cfa-args}(\langle a_1, \ldots, a_n \rangle, \sigma)) \\
&\text{0cfa-call}([\![\texttt{(letrec ((}v_1\ l_1\texttt{) } \cdots \texttt{ (}v_n\ l_n\texttt{)) } c\texttt{)}]\!], \sigma) = \\
&\qquad \textbf{let}\ \ \sigma' = \sigma \sqcup [v_1 \mapsto \{l_1\}] \sqcup \cdots \sqcup [v_n \mapsto \{l_n\}] \\
&\qquad\qquad \sigma'' = \text{0cfa-args}(\langle l_1, \ldots, l_n \rangle, \sigma') \\
&\qquad\ \textbf{in}\ \text{0cfa-call}(c, \sigma'')
\end{aligned}$$

$$\begin{aligned}
&\text{0cfa-app}(f, \langle a_1, \ldots, a_n \rangle, \sigma) = \\
&\qquad \textbf{cond} \\
&\qquad\quad \mathit{isVar}(f): \\
&\qquad\qquad \text{0cfa-abstract-app}(\sigma(f), \langle a_1, \ldots, a_n \rangle, \sigma) \\
&\qquad\quad \mathit{isPrim}(f): \\
&\qquad\qquad \text{0cfa-prim}(f, \langle a_1, \ldots, a_n \rangle, \sigma) \\
&\qquad\quad \mathit{isLam}(f): \\
&\qquad\qquad \textbf{let}\ \ \sigma' = \sigma \sqcup\ [f \downarrow_{\text{formals}_1} \mapsto \text{lookup}(a_1, \sigma)] \sqcup \cdots \\
&\qquad\qquad\qquad\qquad \cdots \sqcup [f \downarrow_{\text{formals}_n} \mapsto \text{lookup}(a_n, \sigma)] \\
&\qquad\qquad\ \textbf{in}\ \text{0cfa-call}(f \downarrow_{\text{body}}, \sigma')
\end{aligned}$$

$$\begin{aligned}
&\text{0cfa-abstract-app}(\emptyset, \langle a_1, \ldots, a_n \rangle, \sigma) = \sigma \\
&\text{0cfa-abstract-app}(S, \langle a_1, \ldots, a_n \rangle, \sigma) = \\
&\qquad \textbf{let}\ \ l = \text{some member of S} \\
&\qquad\qquad \sigma' = \sigma \sqcup\ [l \downarrow_{\text{formals}_1} \mapsto \text{lookup}(a_1, \sigma)] \sqcup \cdots \\
&\qquad\qquad\qquad\qquad \cdots \sqcup [l \downarrow_{\text{formals}_n} \mapsto \text{lookup}(a_n, \sigma)] \\
&\qquad\ \textbf{in}\ \text{0cfa-abstract-app}(S - \{l\}, \langle a_1, \ldots, a_n \rangle, \sigma')
\end{aligned}$$

$$\begin{aligned}
&\text{0cfa-args}(\langle\rangle, \sigma) = \sigma \\
&\text{0cfa-args}(\langle a_1, \ldots, a_n \rangle, \sigma) = \\
&\qquad \textbf{let}\ \ \sigma' = \ \textbf{if}\ \mathit{isLam}(a_1) \\
&\qquad\qquad\qquad\qquad\ \textbf{then}\ \text{0cfa-call}(a_1 \downarrow_{\text{body}}, \sigma) \\
&\qquad\qquad\qquad\qquad\ \textbf{else}\ \sigma \\
&\qquad\ \textbf{in}\ \text{0cfa-args}(\langle a_2, \ldots, a_n \rangle, \sigma')
\end{aligned}$$

$$\begin{aligned}
&\text{0cfa-prim}([\![\texttt{+}]\!], \langle a_1, \ldots, a_3 \rangle, \sigma) = \text{0cfa-args}(\langle a_1, \ldots, a_3 \rangle, \sigma) \\
&\text{0cfa-prim}([\![\texttt{if}]\!], \langle a_1, \ldots, a_3 \rangle, \sigma) = \text{0cfa-args}(\langle a_1, \ldots, a_3 \rangle, \sigma) \\
&\ldots
\end{aligned}$$

Figure 3.3. La 0cfa par interprétation abstraite

$\text{lookup}(e, \sigma) =$
cond
isConst(e) : $\emptyset$
isVar(e) : $\sigma(e)$
isLam(e) : $\{e\}$

Figure 3.4. La 0cfa par interprétation abstraite (suite)

traite non-standard des figures 3.3 et 3.4. J'assumerai ici que tous les programmes fournis à cette analyse sont syntaxiquement valides afin de n'avoir pas à traiter les cas d'erreur. La notation suivante est utilisée : $l \downarrow_{\text{formals}_i}$ est le $i$ème paramètre formel de la procédure l; $l \downarrow_{\text{body}}$ est le corps de la procédure l (un site d'appel). Un environnement abstrait est une fonction dont le domaine est le domaine syntaxique *Var*, qui dénote l'ensemble des variables d'un programme, et qui produit un résultat dans 2^{Lam}, où *Lam* est l'ensemble des λ-expressions d'un programme. L'environnement vide est noté $\sigma_\perp$ ($\sigma_\perp(v) = \emptyset$ pour tout v) et $[v \mapsto S]$ dénote l'environnement σ tel que

$$\sigma(x) = \begin{cases} S & \text{si } x = v \\ \emptyset & \text{sinon} \end{cases}$$

De plus, un opérateur $\sqcup$ existe pour combiner deux environnement en un seul :

$$(\sigma \sqcup \sigma')(v) = \sigma(v) \cup \sigma'(v).$$

Enfin, un environnement σ est mieux défini qu'un environnement σ', noté $\sigma' \sqsubseteq \sigma$ si et seulement si

$$\sigma'(x) \subseteq \sigma(x) \quad \text{pour tout } x.$$

Il est facile de montrer que $\sigma \sqsubseteq \sigma \sqcup \sigma'$ et que $\sigma' \sqsubseteq \sigma \sqcup \sigma'$.

L'analyse *0cfa* d'un programme p est calculée en trouvant le "plus petit" environnement σ tel que $\sigma =$ *0cfa-program*(p, σ). On dit alors que σ est le plus petit *point fixe* de la fonction $\lambda x.$*0cfa-program*(p, x). Ce point fixe peut être trouvé itérativement

par approximation successive en utilisant les équations suivantes :

$$\begin{aligned} \sigma_0 &= \sigma_\perp \\ \sigma_{i+1} &= \textit{0cfa-program}(p, \sigma_i) \end{aligned}$$

Afin de s'assurer de la terminaison de l'algorithme, il faut satisfaire deux conditions :

1. $\sigma_0, \sigma_1, \ldots$ forme une *chaîne ascendante*, c'est-à-dire que $\sigma_0 \sqsubseteq \sigma_1 \sqsubseteq \cdots$.

2. Il existe un $i \geq 0$ tel que $\sigma_i = \sigma_{i+1}$.

La première condition est satisfaite puisque l'algorithme ne fait qu'ajouter des informations à l'environnement abstrait (avec l'opérateur $\sqcup$). La seconde condition est aussi aisément remplie puisqu'il y a seulement, pour un programme donné p, un nombre fini de λ-expressions pouvant être liées à toute variable (i.e. dans le pire case σ_i est l'ensemble universel).

La *0cfa* est généralement réalisée à l'aide d'un tel algorithme. Par exemple, [48] décrit l'analyse effectuée par le compilateur Bigloo. C'est essentiellement celle qui vient d'être décrite. Elle est aussi très proche de celle décrite par Shivers dans [51].

Lorsque de nombreuses itérations sont nécessaires afin d'atteindre le point fixe, une quantité non-négligeable de travail n'ayant pas un impact direct sur le résultat de l'analyse est effectué : chaque itération demande de traverser l'arbre de syntaxe, examinant chaque noeud afin de savoir si c'est une application, une abstraction, etc. C'est le coût de l'interprétation. Lorsque l'on examine attentivement l'algorithme de la figure 3.3, on s'aperçoit que seules trois fonctions peuvent modifier le résultat de l'analyse : *0cfa-call* appliquée à la forme spéciale `letrec`, *0cfa-app* lorsque f est une λ-expression et *0cfa-abstract-app*.

3.2 Compilation abstraite – première tentative

Il serait donc intéressant de se "souvenir" uniquement des endroits qui affectent directement le résultat final de l'analyse. Considérons le programme en forme CPS

```
((λ1 (apply k1)
    (apply (λ2 (x1 k2)
               (+ x1 1 k2))
           (λ4 (t2)
              (apply t2 (λ5 (t3) (t3 2 k1))))))

 (λ6 (f k3)
    (k3 (λ7 (x2 k4)
           (f x2 k4))))

 tl-cont)
```

Figure 3.5. Un programme en forme CPS.

de la figure 3.5, où chaque λ-expression a été numérotée de 1 à 7. Ce programme implémente une version currifiée de la fonction `apply`, une fontion telle que `((apply f) x)` = `(f x)`. Ce programme calcule l'équivalent de `((apply (apply (λ (x) (+ x 1)))) 2)`.

En examinant attentivement le programme, on peut déterminer les sites d'appel qui propagent l'information de flux de contrôle. L'appel $(\lambda_1\ \lambda_6$ `tl-cont`$)$ ajoutera λ_6 à σ(`apply`) et `tl-cont` à σ(`k1`). C'est le cas le plus simple. Considérons maintenant le site d'appel interne `(t3 2 k1)` dans λ_5. L'analyse itérera sur tous les $\lambda_i \in \sigma$(`t3`) et ajoutera σ(`k1`) à $\sigma(\lambda_i \downarrow_{\text{formals}_2})$. En revanche, l'appel `(+ x1 1 k2)` n'ajoute aucune information et n'a donc, par le fait même, aucun impact sur le résultat final.

On note que seul les sites d'appel où l'information est propagée sont utiles au calcul de l'analyse. Une façon d'implémenter cette analyse serait de traverser d'abord l'arbre de syntaxe abstrait et de sauver les sites d'appel intéressants dans une structure de données qui serait ensuite parcourue à chaque itération. Malheureusement, le parcours de cette structure de données peut être considéré comme une couche d'interprétation, bien que moins coûteuse que la première.

La compilation abstraite de cette analyse va permettre de remplacer cette struc-

$$\text{comp-program}(p) = \text{comp-call}(p)$$

$$\begin{array}{l}
\text{comp-call}([\![(f\ a_1 \ldots a_n)]\!]) = \\
\quad \textbf{let}\ \ C_1 = \text{comp-args}(\langle a_1, \ldots, a_n \rangle) \\
\qquad\ \ C_2 = \text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) \\
\quad\ \ \textbf{in}\ [\![(\lambda\ (\sigma)\ \ (C_2\ \ (C_1\ \sigma)))]\!]
\end{array}$$

$$\begin{array}{l}
\text{comp-call}([\![(\texttt{letrec}\ ((v_1\ \ l_1)\ \cdots\ (v_n\ \ l_n))\ \ c)]\!]) = \\
\quad \textbf{let}\ \ C_1 = \text{comp-call}(c) \\
\qquad\ \ C_2 = \text{comp-args}(\langle l_1, \ldots, l_n \rangle) \\
\quad\ \ \textbf{in}\ [\![(\lambda\ (\sigma)\ \ (C_1\ \ (C_2\ \ \sigma \sqcup [v_1 \mapsto \{l_1\}] \sqcup \cdots \sqcup [v_n \mapsto \{l_n\}])))]\!]
\end{array}$$

$$\begin{array}{l}
\text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{cond} \\
\qquad \textit{isVar}(f): \\
\qquad\quad [\![(\lambda\ (\sigma) \\
\qquad\qquad (\texttt{0cfa-abstract-app}\ (\sigma\ \ f)\ \ \sigma\ \ a_1\ \ \ldots\ \ a_n))]\!] \\
\qquad \textit{isPrim}(f): \\
\qquad\quad \text{comp-prim}(f, \langle a_1, \ldots, a_n \rangle) \\
\qquad \textit{isLam}(f): \\
\qquad\quad \textbf{let}\ \ C = \text{comp-call}(f \downarrow_{\text{body}}) \\
\qquad\qquad \textbf{in}\ \ [\![(\lambda\ (\sigma) \\
\qquad\qquad\quad (C\ \ \sigma \sqcup_{i=1\ldots n} [f \downarrow_{\texttt{formals}_i} \mapsto (\texttt{lookup}\ \ a_i\ \ \sigma)]))]\!]
\end{array}$$

$$\begin{array}{l}
\text{comp-args}(\langle\rangle) = [\![(\lambda\ (\sigma)\ \ \sigma)]\!] \\
\text{comp-args}(\langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{if}\ \textit{isLam}(a_1) \\
\qquad \textbf{then}\ \ \textbf{let}\ \ C_1 = \text{comp-call}(a_1 \downarrow_{\text{body}}) \\
\qquad\qquad\qquad\ \ C_2 = \text{comp-args}(\langle a_2, \ldots, a_n \rangle) \\
\qquad\qquad\ \ \textbf{in}\ [\![(\lambda\ (\sigma)\ \ (C_2\ \ (C_1\ \ \sigma)))]\!] \\
\qquad \textbf{else}\ \text{comp-args}(\langle a_2, \ldots, a_n \rangle)
\end{array}$$

$$\begin{array}{l}
\text{comp-prim}([\![\texttt{+}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\text{comp-prim}([\![\texttt{if}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\ldots
\end{array}$$

Figure 3.6. Algorithme de compilation de la *0cfa*

ture de données par la structure de contrôle du programme d'analyse. La seule forme d'interprétation restante sera au niveau du microprocesseur, mais comme elle est inévitable, je n'en tiendrai pas compte. La figure 3.6 donne une première version de l'algorithme de compilation abstraite pour la *0cfa*. Le programme d'analyse est généré dans un langage inspiré de Scheme/Lisp. La fonction *comp-program* prend en entrée un programme p et produit le programme d'analyse p'. Je suppose ici que *0cfa-abstract-app* et *lookup* peuvent être liés d'une manière ou d'une autre au programme d'analyse. Il suffit alors de trouver le point fixe de p' par la technique d'approximations successives, décrite plus haut, afin de trouver un environnement abstrait tel que $\sigma = p'(\sigma)$.

Exemple. Considérons le programme suivant :

```
((λ1 (f c1)
    ((λ2 (x c2)
       (f x c2))
     2
     c1))
 (λ3 (y c3)
    (+ y 1 c3))
 tl-cont)
```

Le compilateur abstrait produit le programme d'analyse suivant :

```
   (λ (σ)
     ((λ (σ)
        ((λ (σ)
           ((λ (σ)
              ((λ (σ)
                 ((λ (σ) (0cfa-abstract-app (σ f) σ x c2))
                  ((λ (σ) σ) σ)))
(1)
(2)               σ ⊔ [x ↦ (lookup 2 σ)] ⊔ [c2 ↦ (lookup c1 σ)]))
               ((λ (σ) σ)
                σ)))
(3)          σ ⊔ [f ↦ (lookup λ3 σ)] ⊔ [c1 ↦ (lookup tl-cont σ)]))
           ((λ (σ) σ)
            σ)))
        ((λ (σ)
           ((λ (σ)
              ((λ (σ) σ)
```

```
        ((λ (σ) σ)
         σ)))
      ((λ (σ) σ)
       σ)))
   σ)))
```

On remarque aisément que seules les lignes *(1)*, *(2)* et *(3)* apportent une contribution non-nulle à l'environnement abstrait. Il y a aussi un certain nombre de calculs qui sont inutiles.

Deux optimisations simples peuvent aider à réduire la quantité de calculs à faire à chaque itération. La première optimisation consiste à éliminer tous les appels à la fonction identité $(\lambda\ (\sigma)\ \sigma)$ en effectuant des η-réductions. Ceci peut être fait à faible coût en ajoutant simplement des conditionnelles additionelles au compilateur abstrait. En supposant que *Id-Funct?* est vrai si son argument est le code de la fonction identité, la fonction *comp-call* devient :

comp-call$([\![(f\ \ a_1 \ldots a_n)]\!]) =$
  let $C_1 =$ *comp-args*$(\langle a_1, \ldots, a_n \rangle)$
      $C_2 =$ *comp-app*$(f, \langle a_1, \ldots, a_n \rangle)$
    in **if** *Id-funct?* (C_1)
        then C_2
        else **if** *Id-funct?* (C_2)
            then C_1
            else $[\![(\lambda\ \ (\sigma)\ \ (C_2\ \ (C_1\ \ \sigma)))]\!]$

La seconde optimisation provient du comportement de la fonction `lookup`. Lorsqu'appliquée à une constante, elle retourne l'ensemble vide; pour une λ-expression, elle retourne le singleton contenant uniquement cette expression. Cette observation permet d'opérer l'optimisation suivante. D'abord, on peut éliminer toutes les contributions de la forme $[v \mapsto$ `(lookup` $c\ \sigma)]$, où v est une variable et c est une constante. De plus, on peut enlever les environnements de la forme $[v \mapsto \{\lambda_k\}]$ et les ajouter directement à l'environnement initial σ_0 (je nommerai ce nouvel ensemble initial σ_0'). Ceci

permet de sauver une itération et, plus important encore, de simplifier le mécanisme de recherche dans l'environnement.

Lorsque ces deux optimisations sont réalisées dans le compilateur abstrait, le programme d'analyse de l'exemple précédent devient :

```
(λ (σ)
  ((λ (σ)
     ((λ (σ) (0cfa-abstract-app (σ f) σ x c2))
      σ ⊔ [c2 ↦ (lookup c1 σ)]))
   σ ⊔ [c1 ↦ (lookup tl-cont σ)]))
```

En prenant $\sigma'_0 = [\texttt{f} \mapsto \{\lambda_3\}]$ (tel que produit par la deuxième optimisation), on trouve $\sigma_1 = [\texttt{f} \mapsto \{\lambda_3\}]$, qui est le point fixe de cette fonction. Il aura donc fallu une seule itération pour le trouver.

3.3 Génération de fermetures

L'algorithme de compilation abstraite présenté précédemment génère un code très compact, mais il demeure insatisfaisant. La couche d'interprétation n'apparaît plus dans le code généré. Malgré tout, le programme d'analyse doit être exécuté d'une manière ou d'une autre, requérant une couche d'interprétation à un autre niveau. Lorsqu'une procédure d'interprétation propre au langage de programmation, comme la fonction `eval` de Scheme, est utilisée, des expérimentations ont montré qu'il demeure plus efficace d'utiliser directement l'interprétation abstraite. Mais il est possible de faire mieux.

Plusieurs langages de programmation fonctionnelle permettent à l'utilisateur de créer de nouvelles fonctions à l'exécution via des λ-expressions. Lorsque ces expressions sont évaluées, elles produisent des *fermetures*, c'est-à-dire des fonctions qui retiennent leur environnement courant.

Il est possible d'utiliser les fermetures afin de surmonter le coût de l'interprétation. L'idée est de représenter une expression compilée par une fermeture. Lorsque cette

$$\text{comp-program}(p) = \text{comp-call}(p)$$

$$\begin{array}{l}
\text{comp-call}([\![(f\ \ a_1 \ldots a_n)]\!]) = \\
\quad \textbf{let}\ \ C_1 = \text{comp-args}(\langle a_1, \ldots, a_n \rangle) \\
\qquad\ \ C_2 = \text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) \\
\quad\ \ \textbf{in}\ \lambda\sigma.C_2(C_1(\sigma))
\end{array}$$

$$\begin{array}{l}
\text{comp-call}([\![\texttt{(letrec ((}v_1\ \ l_1\texttt{) } \cdots \texttt{ (}v_n\ \ l_n\texttt{)) } c\texttt{)}]\!]) = \\
\quad \textbf{let}\ \ C_1 = \text{comp-call}(c) \\
\qquad\ \ C_2 = \text{comp-args}(\langle l_1, \ldots, l_n \rangle) \\
\quad\ \ \textbf{in}\ \lambda\sigma.C_1(C_2(\sigma \sqcup [v_1 \mapsto \{l_1\}] \sqcup \cdots \sqcup [v_n \mapsto \{l_n\}]))
\end{array}$$

$$\begin{array}{l}
\text{comp-app}(f, \langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{cond} \\
\qquad \textit{isVar}(f): \\
\qquad\quad \lambda\sigma.\text{0cfa-abstract-app}(\sigma(f), \sigma, \langle a_1, \ldots, a_n \rangle) \\
\qquad \textit{isPrim}(f): \\
\qquad\quad \text{0cfa-prim}(f, \langle a_1, \ldots, a_n \rangle) \\
\qquad \textit{isLam}(f): \\
\qquad\quad \textbf{let}\ \ C = \text{comp-call}(f \downarrow_{\text{body}}) \\
\qquad\qquad \textbf{in}\ \lambda\sigma.C(\sigma \sqcup_{i=1\ldots n} [f \downarrow_{\text{formals}_i} \mapsto \text{lookup}(a_i, \sigma)])
\end{array}$$

$$\begin{array}{l}
\text{comp-args}(\langle\rangle) = \lambda\sigma.\sigma \\
\text{comp-args}(\langle a_1, \ldots, a_n \rangle) = \\
\quad \textbf{if}\ \textit{isLam}(a_1) \\
\qquad \textbf{then}\ \ \textbf{let}\ \ C_1 = \text{comp-call}(a_1 \downarrow_{\text{body}}) \\
\qquad\qquad\qquad\ \ C_2 = \text{comp-args}(\langle a_2, \ldots, a_n \rangle) \\
\qquad\qquad\quad \textbf{in}\ \lambda\sigma.C_2(C_1(\sigma)) \\
\qquad \textbf{else}\ \text{comp-args}(\langle a_2, \ldots, a_n \rangle)
\end{array}$$

$$\begin{array}{l}
\text{comp-prim}([\![\texttt{+}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\text{comp-prim}([\![\texttt{if}]\!], \langle a_1, \ldots, a_3 \rangle) = \text{comp-args}(\langle a_1, \ldots, a_3 \rangle) \\
\ldots
\end{array}$$

Figure 3.7. Compilation de la *0cfa* par génération de fermetures

fermeture est appliquée sur un environnement abstrait, elle effectue l'analyse de l'expression. Il suffit donc de remplacer la phase de génération de code abstrait par une phase de génération de fermetures. Cette technique a d'ailleurs déjà été proposée pour la compilation de Scheme par Feeley et Lapalme [21]. L'algorithme de compilation abstraite ainsi produit est présenté à la figure 3.7 (sans les optimisations discutées plus haut).

La fonction *comp-app* pourrait être implémenté, en Scheme, de la manière suivante :

```
(define (comp-app f args)
  (cond
   ((var? f)
    (lambda (env)
      (0cfa-abstract-app (env f) env args)))
   ...
   ))
```

Il pourrait sembler, au premier coup d'oeil, que ce nouvel algorithme de compilation n'est pas très différent du premier. C'est vrai en un sens, la logique du compilateur est la même. Par contre, il y a une différence majeure : le code généré n'est plus du tout textuel. Ce changement de représentation a deux avantages principaux. D'abord, le besoin d'une procédure d'interprétation comme `eval` n'est plus nécessaire. Tout langage supportant les fermetures peut être utilisé pour implémenter le compilateur abstrait. Ces fermetures peuvent aussi être simulées par des objets dans un langage orienté-objet. Deuxièmement, en supposant que le compilateur abstrait est lui-même compilé, le compilateur abstrait **et** le programme d'analyse s'exécuteront plus rapidement puisque (1) toutes les expressions de la forme $\lambda\sigma.E$ sont compilées et que (2) la création d'une fermeture, si elle est bien réalisée, est plus rapide que la création du code textuel correspondant.

3.4 *Génération de code machine*

La réalisation d'un compilateur produisant du code machine de qualité est une tâche ardue. Les processeurs sont de plus en plus complexes, les multiples niveaux d'antémémoire rendent difficile la prédiction des temps de calcul, etc. De plus, le travail effectué pour une architecture n'est souvent pas portable vers d'autres architectures. Par contre, la régularité de certains langages, le nombre restreint de constructions de ceux-ci, peuvent rendre très attrayante la compilation vers du code machine.

C'est le cas du compilateur abstrait que je viens de présenter. Nous verrons que des facteurs d'accélération importants peuvent être atteints en compilant les programmes d'analyse vers du code machine. Pour cela, considérons de nouveau le programme d'analyse de la section précédente, que je reproduis ici :

```
(λ (σ)
  ((λ (σ)
     ((λ (σ) (0cfa-abstract-app (σ f) σ x c2))
      σ ⊔ [c2 ↦ (lookup c1 σ)]))
   σ ⊔ [c1 ↦ (lookup tl-cont σ)]))
```

Il est possible de réécrire ce programme fonctionnel en un programme de style impératif équivalent où σ est une variable globale :

$$\sigma := \sigma \sqcup [\texttt{c1} \mapsto \texttt{(lookup tl-cont } \sigma\texttt{)}]$$
$$\sigma := \sigma \sqcup [\texttt{c2} \mapsto \texttt{(lookup c1 } \sigma\texttt{)}]$$
$$\sigma := \texttt{0cfa-abstract-app}(\sigma, \sigma(\texttt{f}), \langle \texttt{x}, \texttt{c2} \rangle)$$

que l'on peut réécrire de la manière suivante

```
join!(c1, tl-cont)
join!(c2, c1)
0cfa-abstract-app!(f, ⟨x, c2⟩)
```

où `0cfa-abstract-app!` est une version destructive de `0cfa-abstract-app`, c'est-à-dire que cette procédure modifie σ directement plutôt que de générer un nouvel environnement, et `join!(x, y)` est équivalent à

$\sigma := \sigma \sqcup [\texttt{x} \mapsto \texttt{(lookup y } \sigma\texttt{)}]$

Si on numérote toutes les variables du programme ainsi que les expressions λ, l'environnement peut être représenté par un vecteur dont les éléments sont des ensembles d'entiers. Le programme d'analyse aurait donc la forme suivante :

```
join!(2, 0)
join!(4, 2)
0cfa-abstract-app!(1, ⟨3,4⟩)
```

En fait, le programme d'analyse de n'importe quel programme CPS-Scheme peut être converti en une suite d'appels à `join!` et `0cfa-abstract-app!` (en faisant abstraction du code pour initialiser le vecteur σ, conséquence de l'optimisation à `lookup`), éliminant ainsi quasi totalement le coût de l'interprétation.

Ce qui vient d'être énoncé n'est pas tout-à-fait exact. Un programme d'analyse comporte en fait deux sections distinctes. D'une part, il y a les instructions que je viens de décrire. Il y a aussi une *table de propagation* utilisée par `0cfa-abstract-app!`. Cette table contient les informations nécessaires pour progager, à un site d'appel dont l'opérateur est une variable `f`, les informations des paramètres actuels aux paramètres formels des expressions λ pouvant être liés à `f` (c'est-à-dire σ(`f`)). En fait, cette propagation a lieu seulement si le nombre de paramètres formels correspond au nombre de paramètres actuels, ce qui est conservateur puisque dans le cas contraire, une erreur d'exécution surviendrait.

Un compilateur abstrait a été réalisé à partir de cette simple observation. Les deux fonctions de base ont été implantées en C et le programme abstrait est produit grâce à quelques routines de génération de code à la volée pour un microprocesseur 64 bits Alpha de DEC roulant OSF/1. En tenant compte de la régularité du code à générer, plusieurs astuces ont permis d'obtenir un code très performant. Par exemple, les adresses des deux fonctions ont été mises dans des registres dédiés (`$s1` et `$s2`) et l'adresse du vecteur σ est mise dans le registre `$s0`. De plus, les paramètres

```
lda   $a0, 2*8, $fp    # join!(2,0)
lda   $a1, 0*8, $fp
addq  $31, $s2, $27
jsr   $26, $27
lda   $a0, 4*8, $fp    # join!(4,2)
lda   $a1, 2*8, $fp
addq  $31, $s2, $27
jsr   $26, $27
lda   $a0, 1*8, $fp     # 0cfa-abstract-app!(1,<3,4>)
lda   $a1, 2, $31       #  <- nombre de param. actuels (2)
lda   $a2, 0, $s3       #  <- index du bloc d'activation
addq  $31, $s1, $27
jsr   $26, $27
```

Figure 3.8. Code Alpha d'un programme d'analyse.

actuels passés à la fonction `0cfa-abstract-app!` sont mis dans des *blocs d'activation abstraits*. Ces blocs d'activation sont des vecteurs contenant les indices dans σ des paramètres actuels. Dans l'exemple, un vecteur de taille 2, dont les éléments sont 3 et 4, serait créé pour $\langle 3, 4 \rangle$. Le code Alpha du programme d'analyse précédent (modulo une vingtaine d'instructions machine permettant de s'interfacer à C) ressemblerait donc à celui de la figure 3.8.

Afin de rendre le programme d'analyse plus performant, la table de propagation a elle aussi été implantée à l'aide d'instructions machine. Ainsi, à chaque expression λ du programme correspond une procédure codée en langage machine. Cette procédure accepte deux arguments : le nombre de paramètres actuels et un pointeur vers le bloc d'activation abstrait. Si le nombre de paramètres actuels correspond au nombre de paramètres formels de l'expression λ, elle appelle `join!` pour chaque couple paramètre formel/paramètre actuel correspondants. Enfin, un pointeur vers cette procédure est mis dans une table qui sera utilisée par `0cfa-abstract-app!`. À titre d'exemple, la procédure de propagation pour `(lambda (f c1) ...)` est illustrée à la figure 3.9.

```
  subq $a0, 2, $t0     # test # arguments
  beq  $t0, propagate
  ret  $31, $26

  ...

propagate:
  subq $sp, 24, $sp    # on sauve quelques registres
  stq  $s3, 0($sp)
  addq $26, $31, $s3
  stq  $s4, 8($sp)
  addq $a1, $31, $s4
  stq  $s0, 16($sp)
  addq $a2, $31, $s4

  lda  $a0, 1*8, $fp   # propagation vers f (1)
  lda  $t0, 0*8,         # parametre actuel 0
  ldq  $a1, 0, $t0
  addq $31, $s0, $s7
  jsr  $26, $27          # appel de Join!

  lda  $a0, 2*8, $fp   # propagation vers c1 (2)
  lda  $t0, 1*8,         # parametre actuel 1
  ldq  $a1, 0, $t0
  addq $31, $s0, $s7
  jsr  $26, $27          # appel de Join!

  addq $s3, $31, $26   # on recupere les registres...
  ldq  $s0, 16($sp)
  ldq  $s4, 8($sp)
  ldq  $s3, 0($sp)
  addq $sp, 24, $sp
  ret  $31, ($26)
```

Figure 3.9. Procédure de propagation.

3.5 Résultats expérimentaux

Trois des différents analyseurs pour la *0cfa* que je viens de décrire ont été implantés afin d'évaluer leurs performances. Il s'agit de l'interprète abstrait, du compilateur générant des fermetures et du compilateur générant du code machine. Ces implantations traitent un sous-ensemble de Scheme plus large que CPS-Scheme. La forme spéciale `set!` et les procédures impératives `set-car!`, `set-cdr!`, etc., sont traités.

Ces implantations traitent aussi les fonctions s'"échappant" en mémoire. Par ceci j'entends des fonctions qui sont stockées dans des structures de données pour être plus tard récupérées et appliquées. Afin de traiter ce cas conservativement, une variable spéciale est introduite, $\mathcal{ESC}$, qui abstrait la mémoire. Par exemple, si le programme à analyser contient le site d'appel `(cons x y k)`, toutes les expressions λ qui peuvent être liées à `x` et `y` à l'exécution sont ajoutés à $\sigma(\mathcal{ESC})$. Inversement, un appel `(car x (lambda (z)` E`))` aura pour effet l'ajout de $\sigma(\mathcal{ESC})$ à σ(`z`). Cette approximation est très grossière, mais elle est conservatrice, facile à implanter et suffisante pour nos tests.

La partie frontale est la même pour les trois implantations. Elle effectue les trois opérations suivantes :

1. Le programme Scheme à analyser est d'abord lu.
2. Un certain nombre de transformations syntaxiques sont effectuées afin d'exprimer le programme dans une forme équivalente mais avec un nombre restreint de formes spéciales (par exemple, `let` et `letrec` peuvent être exprimés en utilisant uniquement des formes spéciales `lambda` et `set!`).
3. Le programme est CPS-converti.
4. Les expressions λ sont numérotées et le programme est α-converti.

Cette dernière opération résulte en un arbre de syntaxe abstrait à partir duquel la *0cfa* sera calculée.

Programme	Description
`conform`	Un programme de manipulation de treillis et d'ordres partiels.
`earley`	Un générateur d'analyseurs syntaxiques pour langages hors-contextes basé sur l'algorithme d'Earley.
`interp`	Un petit interprète pour un langage fonctionnel paresseux.
`lambda`	Un interprète pour le λ-calcul.
`lex`	Un générateur d'analyseurs lexicaux.
`link`	L'éditeur de liens du compilateur Gambit-C.
`lll`	Un générateur d'analyseurs syntaxiques $LL(1)$.
`peval`	Un petit évaluateur partiel pour Scheme.
`source`	Un analyseur syntaxique pour Scheme.

Figure 3.10. Banc d'essai

Les trois systèmes ont été écrits en Scheme et compilés avec le compilateur Gambit-C, générant du code C, sous DEC Alpha. Les opérations de manipulation d'ensembles ont été implantés en C pour des raisons d'efficacité ainsi que d'interface avec le code machine généré. Plusieurs structures de données ont été considérées pour la représentation des ensembles. Une représentation à l'aide de listes était trop coûteuse et les vecteurs de bits occupaient trop de place en mémoire : en général, les ensembles contiennent peu d'éléments et sont très clairsemés. La structure de données adoptée consiste en un vecteur dont les éléments sont triés. L'union de deux ensembles alloue un nouveau vecteur et les éléments sont fusionnés durant la copie.

Le banc d'essai dont je me suis servi pour évaluer les performances des trois systèmes est composé des programmes de la figure 3.10. La figure 3.11 donne les temps d'exécutions des trois implantations sur un DEC AXP3000, un processeur DEC Alpha roulant OSF/1 avec 160MB de mémoire vive. Les temps sont donnés en

Programme	Analyse (secondes)			Génération (secondes)	
	interp.	fermetures	code machine	fermetures	code machine
`conform`	.0648	.0154 (4.2)	.0039 (16.6)	.0262	.2550
`earley`	.0603	.0117 (5.2)	.0020 (30.2)	.0249	.2440
`interp`	.1230	.0435 (2.8)	.0095 (12.9)	.0177	.1570
`lambda`	.1050	.0326 (3.2)	.0055 (19.1)	.0308	.2790
`lex`	.1290	.0266 (4.8)	.0087 (14.8)	.0486	.4590
`link`	.4687	.1322 (3.5)	.0331 (14.2)	.0784	.7890
`lll`	.0940	.0226 (4.2)	.0042 (22.4)	.0244	.2150
`peval`	.1727	.0508 (3.4)	.0102 (16.9)	.0388	.3430
`source`	.0773	.0195 (4.0)	.0033 (23.4)	.0205	.1800

Figure 3.11. Temps d'analyse et de génération des programmes d'analyse.

secondes. La colonne interp donne le temps d'analyse requis par l'interprète abstrait; la colonne fermetures donne le temps d'analyse lorsque le programme est compilé vers des fermetures; la colonne code machine donne le temps d'exécution du programme d'analyse lorsque ce dernier est généré par le compilateur produisant du code machine. Les deux dernières colonnes donnent les temps de génération des programmes d'analyse pour les deux compilateurs abstraits.

On peut donc constater que la technique de compilation abstraite peut accélérer le temps d'analyse par un facteur variant entre 3 et 5 pour la plupart des programmes en générant des fermetures. La seule exception survient dans le cas du programme `interp`. Les données de la figure 3.12 permettent de comprendre la raison de cette différence.

On observe que la taille moyenne des ensembles (dernière colonne) pour ce programme est beaucoup plus élevé que les autres (à l'exception de `peval`) et que le

Programme	Nombre de lignes de code	Nombre d'itérations	Taille moyenne des ensembles
conform	557	3	0.59
earley	648	3	0.41
interp	411	9	3.02
lambda	617	4	0.60
lex	1133	3	0.59
link	1608	6	2.22
lll	613	5	0.43
peval	618	5	6.80
source	453	5	0.77

Figure 3.12. Quelques métriques.

nombre d'itérations est assez élevé lui-aussi. Ceci indique que ce programme fait une grande utilisation des fonctions d'ordre supérieur et/ou qu'un grand nombre de fonctions sont stockées dans des structures de données. Puisque chaque ensemble contient plus d'éléments, plus de temps est passé dans les opérations ensemblistes relativement au temps requis par la couche d'interprétation. Il n'est donc pas étonnant qu'une accélération plus faible en résulte. Notons aussi que même si l'on considère le temps de génération du programme d'analyse, l'accélération globale est près de 2 dans la plupart des cas.

Dans le cas de la génération de code machine, un facteur additionnel de 3 à 6 est atteint, pour une accélération combinée variant d'un facteur de près de 13 à plus de 30. Par contre, on remarque aussi que le temps de génération de code est plus élevé que le temps d'analyse par interprétation abstraite. Ceci est en partie dû au coût de l'interface C de Gambit-C : les routines de génération de code machine, bien

qu'écrites en C, sont appelées du programme Scheme à travers l'interface C, dont le coût est non-négligeable. Or ce coût peut être amorti si le programme d'analyse est réutilisé plusieurs fois. L'architecture que je proposerai au prochain chapitre capitalise sur cette observation.

La figure 3.13 mesure les bénéfices relatifs des optimisations dont j'ai discuté plus haut, soit l'η-réduction et l'optimisation de la fonction `lookup` lorsque le programme d'analyse est compilé vers des fermetures. La première colonne donne les temps d'exécution des programmes d'analyse lorsqu'aucune optimisation n'a lieu. Les deux colonnes suivantes indiquent la réduction en temps d'exécution lorsque les optimisations sont appliquées séparément. Enfin, la dernière colonne donne la réduction lorsque les deux optimisations sont appliquées. Ces chiffres montrent donc que, malgré leur simplicité, certaines optimisations permettent d'obtenir des programmes d'analyse bien plus performants. Il importe donc de bien analyser la nature des programmes d'analyse générés, en fonction de leur implantation, afin d'en éliminer les calculs inutiles ou superflus. Notons que l'η-réduction n'a pas été nécessaire lors de la compilation vers du code machine en raison de la nature du code généré. La deuxième optimisation a, pour sa part, été appliquée systématiquement afin de réduire la complexité du générateur de code.

3.6 Conclusion

Comme nous venons de le voir, la compilation abstraite permet d'accélérer considérablement le temps consacré à l'analyse statique. Malheureusement, plus les opérations élémentaires de l'analyse sont lentes, moins l'approche par compilation abstraite est attrayante puisque le coût de la couche d'interprétation devient faible. Alors, l'analyse décrite précédemment est-elle représentative des "vraies" analyses que l'on retrouve dans les compilateurs?

Je crois que oui. D'ailleurs, c'était une variante de cette analyse qui était implantée

Programme	Aucune	η-réduction	lookup	Toutes
conform	0.0292	-18.7%	-28.8%	-47.3%
earley	0.0274	-26.4%	-37.0%	-57.3%
interp	0.0675	-22.0%	-15.7%	-35.6%
lambda	0.0507	-23.1%	-18.6%	-35.8%
lex	0.0547	-21.6%	-31.9%	-51.4%
link	0.2137	-20.6%	-16.2%	-38.3%
lll	0.0410	-21.5%	-16.9%	-44.9%
peval	0.0846	-20.0%	-17.7%	-40.0%
source	0.0370	-24.7%	-18.5%	-47.5%

Figure 3.13. Bénéfices relatifs de chaque optimisation.

dans la version 1.7 du compilateur Scheme Bigloo. Dans le cas d'analyses pour des programmes logiques, Debray [17] a montré que des accélérations importantes pouvaient être observées. Aussi, pour des analyses plus classiques de flux de données, les opérations de base sont généralement des unions et des différences d'ensembles, comme dans le cas de la *0cfa*.

Une autre conclusion que l'on peut tirer de cette expérience est que plus on désire obtenir un programme d'analyse rapide, plus le temps de génération de ce dernier augmente. Autrement dit, le temps de compilation semble inversement proportionnel au temps d'exécution du programme d'analyse. Il faut donc trouver le point d'équilibre entre les deux. Puisque le coût d'une compilation abstraite vers du code machine surpasse celui d'une interprétation abstraite directe (du moins dans notre cas), comment justifier la réalisation d'une telle technique?

En fait, si le programme d'analyse est exécuté plus d'une fois, on ne paie qu'une

fois le coût de la génération de code. On peut donc considérer que ce coût est amorti sur plusieurs exécutions. Si le nombre d'exécutions est suffisamment élevé, ce coût peut même devenir négligeable. Je développe au prochain chapitre une architecture de compilateur optimisant exploitant cette idée.

Chapitre 4

ARCHITECTURE

Cette section décrit l'approche à la compilation séparée que je propose 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 [14]. Je ne m'intéresserai pas à ces problèmes. Je désire me concentrer uniquement sur les aspects concernant les analyses et 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`).

4.1 L'organisation traditionnelle

L'organisation traditionnelle d'un compilateur supportant la compilation séparée ressemble habituellement à celle de la figure 4.1. Décrivons-la succintement.

Une première phase de compilation génère un fichier "objet" pour chacun des modules[1] `m1`, `m2` et `m3`. Ces compilations sont faites 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 objet.

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

[1] Je ne fais 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 notamment Modula-2, Oberon et Dylan. De toute manière, cette distinction ne changerait pas le modèle de compilation.

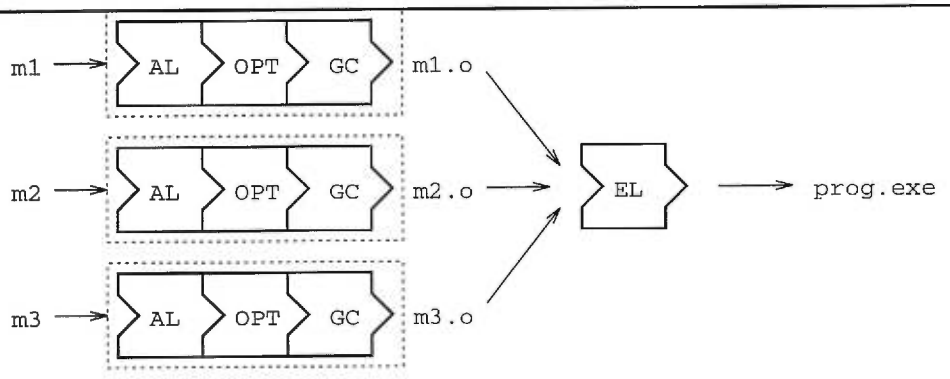

Figure 4.1. Organisation traditionnelle d'une compilation séparée.

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.

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

$$\text{coût}_{\text{all}} = \underbrace{\left(\sum_{m \in M} c_1(\text{taille}(m))\right)}_{\text{compilation}} + \underbrace{\left(\sum_{m \in M \cup L} c_2(\text{taille}(m))\right)}_{\text{édition de liens}} \tag{4.1}$$

et les "`make`" subséquents (après modification d'un module m_i) ont un coût donné

par la formule

$$\text{coût}_{m_i} = c_1(\text{taille}(m_i)) + \left(\sum_{m \in M \cup L} c_2(\text{taille}(m)) \right) \tag{4.2}$$

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 (sauf pour la portée des variables, évidemment).

4.1.1 Variantes de cette architecture

L'architecture qui vient d'être décrite possède des limites intrinsèques évidentes. De nombreuses optimisations souvent très payantes ne peuvent en effet être effectuées au travers la barrière des modules, comme l'allocation globale des registres, l'intégration de fonctions, l'évaluation partielle, la détection de code mort, etc.

Cette architecture se situe en fait à l'extrémité d'un spectre très vaste de possibilités. À l'autre extrémité se situe une architecture où tout le programme source est compilé et optimisé d'un seul coup. Comme je l'ai souligné plus tôt, une telle architecture n'est pas pratique puisqu'elle force l'accès à tout le code source, ce qui n'est pas toujours possible et risque d'être trop coûteux en temps de calcul et en espace mémoire.

D'autres approches ont évidemment été proposées. Ces architectures tombent généralement dans deux catégories. La première 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 infor-

mations 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 [54] et Srivastava et Wall [53] ainsi que ceux de Chow [10] suivent la première approche. Wall [54] 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 [53] ont développé un système permettant des optimisations 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.

Dans le cadre du projet `alto`, Debray et al. [15, 16] ont développé un système effectuant aussi des optimisations intermodules à l'édition de liens à partir du code objet de chacun des modules roulant sur DEC Alpha. Les optimisations réalisées par `alto` sont : la propagation de constantes, l'élimination de code mort, l'élimination d'opérations mémoire superflues, l'intégration de fonctions et l'ordonnancement des instructions. Sur des programmes Scheme préalablement compilés à l'aide de compi-

lateurs produisant du C (Gambit-C [19] et Bigloo [49] en l'occurrence), leurs résultats sont systématiquement meilleurs que ceux obtenus à l'aide du système OM. Ils font état d'accélération de l'ordre de 5% en moyenne pour les programmes compilés avec Gambit-C et de 12% en moyenne pour les mêmes programmes, mais cette fois-ci compilés avec Bigloo.

Chow [10] 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 [42] 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.

Dans le cadre de ses études doctorales, Mary F. Fernández a conçu un éditeur de liens optimisant pour Modula-3 et C++, `mld` [25]. Ce système réalise des optimisations simples semblables à l'évaluation partielle, comme le remplacement d'appels de méthode par des appels directs lorsque le contexte le permet. Il utilise aussi des informations de profilage afin d'optimiser les sections de code qui semblent présenter le plus

d'opportunités. `mld` permet de déterminer, pour chaque module, la représentation exacte de tous les types importés, y compris celle des types *opaques*[2]. Contrairement au système `alto`, `mld` traduit préalablement tous les modules en un code intermédiaire et ne fait pas de compilation vers du code objet proprement dit. Le code objet est produit suite aux optimisations à l'édition des liens.

Il y a aussi plusieurs travaux qui concernent l'optimisation (et la réoptimisation) globale incrémentale de programmes [9, 43]. Mais ils se concentrent surtout sur les problèmes de recompilation et de réoptimisation en présence de modifications dans un module. Burke et Torczon [9], 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 [43] 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.

4.2 Une nouvelle architecture

L'approche que je suggère bouleverse cette organisation afin d'obtenir de meilleures optimisations globales et, en bout de ligne, des programmes plus performants. Cette architecture tire profit de la technique d'analyse statique par compilation abstraite que j'ai décrite précédemment.

L'approche par compilation abstraite est illustrée à la figure 4.2. 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.ap` est généré,

[2] Les types *opaques* sont une caractéristique du langage Modula-3. L'interface d'un module permet la déclaration de types dont la représentation est omise, contrairement à C++.

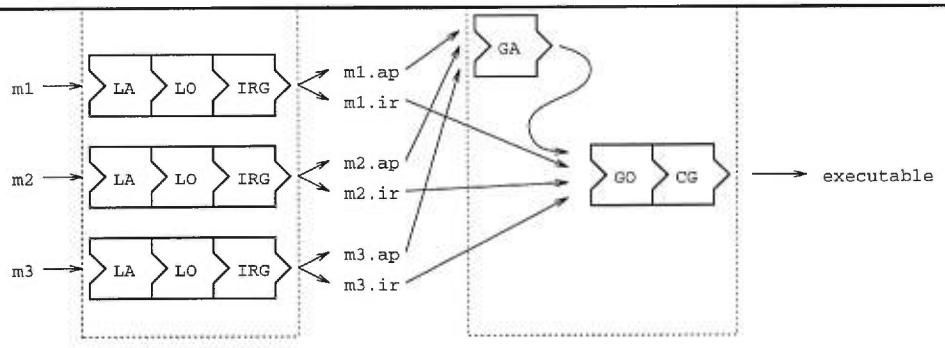

Figure 4.2. Organisation proposée.

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`. Je discuterai plus loin de cette représentation.

La deuxième phase procède en deux étapes :

1. Elle lit d'abord tous les programmes d'analyse des modules d'une application (les fichiers `.ap`) et effectue l'analyse globale (typiquement par itérations successives, jusqu'à l'obtention d'un point fixe, une itération correspondant 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.

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 `.ap`, produisant ainsi des programmes d'analyse pour des programmes d'analyse pour ... etc. Heureusement, je crois que le pro-

gramme d'analyse généré par la première phase du compilateur possédera en général une structure trop simple pour être un bon candidat à d'éventuelles optimisations globales.

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. Je considère 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.

4.2.1 Attraits de cette architecture

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 : il n'est pas nécessaire de développer de nouvelles analyses adaptées à la phase globale, mais plutôt de déterminer comment l'analyse locale sera factorisée et compilée. Il est donc raisonnable de croire que l'effort consacré à un module en particulier lors de l'analyse globale sera moindre que pour l'analyse locale de ce même module. 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. Il suffit alors de fournir au client de la librairie les interfaces nécessaires ainsi que les fichiers `.ir` et `.ap` correspondants qui serviront lors de l'édition de liens.

Évidemment, le coût de la deuxième phase de compilation risque d'être plus élevé qu'avec l'architecture 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

$$\text{coût}'_{\text{all}} = \left(\sum_{m \in M} c'_1(\text{taille}(m))\right) + c'_2(\sum_{m \in M \cup L} \text{taille}(m)) \tag{4.3}$$

pour la compilation initiale et

$$\text{coût}'_{m_i} = c'_1(\text{taille}(m_i)) + c'_2(\sum_{m \in M \cup L} \text{taille}(m)) \tag{4.4}$$

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 très fortement couplés pourraient être compilés et optimisés simultanément, une modification dans l'un entraînant seulement la réoptimisation du groupe auquel il appartient.

4.2.2 Représentations intermédiaires

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'optimisation globale. Il importe, dans le cas des langages typés dynamiquement 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.

De nombreux langages de programmation possèdent des constructions ou 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'optimisation globale 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 m'apparaît donc trop simpliste et manquer de généralité.

Je pense donc qu'une réalisation de l'architecture ici proposée doit impliquer le développement d'une représentation intermédiaire à partir des contraintes spécifiques posées par les optimisations globales désirées. Voici un certain nombre de contraintes qui sont partagées par un grand nombre de langages modernes.

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 portables 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.

Exceptions. De nombreux langages dont C++, Java, SML, etc., possèdent un mécanisme pour traiter les exceptions. La présence de tels mécanismes doit être prise en compte par les analyses statiques et ceux-ci peuvent empêcher certaines optimisations d'être appliquées. À l'opposé, une analyse globale pourrait rendre superflue l'installation des traiteurs d'exceptions (*exception handlers*) en un site particulier du programme si elle détecte qu'un certain type d'exception ne sera jamais levé. Il faut donc que la représentation intermédiaire explicite ces sites d'installation si on veut pouvoir les éliminer lors de l'optimisation globale.

Gestionnaire de mémoire. Certains langages (comme Java, Eiffel, les langages fonctionnels et logiques en général, etc.) possèdent des mécanismes de gestion de mémoire automatique (typiquement un glaneur de cellules ou *GC*). Dans ces systèmes, les objets (au sens générique du terme) sont habituellement tous alloués dans le tas. Or si la durée de vie d'un objet peut être prédite assez précisément par une analyse

statique (comme celles de Hudak [34] et Serrano et Feeley [24]), il est possible dans certains cas d'allouer l'objet sur la pile d'exécution. Ainsi, lors du dépilage d'un bloc d'activation, l'objet est automatiquement désalloué. Si l'on veut effectuer cette optimisation lors de la phase globale, il faut alors que la représentation intermédiaire exprime les allocations comme des opérations de base et non comme des appels à une librairie ou à une routine de l'environnement d'exécution.

4.2.3 Conclusion

En regard des résultats du chapitre précédent, cette nouvelle approche à la compilation séparée semble intéressante et prometteuse, en principe du moins. Il importe donc de montrer qu'elle est aussi intéressante en pratique. Les deux prochains chapitres décrivent une première implantation partielle de cette architecture ainsi que les performances obtenues qui en montrent l'intérêt.

Chapitre 5

UNE IMPLANTATION

5.1 Architecture du système

L'architecture de compilateur présentée au chapitre précédent présente de nombreux défis. L'élaboration d'une représentation intermédiaire adéquate est une entreprise en soi. Je me suis donc concentré à montrer que l'analyse globale par compilation abstraite est effectivement réalisable et procure de bons gains de performance, tout en n'étant pas trop coûteuse en temps de calcul. Je décris donc ici une première implantation partielle de l'architecture idéale proposée. Il s'agit d'un compilateur Scheme supportant des analyses et optimisations intermodules, et développé à partir d'un compilateur Scheme existant, Gambit-C.

5.1.1 Le choix de Scheme

Le choix de Scheme comme langage source n'est pas arbitraire. Scheme, un dialecte de Lisp, est un langage impératif avec un noyau fonctionnel d'ordre supérieur et ayant une sémantique formelle faisant partie du standard [11], ce qui rend possible la construction d'analyses statiques prouvées correctes. Mais c'est surtout un langage très simple dont les concepts sont orthogonaux : toutes les constructions du langage peuvent être ramenées à cinq formes spéciales. Il pourrait donc sembler que Scheme est un langage facile à compiler. C'est vrai s'il s'agit de produire un code de performance moyenne, sous-optimal. Mais pour écrire un compilateur Scheme de haute performance, il en va tout autrement. Voici quelques raisons.

Scheme est un langage typé dynamiquement, i.e. ce ne sont pas les variables du programme source qui sont typées mais bien les valeurs liées à ces variables qui, à

l'exécution, contiennent l'information sur leur propre type. Cela permet d'écrire des fonctions polymorphiques, certes. Mais bon nombre de primitives doivent s'assurer que le type de leurs arguments est correct à l'aide de tests de type implicites pour le programmeur. Cartright et Wright [56] ont décrit des programmes pour lesquels une accélération d'un facteur 3 pouvait être obtenue en éliminant les tests de type superflus. Des analyses statiques doivent donc être utilisées pour déduire, au moins partiellement et de manière conservative, les types des variables.

Les variables auxquelles sont liées les primitives de Scheme peuvent aussi être redéfinies. Dans un système interactif, par exemple, on pourrait redéfinir la variable `+` afin d'imprimer une trace de tous les appels à cette primitive. Comme ces primitives sont généralement de courtes fonctions que le compilateur aurait tout intérêt à intégrer, il faut des mécanismes pour assurer le compilateur qu'il peut faire l'intégration en toute sécurité, généralement des annotations directement dans le code source. Malheureusement, ces annotations sont rarement portables et supposent à tort que le programmeur sait ce qu'il fait.

Scheme est un langage d'ordre supérieur, ce qui implique que les fonctions peuvent être passées en paramètre, retournées par d'autres fonctions, emmagasinées dans des structures de données pour être récupérées et appliquées par la suite. Ceci, combiné au typage dynamique, augmente le coût d'un appel de fonction. Pour un appel `(f a1 ... an)` non-optimisé, il faut à tout le moins :

1. tester que `f` s'évalue bien en une fonction,

2. s'assurer que le nombre d'arguments est bon et ensuite

3. brancher au code de la fonction.

Puisque Scheme encourage l'utilisation de fonctions d'ordre supérieur, l'optimisation des appels de fonctions est primordiale et requiert généralement des analyses de flot de contrôle.

En Scheme, il n'y a pas d'espace de nom séparé pour les variables et les fonctions. Il est donc possible de définir une variable dont la valeur est une fonction et ensuite affecter une nouvelle valeur à cette variable en cours d'exécution. Il en résulte une grande difficulté à faire de l'intégration de fonctions, surtout dans le contexte d'un compilateur supportant la compilation séparée : une variable peut être définie comme une fonction dans un module et redéfinie comme un nombre dans un autre.

La tour des nombres de Scheme est plus riche que dans bon nombre de langages et Scheme force certaines primitives à retourner des valeurs du domaine qui permettent de représenter le résultat le plus exactement possible. Par exemple, la soustraction de deux nombres complexes peut très bien résulter en un entier exact. Il peut donc être très difficile de prédire le type du résultat de primitives numériques en Scheme, même à l'aide d'analyses statiques sophistiquées.

5.1.2 Gambit-C

Écrire un compilateur générant du code de bonne qualité représente une tâche colossale. Il faut soigneusement choisir les représentations intermédiaires, les analyses et optimisations. Dans le cas de Scheme, la qualité de l'environnement d'exécution (GC, représentation des types, etc.) est au moins aussi importante que le compilateur lui-même. C'est pourquoi, plutôt que de développer un autre compilateur Scheme, j'ai décidé de modifier un compilateur existant, même si l'architecture dudit compilateur ne se prêtait pas facilement aux extensions que j'allais lui apporter.

Le compilateur en question, Gambit-C, développé par Marc Feeley à l'Université de Montréal, présente un certain nombre de caractéristiques rendant l'implantation d'analyses intermodules très attrayantes. Mais d'abord, voyons un peu l'historique du compilateur, ce qui permettra de mieux apprécier certaines décisions d'implantation.

Gambit-C est un compilateur Scheme générant du C, basé sur une implantation générant du code machine Motorola 68k, Gambit-68k [20]. Ce compilateur était en fait un compilateur pour MultiLisp [32], c'est-à-dire une extension de Scheme possédant

des primitives pour le traitement parallèle. Ce premier compilateur produisait du code pour une machine à registre abstraite (la GVM [23]), qui était ensuite compilé vers du code machine 68k. La deuxième génération du compilateur, renommé Gambit-C, produit maintenant du code C à partir du code GVM, procurant une plus grande portabilité au compilateur, au prix toutefois d'une perte de performance que je décrirai plus loin.

L'objectif premier de Gambit était de vérifier si la compilation des *futures* pouvait se faire efficacement dans le contexte d'un compilateur qui génère du code rapide. Il importait peu si ce code rapide était obtenu par analyses sophistiquées, comme le font Bigloo [47] de Manuel Serrano et Stalin [52] de Jeffrey Mark Siskind, ou par annotations du programmeur permettant de spécifier certains invariants que le compilateur ne pourrait prouver autrement. De plus, au moment où Gambit a été développé, les analyses statiques de flux de contrôle pour Scheme n'avaient pas encore été mise au point. Gambit-C possède donc un ensemble d'annotations permettant la génération de code plus rapide, au détriment dans certains cas du respect de la sémantique de Scheme. Par exemple, il est possible de spécifier, à l'aide de la déclaration

```
(declare (standard-bindings))
```

que tous les appels à des primitives, dans la portée de cette déclaration, réfèrent bien aux primitives standards de Scheme, autorisant ainsi le compilateur à intégrer celles qui peuvent l'être comme `not` et `char?` ou, à tout le moins, de générer des appels plus efficaces en omettant, par exemple, les tests du nombre d'arguments.

Optimisations

Il va sans dire que Gambit-C, afin de générer un code de grande qualité, utilise un vaste attirail d'autres techniques de compilation. Parmi celles-ci, on retrouve au niveau du code source :

- la conversion des affectations à des variables locales en mutations dans des structures de données;
- la β-réduction (propagation de constantes, propagation de copies, élimination des variables inutiles, intégration de fonctions);
- le *lambda-lifting* [36], qui permet de réduire dans certains cas le nombre de variables libres des procédures dont tous les sites d'appel sont connus statiquement en leur ajoutant des paramètres formels correspondant à ces variables libres; et
- l'optimisation de représentation des fermetures, similaire à celle de Orbit [38].

Au niveau du code GVM généré, plusieurs autres optimisations sont effectuées :

1. l'élimination des branchements en cascade,
2. l'élimination de code mort,
3. l'élimination des branchements inutiles (branchement à l'instruction suivante, qui peuvent être générés par les deux premières optimisations),
4. l'élimination de code commun et
5. le réordonnancement des blocs de base.

Notons que les registres virtuels sont alloués en utilisant une technique d'allocation de registres à la volée (*on-the-fly register allocation*) similaire à celle décrite dans [26].

Appels terminaux

Toute implantation de Scheme fidèle au standard doit faire l'optimisation des appels terminaux (*tail-call optimization*). Un appel de fonction `(p a1 ... an)` apparaissant dans une fonction `f` est **terminal** si un appel à `f` peut être réduit à l'appel en question. Par exemple, dans le code suivant :

```
    (define (fact-iter n k)
      (if (<= n 1)
          k
(1)       (fact-iter (- n 1) (* k n))))
```

l'appel interne à `fact-iter` à la ligne 1 est en position terminale puisque le résultat de cet appel est aussi le résultat de la fonction appelante.

L'optimisation de ces récursions consiste à réutiliser le bloc de continuation de la fonction appelante pour l'appel de la fonction. L'optimisation des récursions terminales permet d'assurer qu'une certaine classe de fonctions récursives puissent être compilées comme des fonctions itératives, c'est-à-dire prenant un espace constant pour la continuation[1].

Le compilateur Gambit original, produisant du code machine pour processeur 68k, faisait cette optimisation sans trop de problèmes : les instructions machine ne posaient pas d'obstacle insurmontable à la réutilisation des blocs de continuation puisque la convention d'appel pouvait être entièrement définie.

Dans Gambit-C, il en va tout autrement. Les buts qui ont guidé le développement du compilateur ont posé un certain nombre de contraintes dès le départ. Par exemple, le code C généré devait pouvoir s'interfacer avec des routines externes écrites en C. De plus, Gambit-C devait supporter le modèle de compilation séparé traditionnel. Autrement dit, chaque fichier Scheme devait être compilé en un fichier C correspondant. Enfin, Gambit-C devait supporter le chargement dynamique.

Compte tenu que la convention d'appel de C peut varier d'une architecture à l'autre et même d'un compilateur à l'autre pour une architecture donnée, Gambit-C implante sa propre pile d'exécution parallèlement à celle de C. Seuls les appels à des fonctions externes en C sont toutefois effectués sur la pile de C.

---

[1] Cette caractérisation est un peu boîteuse, j'en conviens. Une implantation pourrait très bien allouer tous les blocs de continuation dans le tas, ce qui fait que toute fonction récursive puisse être exécutée en prenant un espace constant sur la pile. Je fais donc référence ici uniquement aux implantations qui allouent leurs blocs de continuation sur la pile.

Gambit-C peut donc optimiser les appels terminaux des programmes Scheme puisqu'il a le plein contrôle de la pile de Scheme, et donc de la convention d'appel. Le seul obstacle réside au niveau des appels entre modules. Puisque chaque module est compilé en un fichier C contenant des fonctions C, un appel intermodule doit donc ultimement résulter en un appel à une fonction C, utilisant donc la pile de C. Pour résoudre ce problème, Gambit-C utilise une technique de **trampoline**. Chaque fichier Scheme est compilé en une unique fonction C qui sert de point d'entrée au module. Chaque fonction locale est accédée à partir de cette fonction à l'aide de branchements (`goto`). Un appel de fonction entre modules consiste donc en 5 opérations :

1. on branche à la fin de la fonction C du module de la fonction appelante,
2. la fonction C retourne le contrôle au noyau de Gambit-C,
3. le noyau de Gambit-C détermine dans quel module il faut brancher,
4. on entre dans la fonction C du module de la fonction appelée,
5. on branche à l'étiquette de la fonction appelée à l'aide d'un `switch`.

Lors du retour de la fonction appelée, les mêmes étapes sont exécutées afin de retourner dans le module de la fonction appelante. On voit donc qu'il y a un coût non-négligeable aux appels entre modules. Ce coût est d'autant plus important que l'absence d'annotations adéquates forcera tout appel à une primitive à passer par ce mécanisme.

Notons qu'un appel à une fonction et le retour à une fonction dans le même module peuvent être compilés plus efficacement. Grosso modo, il faut seulement un `goto` si la destination est connue à la compilation et l'étape 5 sinon. Seule l'étape 5 est nécessaire pour le retour à une fonction dans le même module.

5.1.3 GOLD

Gambit-C génère déjà du code de très bonne qualité en présence d'annotations du programmeur. Or ces annotations ont plusieurs désavantages. Entre autres, elles ne sont pas portables d'un système Scheme à l'autre. De plus, elles forcent le compilateur à faire aveuglément confiance au programmeur.

L'extension à Gambit-C que j'ai développée, GOLD (*Gambit Optimizing Linker*), tente de pallier à ces lacunes. Pour cela il essaie, à partir d'un programme Scheme multi-modules, de déduire automatiquement les annotations que le programmeur y aurait mises en vue de rendre possible la réalisation, par Gambit-C, des optimisations suivantes :

- le court-circuitage du mécanisme de trampoline pour les appels intra-module;
- l'intégration de primitives;
- la spécialisation des appels de primitives numériques.

En plus de ces optimisations, GOLD fait aussi :

- l'intégration de fonctions au-delà de la barrière des modules, i.e. que des fonctions globales définies dans un certain module (ici un fichier) peuvent être intégrés dans d'autres, permettant ainsi de sauver le coût d'un appel via la trampoline.

GOLD est écrit en Scheme. Son architecture est simple. Le système maintient un ensemble des modules qui composent l'application courante (ou projet). Chaque module est analysé séparément. L'analyse est opérée après la phase d'optimisation du code source de Gambit-C. Il n'y a donc pas de compilation vers C à ce moment. En effet, cette première étape ne fait que produire le programme d'analyse du module, un fichier intermédiaire contenant le résultat de l'analyse locale et ajoute, à une base

de donnée globale au projet, l'ensemble des fonctions dudit module pouvant être intégrées dans d'autres modules (le critère déterminant si une fonction est candidate pour l'intégration intermodule sera décrit plus loin).

Lorsque tous les modules ont été compilés, l'utilisateur peut alors faire l'édition des liens, ce qui implique :

1. une analyse globale,

2. l'optimisation de chacun des modules et

3. la génération des fichiers C correspondant aux modules du programme.

La prochaine section présente l'analyse statique calculée par GOLD. Cette analyse est effectuée sur chaque module d'abord et ensuite sur l'ensemble des modules composant le projet. La section suivante présentera le détail des optimisations globales intermodules.

5.2 L'analyse

L'analyse de GOLD est basée principalement sur la 0CFA de Shivers, qui est une analyse de flot de contrôle monovariante pour langage d'ordre supérieur par interprétation abstraite. Tout comme pour la 0CFA, mon système effectue le calcul de l'analyse par approximations successives jusqu'à l'obtention d'un point fixe. Je ne donnerai donc pas ici tous les détails de l'analyse.

Malgré tout, cette analyse diffère de celle de Shivers à plusieurs niveaux. Les sections suivantes discutent de ces différences. Mais voyons d'abord le langage source sur lequel opère l'analyse de GOLD.

Bien que GOLD soit un compilateur pour tout le langage Scheme, je m'intéresserai ici à un sous-ensemble seulement de Scheme. Toutefois, ce sous-ensemble, décrit à la figure 5.1, est assez complet pour qu'une traduction d'un programme Scheme

$\Pi \in$ *Programme*
$D \in$ *Définition*
$E \in$ *Expression*
$V \in$ *Variable*
$K \in$ *Constante*
$P \in$ *Primitive* (`cons`, `car`, `+`, *etc.*)
$l \in$ *Etiquette*

$$\Pi ::= \{D^l \mid E^l\}$$

$$D ::= (\texttt{define}\ V^{l_1}\ E^{l_2})^l$$

$$\begin{aligned} E ::= {} & K^l \\ \mid {} & V^l \\ \mid {} & (\texttt{set!}\ V^{l_1}\ E^{l_2})^l \\ \mid {} & (\texttt{if}\ E_t^{l_t}\ E_c^{l_c}\ E_a^{l_a})^l \\ \mid {} & (\texttt{letrec}\ ((V_1\ E_1^{l_1})\ \cdots\ (V_n\ E_n^{l_n}))\ E)^l \\ \mid {} & (\texttt{lambda}\ (V_1\ \ldots\ V_n)\ E^{l_e})^l \\ \mid {} & (E^{l_0}\ E_1^{l_1}\ \ldots\ E_n^{l_n})^l \\ \mid {} & (P^{l_0}\ E_1^{l_1}\ \ldots\ E_n^{l_n})^l \end{aligned}$$

Figure 5.1. Syntaxe abstraite d'un sous-ensemble de Scheme

vers ce sous-ensemble soit toujours possible. En fait, ce sous-ensemble correspond *grosso modo* à la représentation intermédiaire qu'utilise GOLD. Quelques commentaires s'imposent toutefois.

Tout d'abord, pour les besoins de l'analyse, je supposerai que le programme a été α-converti (cette transformation donne des noms uniques á toutes les variables du programme). Ensuite, j'ai décidé de distinguer les appels de primitives (procédures standards de Scheme) des autres appels de fonctions. Cette distinction est parti-

culièrement importante si l'on veut obtenir des approximations plus précises dans un contexte de compilation séparée. Je discuterai de ce point précis à la section 5.2.3. De plus, je supposerai que le programme a subi une conversion des affectations (*assignment-conversion*). Ainsi, les variables mutées à l'aide d'expressions `set!` ne peuvent être que des variables globales. Finalement, les programmes ne sont pas CPS-convertis, comme pour l'analyse de Shivers. Du point de vue de l'analyse, il n'y a pas nécessairement d'avantage à utiliser la forme CPS ou directe (la combinaison d'un heuristique d'intégration á une analyse en style direct est comparable à une analyse sur un programme converti en forme CPS, tel que montré par Felleisen et al. [46]). Étant donné que Gambit-C utilise la forme directe comme représentation intermédiaire, j'ai pris le parti d'implanter mon analyse à partir de cette forme, ce qui évite le coût d'une transformation supplémentaire.

5.2.1 Les valeurs abstraites

Les valeurs abstraites manipulées par GOLD sont beaucoup plus nombreuses et plus riches que celles de Shivers (cet ensemble de valeurs sera défini formellement plus loin) et comparables à celles du compilateur Bigloo (à l'exception des approximations de valeurs numériques). Ces valeurs abstraites peuvent être classées en 3 catégories de la manière suivante : les valeurs atomiques, les valeurs numériques et les valeurs structurées.

L'élaboration de l'ensemble des valeurs abstraites m'a permis de constater un fait intéressant : la conception d'une analyse statique ne peut se faire sans avoir une idée, ne serait-ce que vague, des optimisations que l'on désire réaliser, ainsi que du système dans lequel cette analyse sera implantée.

Par exemple, pour les valeurs numériques, je voulais d'abord abstraire la tour des nombres de Scheme sans me soucier de leur représentation dans la machine. En suivant cette approche, je n'aurais pas introduit la valeur `Flonum`. Or une classe importante d'optimisations présuppose la possibilité de distinguer les expressions dont la valeur

est toujours un nombre en point flottant. Considérons par exemple l'expression `(+ x y)`. Typiquement, la primitive `+` teste le type de ses arguments et effectue au besoin des conversions avant d'appeler une fonction d'addition plus spécialisée (pour les entiers de taille fixe, les nombres en point flottant, etc.). Si l'analyse réussit à prouver que les variables `x` et `y` ne peuvent être liées qu'à des nombres en point flottant et que `+` est lié à la primitive Scheme, alors il est possible de générer un code beaucoup plus efficace pour l'expression en question, en éliminant toute forme de discrimination sur le type des arguments et en appelant directement la fonction spécialisée (`##flonum.+` dans le cas de Gambit-C). Cette optimisation est essentielle pour la compilation d'applications scientifiques pour lesquels l'efficacité des opérations sur les nombres en point flottant est cruciale. J'ai donc finalement décidé d'inclure `Flonum` dans l'ensemble des valeurs abstraites numériques.

Voici maintenant la description détaillée des 3 catégories de valeurs abstraites :

Valeurs atomiques Les valeurs atomiques dénotent soit des valeurs uniques, comme `True`, `False`, `Null` et `Eof`, soit des ensembles finis ou infinis dénombrables de valeurs de même type, comme `Char`, `String`, `Symbol`, `Port`.

Valeurs numériques Ces valeurs abstraites sont utilisées pour représenter des intervalles de valeurs Scheme de type numérique. Parmi ces valeurs abstraites, on retrouve `Flonum`, `Real` et `Complex`. Ces valeurs représentent respectivement les nombres réels représentables par la machine par des nombres en point flottant (communément appelés *flonums*), des nombres réels quelconques et les nombres complexes.

À cela, il faut ajouter un ensemble de valeurs qui représentent des intervalles d'entiers représentables par un mot machine de 32 bitsn (moins les 2 bits pour le type), les *fixnums*. Ces valeurs sont de la forme `Integer`$(\min, \max)$ où

$$\texttt{MINFIXNUM} \leq \min \leq \max \leq \texttt{MAXFIXNUM}$$

où `MAXFIXNUM` $= 2^{29}-1$ et `MINFIXNUM` $= -2^{29}$, et une valeur spéciale représentant tout l'ensemble des entiers naturels, `Integer`$(-\infty, \infty)$.

Valeurs structurées Les valeurs structurées représentent des valeurs Scheme pouvant contenir des références à d'autres valeurs, comme les paires, les vecteurs et les fermetures (par le biais de l'environnement). Afin d'abstraire ces valeurs concrètes, l'analyse introduit une notion (assez classique) d'*adresse virtuelle* faisant référence aux sous-éléments de ces structures (voir par exemple [4]). L'ensemble des adresses virtuelles utilisées lors de l'analyse dépend du programme source.

Les valeurs structurées sont de la forme :

- `Pair`$^l(a_1, a_2)$: une telle valeur représente une paire créée par un appel à toute primitive allouant une ou des paires (comme `cons`, `list`, `map`, `append`, etc.) et dont l'approximation du premier élément se trouve à l'adresse a_1 et l'approximation du second élément à l'adresse a_2.
- `KVector`$^l(a_1, \ldots, a_n)$: ces valeurs sont utilisées pour représenter des vecteurs de taille fixe créés par une application de la procédure Scheme `vector`. a_i est l'adresse de l'approximation du $i$ème élément du vecteur.
- `UVector`$^l(a_n, a_e)$: ces valeurs correspondent à des vecteurs dont la taille n'est pas connue statiquement, résultant d'un appel à `make-vector`, par exemple, ou d'un appel `(apply vector l)`. a_n est l'adresse de l'approximation de la longueur du vecteur et a_e celle de l'approximation de la valeur des éléments du vecteur (tous éléments confondus).
- `Closure`$^l(a_R, a_1, \ldots, a_n)$: ces valeurs représentent des fermetures dont le résultat est approximé à l'adresse a_R et le $j$ème paramètre formel est approximé à l'adresse a_j.

Les adresses virtuelles d'une valeur structurée sont toutes distinctes (deux valeurs structurées ne peuvent faire référence à la même adresse virtuelle). Ces adresses seront omises lorsque le contexte ne les requiert pas. L'exposant l, une étiquette, est aussi unique pour chaque valeur et correspond à l'étiquette de l'expression du programme qui a permis de générer la valeur abstraite (généralement le site d'appel d'une primitive d'allocation). La génération de l'ensemble des valeurs structurées d'un programme est décrite plus loin.

L'ensemble des valeurs abstraites manipulées par GOLD lors d'une analyse dépend du programme source. En effet, l'ensemble des valeurs structurées est propre au programme source puisque ces valeurs sont associées à des sites d'appel particuliers du programme. Les prochaines définitions synthétisent cette idée.

Définition 5.1

$\mathcal{V}_a$ dénote l'ensemble fini des valeurs abstraites atomiques (cet ensemble ne varie pas d'un programme à l'autre).

Définition 5.2

$\mathcal{V}_n$ dénote l'ensemble (fini) des valeurs abstraites de type numérique (cet ensemble ne varie pas d'un programme à l'autre).

Définition 5.3

Je noterai $\mathcal{V}_s^{\Pi}$ l'ensemble des valeurs abstraites structurées engendrées par le programme Π et par $\mathcal{L}^{\Pi}$ l'ensemble des adresses virtuelles engendrées par Π.

Définition 5.4

$\mathcal{V}^{\Pi}$ dénote l'ensemble des valeurs abstraites pouvant possiblement être utilisées lors de l'analyse du programme Π. Cet ensemble est défini par l'équation suivante :

$$\mathcal{V}^{\Pi} = \mathcal{V}_a \cup \mathcal{V}_n \cup \mathcal{V}_s^{\Pi}$$

Définition 5.5

Une ***valeur abstraite*** *est donc un élément de $\mathcal{V}^{\Pi}$.*

À une exception près, toutes les valeurs abstraites qui viennent d'être décrites sont disjointes des autres valeurs, c'est-à-dire qu'une approximation peut contenir toutes ces valeurs abstraites simultanément. L'exception concerne les valeurs abstraites représentant les nombres de Scheme. Le treillis $(\mathcal{V}_n \cup \{\bot_n\}, \sqsubset)$ des valeurs abstraites numériques est défini de telle sorte que :

$$\bot_n \sqsubset \texttt{Integer}(i,j) \sqsubset \texttt{Real} \sqsubset \texttt{Complex},$$
$$\bot_n \sqsubset \texttt{Flonum} \sqsubset \texttt{Real}$$

et

$$\texttt{Integer}(i,j) \sqcup \texttt{Integer}(i',j') = \texttt{Integer}(\min(i,i'), \max(j,j')),$$
$$\texttt{Integer}(i,j) \sqsubset \texttt{Integer}(i',j') \iff i' \leq i \leq j \leq j'.$$

Ainsi, par exemple, $\texttt{Integer}(i,j) \sqcup \texttt{Flonum} = \texttt{Real}$. Ce treillis est consistant avec la tour des nombres de Scheme ($\bot_n$ représente l'absence d'approximation numérique).

Définition 5.6

Soit

- $T_a = (\mathcal{P}(\mathcal{V}_a), \subseteq)$ *le treillis des sous-ensembles de* $\mathcal{V}_a$,
- $T_s^{\Pi} = (\mathcal{P}(\mathcal{V}_s^{\Pi}), \subseteq)$ *le treillis des sous-ensembles de* $\mathcal{V}_s^{\Pi}$, *et*
- $T_n = (\mathcal{V}_n \cup \{\bot_n\}, \sqsubset)$ *le treillis des nombres décrit précédemment.*

Une ***approximation*** X *est un élément de* $T_a \times T_s^{\Pi} \times T_n$. *Cet ensemble forme bien un treillis. Si* $X = \langle x_a, x_s, x_n \rangle$ *et* $Y = \langle y_a, y_s, y_n \rangle$, *il suffit de poser*

$$X \sqcup Y = \langle x_a \cup y_a, x_s \cup y_s, x_n \sqcup y_n \rangle,$$
$$X \sqsubset Y \iff x_a \subseteq y_a \wedge x_s \subseteq y_s \wedge x_n \sqsubset y_n.$$

Définition 5.7

Dans le reste du texte, $\mathcal{A} : Var \cup \mathcal{L}^{\Pi} \rightarrow T_a \times T_s^{\Pi} \times T_n$ *dénotera la fonction qui, étant donné une variable ou une adresse, retourne l'approximation correspondant à cette variable ou à cette adresse.*

Afin d'alléger la notation, j'utiliserai la convention suivante : soit V *une valeur abstraite, alors*

$$V = \mathcal{A}(\mathtt{x}) \iff \mathcal{A}(\mathtt{x}) = \langle \{V\}, \emptyset, \bot_n \rangle \vee \mathcal{A}(\mathtt{x}) = \langle \emptyset, \{V\}, \bot_n \rangle \vee \mathcal{A}(\mathtt{x}) = \langle \emptyset, \emptyset, V \rangle$$

$$V \in \mathcal{A}(\mathtt{x}) \iff V \in x_a \vee V \in x_s \vee V = x_n \quad \text{où} \quad \mathcal{A}(\mathtt{x}) = \langle x_a, x_s, x_n \rangle$$

où x *est une variable ou une adresse.*

Généralement, les analyses statiques sont décrites en utilisant des fonctions pures (au sens mathématique), i.e. où les effets de bord ne sont pas permis. Pour y arriver, ces fonctions utilisent des caches qui sont elles-même des fonctions et qui modélisent des tables dans lesquelles sont stockées des informations pertinentes à l'analyse. Malheureusement, ces tables doivent être passées en paramètre à toutes les fonctions intermédiaires qui sont définies, alourdissant d'autant la notation.

Bien que la rigueur mathématique soit de mise, j'ai opté ici pour une notation beaucoup plus proche des langages impératifs pour la description des éléments de l'analyse qui suivront. La raison principale est que la description du compilateur abstrait en sera simplifiée puisque les programmes d'analyse ressemblent plus à des programmes en assembleur qu'à des programmes fonctionnels.

Bien sûr, j'ai vérifié que la notation utilisée peut être exprimée mathématiquement. Par exemple, il est possible de modifier la table des approximations des variables $\mathcal{A}$ à l'aide de l'opérateur d'affectation: $\mathcal{A}(\mathtt{x}) \leftarrow A$ change l'approximation de x pour A. Formellement, il faudrait passer la table (fonction) $\mathcal{A}$ en paramètre à toutes les fonctions définissant l'analyse et simuler l'affectation en étendant la table de manière purement fonctionnelle : $\mathcal{A}' = \mathcal{A}[x \mapsto A]$. Mais le résultat recherché est le même.

5.2.2 Structure plate.

Lors de l'analyse d'un programme à l'aide de la 0CFA, une application abstraite force l'évaluation (abstraite) du code des fermetures pouvant être liées à l'expression en position applicative. Cette approche, combinée à une technique de *timestamp,* permet, lors du calcul du graphe de flot de contrôle, de déterminer les fonctions inutilisées dans un programme, tout en évitant de boucler même si le programme est récursif.

L'analyse implantée dans GOLD est essentiellement une simplification de celle de Shivers, résultant en une analyse dans certains cas plus conservatrice, mais plus simple à compiler efficacement. L'idée de base est de parcourir le code de toutes les expressions du programme Scheme une seule fois lors de chaque itération de l'algorithme de recherche du point fixe. Dans ce contexte, une application abstraite n'est plus traitée que comme une propagation d'informations des paramètres actuels vers les paramètres formels des fermetures pouvant être liées à l'opérateur de l'application.

Cette technique permet d'éviter la gestion des *timestamps* puisque le code de chaque procédure n'est parcouru qu'une seule fois par itération. Par contre, il devient plus difficile de détecter les procédures qui ne seront pas appelées lors de l'exécution du programme (limitant ainsi l'élimination de code mort). Dans l'exemple suivant :

```
(define (foo x) (+ x 1))

(define (bar) (foo 'a))

(display (foo 2))
```

le corps de la fonction `bar` sera parcouru à chaque itération, augmentant $\mathcal{A}(\mathtt{x})$ du symbole `'a`, et ce même si `bar` n'est jamais appelé. Ceci résulte donc en une approximation conservatrice pour `x`.

Cette technique offre toutefois un avantage considérable: elle rend la compilation de l'analyse presque triviale. En effet, puisque le code de chaque procédure est parcouru une et une seule fois lors de chaque itération, il n'est pas nécessaire d'inclure

un pointeur de code dans la représentation des fermetures abstraites, uniquement des approximations pour les paramètres formels et pour le résultat. Une application abstraite peut alors être compilée en une seule instruction opérant la propagation des valeurs des paramètres actuels vers les paramètres formels. Ainsi, il est possible de compiler (abstraitement) un programme en une suite d'instructions abstraites exécutées séquentiellement sans branchement, à l'exception des expressions conditionnelles (`if`, `and` et `or`), qui peuvent introduire des branchements conditionnels vers l'avant.

5.2.3 Détection des primitives

La sémantique de Scheme a cela de particulier qu'elle permet la redéfinition des primitives du langage. Cette propriété du langage s'avère utile dans plusieurs contextes. Par exemple, dans un but de déverminage, un programmeur pourrait arrêter momentanément l'exécution de son programme, redéfinir la primitive `cons` et ensuite continuer l'exécution. Aussi, certaines primitives peuvent être redéfinies afin d'être applicables à de nouveaux types de données. Par exemple, on peut vouloir redéfinir `for-each` pour qu'il soit possible d'appliquer une fonction sur d'autres types de séquences de valeurs que les listes.

S'il veut respecter la sémantique du langage, le compilateur ne peut donc pas faire l'intégration des primitives, au prix d'une sérieuse perte de performance. Lors de la production d'un système final, par contre, le programmeur peut décider d'indiquer au compilateur, à l'aide d'annotations au programme ou d'options de compilation, que certaines primitives ne sont pas redéfinies par le programme et qu'elles peuvent être intégrées au besoin.

Comme j'ai pris le parti de faire l'analyse statique des programmes vierges de toutes annotations, la tâche du compilateur s'en trouve plus ardue. En effet, un problème se pose : comment déterminer, lors de la compilation d'un module A, qu'une primitive n'a pas été redéfinie dans un module B? D'autant plus que cette propriété

est en général globale et dynamique. Considérons le bout de programme suivant :

```
(if *debug*
    (set! cons
          (let ((cons cons))
            (lambda (x y)
              (display (list 'cons x y)
              (newline)
              (cons x y))))))
```

où la variable `*debug*` est définie dans un autre module. Si `*debug*` est toujours liée à `#f`, alors la primitive `cons` ne sera pas redéfinie. Or seule une analyse au niveau global pourrait permettre de déterminer si `*debug*` est toujours liée à `#f`.

Afin de résoudre ce problème, GOLD procède par itérations successives. Il suffit de considérer le programme d'analyse globale comme une fonction prenant en entrée un ensemble de primitives redéfinies et calculant l'ensemble des primitives redéfinies par le programme. Je noterai $\mathcal{A}_i : \mathcal{P}(Prim) \rightarrow \mathcal{P}(Prim)$ une telle fonction.

La première étape consiste à produire tous les programmes d'analyse des différents modules en supposant d'abord qu'aucune primitive n'est redéfinie. Le fait de savoir qu'une primitive est redéfinie empêche le traitement polyvariant de cette primitive, ce dont je discute à la section 5.2.4.

L'analyse globale est ensuite lancée, calculant $P_1 = \mathcal{A}_1(\emptyset)$. Si $P_1 \neq \emptyset = P_0$, alors tous les modules contenant des applications de primitives contenues dans P_1 sont *invalidés*, ce qui veut dire qu'il faut les réanalyser localement et produire de nouveaux programmes d'analyse. L'analyse globale est lancée de nouveau, produisant un ensemble $P_2 = P_1 \cup \mathcal{A}_2(P_1)$ (il faut noter que la fonction n'est plus la même puisque certains programmes d'analyse auront changé). Si $P_2 \neq P_1$, on invalide certains modules et on recommence. Le système atteint un point fixe lorsque $P_n = P_{n-1}$ $(n \geq 1)$. Notons que puisque $P_n \subseteq P_{n-1}$ et que $Prim$ est un ensemble fini, le point fixe est toujours atteint en un nombre fini d'itérations.

Pour un programme ne redéfinissant aucune primitive, cette méthode a l'avantage

de ne requérir qu'une seule itération. Si la supposition initiale était que toutes les primitives sont redéfinies, il faudrait dans ce cas au moins deux itérations.

5.2.4 Polyvariance des primitives

Comme il a déjà été mentionné à de nombreuses reprises, l'analyse effectuée par GOLD est monovariante. Malgré tout, afin de produire des approximations assez précises, certaines primitives Scheme doivent être traitées de manière polyvariante. Par exemple, il serait insensé de fusionner tous les appels à `cons` et de manipuler une seule approximation grossière d'une paire lors de l'analyse d'un programme de plusieurs dizaines de milliers de lignes.

Comme je l'ai déjà mentionné, toutes les primitives d'allocation de base (`cons`, `vector` et `make-vector`) sont traitées de manière polyvariante par l'analyse. Autrement dit, chaque site d'appel engendre une valeur structurée distincte, dont les adresses sont aussi distinctes des adresses référencées par les autres valeurs structurées. Lors de l'analyse proprement dite, les approximations des sous-structures sont augmentées des approximations correspondant aux arguments passés à ces primitives. L'allocation des valeurs structurées est faite lors de l'initialisation de l'analyse locale en parcourant l'arbre d'analyse du module. Par exemple, une valeur structurée $\texttt{Pair}^l$ est créée pour chaque application explicite de `cons` dans le programme[2]. Il en va de même pour les autres primitives d'allocation ainsi que pour les formes `lambda` qui, elles, alloueront des valeurs $\texttt{Closure}^l$.

Afin d'obtenir une analyse plus précise encore, quelques autres primitives de Scheme sont aussi traitées comme primitives d'allocation par la phase d'analyse. Pour chacune de ces primitives, une primitive abstraite existe dans GOLD. Des valeurs structurées sont allouées pour les appels à ces primitives au cours de l'initialisation. Quelques une de ces primitives, avec les valeurs structurées qui sont allouées, sont

[2] En autant que `cons` ne soit pas redéfinie dans le programme.

Primitive	Valeur allouée	Initialisation

`append`
`reverse`
`vector->list`
`map`
	$\texttt{Pair}^l(a_1, a_2)$	$\mathcal{A}(a_1) = \langle \emptyset, \emptyset, \bot_n \rangle$
		$\mathcal{A}(a_2) = \langle \{\texttt{Null}\}, \{\texttt{Pair}^l\}, \bot_n \rangle$

`list->vector`	$\texttt{UVector}^l(a_1, a_2)$	$\mathcal{A}(a_1) = \texttt{Integer}(0, \texttt{MAXFIXNUM}/1024)$ $\mathcal{A}(a_2) = \bot$

Figure 5.2. Autres primitives allouantes.

données à la figure 5.2.

Notons enfin que toutes les autres primitives du langage, comme `+`, `car`, `integer->char`, etc., sont aussi traitées de manière polyvariante, mais que les appels à ces primitives ne nécessitent pas d'allocation de valeurs structurées à l'initialisation.

5.2.5 Filtrage et intervalles d'entiers

Afin d'obtenir des approximations plus précises sans compromettre la vitesse d'analyse, GOLD réalise deux techniques complémentaires: le filtrage de types (voir [4, 35]) et le rétrécissement d'intervalle, technique calquée sur la précédente mais appliquée aux intervalles d'entiers de taille fixe. Ces techniques sont toutes deux utilisées dans le contexte des expressions conditionnelles, mais permettent d'atteindre des buts différents.

Le filtrage de types. Grâce au typage dynamique, Scheme permet l'écriture de fonctions polymorphiques. Le programmeur doit donc inclure lui-même des tests de type explicites afin de discriminer sur le type des arguments. Considérons le court programme suivant :

```
(define (bar x)
  (+ x 1))
```

```
(define (foo y)
  (if (integer? y)
      (bar y)
      0))

(+ (foo 4) (foo 'quux))
```

où la fonction `foo` est polymorphique. Puisque l'analyse décrite ici est basée sur la 0CFA et par conséquent monovariante, on aura $\texttt{Symbol} \in \mathcal{A}(\texttt{y})$ et $\texttt{Integer}(4,4) \in \mathcal{A}(\texttt{y})$. De plus, $\mathcal{A}(\texttt{x})$ devrait contenir la même information (`y` est passé en paramètre dans l'appel à `bar`). Or l'appel `(bar y)` est exécuté seulement si `y` est un entier, ce qui implique que `x` ne sera jamais un symbole lors de l'exécution du programme, seulement un entier.

Afin de restreindre la taille de $\mathcal{A}(\texttt{x})$, l'analyse de GOLD utilise la technique de *filtrage de types*. L'idée est d'utiliser l'information procurée par le test de type d'une expression conditionnelle afin de restreindre certaines approximations dans le conséquent et l'alternative. Cette idée n'est pas nouvelle. Elle apparaît dans la thèse de Shivers au chapitre 9. Par contre, la réalisation de cette idée dans GOLD ne rencontre pas le problème des abstractions discuté à la page 120 de ladite thèse. Mon analyse ne requiert donc pas l'utilisation de contours spéciaux ou d'une sémantique de reflux (*reflow semantics*).

Pour y arriver, le compilateur transforme et annote, uniquement pour les besoins de l'analyse, les expressions conditionnelles impliquant des tests de type en utilisant la règle suivante (où `v` n'est pas une variable globale) :

$$\texttt{(if } (T_? \texttt{ v}) \ E_c \ E_a\texttt{)} \quad \Rightarrow \quad \begin{array}{l} \texttt{(if } (T_? \texttt{ v}) \\ \quad \texttt{(let ((v1 v}\downarrow_{\sqsubseteq \hat{T}}\texttt{))} \ E_c[\texttt{v/v1}]\texttt{)} \\ \quad \texttt{(let ((v2 v}\downarrow_{\not\sqsubseteq \hat{T}}\texttt{))} \ E_a[\texttt{v/v2}]\texttt{))} \end{array}$$

où $T_?$ dénote la primitive testant l'appartenance au type (concret) T, $\hat{T}$ est le type abstrait correspondant au type concret T et $E[X/Y]$ dénote la substitution des oc-

curences libres de X dans E par Y. $\mathtt{v}\downarrow_{\sqsubseteq\hat{T}}$ est une annotation indiquant que lors de l'analyse, uniquement les valeurs abstraites de $\mathcal{A}(\mathtt{v})$ qui sont de type abstrait $\hat{T}$ doivent être ajoutées à $\mathcal{A}(\mathtt{v1})$ (lors de l'analyse de `(let ((x y))...)`, la liaison force l'ajout des valeurs abstraites de $\mathcal{A}(\mathtt{y})$ aux valeurs abstraites de $\mathcal{A}(\mathtt{x})$). À l'opposé, $\mathtt{v}\downarrow_{\not\sqsubseteq\hat{T}}$ filtre les valeurs abstraites de $\mathcal{A}(\mathtt{v})$ qui ne sont pas de type abstrait $\hat{T}$ et les ajoute à $\mathcal{A}(\mathtt{v2})$.

Cette transformation n'est pas sémantiquement correcte si v peut être mutée lors de l'évaluation de E_c ou E_a. Dans ce cas, $\mathcal{A}(\mathtt{v1})$ (resp. $\mathcal{A}(\mathtt{v2})$) ne reflèterait pas cette mutation. C'est pourquoi j'ai restreint v à n'être qu'une variable locale. Notons d'ailleurs que cette transformation ne pourra être appliquée qu'à des expressions conditionnellles dont la variable locale v n'est pas mutée puisque le programme a subi une conversion des affectations. En effet, dans le cas d'une variable locale mutée, l'expression conditionnelle serait de la forme `(if (`$T_?$ `(##cell-ref v)) ...)`.

Ce filtrage de types est effectué si l'opérateur du test est une des primitives suivantes : `null?`, `boolean?`, `char?`, `string?`, `symbol?`, `pair?`, `vector?`, `procedure?`.

Les formes `if` dont le test est une variable sont aussi soumises au filtrage de types. Une telle expression apparaît souvent suite à un appel à `assq` :

```
(define (find-val key default-val)
  (let ((x (assq key *a-list*)))
    (if x (cdr x) default-val)))
```

Dans ce cas, l'approximation de x est restreinte aux valeurs abstraites différentes de `False` dans le conséquent (`False` est ainsi filtré) et $\mathcal{A}(\mathtt{x}) = \mathtt{False}$ dans l'alternative.

Approximations d'intervalles. Le treillis des valeurs abstraites qui a été décrit précédemment permet à l'analyse de dériver des intervalles de valeurs pour les variables contenant uniquement des *fixnums*. Du moins en théorie. En pratique, la principale difficulté consiste à limiter la taille de ces intervalles sans pour autant augmenter trop drastiquement le temps de calcul consacré à l'analyse. À titre d'exemple,

considérons la boucle suivante :

```
(let loop ((i 0))
  (if (> i len)
      i
      (begin
        (foo i)
        (loop (+ i 1)))))
```

Une première tentative pourrait être d'approximer `i` par $\texttt{Integer}(0,0)$ en raison de la liaison à la première ligne, ensuite par $\texttt{Integer}(0,1)$ à cause de l'appel `(loop (+ i 1))`, ensuite par $\texttt{Integer}(0,2)$ à cause du même appel, etc. En supposant que $\mathcal{A}(\texttt{len}) = \texttt{Integer}(0,255)$, il faudrait 256 itérations pour déduire que $\mathcal{A}(\texttt{i}) = \texttt{Integer}(0,256)$. C'est prohibitif!

Une approche diamétralement opposée consisterait à approximer `i` par $\texttt{Integer}(0,0)$ d'abord, à cause de la liaison initiale, et ensuite par $\texttt{Integer}(-\infty,\infty)$ puisque l'équation récursive $I = 0 \mid I + 1$ définit en fait les entiers positifs (`i` est lié soit à 0, soit à `(+ i 1)`).

En général, lorsque $\texttt{Integer}(-\infty,\infty) \in \mathcal{A}(\texttt{len})$, l'analyse ne peut faire mieux que d'approximer `i` par $\texttt{Integer}(-\infty,\infty)$, mais elle le fait beaucoup plus rapidement qu'en utilisant la première approche. L'analyse approxime donc `i` par $\texttt{Integer}(0,0)$ à cause de la liaison initiale. Ensuite, le résultat de `(+ i 1)` est approximé par $\texttt{Integer}(0,\texttt{MAXFIXNUM})$. En effet, puisque la borne supérieure de l'approximation de `i` est plus petite que `MAXFIXNUM` et qu'on ajoute 1 à `i`, alors `(+ i 1)` doit donc appartenir à l'intervalle $[0,\texttt{MAXFIXNUM}]$, ce qui est une approximation très conservatrice. Cette valeur est donc ajoutée à $\mathcal{A}(\texttt{i})$ en raison de l'appel à `loop`. Enfin, à la deuxième itération, l'appel à `(+ i 1)` sera approximé par $\texttt{Integer}(-\infty,\infty)$ puisque $\texttt{MAXFIXNUM} + 1$ est un bignum, un entier non-borné.

Dans certains cas particuliers, par contre, elle peut faire mieux. Pour cela, mon analyse implante une technique de *rétrécissement d'intervalle.* Le rétrécissement d'intervalle est appliqué lors de l'analyse des sous-expressions de certaines expressions

conditionnelles, tout comme pour le filtrage de types.

L'idée générale du rétrécissement d'intervalle est inspirée du filtrage de types. Considérons le fragment de programme suivant :

```
(if (< n len) A B)
```

où `A` et `B` sont deux expressions quelconques. Supposons qu'avant l'analyse de cette expression lors d'une certaine itération, $\texttt{Integer}(i, j) \in \mathcal{A}(\texttt{n})$ et que $\texttt{Integer}(k, l) \in \mathcal{A}(\texttt{len})$. Lors de l'analyse de `A`, on peut donc faire la supposition que $\texttt{n} < \texttt{len}$, ce qui implique que $\mathcal{A}(\texttt{n})$ peut être réduit à $\texttt{Integer}(i, \min(j, l-1))$. Similairement, $\mathcal{A}(\texttt{len})$ peut être réduit à $\texttt{Integer}(\max(i+1, k), l)$. (Des observations semblables peuvent être faites sur $\mathcal{A}(\texttt{i})$ et $\mathcal{A}(\texttt{len})$ lors de l'analyse de `B`.)

Partant de ces simples observations, le compilateur transforme et annote les expressions conditionnelles en utilisant un ensemble de règles, dont la suivante pour la primitive `<` :

$$
\begin{array}{lll}
 & & \texttt{(if (< x y)} \\
 & & \quad \texttt{(let ((x1 x}\downarrow_{<\texttt{y}}\texttt{) (y1 y}\downarrow_{>\texttt{x}}\texttt{))} \\
\texttt{(if (< x y)}\ E_c\ E_a\texttt{)} & \Rightarrow & \quad\quad E_c[\texttt{x/x1, y/y1}] \\
 & & \quad \texttt{(let ((x2 x}\downarrow_{\geq\texttt{y}}\texttt{) (y2 y}\downarrow_{\leq\texttt{x}}\texttt{))} \\
 & & \quad\quad E_a[\texttt{x/x2, y/y2}]\texttt{))}
\end{array}
$$

où $\texttt{x}\downarrow_{<\texttt{y}}$ est une annotation indiquant que lors de l'analyse, si $\texttt{Integer}(i, j) \in \mathcal{A}(\texttt{x})$ et $\texttt{Integer}(k, l) \in \mathcal{A}(\texttt{y})$, alors $\mathcal{A}(\texttt{x1}) \leftarrow \texttt{Integer}(i, \min(j, k))$, sinon $\mathcal{A}(\texttt{x1}) \leftarrow \mathcal{A}(\texttt{x})$. Les autres annotations suivent ce même principe. (De même que pour le filtrage de types, ces transformations ne peuvent être appliquées que si `x` et `y` sont des variables locales.)

L'avantage de cette technique est qu'elle permet d'accélérer le processus d'obtention du point fixe. Considérons l'exemple suivant, où `v` est une variable liée à un vecteur :

```
(let ((len (vector-length v)))
  (let loop ((i 0))
    (if (< i len)
        (begin
          (foo (vector-ref v i))
          (loop (+ i 1)))
        #f)))
```

Puisque la variable `i` itère sur tous les éléments d'un vecteur et que, en Gambit-C, la taille d'un vecteur est limitée à une taille beaucoup plus petite que le *fixnum* maximal (`MAXFIXNUM`/1024 − 1), il résulte que `i` ne peut être qu'un *fixnum*. Mon analyse, en utilisant les techniques qui viennent d'être décrites, permet d'arriver à cette conclusion en deux itérations seulement.

5.3 Le programme d'analyse

Le programme d'analyse, autant local que global, est représenté à l'exécution par des fermetures. Le compilateur abstrait qui le génère est similaire à celui décrit à la section 3.3. Évidemment, l'analyse de GOLD est plus sophistiquée et plus précise. Les différences majeures sont présentées ici. Lorsque nécessaire, l'algorithme de compilation abstraite sera décrit en utilisant une syntaxe très proche de Scheme.

Trois étapes sont nécessaires pour la compilation abstraite d'un module :

1. Les adresses virtuelles sont allouées. Ces adresses sont en fait des indices dans un tableau d'approximations et de valeurs abstraites structurées. Une adresse virtuelle est allouée pour chaque variable ainsi que pour toute expression autre qu'une référence à une variable. Le programme est annoté de ces adresses virtuelles. Les valeurs structurées et les adresses virtuelles de leurs champs sont allouées pour les appels à des primitives d'allocation comme `cons`, `vector`, etc.

   Des valeurs structurées sont aussi allouées pour les listes et vecteurs constants. Par contre, un algorithme de compression est appliqué afin de limiter le nombre

de valeurs structurées engendrées par de telles constantes. Plutôt que d'allouer une paire pour chaque élément d'une liste constante, une seule paire abstraite est allouée et le premier élément de cette paire contient l'union des approximations de tous les éléments de la liste, alors que le deuxième élément de la paire abstraite fait référence à la paire elle-même et contient aussi `Null` (si la liste est propre, bien entendu). Dans le cas d'un vecteur constant, un vecteur abstrait de type `UVect` est alloué. Si les éléments d'une structure constante (liste ou vecteur) sont des listes ou des vecteurs, une seule valeur abstraite de type `Pair` ou `UVect` sera créée pour tous ces éléments. Autrement dit, cet algorithme permet de conserver la profondeur d'une structure de données constante, sans toutefois générer un trop grand nombre de valeurs abstraites. Par exemple, la définition suivante :

```
(define x '(1 2 #(1 2) #(sym sym) (3 4) (x y)))
```

ne génère que trois valeurs structurées distinctes, soit une paire abstraite pour la liste liée à `x`, un vecteur abstrait pour les deux vecteurs et une autre paire abstraite pour les deux listes internes. Sans cet algorithme, il eut fallu allouer douze valeurs abstraites différentes.

2. Le tableau contenant toutes les approximations est alloué et les constantes sont initialisées.

3. Enfin, il y a la génération des fermetures proprement dite.

Une fois ces trois étapes accomplies, l'analyse locale est démarrée. Le tableau d'approximations est ensuite écrit sur disque ainsi que le programme d'analyse, dont la représentation externe sera décrite plus loin.

Conditionnelles. Pour la compilation de la 0CFA, une expression conditionnelle (expression `if`) était considérée comme un appel à une primitive, si bien que le code abstrait du conséquent et de l'alternative étaient exécutés, sans égard à la valeur abstraite du test. GOLD tient toutefois compte de cette valeur. Le code abstrait d'une telle expression est produit par une procédure Scheme semblable à celle-ci :

```
(define (closure-gen:IF tst-adr conseq altern)
  (let ((approx (vector-ref *approx-table* tst-adr)))
    (lambda ()
      (if (approx:is-true? approx)
          (conseq))
      (if (approx:is-false? approx)
          (altern)))))
```

On remarque que le test doit déjà avoir été compilé et que l'approximation du résultat de ce test doit se trouver à l'adresse virtuelle `tst-adr`.

Primitives d'allocation. Les primitives d'allocation sont, elles aussi, traitées de manière spéciale. Un appel à la primitive `cons`, par exemple, est compilé par une procédure semblable à celle-ci :

```
(define (closure-gen:CONS pair-adr arg1-adr arg2-adr)
  (let ((approx-arg1 (vector-ref *approx-table* arg1-adr))
        (approx-arg2 (vector-ref *approx-table* arg2-adr))
        (car-approx  (abst-pair:car pair-adr))
        (cdr-approx  (abst-pair:cdr pair-adr)))
    (lambda ()
      (types:approx-join car-approx approx-arg1)
      (types:approx-join cdr-approx approx-arg2))))
```

où l'appel à `abst-pair:car` (`abst-pair:cdr`) extrait l'approximation du premier (resp. deuxième) élément de la paire abstraite contenue à l'adresse virtuelle `pair-adr`.

Autres primitives. Les autres primitives de Scheme ont toutes un équivalent abstrait. Un appel C à une primitive P est compilé en un appel à sa version abstraite $\hat{P}$,

qui prend en paramètre l'adresse virtuelle de l'approximation du résultat ainsi que la liste des adresses virtuelles des approximations des paramètres actuels.

5.3.1 Factorisation

Dans tout module, il existe des expressions ne contribuant pas directement au résultat de l'analyse globale. Prenons par exemple l'expression suivante :

```
(let ((v (make-vector 1000 0)))
  (let loop ((i 0))
    (if (< i 1000)
        (begin
         (vector-set! v i i)
         (loop (+ i 1)))
        v)))
```

La valeur de la variable `i` ne dépend pas de l'interaction avec d'autres modules. C'est donc dire que l'analyse locale est suffisante pour déterminer l'appromixation de `i`. Il est donc inutile, lors de la génération du programme d'analyse, de produire du code en vue d'approximer davantage `i`.

Afin d'effectuer correctement cette optimisation des programmes d'analyse, GOLD propage une valeur abstraite spéciale, `Top`. Cette propagation se fait de la manière suivante, à chaque itération de l'analyse locale :

1. Au début de l'itération, `Top` est ajouté à toutes les approximations des variables globales, indiquant que ces variables peuvent être accédées ou mutées dans d'autres modules.

2. Les primitives abstraites propagent `Top` lors de l'exécution du programme d'analyse.

3. À la fin de l'itération, `Top` est propagé d'une approximation A à tous les champs des valeurs structurées contenues dans A. Par exemple, si $\texttt{Pair}^l(a_1, a_2) \in A$ et $\texttt{Top} \in A$, alors `Top` est ajouté à $\mathcal{A}(a_1)$ et $\mathcal{A}(a_2)$.

Lors de l'écriture sur disque du résultat de l'analyse locale, la valeur `Top` est enlevée de toutes les approximations. De plus, uniquement le code des expressions dont l'approximation d'au moins un des arguments ou de l'opérateur contient `Top` est préservé sur disque en vue de l'analyse globale.

5.3.2 Représentation sur disque

La dernière étape de la compilation locale d'un module est l'écriture sur disque de son programme d'analyse. Le fichier ainsi produit est une représentation du programme d'analyse sous la forme d'une séquence d'instructions pour une machine virtuelle dédiée à l'analyse statique de Scheme. Ces instructions correspondent essentiellement aux différentes fermetures créées pour l'analyse locale.

La seule différence notable est que cette représentation possède une notion d'étiquette, comme tout langage assembleur. Ces étiquettes sont utilisées dans les expressions conditionnelles. Par exemple, l'expression `(if tst conseq altern)` produirait la séquence d'instructions suivante :

```
    [ code de tst ]
    if-not-true? tst-adr etiq-1
    [ code de conseq ]
etiq-1:
    if-not-false? tst-adr etiq-2
    [ code de altern ]
etiq-2:
```

où `tst-adr` est l'adresse virtuelle contenant l'approximation du test `tst`. Les instructions `if-not-true?` et `if-not-false?` branchent à l'étiquette indiquée si l'approximation à l'adresse virtuelle donnée contient `False` (resp. contient autre chose que `False`).

Cette représentation est intéressante si l'on désire produire du code machine directement plutôt que de générer des fermetures pour l'analyse globale.

5.4 *Analyse globale et édition abstraite des liens*

La phase d'analyse globale comprend une étape d'édition *abstraite* des liens. Elle ressemble à une édition de liens conventionnelle. Les programmes d'analyse sont d'abord tous lus, ainsi que les résultats des analyses locales. Ensuite l'édition des liens remplace, dans chaque programme d'analyse local, les références à des variables libres (définies à l'extérieur du module) par des référence aux approximations de ces variables.

La technique utilisée pour faire cette "relocalisation" s'apparente à celle décrite dans [44], où il est question d'une technique de relocalisation de code machine basée sur la currification. Il s'agit de générer, pour chaque instruction virtuelle, non pas une mais deux fermetures imbriquées. La première prend en argument une table donnant, pour chaque variable globale du projet, l'adresse de l'approximation correspondante. Cette fermeture calcule les adresses virtuelles manquantes en vue de l'exécution de l'instruction virtuelle et retourne une seconde fermeture exécutant effectivement cette instruction. Dans les faits, les arguments de cette instruction virtuelle sont également des fermetures. Par exemple, la procédure de génération de code de relocalisation pour l'instruction virtuelle correspondant à $\mathcal{A}(\texttt{dest}) \leftarrow \mathcal{A}(\texttt{src})$ ressemble à la fonction Scheme suivante :

```
(define (codegen:join-approx dest src)
  (lambda (table)
    (let ((reloc-dest (dest table))  ; relocalisation
          (reloc-src  (src table)))  ; relocalisation
      (lambda ()
        (vm:join-approx reloc-dest reloc-src)))))
```

Une fois les programmes d'analyse ainsi retouchés, l'analyse globale peut être lancée. Comme pour l'analyse locale, il s'agit d'exécuter les programmes d'analyse des différents modules de l'application, autant de fois qu'il est nécessaire afin d'obtenir un point fixe.

5.5 *Les optimisations*

Le compilateur GOLD réalise trois types d'optimisations à partir des informations recueillies grâce à l'analyse globale intermodules. Ces optimisations sont l'intégration de fonctions au travers la barrière des modules, l'ajout d'annotations spécifiques au compilateur Gambit-C et l'intégration de certaines primitives Scheme.

Une fois ces optimisations faites, la β-réduction est appliquée de nouveau, suivie du λ-*lifting*. L'intérêt d'appliquer ces optimisations de nouveau est simple. L'efficacité de ces optimisations dépend fortement des annotations apportées par le programmeur. Par exemple, les fonctions définies au niveau global ne peuvent être intégrées à moins que le programme soit annoté de la déclaration `block`.

Je vais maintenant décrire chaque optimisation globale en détail et discuter de leur impact sur les autres optimisations de Gambit-C.

5.5.1 Annotations

La première optimisation consiste à annoter le programme de déclarations propres au compilateur Gambit-C. Comme je l'ai décrit plus haut, ces déclarations informent Gambit-C d'invariants de programme qui l'autoriseront à mieux optimiser le programme source. En fonction des résultats de l'analyse globale, GOLD annote les différentes expressions d'un programme des déclarations suivantes.

`(block).` Cette annotation indique que les variables globales définies dans la portée de cette annotation ne sont pas mutées à l'extérieur du module où apparaissent ces définitions. Dans le cas où une telle variable V est liée uniquement à une fonction, cette annotation permet à Gambit-C de faire l'intégration de cette fonction dans le module où elle apparaît lors de la phase de β-réduction. Également, les appels à ces fonctions sont compilés plus efficacement, puisqu'il n'est pas nécessaire de compiler la référence à la variable V et de tester le type du résultat pour s'assurer qu'il s'agit

bien d'une procédure.

Ainsi, une expression `(define` V `(lambda` $(V_1 \ \ldots \ V_n)$ $E)^l$) est annotée de la déclaration `block` si et seulement si $\mathcal{A}(V) = \texttt{Closure}^l$.

`(standard-bindings).` Cette annotation indique que, dans sa portée, les variables prédéfinies contiennent la primitive prédéfinie correspondante. Gambit-C peut ainsi générer des appels plus directs à ces primitives. Sans annotation, un appel à `car`, par exemple, implique de déréférencer la variable globale `car`, de tester si la valeur ainsi obtenue est une fonction et finalement de brancher au code de cette fonction. Grâce à cette annotation, seul le branchement au code de la primitive en question est effectué. Dans le cas de primitives qui ne peuvent causer d'erreur à l'exécution (`not`, `char?` , ...), la primitive peut être intégrée.

GOLD annote tous les appels aux primitives de Scheme qui ne sont pas redéfinis dans le projet. Les primitives ainsi appelées effectuent toutes les tests de type adéquats sur leurs arguments.

`(not safe).` Gambit-C permet deux modes de compilation, `safe` et `not safe`, qui peuvent être sélectionnés expression par expression. Dans le premier cas, tous les tests de type et de bornes sont effectués. Dans le deuxième cas, Gambit-C élimine tous les tests de type et de borne (sauf ceux explicites dans le programme, bien évidemment). En particulier, il n'y a pas de test s'assurant qu'une expression en position opérateur s'évalue bien en une fonction[3]. Dans le cas de très courtes fonctions, l'élimination de ce test peut être significatif.

GOLD annote le programme de cette déclaration dans deux contextes différents. D'abord, il y a les appels à des primitives, si le nombre des paramètres formels et actuels correspondent et que les types des arguments attendus sont bons. Combiné

[3] Avec Gambit-C, le test pour s'assurer que le nombre de paramètres actuels correspond bien au nombre de paramètres formels attendus est effectué par la fonction appelée et non par l'appelant.

```
+           -           *             modulo          quotient
remainder   <           <=            >               >=
=           zero?       odd?          even?
negative?   positive?   char->integer integer->char
```

Figure 5.3. Primitives considérées pour l'optimisation (`fixnum`).

à la déclaration `standard-bindings`, Gambit-C peut ainsi faire appel à des versions des primitives qui ne testent pas leurs arguments. Dans le cas de plusieurs primitives, comme `car`, `cdr`, etc., Gambit-C peut même faire l'intégration complète de ces primitives. Ce sont typiquement les primitives qui s'implantent par un accès dans une structure de données, des opérations arithmétiques, des tests divers, etc. En faisant l'intégration de ces primitives, Gambit-C élimine complètement le besoin d'un appel par le mécanisme de trampoline.

L'autre contexte correspond aux autres appels de fonctions. Un appel $(F\ E_1\ \ldots\ E_N)$ est annoté de cette déclaration si et seulement si $\mathcal{A}(F)$ ne contient que des valeurs structurées de type $\texttt{Closure}^l$.

`(fixnum)` et `(flonum)`. Certaines primitives arithmétiques, pour être intégrées complètement par Gambit-C (produisant ainsi un code sans appel de fonction), nécessitent l'introduction de l'annotation `(fixnum)` ou `(flonum)`. Ces annotations indiquent que la version de ces primitives optimisées pour les entiers représentables par un mot machine (typiquement 30 bits en Gambit-C) ou par un nombre en point flottant peut être utilisée.

Ainsi GOLD annote, si leurs arguments (et dans certains cas le résultat aussi) ont comme seule approximation $\texttt{Integer}(\alpha, \beta)$, avec $\alpha \neq -\infty$ et $\beta \neq \infty$, les appels aux primitives de la figure 5.3. L'annotation `(flonum)` est, pour sa part, appliquée aux appels aux primitives de la figure 5.4, si leurs arguments ont comme seule approximation `Flonum`.

```
+        -          *          /
floor    ceiling    round      expt  abs
sin      cos        tan        asin  acos
<        <=         >          >=    =
zero?    negative?  positive?
```

Figure 5.4. Primitives considérées pour l'optimisation (`flonum`).

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(fib 30)
(fib 30)
(fib 30)
(fib 30)
(fib 30)
```

Figure 5.5. La fonction de Fibonacci.

5.5.2 Intégration de primitives

Comme il a déjà été mentionné précédemment, le coût d'un appel à certaines primitives au travers la barrière des modules peut être grand. Pour s'en convaincre, considérons le programme de la figure 5.5, calculant 5 fois le 30ième nombre de Fibonacci.

Sur un ordinateur Pentium II de 400MHz avec 512Mo de mémoire vive, ce petit programme prend 25.4 secondes à s'exécuter lorsque compilé avec Gambit-C. Si on lui ajoute les déclarations suivantes (qui sont correctes puisque le résultat de (`fib` n) est bien un *fixnum* quand $n \leq 30$) :

```
(declare (standard-bindings)
         (not safe)
         (block)
```

```
(let ((y arg1))
  (if (let ()
        (declare (standard-bindings) (not safe))
        (pair? y))
      (let ()
        (declare (standard-bindings) (not safe))
        (car y))
      (car y)))
```

Figure 5.6. Transformation source-à-source de `car`.

```
        (fixnum))
```

le même programme s'exécute 0.86 secondes, soit près de 30 fois plus rapidement. C'est la version la plus efficace possible avec Gambit-C. Or l'analyse de GOLD ne parvient pas à déduire que le paramètre formel et le résultat de la fonction `fib` sont toujours des *fixnums*, ce qui l'empêche d'annoter le programme de manière optimale lors de l'optimisation globale. En fait, ceci se produit pour tous les appels à des primitives dont l'approximation d'au moins un des paramètres actuels est trop large. Dans le cas de `fib`, si on enlève seulement la déclaration `(fixnum)`, le programme s'exécute en 24.0 secondes, soit plus de 27 fois plus lentement que la version optimale. Et ce, simplement parce que les appels aux primitives arithmétiques et de comparaison doivent traverser la barrière des modules.

Dans le cas de plusieurs primitives, comme `car` et `cdr`, cela résulte en général de l'incapacité de GOLD à restreindre suffisamment les approximations dans les clauses d'une expression `if` trop complexe. Par exemple, mon analyse ne pourra pas déduire que l'appel à `caar` peut se faire sans test de type dans le code suivant :

```
(if (and (pair? x)
         (pair? (car x)))
    (let ((y (caar x)))
      ...
```

Afin de rendre plus efficaces ces appels sans compromettre la sémantique de

```
(let ((v arg1)
      (i arg2))
  (if (let ()
        (declare (standard-bindings) (extended-bindings)
                 (fixnum) (not safe))
        (and (vector? v)
             (##fixnum? i)
             (>= i 0)
             (< i (vector-length v))))
      (let ()
        (declare (standard-bindings) (fixnum) (not safe))
        (vector-ref v i))
      (vector-ref v i)))
```

Figure 5.7. Transformation source-à-source de `vector-ref`.

Scheme, GOLD fait lui-même l'intégration de certaines primitives par le biais de transformations source-à-source (équivalentes à des expansions de macros). En fait, GOLD ajoute des tests de type explicites pour s'assurer que la version la plus efficace de la primitive est appelée sur des arguments du bon type. Un appel à `(car arg1)`, par exemple, dont l'analyse n'a pu prouver que `arg1` est bien une paire, se verra transformé en l'expression de la figure 5.6.

Autrement dit, si l'argument à `car` n'est pas une paire, la version *safe* sera appelée, qui générera une erreur d'exécution tel que prescrit par le standard. Dans le cas contraire, la primitive la plus efficace sera appelée et aucun appel entre modules n'aura été effectué.

D'autres primitives, comme `vector-ref` et `string-set!`, doivent tester le type de plus d'un argument et faire aussi des tests de bornes. Un appel à la primitive `vector-ref`, par exemple, est transformé en une expression illustrée à la figure 5.7. Si l'analyse permet de déduire que le premier argument est effectivement un vecteur et que le deuxième argument est bien un *fixnum*, il est alors possible de réduire le test à

```
(let ((x1 arg1)
      (x2 arg2))
  (declare (standard-bindings))
  (if (let ()
        (declare (not safe))
        (and (##fixnum? a1)
             (let ()
               (declare (standard-bindings) (fixnum) (not safe))
               (let ((r (+ a1 a2)))
                 (if (negative? a2)
                     (< r a1)
                     (>= r a1))))))
      (let ()
        (declare (standard-bindings) (fixnum) (not safe))
        (+ a1 a2))
      (+ a1 a2)))
```

Figure 5.8. Transformation source-à-source de +.

```
        (and (>= i 0)
             (< i (vector-length v))))
```

En fait, chacune des sous-expressions de l'expression and peut être évitée. Les tests effectivement produits dépendent, à chaque site d'appel, du résultat de l'analyse globale.

Les primitives numériques sont certainement celles qui requièrent le plus de soin à implanter, du moins certaines. Considérons +, par exemple. Même si les arguments à + sont des *fixnums*, le résultat peut bien être un *bignum*. Or la primitive `##fixnum.+`, celle qui prend en argument seulement des *fixnums* et produit un *fixnum*, pourrait dans ce cas retourner un résultat erroné (une valeur négative lorsque les deux arguments sont positifs, par exemple). Il faut donc être extrêmement prudent lors de l'intégration. Dans le cas de +, la transformation produit, pour un appel `(+ arg1 arg2)` où `arg2` est un entier mais pas nécessairement un *fixnum*, l'expression de la figure 5.8.

Dans le cas d'un site d'appel à + pour lequel un des arguments est un *fixnum* et l'autre un *flonum*, GOLD introduit une conversion explicite du *fixnum* vers un *flonum* et remplace l'appel à + par un appel à `##flonum.+`. Il faut noter que cette optimisation n'est valide que si le *fixnum* en question est représentable sur 53 bits (bits dans la mantisse d'un *flonum*) ou moins, ce qui est le cas avec l'analyse de GOLD puisqu'un entier est considéré un *fixnum* si et seulement si son approximation entre dans l'intervalle [`MINFIXNUM`, `MAXFINUM`].

Les primitives intégrées suivant ce modèle, ainsi que le code généré pour chacune d'elles, sont décrites à l'annexe B. Bien entendu, l'implantantion de cette transformation dans GOLD suppose que les opérations sur des *fixnums* ou des *flonums* prédominent. Dans le cas d'applications traitant massivement des nombres rationnels, complexes ou entiers non-bornés (*bignum*), il faudrait prévoir une option inhibant cette optimisation pour les primitives arithmétiques afin de réduire leur coût à l'appel intermodules uniquement.

En appliquant ces transformations sur la fonction de Fibonacci, le programme ne prend plus que 1.07 secondes à s'exécuter, soit seulement 24% plus lentement que le programme optimal.

Notons enfin que cette seconde optimisation n'est rendue possible que grâce à une analyse globale du code. En effet, sans cette analyse, il ne serait pas possible d'être sûr que la fonction appelée est bien la primitive en question, du moins sans annotation de quelque nature que ce soit. Le compilateur BIGLOO, quant à lui, infère cette information localement grâce à un système de modules qui lui permet de déterminer quelles sont les fonctions ou variables importées. L'extension au langage Scheme acceptée par BIGLOO permet aussi d'annoter certaines variables de types, aidant davantage le compilateur lors de sa phase d'analyse de flot de contrôle.

5.5.3 Intégration intermodule

La programmation fonctionnelle favorise l'abstraction. En Scheme, en particulier, l'absence de structures de données définissables par le programmeur force ce dernier à utiliser des listes ou des vecteurs afin de simuler l'équivalent des `struct` de C ou des `record` de Pascal. Ces structures sont typiquement accédées via de courtes fonctions implantant les constructeurs, accesseurs et mutateurs, et regroupées dans un seul module/fichier. Leur utilisation dans d'autres modules peut donc entraîner un coût non-négligeable en raison du mécanisme de trampoline de Gambit-C.

Réalisation

Afin de réduire le coût d'appel de ces fonctions, GOLD tente d'intégrer les *courtes* fonctions au travers la barrière des modules (la classe des fonctions *candidates* pour cette intégration sera définie plus loin). Lorsqu'un module est analysé localement, les fonctions candidates sont encodées et emmagasinées dans une base de données globale, qui peut être modélisée par une fonction partielle

$$FC : \{l \mid \texttt{Closure}^l \in \mathcal{V}_s^{\Pi}\} \mapsto E_{\perp}.$$

telle que $FC(l) = \perp$ si la fonction correspondant à $\texttt{Closure}^l$ n'est pas candidate à l'optimisation globale, c'est-à-dire qu'elle ne satisfait pas le critère qui sera énoncé plus loin ($E_{\perp}$ est l'ensemble *pointé* $E \cup \{\perp\}$). Notons que cette "sérialisation" conserve exactement la structure des arbres de syntaxe abstraite de ces fonctions, y compris le partage de certains noeuds et les annotations.

Ces arbres de syntaxe sont reconstruits lors de la phase d'optimisation globale, sur demande seulement. Lors de cette phase, le programme source est traversé de nouveau. Si GOLD rencontre une expression `(V E1 ... En)` telle que $\mathcal{A}(\texttt{V}) = \texttt{Closure}^l$ et que $FC(l) \neq \perp$, elle sera transformée en une expression $(FC(l)$ `E1 ... En)` qui sera elle-même optimisée (récursivement). GOLD ne fait pas l'intégration de fonctions résultant

de l'évaluation d'expressions plus complexes. Pour ce faire, il faudrait déterminer si l'expression en question produit des effets de bord, ce que GOLD ne fait pas (il s'agirait d'une autre analyse globale qui n'a pas été implantée).

Fonctions candidates

L'intégration de fonctions est un sujet complexe en lui-même et faisant l'objet d'une littérature abondante. Je ne prétends donc pas proposer ici une solution nouvelle à ce problème, mais bien plutôt de motiver le choix d'un critère à la fois simple et utile.

Le but de cette intégration est avant tout d'*éviter de trop nombreux recours au coûteux mécanisme de trampoline pour les appels intermodules,* sans augmenter outre mesure la taille du code objet généré. Le critère premier est donc la vitesse d'exécution du programme optimisé.

En raison de la trampoline, il est très difficile de déterminer lors de l'analyse locale quelles fonctions sont susceptibles d'accélérer le programme si elles sont intégrées dans d'autres modules. Typiquement, les critères d'intégration dépendent à la fois de la fonction appelée et du site d'appel (voir [2] par exemple). Dans le cas de GOLD, deux problèmes se posent. Tout d'abord, les sites d'appels externes ne sont évidemment pas connus lors de l'analyse locale. De plus, afin d'éviter une utilisation excessive de la mémoire, la phase d'optimisation globale ne conserve pas une représentation interne de toutes les fonctions candidates. Elles sont créées au besoin.

Pour résoudre ce problème, seule une petite classe de fonctions peuvent être candidates à l'intégration globale. Le critère est appliqué lors de l'analyse locale seulement. Lors de l'intégration globale, tout appel d'une fonction candidate est optimisé sans autre forme de procès.

Le critère est le suivant : une fonction est *candidate* si et seulement *(1)* tous les appels internes sont faits à des primitives et *(2)* la taille de l'arbre de syntaxe abstrait (nombre de noeuds) du corps de la fonction est plus petit ou égal à $k \times n$ où n est le nombre de paramètres formels de la fonction. La partie *(1)* permet d'éviter qu'un

appel intermodule ne soit remplacé par plusieurs autres appels intermodules et évite d'avoir à traiter les intégrations de fonctions mutuellement récursives. La partie *(2)* limite l'expansion du code résultant de l'intégration. Au chapitre suivant, je mesurerai l'impact de plusieurs valeurs de k sur la taille de l'exécutable généré et sur la vitesse d'exécution.

Bien que ce critère soit très arbitraire, il semble donner de bons résultats en pratique, comme il sera montré au prochain chapitre.

Impact des autres optimisations

Les fonctions intégrées lors de cette phase d'optimisation conservent toutes les annotations permettant d'effectuer toutes les autres optimisations globales, en plus de la β-réduction et du λ-lifting de Gambit-C. Il est donc possible que cette intégration puisse entraîner d'autres optimisations.

Chapitre 6

RÉSULTATS EXPÉRIMENTAUX

Dans ce chapitre, les performances du système GOLD, décrit au chapitre précédent, sont évaluées. Dans un premier temps, le système est évalué sur un certain nombre de programmes unimodules. Ensuite, le système est évalué sur deux programmes multimodules de plus grande taille. Enfin, les conclusions de cette expérience sont tirées à partir des résultats obtenus.

Dans tous les cas, les résultats ont été obtenus sur une machine à deux processeurs Pentium II de 400 MHz avec 512 Mo de mémoire vive roulant le système d'exploitation Linux version 2.2.9. À l'exclusion de la quantité de mémoire vive, qui est particulièrement élevée puisque la machine en question sert de serveur de laboratoire, cette configuration correspond essentiellement à un poste de travail typique et n'a rien d'une machine idyllique rendant les résultats quelque peu irréalistes.

À moins d'indication contraire, tous les temps donnés sont en secondes.

6.1 Programmes unimodules

Afin de montrer l'intérêt de l'analyse statique de GOLD ainsi que l'efficacité de ses optimisations, un certain nombre de programmes de tests unimodules ont été compilés. L'absence d'appels entre modules autres que pour les primitives de Scheme permet de mieux cerner l'impact de certaines optimisations puisque, en principe, tous les appels intramodules sont compilés plus efficacement, voire même intégrés.

Une description des divers programmes du banc d'essai utilisé apparaît au tableau 6.1. Il s'agit de programmes typiques de bancs d'essais pour compilateurs Scheme optimisants, dont la taille s'échelonne de quelques dizaines de lignes à quel-

programme	lignes	description
`tak`	12	La fonction de Takeuchi.
`fib`	12	La fonction de Fibonacci.
`fibfp`	13	La fonction de Fibonacci en virgule flottante.
`takl`	27	La fonction de Takeuchi utilisant des listes comme compteurs.
`cpstak`	27	Une version CPS de la fonction de Takeuchi.
`primes`	28	Calcul des nombres premiers plus petits que 1000.
`nqueens`	32	L'algorithme de placement de 8 reines sur un échiquier.
`pnpoly`	44	Test de l'appartenance d'un point à l'intérieur d'un polygone bidimensionnel.
`mbrot`	48	Le calcul d'une partie de l'ensemble de Mandelbrot.
`dderiv`	82	Un programme de dérivation symbolique
`puzzle`	141	Résolution du casse-tête de Forest Baskett.
`simplex`	184	L'algorithme du simplexe.
`browse`	187	Programme de création et de parcours d'une base de données d'unités (en intelligence artificielle).
`boyer`	203 (+350)	Un programme de dérivation de preuve de théorèmes.
`earley`	648	Un générateur d'analyseurs syntaxiques basé sur l'algorithme d'Earley et l'exécution d'un analyseur.
`nucleic`	1000 (+2450)	Un algorithme de détermination de la structure tridimensionnelle d'un acide nucléique.
`slatex`	2337	Un convertisseur Scheme vers LaTeX.

Figure 6.1. Description des programmes unimodules.

ques milliers de lignes. La première colonne donne le nom du programme. La deuxième donne le nombre de lignes de code du programme et, dans deux cas, le nombre de lignes utilisées pour représenter les données relatives à l'exemplaire du problème à résoudre. Enfin, la dernière colonne décrit brièvement le programme.

6.1.1 Temps de compilation

Le tableau 6.2 donne quatre mesures relatives à l'analyse et à l'optimisation de chacun des programmes. La colonne **iter** donne le nombre d'itérations nécessaires pour l'obtention du point fixe lors de l'analyse locale. Ce nombre est relativement petit (entre 4 et 15), à l'exception de `nucleic` qui en requiert 46. Ceci est attribuable à la structure même du programme, et non à la présence de nombreuses structures constantes constituant la base de données de nucléotides.

La colonne **analyse** donne le temps d'exécution de l'analyse locale, qui est évidemment fonction du nombre d'itérations, de la taille et de la nature du programme. À l'exception de `nucleic` et `slatex`, ce temps est de moins d'une seconde et souvent négligeable. Il faut cependant noter que ces temps ne comprennent pas le temps d'initialisation de l'analyse (allocation des valeurs structurées et adresses virtuelles, génération des fermetures) ni le temps de sauvegarde du programme d'analyse sur disque. Le temps total est indiqué entre parenthèses. Notons aussi que la sauvegarde des programmes d'analyse n'a pas été implantée avec un souci d'efficacité. De plus, Gambit-C n'est pas une implantation particulièrement rapide lorsqu'il s'agit des entrées/sorties.

La colonne **liens** donne le temps total requis pour l'analyse globale, comprenant le chargement des programmes d'analyse (encore une fois pas très efficace), la génération du programme d'analyse globale, l'exécution de l'analyse, les optimisations et la génération du fichier C correspondant. Puisque ces programmes ne comportent qu'un seul module, une seule itération du programme d'analyse globale est requise pour l'obtention du point fixe : le résultat de l'analyse globale est le même que celui de

programme	iter	analyse	liens	gcc
fibfp	4	< 0.01 (0.07)	0.17	0.35 (0.51)
fib	4	< 0.01 (0.07)	0.20	0.44 (0.55)
tak	5	< 0.01 (0.07)	0.31	0.55 (0.54)
takl	5	< 0.01 (0.09)	0.39	0.78 (0.54)
primes	5	< 0.01 (0.10)	0.25	0.57 (0.53)
cpstak	5	< 0.01 (0.10)	0.52	0.71 (0.55)
nqueens	6	< 0.01 (0.12)	0.45	0.65 (0.52)
mbrot	6	< 0.01 (0.14)	0.50	0.92 (0.52)
dderiv	8	< 0.01 (0.17)	0.62	1.15 (0.55)
pnpoly	4	< 0.01 (0.21)	0.82	1.11 (0.56)
puzzle	8	0.04 (0.57)	1.75	2.22 (0.58)
simplex	8	0.10 (1.30)	4.44	9.42 (0.57)
browse	11	0.12 (0.69)	2.09	3.14 (0.54)
boyer	13	0.16 (0.65)	3.67	2.86 (0.71)
earley	13	0.40 (2.39)	6.71	14.30 (0.54)
slatex	13	1.40 (5.75)	20.96	26.98 (1.18)
nucleic	46	3.89 (6.87)	33.19	29.62 (0.91)

Figure 6.2. Mesures sur l'analyse locale des programmes unimodules.

l'analyse locale. C'est donc dire que le temps de génération de code C par Gambit-C prédomine.

Enfin, la colonne **gcc** donne le temps de compilation par `gcc` (version 2.7.2.3) du programme C résultant ainsi que du fichier C produit par la propre phase d'édition de liens de Gambit-C (le temps de génération de ce fichier est indiqué entre parenthèses). Le compilateur `gcc` est appelé avec les options `-O3 -fomit-frame-pointer`. On remarque que dans la plupart des cas, le temps d'analyse globale est plus faible que le temps de compilation du programme C. Seuls `boyer` et `nucleic` montrent des temps plus élevés. Ce sont en fait les deux seuls programmes qui contiennent de grandes quantités de données (plus importantes d'ailleurs que le programme lui-même). Il se trouve que la génération par Gambit-C des constantes dans le code C n'est pas très efficace (c'est très volumineux).

6.1.2 Temps d'exécution

La figure 6.3 donne les temps d'exécution des différents programmes du banc d'essai considéré. Trois temps ont été calculés pour chaque programme. La colonne **annot** donne le temps d'exécution du programme lorsque ce dernier est annoté manuellement afin d'obtenir des performances optimales. La colonne **gold** donne le temps d'exécution lorsque le programme est optimisé par GOLD. Le nombre entre parenthèses est le rapport entre ce temps et celui de la colonne **annot**. Enfin, la dernière colonne (**no annot**) donne le temps d'exécution du programme lorsqu'aucune annotation n'est ajoutée au programme. Le nombre entre parenthèses est le rapport entre ce temps et celui de la colonne **annot**.

On peut d'abord observer que près de la moitié des programmes optimisés par GOLD sont moins de 15% plus lents que la version annotée manuellement et que seuls cinq programmes sont plus de 40% plus lents. De plus, les programmes compilés par mon système sont, dans 10 cas, entre 10 et 25 fois plus rapides que le programme correspondant compilé sans aucune annotation. Analysons maintenant les écarts entre

programme	annot	gold		no annot	
dderiv	2.11	(0.95)	2.00	(4.38)	8.77
fibfp	15.46	(1.02)	15.70	(6.24)	98.03
slatex	6.23	(1.04)	6.46	(1.76)	11.38
cpstak	13.14	(1.05)	13.84	(8.85)	122.52
primes	7.35	(1.09)	8.02	(14.21)	113.98
mbrot	13.35	(1.11)	14.80	(8.61)	127.47
browse	5.86	(1.12)	6.55	(17.07)	111.86
boyer	4.21	(1.13)	4.77	(23.53)	112.21
nqueens	4.27	(1.19)	5.06	(21.58)	109.18
takl	3.62	(1.31)	4.75	(24.83)	117.83
fib	3.67	(1.36)	5.01	(23.28)	116.56
tak	3.63	(1.40)	5.09	(23.69)	120.51
earley	7.89	(1.90)	14.95	(3.27)	48.95
pnpoly	1.61	(2.03)	3.27	(22.12)	72.41
puzzle	3.54	(2.82)	9.98	(21.45)	214.09
simplex	4.29	(3.26)	13.99	(10.18)	142.37
nucleic	9.95	(4.84)	48.20	(2.79)	134.29

Figure 6.3. Temps d'exécution des programmes unimodules.

les temps de la colonne **annot** et ceux de la colonne **gold** avec plus de détails.

`boyer`, `browse` et `dderiv` sont des applications symboliques manipulant presque exclusivement des symboles et des listes. Une grande partie des tests de type ont été éliminés de ces programmes, expliquant les faibles différences entre les programmes optimisés par GOLD et ceux annotés manuellement. Dans le cas de `dderiv` la différence négative pourrait être expliquée par un ordonnancement différent des blocs de base au niveau du code généré par Gambit-C, impliquant de meilleures performances au niveau de la cache.

`tak`, `takl` et `cpstak` calculent tous les trois la fonction de Takeuchi. Dans le cas de `tak`, l'analyse est capable de déterminer uniquement que les valeurs numériques passées en paramètre sont des entiers, et non des *fixnum*. De ce fait, une soustraction est beaucoup plus coûteuse que dans la version optimisée manuellement, ce qui explique la différence de 40%[1]. Dans le cas de `takl`, la version utilisant des listes, aucun des appels à `cdr` ne peut être fait sans test de type (GOLD ne faisant pas de filtrage de type dans une forme `and` ou `or`), expliquant ainsi la différence de 31%. En ce qui concerne `cpstak`, la différence devrait en principe être similaire à celle observée pour `tak`. Or elle est seulement de 5%. Ceci s'explique par le fait que le coût d'une soustraction est amorti par le coût des nombreuses fermetures créées à l'exécution

[1] L'addition et la soustraction de deux *fixnum* peuvent être réalisées dans Gambit-C à l'aide d'une seule instruction machine (en faisant abstraction des tests de débordement, bien entendu) grâce à l'encodage particulier des types, soit les 2 bits les moins significatifs à 0. Les déclarations `(not safe)` `(fixnum)` doivent évidemment être présentes. Or si on ajoute des tests de type et de débordement, le coût de ces opérations devient beaucoup plus grand. Quelques tests sommaires ont montré qu'en testant uniquement le type d'un des deux arguments, une addition est plus de 5 fois plus lente. Les additions et soustractions de *flonum* sont, elles, plus de 2 fois plus lentes lorsque des tests de type sont ajoutés. De plus, chaque valeur intermédiaire en virgule flottante sera allouée dans le tas, produisant ainsi des coûts supplémentaires comme le *boxing/unboxing* et des appels supplémentaires au glaneur de cellules et empêchant Gambit-C de générer du code plus efficace. Tout ceci a pour effet de rendre plus important le coût de ces opérations.

pour les continuations explicites et donc du plus grand nombre de glanage de cellules, soit aucun *GC* pour `tak` contre 280 pour `cpstak`.

`fib` et `fibfp` calculent tous deux la fonction de Fibonacci, à la différence près que `fib` le fait à l'aide de nombres entiers alors que `fibfp` le fait à l'aide de nombres réels inexacts. Dans le premier cas, GOLD ne peut prouver que le résultat de la fonction `fib` est un *fixnum* alors que dans le deuxième cas, il peut montrer que le résultat est toujours un *flonum*. Compte tenu du coût d'une addition et d'une soustraction sur des *fixnum* (tests des type et de débordement), ceci explique que `fib` soit 35% plus lent et que `fibfp` soit uniquement 2% plus lent.

La version de `slatex` optimisée par GOLD est seulement 4% plus lente que la version annotée manuellement, ce qui est un excellent résultat pour un programme de cette taille. Par contre, la différence de 79% entre la version optimisée et la version non-optimisée semble indiquer que le coût des entrées/sorties prédomine dans cette application.

Pour `mbrot`, la plupart des opérations arithmétiques sont faites sur des nombres en virgule flottante. Il y a tout de même quelques opérations sur des entiers et des accès dans des tableaux. Ces dernières opérations ne peuvent être optimisées complètement par GOLD, ce qui explique l'écart de 11% entre la version optimisée et la version annotée manuellement. Il est intéressant de noter que le style d'écriture du programme empêche GOLD de faire un certain nombre d'optimisations. L'exemple le plus flagrant se produit à la ligne 14 (voir l'annexe C). La boucle de la fonction `count` effectue au plus 64 itérations. Un des critères d'arrêt de la boucle est `(if (= c max-count) ...)`. Or GOLD ne fait pas de rétrécissement d'intervalle lorsque le test est un `=`. Si le test avait été remplacé par `(>= c max-count)`, GOLD aurait pu prouver que `c` est un *fixnum*, produisant ainsi des appels plus performants à `=` (ligne 14) et `+` (ligne 22).

En ce qui concerne `nqueens` et `primes`, ces deux programmes manipulent à la fois des nombres entiers et des paires. Or ils sont écrits de telle sorte que, grâce au filtrage de types, seules les opérations arithmétiques requièrent des tests de type et de

débordement, expliquant ainsi les écarts de 9% pour `primes` et 19% pour `nqueens`. Notons que dans le cas `nqueens`, il y a un appel à `+` qui requiert plusieurs opérations arithmétiques ainsi que des tests (voir annexes).

`pnpoly` et `puzzle` sont deux programmes dont les opérations prédominantes sont les accès et les mutations dans des vecteurs. Or dans la majorité des cas, GOLD ne peut éviter les tests de bornes et le test du type de l'indice, ce dernier étant souvent approximé par `Integer`$(-\infty, \infty)$. Par conséquent, GOLD ne peut faire mieux que réduire l'écart à près d'un facteur entre 2 et 3 avec la version annotée manuellement[2].

Enfin, `simplex` et `nucleic` sont les deux programmes pour lesquels l'écart observé est le plus important, soit un facteur de 3.26 pour `simplex` et de 4.84 pour `nucleic`. Dans les deux cas, une grande proportion du temps de calcul est consacré à accéder et muter des valeurs dans des vecteurs et à faire des calculs en virgule flottante. Dans le cas de `simplex`, la plupart des opérations arithmétiques peuvent être optimisées pour les *flonum*, ce qui n'est pas le cas pour `nucleic`, et ce pour deux raisons. D'abord il y a quelques appels à `sqrt` dont l'approximation du résultat est un nombre complexe et non un *flonum* (la racine carrée d'un nombre négatif est un nombre complexe et GOLD ne fait pas la distinction entre un *flonum* négatif ou positif). De plus, la base de données de nucléotides est représentée par des vecteurs hétérogènes (contenant des vecteurs, des symboles et des nombres) implémentant une forme d'héritage simple. Les accès à ces données produisent alors des approximations trop grossières (comprenant par exemple `Flonum`, `Symbol` et des vecteurs de taille connue). Il en résulte que GOLD ne peut éliminer qu'un faible nombre de tests de type, obligeant par le fait même Gambit-C à *boxer* et *unboxer* toutes les valeurs intermédiaires.

[2] Afin de mieux évaluer le coût d'une mutation dans un vecteur, j'ai compilé un programme de test faisant 200 mutations consécutives dans un vecteur. Par rapport à une version annotée manuellement, ce programme s'exécute plus de 3.3 fois plus lentement lorsque les appels à `vector-set!` sont intégrés comme dans GOLD avec tous les tests de type et de borne, et plus de 10 fois plus lentement si le programme n'est ni annoté ni optimisé.

module	lignes	taille	iter	local	la	ap
err.scm	31	635	2	0.04 (0.00)	680	626
bv.scm	68	2278	5	0.16 (0.01)	3723	3534
util.scm	83	2424	3	0.18 (0.00)	4136	2884
lr-dvr.scm	85	3667	5	0.12 (0.00)	1514	1754
obj.scm	90	4396	4	0.08 (0.00)	1003	1699
globals.scm	93	1716	2	0.05 (0.00)	1561	1090
fv.scm	200	6263	6	0.32 (0.05)	11095	6664
match.scm	426	13609	5	0.55 (0.05)	14554	12790
comp.scm	1354	44156	10	2.32 (0.54)	75658	35897
erlang.y.scm	1392	61582	3	2.38 (0.07)	47871	38966
ast.scm	1418	46674	6	2.19 (0.25)	63377	43400
erlang.l.scm	1826	63609	17	5.25 (1.81)	161288	43660
Total	7066	251009	—	13.64 (2.78)	386460	192964

Figure 6.4. Mesures sur les modules de etos.

6.2 Programmes multimodules

Dans cette section, j'analyse les performances de GOLD sur deux programmes multimodules de taille plus considérable :

etos: un compilateur Erlang [3] produisant du Scheme écrit par Marc Feeley et Martin Larose [22], version 1.4.

Gambit-C: le compilateur Scheme de Marc Feeley, version 2.8a [19].

etos est composé de 12 modules. Ces modules sont donnés au tableau 6.4. La colonne **lignes** donne le nombre de lignes de chacun et la colonne suivante le nombre de caractères (ou octets). Comme on peut le constater, certains modules sont

module	lignes	taille	iter	local	la	ap
gsc.scm	9	271	2	0.02 (0.00)	72	51
_parms.scm	199	9724	2	0.24 (0.00)	6681	5919
_back.scm	219	8605	2	0.16 (0.00)	3062	2642
_t-c-3.scm	274	7484	4	0.35 (0.01)	6634	8974
_env.scm	359	10663	7	0.48 (0.04)	10488	8615
_host.scm	522	13971	4	0.46 (0.02)	10150	9125
_prims.scm	528	35742	1	0.15 (0.00)	219	31
_utils.scm	545	15060	8	1.06 (0.12)	27598	19621
_source.scm	1234	43971	8	1.78 (0.19)	40222	33309
_ptree2.scm	1709	58867	7	2.57 (0.28)	48603	61406
_t-c-1.scm	1709	53758	11	3.31 (0.69)	97788	46830
_gvm.scm	1841	65290	14	6.30 (1.26)	262513	63363
_ptree1.scm	2301	81829	10	4.68 (0.93)	141979	73054
_front.scm	2668	94875	7	4.57 (0.46)	95797	74451
_t-c-2.scm	3059	108887	6	3.64 (0.36)	103835	63085
Total	17176	608997	—	29.77 (4.32)	855641	470476

Figure 6.5. Mesures sur les modules de Gambit-C.

aussi petits qu'une trentaine de lignes alors que les 4 plus gros comportent plus de 1300 lignes. Parmi ces derniers, `erlang.l.scm` et `erlang.y.scm` ont été produits automatiquement par des outils de génération d'analyseurs lexicaux et syntaxiques respectivement. Gambit-C est, pour sa part, composé de 15 modules (tableau 6.5). Contrairement à **etos**, aucun des modules de Gambit-C n'a été généré automatiquement.

6.2.1 Analyse locale

Les tableaux 6.4 et 6.5 contiennent aussi diverses mesures sur l'analyse locale de chacun des modules des deux programmes. La colonne **iter** donne le nombre d'itérations requises pour l'obtention du point fixe. La colonne **local** donne le temps total de l'analyse locale comprenant l'initialisation, la génération des fermetures, la génération du programme d'analyse et la sauvegarde des données locales sur disque. Le temps de calcul du point fixe proprement dit est donné entre parenthèses (une valeur de 0.00 indiquant un temps négligeable).

Pour les modules de plus grande taille, ce temps est de plus de deux secondes. Mais outre `erlang.y.scm` pour **etos** et `_gvm.scm` pour Gambit-C, on remarque que le temps d'analyse locale proprement dite est bien en deçà d'une seconde. C'est donc qu'une grande proportion du temps de calcul n'est pas consacrée à l'analyse mais à d'autres traitements, tels la lecture du code source, les transformations et optimisations locales effectuées par Gambit-C, la génération du programme d'analyse, la sauvegarde du résultat de l'analyse, etc. Malgré tout, ce sont des temps raisonnables qui peuvent être réduits en optimisant chaque élément de cette chaîne.

Les deux dernières colonnes, **la** et **ap** donnent respectivement la taille du fichier contenant le résultat de l'analyse locale et la taille du fichier contenant le programme d'analyse du module. Dans tous les cas, ces fichiers sont d'une taille du même ordre de grandeur que le module lui-même. Même si aucun soin particulier n'a été pris pour minimiser la taille de ces fichiers, ils demeurent d'une taille raisonnable compte tenu de la capacité grandissante des disques durs actuels.

6.2.2 Édition de liens

Le tableau 6.6 donne certaines mesures relatives à la génération des deux exécutables à l'aide de GOLD. Dans la première partie du tableau, on retrouve le nombre d'itérations nécessaires pour l'obtention du point fixe, le temps de chargement des fichiers d'analyse

Programme original	**etos**	**Gambit-C**
Nombre d'itérations	16	15
Chargement des fichiers	5.5	14.9
Analyse globale (sec.)	27.5	699.9
Éditions de liens (sec.)	111.8	862.0
gcc (sec.)	68.7	141.7
Un seul module	**etos**	**Gambit-C**
Analyse	50.2	1350.1
Nombre d'itérations	20	18
Gambit-C (sec.)	95.8	392.7
gcc (sec.)	147.2	315.23

Figure 6.6. Statistiques de génération des exécutables.

et des résultats locaux, le temps d'exécution du programme d'analyse globale, le temps total de l'édition de liens réalisée par GOLD (chargement des programmes d'analyse, des résultats locaux, génération du programme d'analyse globale, son exécution, optimisations et génération du code C) et le temps de génération de l'exécutable par `gcc`. La deuxième partie du tableau donne quelques mesures semblables lorsque les deux programmes sont compilés en un seul module. Ces mesures sont : le temps d'analyse ainsi que le nombre d'itérations effectuées, le temps de compilation par **Gambit-C** et le temps de génération de l'exécutable par `gcc`.

etos. On remarque que le temps consacré à l'optimisation et la génération de code (C et code objet) surpasse grandement le temps d'analyse globale. Ceci est un résultat intéressant, montrant ainsi que l'analyse globale n'est, dans ce cas-ci, pas si coûteuse (d'autant plus que le programme d'analyse global n'est pas compilé vers du code machine). De plus, l'analyseur syntaxique (le module `erlang.y.scm`) contient un vecteur

dont 167 éléments sont des fermetures. Ce sont les actions associées aux réductions dans une table d'analyse LALR(1) (voir [1]). Si l'analyseur syntaxique de **etos** avait été écrit directement plutôt que produit par un générateur d'analyseurs syntaxiques, le temps d'analyse de ce module eût été certes plus court, rendant probablement l'analyse globale plus rapide.

Gambit-C. Dans le cas de Gambit-C, l'analyse globale est beaucoup plus coûteuse (sans être toutefois excessive), soit près de 12 minutes. Il se trouve que Gambit-C est écrit dans un style très fonctionnel avec beaucoup d'appels à des primitives d'allocation comme `map`, `reverse`, `append`, etc. Ceci accroît le nombre de valeurs abstraites manipulées, faisant ainsi gonfler la taille des approximations (qui sont de 630 en moyenne pour les ensembles contenant au moins deux éléments). On peut donc penser qu'une implantation plus efficace des routines de gestion des ensembles de valeurs abstraites permettrait d'obtenir de meilleurs résultats. La compilation du programme d'analyse vers du code machine ne nuirait pas non plus.

On remarque que, dans les deux cas, l'analyse globale est près de deux fois plus rapide que l'analyse locale d'un seul fichier contenant le code de toute l'application. De plus, même en ajoutant au temps de l'analyse globale la somme des temps d'analyse locale des modules, on est loin du compte (41.1 secondes contre 50.2 pour **etos** et 729.7 contre 1350.1 pour **Gambit-C**). Il y a quelques facteurs qui expliquent ce résultat. D'abord, il y a plus d'itérations effectuées lorsque tous les modules sont analysés dans un seul fichier (20 pour **etos**, 18 pour **Gambit-C**). Or les dernières itérations sont les plus coûteuses puisque les ensembles de valeurs abstraites contiennent plus d'éléments. De plus, lors de l'analyse globale, une partie du code n'est pas réanalysés grâce à la factorisation de l'analyse. Enfin, il faut prendre en compte la complexité de l'analyse, qui est cubique en pire cas. On remarque aussi que les temps de compilation par **Gambit-C** et `gcc` sont aussi plus grands (plus d'un facteur 2 pour `gcc`). Ceci est

programme	annot	gold		no annot	
etos	7.43	(**1.28**)	9.57	(2.23)	21.33
Gambit-C	8.91	(**1.29**)	11.51	(3.61)	41.53

Figure 6.7. Temps d'exécution.

en grande partie dû au fait que les algorithmes utilisés en compilation sont souvent plus que linéaires.

6.2.3 Exécution

Le tableau 6.7 donne les temps d'exécution des deux programmes compilés de trois manières différentes et utilisés, **etos** pour compiler 50 fois un programme Erlang de 165 lignes et Gambit-C pour compiler le module `_t-c-2.scm`, le plus gros de ses modules. Dans un premier temps (colonne **annot**), tous les modules ont été préfixés manuellement des déclarations suivantes :

```
(declare
  (block)
  (not safe)
  (standard-bindings)
  (fixnum))
```

à l'exception de quelques modules requérant une arithmétique générique. Dans ces quelques cas, l'annotation `(fixnum)` a été remplacée par `(generic)`. Ensuite, chacun des deux programmes a été compilé à l'aide de GOLD avec un facteur d'intégration $k =$ 6 pour **etos** et $k = 20$ pour Gambit-C (ce sont les facteurs qui permettent d'obtenir les meilleurs résultats). C'est la colonne **gold**. Enfin, la colonne **no annot** donne le temps d'exécution lorsque le programme est compilé par Gambit-C sans aucune optimisation ni annotation. Les nombres entre parenthèses donnent le rapport entre le temps obtenu pour une méthode donnée et le temps obtenu lorsque le programme est annoté manuellement.

On remarque donc que les programmes produits par GOLD sont moins de 30% plus lent que les mêmes programmes annotés manuellement et plus de deux fois plus rapides que les programmes sans annotation (plus de 3.5 fois pour Gambit-C). Plusieurs facteurs expliquent ces résultats :

1. Les noeuds des arbres de syntaxe abstraite manipulés par **etos** et Gambit-C sont représentés en Scheme par des vecteurs. Or GOLD n'est pas en mesure d'utiliser les prédicats testant les types des noeuds définis dans `ast.scm` et `_ptree1.scm` afin d'optimiser les accès aux champs de ces noeuds. Les accès aux éléments des vecteurs sont donc toujours relativement coûteux.

2. Les deux programmes sont écrits dans un style faisant un grand usage de primitives Scheme (comme `map`, `reverse`, `assq`, etc.) qui requièrent invariablement des appels intermodules via la trampoline (ces primitives sont implantées dans un module séparé). Tout ce que GOLD permet de sauver, dans la plupart des cas, c'est la référence à la variable globale à laquelle la primitive est liée et le test de type s'assurant que cette variable est bien liée à une procédure.

3. Les deux programmes sont essentiellement des applications symboliques. Or les tests de type qui ne peuvent être optimisés par GOLD sont souvent peu coûteux en comparaison du coût d'une opération arithmétique qui ne peut être optimisée.

4. Les entrées/sorties (d'ailleurs très coûteuses avec Gambit-C), comptent pour une certaine part du temps de calcul, amortissant ainsi certainement l'impact des optimisations de GOLD.

5. Il semble que les optimisations `(block)` et `(not safe)` permettent à Gambit-C d'effectuer de meilleures optimisations en réduisant le coût des appels intramodules et en intégrant bon nombre de fonctions.

k	**exécution** (sec.)	**exécutable** (Ko)	**gcc** (sec.)
0	10.31 (1.00)	772.5 (1.00)	55.8 (1.00)
2	10.00 (0.97)	789.5 (1.02)	57.5 (1.03)
4	10.03 (0.97)	845.9 (1.10)	65.0 (1.16)
6	9.57 (0.93)	879.6 (1.14)	68.7 (1.23)
8	9.79 (0.95)	1007.0 (1.30)	122.2 (2.18)
10	9.87 (0.96)	1005.5 (1.30)	119.0 (2.13)
20	9.82 (0.95)	1005.5 (1.30)	120.0 (2.15)
50	9.88 (0.95)	1005.5 (1.30)	120.6 (2.06)

Figure 6.8. Impact du facteur d'intégration (k) pour etos.

Comme nous allons le constater, l'intégration intermodules ne compte que pour 5% de cette différence dans le cas de **etos**, mais pour plus de 25% dans le cas de Gambit-C. Voyons donc l'impact du facteur d'intégration sur quelques aspects du système.

Intégration intermodule. Les tableaux 6.8 et 6.9 donnent trois mesures obtenues pour différentes valeur de k, le facteur d'intégration décrit au chapitre précédent. Ces mesures sont, dans l'ordre, le temp d'exécution du programme résultant, la taille de l'exécutable (en Ko) et le temps total de compilation du programme par gcc. Les nombres entre parenthèses donnent le rapport entre une mesure pour un k donné et la même mesure lorsque $k = 0$ (une valeur de 0 désactivant l'intégration).

Ces mesures montrent que les temps d'exécution peuvent être améliorés de manière significative grâce à l'intégration intermodule. **etos** s'exécute jusqu'à 7% plus rapidement lorsque $k = 6$ et Gambit-C jusqu'à 27% lorsque $k = 20$, et ce avec des exécutables 14% et 16% plus gros respectivement. La compilation des fichiers C résultants prend, respectivement, 23% et 36% plus de temps.

k	**exécution** (sec.)	**exécutable** (Ko)	**gcc** (sec.)
0	15.74 (1.00)	1785.7 (1.00)	104.5 (1.00)
2	14.98 (0.95)	1780.3 (1.00)	104.4 (1.00)
4	13.15 (0.84)	1935.4 (1.08)	126.1 (1.21)
6	12.74 (0.81)	1941.8 (1.09)	126.7 (1.21)
8	12.82 (0.81)	1948.4 (1.09)	128.2 (1.23)
10	11.66 (0.74)	1969.3 (1.10)	130.0 (1.24)
20	11.51 (0.73)	2077.6 (1.16)	141.7 (1.36)
50	11.63 (0.74)	2081.5 (1.17)	142.2 (1.36)

Figure 6.9. Impact du facteur d'intégration (k) pour Gambit-C.

Chose surprenante, le temps de compilation de **etos** par `gcc` est multiplié par 2 lorsque $k \geq 8$. Il se trouve qu'un module en particulier, `erlang.y.scm`, est responsable de cette augmentation prononcée. En effet, le fichier C produit pour ce module est près de 2.5 fois plus gros (673Ko lorsque $k = 0$ et 1595Ko lorsque $k = 8$), prend plus de 4 fois plus de temps à être compilé par `gcc` (13 secondes lorsque $k = 0$ et 65 secondes lorsque $k = 8$) et produit un fichier objet 2 fois plus gros (157Ko quand $k = 0$ versus 316Ko quand $k = 8$). Ceci s'explique par le fait qu'une fonction relativement grosse est intégrée 167 fois. Il se peut alors que le code ainsi produit exhibe un comportement superlinéaire de `gcc`.

Ces mesures montrent aussi que passé un certain seuil, le facteur d'intégration n'a plus d'impact vraiment significatif sur le temps d'exécution et que ce seuil est fonction du programme source. Dans le cas de **etos**, ce seuil est de 8, tandis qu'il se situe entre 10 et 20 pour Gambit-C. Ces résultats ne sont pas vraiment surprenants (ni intéressants en ce qui concerne le choix de k vu l'heuristique simpliste utilisé). Plus on permet à GOLD d'intégrer de grosses fonctions, plus l'exécutable sera gros et le

temps de compilation élevé. À l'opposé, les gains (en temps d'exécution) seront moins prononcés puisqu'à mesure que la fonction à intégrer est grosse, le temps consacré à l'appel intermodule via la trampoline devient plus négligeable. Il faut aussi mentionner que Gambit-C effectue déjà une phase d'intégration locale beaucoup plus sophistiquée.

6.3 Conclusion

Quelles conclusions peut-on tirer à la lumière de tous ces résultats?

Tout d'abord, même si GOLD ne permet pas d'obtenir des programmes aussi performants qu'en annotant explicitement le programme source, les temps obtenus s'en approchent, montrant que l'approche est valable.

Malgré tout, il est clair que l'analyse globale pourrait être plus rapide, surtout dans le cas de Gambit-C, même si elle n'est pas excessive. Un tel système pourrait s'avérer très utile dans un environnement de production, lors de la génération de l'exécutable final, environnement pour lequel le temps de compilation est souvent moins critique.

Mais il est indéniable que le temps d'analyse pourrait être réduit considérablement par la génération de code machine plutôt que de fermetures. L'étude de cas du chapitre 3 le démontre bien. La phase d'analyse locale pourrait produire un programme d'analyse déjà compilé vers du code natif. La phase d'édition de liens des programmes d'analyse ne comprendrait donc que le chargement de ces programmes ainsi que la relocalisation des variables importées et exportées.

Ensuite, il est évident que l'analyse de GOLD n'est pas suffisamment précise dans certains cas. La tour des nombres de Scheme rend beaucoup moins précise l'analyse de programmes essentiellement numériques. Il serait certainement possible de combiner l'analyse avec des options sur la ligne de commande permettant de restreindre cette tour à deux types de valeurs numériques, soit aux *flonums* et aux *fixnums* tout comme Bigloo.

L'existence de formes spéciales pour la définition de structures de données (comme `define-structure`) aiderait certainement l'analyse à obtenir des approximations plus précises. L'accès à un élément quelconque d'un vecteur retourne l'approximation de tous les éléments. Cette situation est particulièrement problématique dans le cas de vecteurs hétérogènes, i.e. comprenant des valeurs de types potentiellement différents. Les approximations ainsi calculés sont beaucoup trop grossières, rendant l'analyse quelque peu inutile et surtout trop coûteuse.

L'algorithme d'intégration intermodules pourrait être plus sophistiqué. Malgré tout, il permet dans certains cas d'obtenir d'intéressants gains de performance en limitant les appels intermodules.

Finalement, il eut été intéressant de comparer les temps des programmes compilés par GOLD avec ceux d'un système effectuant uniquement des optimisations locales basées sur une analyse de flot de contrôle. Or il se trouve que la sémantique très particulière de Scheme rende quasi impossible les optimisations payantes dans le contexte de programmes multimodules sans l'aide d'annotations spécifiques, d'un système de modules ou d'une forme d'analyse globale intermodule. À moins, évidemment, de faire des suppositions qui pourraient se révéler erronnées. Le compilateur ne peut, localement du moins, prouver qu'une primitive ou une procédure n'est pas redéfinie ailleurs dans un autre module.

Chapitre 7

CONCLUSION

J'ai montré par cette thèse qu'il est possible d'effectuer l'analyse globale de programmes multimodules de façon efficace grâce à une architecture basée sur la compilation abstraite. L'apport de cette thèse peut être résumé en trois points :

1. Dans un premier temps, j'ai montré que la compilation abstraite permet d'accélérer de manière significative la phase d'analyse statique. Grâce à une étude de cas détaillée, diverses stratégies de compilation abstraite ont été évaluées et comparées, montrant qu'il existe un large spectre de possibilités qui sont fonction du temps de génération du programme d'analyse et du temps d'exécution de ce dernier.

2. J'ai aussi développé une architecture de compilateur supportant l'analyse globale et les optimisations de haut niveau de programmes multimodules à l'édition des liens. Cette architecture permet, entre autres, de spécialiser des librairies fournies par des tiers sans nécessiter le recours au code source. De plus, le système opérant sur une représentation de haut niveau du code source, il est possible de développer des optimisations au niveau des constructions du langage source, ce qui est à peu près impossible pour les optimiseurs opérant uniquement à partir de code machine.

3. Enfin, un compilateur pour le langage Scheme réalisant partiellement l'architecture proposée a été développé et évalué. Il permet d'obtenir des programmes s'exécutant presqu'aussi rapidement qu'avec l'ajout de directives de compilation spécifiques au compilateur. Bien sûr, ce système possède ses faiblesses et

pourrait être amélioré considérablement. Malgré tout, les résultats présentés sont très encourageants et font présager un avenir prometteur à l'architecture proposée.

7.1 Travaux connexes

Un certain nombre de travaux s'apparentent de près ou de loin au présent ouvrage. Comme nous allons le constater, les buts poursuivis par ces systèmes sont suffisamment différents des miens pour rendre oiseuse toute comparaison directe.

7.1.1 Bigloo

Bigloo [47, 49] est un compilateur pour les langages Scheme et ML produisant du code C véloce, écrit par Manuel Serrano. À la différence de Gambit-C, Bigloo ne respecte pas le standard Scheme en tous points, sacrifiant au standard afin de permettre une génération de code plus efficace. Par exemple, chaque fonction Scheme est compilée en une fonction C équivalente. Comme je l'ai expliqué précédemment, ceci fait en sorte qu'il est impossible de garantir l'exécution correcte des appels terminaux, sauf certains cas bien précis.

Puisqu'un des buts de Bigloo est de supporter la compilation modulaire sans affecter la qualité du code généré, Bigloo possède un système de modules très étoffé, faisant donc dévier le langage source du standard Scheme. Les déclarations de modules de Bigloo permettent de spécifier la signature des variables exportées par un module ainsi que l'ensemble des modules qui sont importés. Ceci permet à l'analyse de flot de contrôle [24, 48] (qui est d'ailleurs assez semblable à celle décrite ici) de faire des suppositions qui ne seraient peut-être pas vraies autrement. Par exemple, si la fonction `car` n'est pas importée d'un autre module et qu'elle n'est pas redéfinie par le module courant, alors l'analyse peut supposer qu'il s'agit bien de la primitive du système.

Notons aussi que Bigloo ne supporte que deux types numériques, les *fixnums* et les *flonums*. Ceci permet d'obtenir des approximations plus précises sur les types des variables liées uniquement à des nombres lors de l'analyse statique.

Il est intéressant de constater que, même à l'aide d'un système de modules sophistiqué et une analyse statique à la fine pointe de la technologie, l'obtention de performances optimales nécessite quand même l'utilisation d'options de compilation supprimant l'émission de tests de type dynamiques. Par exemple, sur un Pentium 133MHz roulant Linux, le programme `simplex` roule 3.16 fois plus rapidement lorsque l'option `-unsafe` est spécifiée sur la ligne de commande, indiquant à Bigloo de supprimer tous les tests de type dynamiques restants après la phase d'optimisation. Pour `boyer`, la différence est de 31% seulement, alors que pour `mbrot` la différence est de 30% (ce qui est moins bon qu'avec GOLD, voir la table6.3).

Malgré tout, il demeure que Bigloo est un des compilateurs Scheme, produisant du C, parmi les plus efficaces. Son auteur a pu exhiber des programmes de taille réelle pour lesquels les performance obtenues avec Bigloo étaient égales ou supérieurs à des programmes équivalents écrits en C.

7.1.2 Stalin

Stalin [52] est un autre compilateur Scheme produisant du C. Il est développé par Jeffrey Mark Siskind. Tout comme GOLD, Stalin ne requiert aucun annotation de la part du programmeur afin de générer un code efficace. Ceci est rendu possible grâce à une analyse de flux de contrôle ensembliste (*set-based analysis*) très sophistiquée. À l'opposé de GOLD et Bigloo, Stalin ne supporte pas vraiment la compilation séparée : il faut lui fournir tout le code de l'application dans un seul fichier. Cette approche ne semble pas appropriée dans un contexte où certaines librairies ne sont pas disponibles sous forme de code source. Malheureusement, aucune littérature ne traite de cette analyse et de l'architecture du compilateur.

7.1.3 `mld`

`mld` [25] est un éditeur de liens optimisant pour Modula-3. Ce système a été décrit au chapitre 4. Tout comme l'architecture décrite ici, l'édition des liens ne se fait pas à partir de code objet mais sur un code intermédiaire adapté aux besoins des optimisations globales. Ceci semble valider en quelque sorte l'architecture que je propose.

La plus grande différence entre `mld` et mon architecture réside dans le pré-traitement du programme source, résultant en un programme d'analyse afin d'accélérer la phase d'analyse globale, ainsi que le concept d'édition de liens des programmes d'analyse. La vitesse des analyses de `mld` ne semble pas une préoccupation très profonde de Fernández, mais il faut dire que les analyses sont assez simples et pas aussi coûteuses en temps de calcul que celles de GOLD ou de `alto`. Il faut dire qu'elle a surtout mis l'emphase sur la portabilité du système ainsi que la nature des optimisations, qui sont surtout orientées vers les langages aux liaisons tardives (*late binding*). Dans ces langages, les types sont connus statiquement, mais la représentation exacte d'un objet n'est souvent connue qu'à l'exécution. C'est surtout le cas des langages orienté objet comme Modula-3 et C++.

7.1.4 Alto

`alto` [15, 16] est un éditeur de liens opérant des analyses statiques et des optimisations directement sur le code objet produit par la phase de compilation séparée, que j'ai décrit plus amplement au chapitre 4. Cette approche est intéressante puisqu'elle ne nécessite pas la présence du code source des modules. Par contre, elle possède une limite intrinsèque : l'absence d'information de haut niveau limite la classe d'optimisations applicables.

En particulier, les langages typés posent des problèmes qui leur sont propres. Les types sont souvent encodés à même les valeurs manipulées. Par exemple, dans Gambit-C, les deux bits les moins significatifs d'une valeur de type "fermeture" sont

mis à `01`, et le pointeur de code de la fermeture en question se trouve dans une structure à l'adresse obtenue en soustrayant 1 de la valeur en question. Dans le cas de programmes unimodules compilés avec Gambit-C, `alto` permet d'obtenir des exécutables en moyenne 10% plus rapide, ce qui ne me semble pas faramineux, compte tenu du niveau de sophistication des analyses et optimisations déployées par ce système. Lorsque ces mêmes programmes sont compilés avec Bigloo, des accélérations de 19% en moyenne peuvent être observées. Mais il faut dire que le mécanisme de trampoline utilisé par Gambit-C, combiné à l'arithmétique de pointeurs pour la représentation des types, complique considérablement la tâche de l'analyse statique d'`alto`.

Avec `alto`, l'édition des liens (analyse et optimisation) du compilateur `gcc`, qui représente environ 193000 lignes de code, prend 1083 secondes (soit plus de 18 minutes) sur un DEC Alpha 21164 de 300 MHz équipé de 512 Mo de mémoire vive. Le nombre d'instructions machine générées pour `gcc` est de 353000, comprenant les librairies, puisqu'`alto` optimise autant le programme source que les librairies que ce dernier utilise (pour les chiffres exacts, voir [16]). Le programme Scheme `earley`, lorsque compilé par **Gambit-C** et ensuite `gcc`, comprend 191000 instructions machine, librairies comprises. Or ce programme Scheme a moins de 650 lignes. En extrapolant, on peut imaginer que **Gambit-C** comprendrait certainement plus de 353000 instructions et que son optimisation par `alto` serait au moins aussi longue que pour `gcc`, montrant que le système GOLD est comparable à `alto` en terme de temps de calcul. Et ce même en utilisant des fermetures pour la représentation des programmes d'analyse.

7.1.5 CM

CM est un système de gestion de compilation développé par Matthias Blume pour le langage SML/NJ [6]. Ce système effectue automatiquement l'analyse des dépendances entre les modules. Ceux-ci peuvent être rassemblés pour former des *groupes*, qui peuvent eux-mêmes être regroupés pour former des super-groupes, etc.

De plus, CM peut faire de l'intégration de fonctions à travers la barrière des modules lors de la compilation (et non de l'édition des liens) en raison de l'absence de dépendances cycliques entre les modules. Ceci est réalisé à l'aide d'une technique appelée *λ-splitting*. L'idée de cette technique est de considérer chaque unité de compilation $U : I \rightarrow E$ comme une fonction des importations vers ses exportations. L'objectif de cette technique est de produire un triplet $(T, U_i : T \rightarrow E, U_e : I \rightarrow T)$ tel que $U_i \circ U_e$ donne U. La composante U_i comprend toutes les fonctions intégrables à travers la barrière des modules, tandis que U_e contient le reste du code compilé de U. Des heuristiques pour la détermination des fonctions intégrables y sont présentées.

Contrairement à GOLD, CM ne choisit pas les sites d'intégration de fonctions en se basant sur le résultat d'une analyse statique, mais uniquement sur des critères syntaxiques. Ceci limite certainement les possibilités d'intégration.

7.2 Travaux futurs

Les travaux qui viennent d'être présentés ouvrent la porte à un vaste éventail d'autres recherches. Je décris ici un certain nombre d'avenues de recherche qui seraient intéressantes à explorer, ainsi que des améliorations souhaitables au système GOLD.

7.2.1 Optimisations de l'analyse globale

La phase d'analyse globale pourrait être grandement accélérée par la génération de code machine plutôt que des fermetures et par l'écriture en C des routines de gestion des approximations plutôt qu'en Scheme. Todd Proebsting a développé un générateur de code à la volée dont les performances sont impressionnantes : il est question d'une émission de 3.6 Mo de code à la seconde sur un Pentium 266MHz! Aussi, une génération de code à la volée inspirée des travaux de Franz [30] pourrait s'avérer très performante. Il y est question d'une représentation intermédiaire compacte (les entrées de dictionnaires sémantiques ou SDE) développée en vue d'obtenir une génération de

code à la volée très rapide. En fait, cette représentation permet la génération de code presqu'aussi rapidement que le chargement dynamique de code objet. Ces techniques sont implantées dans un compilateur pour le langage Oberon [55], MacOberon. De nombreux autres travaux traitent de la génération de code machine à la volée [18, 37, 39, 40, 57].

La factorisation de l'analyse pourrait être améliorée. En effet, les programmes d'analyse qui sont conservés pourraient être optimisés. Il serait intéressant d'adapter les algorithmes de simplification de contraintes décrits dans [27] aux besoins de l'analyse de GOLD. Cet article présente une analyse de composantes à base d'ensembles (*componential set-based analysis*) qui permet l'analyse et la réanalyse de programmes Scheme multi-modules. Des idées similaires à celles qui sont présentées ici y sont décrites. Malheureusement, les auteurs ne présentent pas l'impact de cette analyse sur les optimisations qui pourraient être effectuées globalement.

Au delà de la simple implémentation de l'analyse globale, il serait intéressant de regrouper certains modules, dont les dépendances sont plus fortes. Ainsi, lorsqu'un module est modifié, il serait seulement nécessaire d'effectuer l'analyse globale du groupe de modules auquel il appartient. Ceci permettrait d'accélérer l'analyse globale mais aussi toute la phase d'édition des liens, puisque seule la recompilation et l'optimisation des modules impliqués serait requise.

7.2.2 Représentations intermédiaires

Bien que l'architecture de compilateur présentée au chapitre 4 fasse état d'une représentation intermédiaire sur laquelle les optimisations globales sont effectuées, GOLD ne matérialise pas cette idée. La raison principale est que l'élaboration d'une telle représentation intermédiaire aurait certainement requis autant de travail que la présente thèse.

L'impact d'une telle représentation aurait été certain. D'une part, la génération de code machine aurait certainement été plus rapide que dans le cas de GOLD, ainsi que

la phase d'optimisation. En effet, le coût de la génération de C et de la compilation de C serait supprimé. Dans [25], Fernández mentionne que l'éditeur de liens `mld` est toujours plus rapide lorsqu'il génère du code machine plutôt que de l'assembleur qui doit être envoyé à `as` par la suite (entre 1.3 et 1.5 fois plus rapide sur un processeur Intel 486, 1.7 à 2.1 fois plus rapide sur un SPARC et jusqu'à 3 fois plus rapide sur MIPS).

Mentionnons aussi qu'un compilateur générant un code intermédiaire est aussi plus portable, qu'il peut être utilisé comme compilateur croisé (*cross-compiler*), permettant la compilation sur une architecture différente de celle où roulera l'exécutable résultant. Seul l'éditeur doit posséder plusieurs parties arrière (*back-end*) afin de générer l'exécutable sur plusieurs architecture-cibles. De plus, seule la phase de l'éditeur de liens faisant la génération de code doit être récrit. Toute la phase d'analyse et d'optimisation peut être écrite de manière à être indépendante de l'architecture-cible.

7.2.3 D'autres analyses, d'autres langages

Il serait certainement intéressant d'appliquer le concept de compilation abstraite à d'autres analyses statiques et/ou à d'autres langages de programmation.

Pour les langages paresseux, l'analyse de nécessité serait une candidate intéressante, tout comme la 1CFA [51] (qui serait aussi intéressante à appliquer aux langages stricts comme Lisp/Scheme ou ML). Cette dernière est plus coûteuse (en temps et en espace) que la 0CFA, mais permet de meilleures optimisations puisqu'elle est polyvariante : l'analyse du code d'une procédure est fonction du site d'appel. Ceci contraste avec la 0CFA puisque, dans ce cadre, l'analyse d'une procédure est fonction de l'ensemble de ses sites d'appel. L'algorithme de compilation abstraite d'une analyse polyvariante serait évidemment plus complexe que celle présentée dans le présent traval et représenterait un défi intéressant.

Les langages comme ML et Haskell, bien que fortement typés, permettent l'écriture de fonctions polymorphiques. Ceci permet leur utilisation sur des types de données

différents. Un compilateur abstrait pour de tels langages pourrait générer des versions différentes du programme d'analyse d'une telle fonction, représentant une complexité équivalente à celle de la 1CFA.

Si on sort du cadre des langages fonctionnels, des langages comme Java ou Erlang [3] seraient aussi de bons candidats. Java [31], par exemple, est un langage orienté objet inspiré de C++ mais conçu principalement pour les applications distribuées et sécures. Afin de faciliter le transport d'une application Java d'une plateforme à une autre, les classes Java sont compilées en du code pour une machine virtuelle, la JVM. Pour obtenir des performances décentes, la plupart des environnements d'exécution Java chargent le code virtuel et le compile sur le champ ou bien compilent les méthodes qui sont appelées le plus souvent. On peut donc envisager d'appliquer des analyses statiques afin d'optimiser le code machine ainsi générer. La vitesse d'exécution de la phase d'analyse est donc cruciale dans un tel contexte.

7.2.4 Un système complet

Le système que j'ai présenté dans les derniers chapitres a été conçu à partir d'un système existant, Gambit-C. Bien que limitant la quantité de travail requis pour la réalisation d'un tel système, il n'en demeure pas moins que cela a contraint fortement l'architecture de GOLD. Par exemple, il a été impossible de réaliser complètement l'architecture proposée étant donné que GOLD devait interférer le moins possible avec le reste du compilateur. De plus, il se peut que certaines optimisations aient été possibles si GOLD avait été conçu à partir de zéro. L'utilisation de résultats de profilage pourrait par exemple guider la phase d'optimisation afin qu'elle se concentre sur les parties du code qui sont exécutées le plus souvent.

RÉFÉRENCES

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

[2] A. W. Appel. *Compiling with continuations.* Cambridge University Press, Cambridge, 1992.

[3] J. L. Armstrong, S. R. Virding, et C. Wikström. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[4] M. Ashley. A Practical and Flexible Flow Analysis for Higher-Order Languages. Dans *Proceedings of the 1996 ACM Conference on Principles of Programming Languages*, 1996.

[5] A. E. Ayers. *Abstract Analysis and Optimization of Scheme.* PhD thesis, Massachussets Institute of Technology, September 1993.

[6] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization.* PhD thesis, Princeton University, Novembre 1997.

[7] Anders Bondorf. Similix 5.0 manual. Rapport technique, DIKU, Département d'informatique, Université de Copenhagen, Mai 1993.

[8] D. Boucher et M. Feeley. Abstract compilation: a new implementation paradigm for static analysis. Dans *Proceedings of the 1996 International Conference on Compiler Construction*, 1996.

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

[10] F. C. Chow. Minimizing register usage penalty at procedure calls. Dans *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, 1988.

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

[12] C. Consel et O. Danvy. Tutorial Notes on Partial Evaluation. Dans *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[13] P. Cousot et R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximations of fixpoints. Dans *Fourth Symposium on Principles of Programmings Languages*, pages 238–252, 1977.

[14] R. Crelier. *Separate Compilation and Module Extension.* PhD thesis, Swiss Federal Institute of Technology, Zurich, 1994.

[15] S. Debray, R. Muth, et S. Watterson. Link-time Improvement of Scheme Programs. Dans *Proc. 8th International Conference on Compiler Construction (CC'99)*, pages 76–90, Mars 1999.

[16] S. Debray, R. Muth, S. Watterson, et K. De Bosschere. `alto`: A Link-Time Optimizer for the DEC Alpha. Rapport Technique 98-14, Université d'Arizona, Décembre 1998.

[17] S. K. Debray et D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3): 207–229, 1988.

[18] Dawson R. Engler et Todd A Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generator. Dans *Sixth International Conference on Architectural Support for Programming Languages a nd Operating Systems (ASPLOS-VI)*, pages 263–273, October 1994.

[19] M. Feeley. Gambit-C version 2.8. `http://www.iro.umontreal.ca/~gambit`.

[20] M. Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* Thèse de doctorat, Brandeis University, Avril 1993.

[21] M. Feeley et G. Lapalme. Using closures for code generation. *Computer Languages*, 12(1): 47–66, 1987.

[22] M. Feeley et M. Larose. Etos: an Erlang to Scheme Compiler. Rapport Technique 1079, Université de Montréal, Août 1997.

[23] M. Feeley et J. Miller. A parallel virtual machine for efficient Scheme compilation. Dans *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[24] M. Feeley et M. Serrano. Storage use analysis and its applications. Dans *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, 1996.

[25] M. F. Fernández. Simple and Effective Link-Time Optimization of Modula-3 Programs. Dans *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, Juin 1995.

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

[27] C. Flanagan et M. Felleisen. Componential Set-Based Analysis. *ACM Transactions on Programming Languages and Systems*, Février 1999.

[28] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, et M. Felleisen. Catching bugs in the web of program invariants. Dans *Proceedings of the 1996 Conference on Programming Languages Design and Implementation*, 1996.

[29] M. Flanagan et M. Felleisen. Set-based analysis for full scheme and its use in soft-typing. Rapport technique, Rice University, October 1995.

[30] M. S. Franz. *Code-Generation On-the-Fly: A Key to Portable Software.* PhD thesis, Swiss Federal Institute of Technology, Zurich, 1994.

[31] J. Gosling. Java intermediate bytecode. *ACM Workshop on Intermediate Representations, SIGPLAN Notices*, 30(3): 111–118, 1995.

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

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

[34] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). Dans *Conference of Lisp and Functional Programming*, pages 351–363, 1986.

[35] S. Jagannathan et A. Wright. Effective flow-analysis for avoiding runtime checks. Dans *Proceedings of the 1995 ACM Conference on Principles of Programming Languages*, Septembre 1995.

[36] T. Johnsson. Lambda-lifting: Transforming programs to recursive equations. Dans *Proc. Conference on Functional Programming and Computer Architecture.* Springer-Verlag, 1985.

[37] David Keppel, Susan J. Eggers, et Robert R. Henry. A case for runtime code generation. Rapport Technique 91-11-04, Department of Computer Science and Enginnering, University of Washington, November 1991.

[38] D. A. Kranz. ORBIT*: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, 1988.

[39] Mark Leone et Peter Lee. Deferred Compilation: The Automation of Run-Time Code Generation. Rapport Technique CMU-CS-93-225, Carnegie-Mellon, Department of Computer Science, Pittsburgh, PA 15212, December 1993.

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

[41] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, Univeristy of Edinburgh, 1981.

[42] D. Odnert et V. Santhanam. Register allocation across procedure and module boundaries. Dans *Proceedings of the 1990 ACM Conference on Programming Language Design and Implementation*, pages 28–39, June 1990.

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

[44] N. Ramsey. Relocating machine instructions by currying. Dans *ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, pages 226–236, Mai 1996.

[45] G. J. Rozas. Taming the Y operator. Dans *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 226–234, 1992.

[46] A. Sabry et M. Felleisen. Is Continuation-Passing Style Useful for Data Flow Analysis? Dans *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[47] M. Serrano. *Vers une compilation portable* et *performante des langages fonctionnels.* Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France, Decembre 1994.

[48] M. Serrano. Control flow analysis: a compilation paradigm for functional language. Dans *Proceedings of SAC 95*, 1995.

[49] M. Serrano. Bigloo User's Manual. Rapport technique, Inria, Rocquencourt, March 1994.

[50] O. Shivers. Control Flow Analysis in Scheme. Dans *SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[51] O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[52] J. M. Siskind. Stalin, an Optimizing Compiler for Scheme. `ftp://ftp.nj.nec.com/pub/qobi/stalin.tar.Z`.

[53] A. Srivastava et D. W. Wall. A practical system for intermodule code optimization at link-time. Rapport Technique 92/6, DEC Western Research Laboratory, Palo Alto, California, December 1992.

[54] D. W. Wall. Global register allocation at link time. Rapport Technique 86/3, DEC Western Research Laboratory, Palo Alto, California, October 1986.

[55] N. Wirth. The Programming Language Oberon. *Software – Practice and Experience*, 18(7): 671–690, 1988.

[56] Andrew K. Wright et Robert Cartwright. A practical soft type system for Scheme. Dans *Proc. ACM Conference on Lisp and Functional Programming*, pages 250–262, 1994.

[57] Curtis Yarvin et Adam Sah. Portable Runtime Code Generation in C. Rapport Technique UCB//CSD-94-792, University of California Berkeley, Department of Computer Science, 1994.

Annexe A

PROGRAMME D'ANALYSE DE FLUX DE DONNÉES

```
;; ------------------------------------------------------------ ;;
;; FICHIER : dfa.sim                                            ;;
;; ------------------------------------------------------------ ;;

(loadt "dfa.adt")

(define (dfa adj n gen kill sol)
  (let loop ((i 0))
    (if (= i n)
        sol
        (begin
          (my-vector-set! sol i (out-b adj gen kill i sol))
          (loop (+ i 1))))))


(define (in-b adj i sol)
  (let loop ((l    (pred adj i))
     (lub  '()))
    (if (null?  l)
        lub
        (let ((j (car l)))
          (loop (cdr l) (join (my-vector-ref sol j) lub))))))

(define (pred adj i)
  (vector-ref adj i))

(define (out-b adj gen kill i sol)
  (join (vector-ref gen i)
        (diff (in-b adj i sol)
            (vector-ref kill i))))
```

```
;; ------------------------------------------------------------------ ;;
;; FICHIER : dfa.adt                                                  ;;
;; ------------------------------------------------------------------ ;;

;; Union de deux ensembles
(defprim-dynamic (join l1 l2)
  (cond
   ((null?  l1) l2)
   ((null?  l2) l1)
   (else
    (let ((x (car l1))
          (y (car l2)))
      (cond
       ((= x y) (cons x (join (cdr l1) (cdr l2))))
       ((< x y) (cons x (join (cdr l1) l2)))
       ((> x y) (cons y (join l1 (cdr l2)))))))))

;; Difference de deux ensembles
(defprim-dynamic (diff l1 l2)
  (cond
   ((null?  l2) l1)
   ((null?  l1) '())
   (else
    (let ((x (car l1))
          (y (car l2)))
      (cond
       ((= x y) (diff (cdr l1) (cdr l2)))
       ((< x y) (cons x (diff (cdr l1) l2)))
       ((> x y) (diff l1 (cdr l2)))))))))


(defprim-opaque 2 my-vector-ref vector-ref)
(defprim-opaque 3 my-vector-set! vector-set!)
```

Annexe B

RÈGLES D'INTÉGRATION DES PRIMITIVES

Je décris ici les règles d'intégration de GOLD. Il s'agit essentiellement de transformations source-à-source guidées par le résultat de l'analyse globale. Cette transformation assume évidemment que GOLD a prouvé que les primitives ainsi intégrées ne sont pas redéfinies dans un des modules du programme.

B.1 Fonctions de génération de tests de type

Afin d'alléger l'écriture des règles d'intégration, plusieurs fonctions auxiliaires de générations de tests de type sont requises. Ces fonctions sont données ici. Elles prennent en argument une expression `e` et une variable temporaire `tmp`. Si l'approximation de `e` contient uniquement le type attendu, la fonction retourne la constante `#t`, sinon elle retourne un test de type `(type? tmp)`.

$\mathcal{T}_{\texttt{Char}}[\![\texttt{e}]\!][\![\texttt{tmp}]\!]$ =
    Si $\mathcal{A}(\texttt{e}) = \texttt{Char}$,
        `#t`
    sinon
        `(char? tmp)`

$\mathcal{T}_{\texttt{String}}[\![\texttt{e}]\!][\![\texttt{tmp}]\!]$ =
    Si $\mathcal{A}(\texttt{e}) = \texttt{String}$,
        `#t`
    sinon
        `(string? tmp)`

$\mathcal{T}_{\texttt{Symbol}}[\![\texttt{e}]\!][\![\texttt{tmp}]\!]$ =
    Si $\mathcal{A}(\texttt{e}) = \texttt{Symbol}$,
        `#t`
    sinon
        `(symbol? tmp)`

$\mathcal{T}_{\texttt{Fixnum}}[\![\texttt{e}]\!][\![\texttt{tmp}]\!]$ =
Si $\mathcal{A}(\texttt{e}) = \texttt{Integer}(x, y)$ et $\texttt{MINFIXNUM} \leq x \leq y \leq \texttt{MAXFIXNUM}$
```
#t
```
sinon
```
(let () (declare (extended-bindings))
  (##fixnum? tmp))
```

$\mathcal{T}_{\texttt{Flonum}}[\![\texttt{e}]\!][\![\texttt{tmp}]\!]$ =
Si $\mathcal{A}(\texttt{e}) = \texttt{Flonum}$
```
#t
```
sinon
```
(let () (declare (extended-bindings))
  (##flonum? tmp))
```

$\mathcal{T}_{\text{Vect}}[\![\texttt{e}]\!][\![\texttt{tmp}]\!]$ =
Si $\mathcal{A}(\texttt{e}) = \langle \emptyset, \{ \texttt{KVect}_{i_1}, \ldots, \texttt{KVect}_{i_n}, \texttt{UVect}_{j_1}, \ldots, \texttt{UVect}_{j_n} \}, \bot_n \rangle$,
```
#t
```
sinon
```
(vector? tmp)
```

B.2 Règles d'intégration

Les règles sont données aux pages suivantes et sont regroupées en fonction du type attendu de leurs arguments. Elles ont toutes le format suivant :

nom

$\mathcal{I}[\![\texttt{(nom x1 ... xn)}]\!]$
Si *condition*,
```
... code ...
```
sinon,
```
... code ...
```

Lorsque plusieurs primitives possèdent une expansion semblable, le nom de la primitive est remplacé par P, indiquant qu'il s'agit en fait d'une famille de règles ne différant que dans le nom de la primitive.

B.2.1 Opérations sur les paires

car, cdr

$\mathcal{I}[\![$`(P x)`$]\!] =$
si $\mathcal{A}$(`x`) $= \langle \emptyset, \{\texttt{Pair}_{i_1}, \ldots, \texttt{Pair}_{i_n}\}, \bot_n \rangle$,

```
(let ((tmp x))
  (declare (standard-bindings) (not safe))
  (P tmp))
```

sinon

```
(let ((tmp x))
  (declare (standard-bindings))
  (P tmp))
```

set-car! , set-cdr!

$\mathcal{I}[\![$`(P x1 x2)`$]\!] =$
si $\mathcal{A}$(`x1`) $= \langle \emptyset, \{\texttt{Pair}_{i_1}, \ldots, \texttt{Pair}_{i_n}\}, \bot_n \rangle$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings) (not safe))
  (P tmp1 tmp2))
```

sinon

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (P tmp1 tmp2))
```

B.2.2 Opération sur les symboles

`symbol->string`

$\mathcal{I}[\![$`(symbol->string x)`$]\!] =$
si $\mathcal{A}$(x) $= \langle \{$ Symbol $\}, \emptyset, \bot_n \rangle$,

```
(let ((tmp x))
  (declare (standard-bindings) (not safe))
  (symbol->string tmp))
```

sinon

```
(let ((tmp x))
  (declare (standard-bindings))
  (symbol->string tmp))
```

B.2.3 Opérations sur les caractères

`char=? , char<? , char<=? , char>? , char>=?`

`char-ci=? , char-ci<? , char-ci<=? , char-ci>? , char-ci>=?`

$\mathcal{I}[\![$`(P x1 x2)`$]\!] =$

```
(let ((tmp1 x1) (tmp2 x2))
  (if (let ()
        (declare (standard-bindings) (not safe))
        (and T_Char[[x1]][[tmp1]]
             T_Char[[x2]][[tmp2]] ))
      (let ()
        (declare (standard-bindings) (not safe))
        (P tmp1 tmp2))
      (let ()
        (declare (standard-bindings))
        (P tmp1 tmp2))))
```

B.2.4 Opérations sur les chaînes de caractères

string-length

$\mathcal{I}[\![$(string-length x)$]\!] =$
si $\mathcal{A}$(x) $= \langle \{$ String $\}, \emptyset, \bot_n \rangle$,

```
(let ((tmp x))
  (declare (standard-bindings) (not safe))
  (string-length tmp))
```

sinon

```
(let ((tmp x))
  (declare (standard-bindings))
  (string-length tmp))
```

string-ref

$\mathcal{I}[\![$(string-ref s i)$]\!] =$

```
(let ((tmp-s s) (tmp-i i))
  (if (let ()
        (declare (fixnum) (standard-bindings) (not safe))
        (and T_String[[s]][[tmp-s]]
             T_Fixnum[[i]][[tmp-i]]
             (>= tmp-i 0)
             (< tmp-i (string-length tmp-s))))
      (let ()
        (declare (standard-bindings) (not safe))
        (string-ref tmp-s tmp-i))
      (let ()
        (declare (standard-bindings))
        (string-ref tmp-s tmp-i))))
```

string-set!

$\mathcal{I}[\![$(string-set! s i c)$]\!] =$

```
(let ((tmp-s s) (tmp-i i) (tmp-c c))
  (if (let ()
        (declare (fixnum) (standard-bindings) (not safe))
        (and T_String[[s]][[tmp-s]]
             T_Fixnum[[i]][[tmp-i]]
             T_Char[[c]][[tmp-c]]
             (>= tmp-i 0)
             (< tmp-i (string-length tmp-s))))
      (let ()
        (declare (standard-bindings) (not safe))
        (string-set! tmp-s tmp-i tmp-c))
      (let ()
        (declare (standard-bindings))
        (string-set! tmp-s tmp-i tmp-c))))
```

B.2.5 Opérations sur les vecteurs

vector-length

$\mathcal{I}[\![$(vector-length x)$]\!] =$
si $\mathcal{A}$(x) $= \langle \emptyset, \{$ KVect$_{i_1}, \ldots,$KVect$_{i_n}$,UVect$_{j_1}, \ldots,$UVect$_{j_n}$ $\}, \bot_n \rangle$,

```
(let ((tmp x))
  (declare (standard-bindings) (not safe))
  (vector-length tmp))
```

sinon

```
(let ((tmp x))
  (declare (standard-bindings))
  (vector-length tmp))
```

vector-ref

$\mathcal{I}[\![$(vector-ref v i)$]\!] =$

```
(let ((tmp-v v) (tmp-i i))
  (if (let ()
        (declare (fixnum) (standard-bindings) (not safe))
        (and 𝒯Vect⟦v⟧⟦tmp-v⟧
             𝒯Fixnum⟦i⟧⟦tmp-i⟧
             (>= tmp-i 0)
             (< tmp-i (vector-length tmp-v))))
      (let ()
        (declare (standard-bindings) (not safe))
        (vector-ref tmp-v tmp-i))
      (let ()
        (declare (standard-bindings))
        (vector-ref tmp-v tmp-i))))
```

vector-set!

$\mathcal{I}[\![$(vector-set! v i e)$]\!] =$

```
(let ((tmp-v v) (tmp-i i) (tmp-e e))
  (if (let ()
        (declare (fixnum) (standard-bindings) (not safe))
        (and T_Vect[[v]][[tmp-v]]
             T_Fixnum[[i]][[tmp-i]]
             (>= tmp-i 0)
             (< tmp-i (vector-length tmp-v))))
      (let ()
        (declare (standard-bindings) (not safe))
        (vector-set! tmp-v tmp-i tmp-e))
      (let ()
        (declare (standard-bindings))
        (vector-set! tmp-v tmp-i tmp-e))))
```

B.2.6 Opérations sur les nombres

positive? , negative? , zero?

x $\mathcal{I}[\![$(*P* x)$]\!]$ =

si $\mathcal{A}$(x) = Integer(k, l) et MINFIXNUM $\leq k \leq l \leq$ MAXFIXNUM,

```
(let ((tmp x))
  (declare (standard-bindings) (fixnum) (not safe))
  (P tmp))
```

sinon, si Integer$(-\infty, +\infty) \in \mathcal{A}$(x),

```
(let ((tmp x))
  (declare (standard-bindings))
  (if (let () (declare (extended-bindings) (not safe))
        (##fixnum? tmp))
      (let () (declare (fixnum) (not safe))
        (P tmp))
      (P tmp)))
```

sinon, si $\mathcal{A}$(x) = Flonum,

```
(let ((tmp x))
  (declare (standard-bindings) (flonum) (not safe))
  (P tmp))
```

sinon

```
(let ((tmp x))
  (declare (standard-bindings))
  (if (let () (declare (extended-bindings) (not safe))
        (##fixnum? tmp))
      (let () (declare (fixnum) (not safe))
        (P tmp))
      (if (let () (declare (extended-bindings) (not safe))
            (##flonum? tmp))
          (let () (declare (flonum) (not safe))
            (P tmp))
          (P tmp))))
```

odd? , even?

$\mathcal{I}[\![$(*P* x)$]\!] =$

```
(let ((tmp x))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        T_Fixnum[[x]][[tmp]] )
      (let () (declare (not safe) (fixnum))
        (P tmp))
      (P tmp)))
```

abs, floor, ceiling, round, sin, cos, tan, asin, acos, atan

$\mathcal{I}[\![$(*P* x)$]\!] =$

```
(let ((tmp x))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        T_Flonum[[x]][[tmp]] )
      (let () (declare (not safe) (flonum))
        (P tmp))
      (P tmp)))
```

exact->inexact

$\mathcal{I}[\![$(exact->inexact x)$]\!] =$

```
(let ((tmp x))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        T_Fixnum[[x]][[tmp]] )
      (let () (declare (not safe) (fixnum))
        (exact->inexact tmp))
      (exact->inexact tmp)))
```

`=, <, <=, >, >=`

$\mathcal{I}[\![$`(P x1 x2)`$]\!] =$

Si `Integer`$(x, y) \in \mathcal{A}$(`x1`) et `Integer`$(x, y) \in \mathcal{A}$(`x2`),

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and T_Fixnum[[x1]][[tmp1]]
             T_Fixnum[[x2]][[tmp2]] ))
      (let () (declare (fixnum) (not safe))
        (P tmp1 tmp2))
      (P tmp1 tmp2)))
```

sinon, si `Flonum` $\in \mathcal{A}$(`x1`) et `Flonum` $\in \mathcal{A}$(`x2`),

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and T_Flonum[[x1]][[tmp1]]
             T_Flonum[[x2]][[tmp2]] ))
      (let () (declare (flonum) (not safe))
        (P tmp1 tmp2))
      (P tmp1 tmp2)))
```

sinon

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and T_Fixnum[[x1]][[tmp1]]
             T_Fixnum[[x2]][[tmp2]] ))
      (let () (declare (fixnum) (not safe))
        (P tmp1 tmp2))
      (if (let () (declare (not safe))
            (and T_Flonum[[x1]][[tmp1]]
                 T_Flonum[[x2]][[tmp2]] ))
          (let () (declare (flonum) (not safe))
            (P tmp1 tmp2))
          (P tmp1 tmp2))))
```

quotient, modulo, remainder

$\mathcal{I}[\![$(P x1 x2)$]\!] =$

Si Integer$(x, y) \in \mathcal{A}$(x1) et Integer$(x, y) \in \mathcal{A}$(x2),

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe)) (fixnum)
        (and T_Fixnum[[x1]][[tmp1]]
             T_Fixnum[[x2]][[tmp2]]
             (not (= tmp2 0))))
      (let () (declare (not safe) (fixnum))
        (P tmp1 tmp2))
      (P tmp1 tmp2)))
```

sinon

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (P tmp1 tmp2))
```

sqrt

$\mathcal{I}[\![$(sqrt x)$]\!] =$

```
(let ((tmp x))
  (declare (standard-bindings))
  (if (let () (declare (not safe)) (flonum)
        (and T_Flonum[[x]][[tmp]]
             (>= tmp 0.0)))
      (let () (declare (not safe) (flonum))
        (sqrt tmp))
      (sqrt tmp)))
```

/

$\mathcal{I}[\![$(/ x1 x2)$]\!] =$

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe)) (flonum)
        (and T_Flonum[[x1]][[tmp1]]
             T_Flonum[[x2]][[tmp2]]
             (not (= tmp2 0.0))))
      (let () (declare (not safe) (flonum))
        (/ tmp1 tmp2))
      (/ tmp1 tmp2)))
```

+

$\mathcal{I}[\![$(+ x1 x2)$_l]\!]$

Si $\mathcal{A}$(x1) = Flonum et $\mathcal{A}$(x1) = Flonum,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings) (flonum) (not safe))
  (+ tmp1 tmp2))
```

sinon, si $\mathcal{A}$(x1) = Flonum et $\mathcal{A}$(x2) = Integer(x, y), $x \neq -\infty$, $y \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (extended-bindings)
           (standard-bindings) (flonum) (not safe))
  (+ tmp1 (##flonum.<-fixnum tmp2)))
```

sinon, si $\mathcal{A}$(x2) = Flonum et $\mathcal{A}$(x1) = Integer(x, y), $x \neq -\infty$, $y \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (extended-bindings)
           (standard-bindings) (flonum) (not safe))
  (+ (##flonum.<-fixnum tmp1) tmp2))
```

sinon, si Flonum $\in \mathcal{A}$(x1) ou Flonum $\in \mathcal{A}$(x2),

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and T_Flonum[[x1]][[tmp1]]
             T_Flonum[[x2]][[tmp2]] ))
      (let () (declare (flonum) (not safe))
        (+ tmp1 tmp2))
      (+ tmp1 tmp2)))
```

sinon, si Integer$(x, y) = \mathcal{A}$(x1), Integer$(x', y') = \mathcal{A}$(x2) et Integer$(x'', y'') = \mathcal{A}(l)$, où $x, x', x'' \neq -\infty$ et $y, y', y'' \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings) (fixnum) (not safe))
  (+ tmp1 tmp2))
```

sinon, si $\texttt{Integer}(x, y) = \mathcal{A}(\texttt{x1})$ et $\texttt{Integer}(x', y') = \mathcal{A}(\texttt{x2})$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and T_Fixnum[[x1]][[tmp1]]
             T_Fixnum[[x2]][[tmp2]]
             (let () (declare (fixnum))
               (let ((tmp-r (+ tmp1 tmp2)))
                 (if (negative?  tmp2)
                     (< tmp-r tmp1)
                     (>= tmp-r tmp1))))))
      (let () (declare (fixnum) (not safe))
        (+ tmp1 tmp2))
      (+ tmp1 tmp2)))
```

sinon

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (+ tmp1 tmp2))
```

-

$\mathcal{I}[\![$(- x1 x2)$_l]\!]$

Si $\mathcal{A}$(x1) = Flonum et $\mathcal{A}$(x1) = Flonum,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings) (flonum) (not safe))
  (- tmp1 tmp2))
```

sinon, si $\mathcal{A}$(x1) = Flonum et $\mathcal{A}$(x2) = Integer(x, y), $x \neq -\infty$, $y \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (extended-bindings)
           (standard-bindings) (flonum) (not safe))
  (- tmp1 (##flonum.<-fixnum tmp2)))
```

sinon, si $\mathcal{A}$(x2) = Flonum et $\mathcal{A}$(x1) = Integer(x, y), $x \neq -\infty$, $y \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (extended-bindings)
           (standard-bindings) (flonum) (not safe))
  (- (##flonum.<-fixnum tmp1) tmp2))
```

sinon, si Flonum $\in \mathcal{A}$(x1) ou Flonum $\in \mathcal{A}$(x2),

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and T_Flonum[[x1]][[tmp1]]
             T_Flonum[[x2]][[tmp2]] ))
      (let () (declare (flonum) (not safe))
        (- tmp1 tmp2))
      (- tmp1 tmp2)))
```

sinon, si Integer$(x, y) = \mathcal{A}$(x1), Integer$(x', y') = \mathcal{A}$(x2) et Integer$(x'', y'') = \mathcal{A}(l)$, où $x, x', x'' \neq -\infty$ et $y, y', y'' \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings) (fixnum) (not safe))
  (- tmp1 tmp2))
```

sinon, si $\texttt{Integer}(x, y) = \mathcal{A}(\texttt{x1})$ et $\texttt{Integer}(x', y') = \mathcal{A}(\texttt{x2})$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and T_Fixnum[[x1]][[tmp1]]
             T_Fixnum[[x2]][[tmp2]]
             (let () (declare (fixnum))
               (let ((tmp-r (- tmp1 tmp2)))
                 (if (negative?  tmp2)
                     (> tmp-r tmp1)
                     (<= tmp-r tmp1))))))
      (let () (declare (fixnum) (not safe))
        (- tmp1 tmp2))
      (- tmp1 tmp2)))
```

sinon

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (- tmp1 tmp2))
```

*****

$\mathcal{I}[\![$`(* x1 x2)`$_l$$]\!]$

Si $\mathcal{A}$(`x1`) = `Flonum` et $\mathcal{A}$(`x1`) = `Flonum`,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings) (flonum) (not safe))
  (* tmp1 tmp2))
```

sinon, si $\mathcal{A}$(`x1`) = `Flonum` et $\mathcal{A}$(`x2`) = `Integer`(x, y), $x \neq -\infty$, $y \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (extended-bindings)
           (standard-bindings) (flonum) (not safe))
  (* tmp1 (##flonum.<-fixnum tmp2)))
```

sinon, si $\mathcal{A}$(`x2`) = `Flonum` et $\mathcal{A}$(`x1`) = `Integer`(x, y), $x \neq -\infty$, $y \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (extended-bindings)
           (standard-bindings) (flonum) (not safe))
  (* (##flonum.<-fixnum tmp1) tmp2))
```

sinon, si `Flonum` $\in \mathcal{A}$(`x1`) ou `Flonum` $\in \mathcal{A}$(`x2`),

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe))
        (and 𝒯Flonum⟦x1⟧⟦tmp1⟧
             𝒯Flonum⟦x2⟧⟦tmp2⟧ ))
      (let () (declare (flonum) (not safe))
        (* tmp1 tmp2))
      (* tmp1 tmp2)))
```

sinon, si `Integer`(x, y) = $\mathcal{A}$(`x1`), `Integer`(x', y') = $\mathcal{A}$(`x2`) et `Integer`(x'', y'') = $\mathcal{A}(l)$, où $x, x', x'' \neq -\infty$ et $y, y', y'' \neq +\infty$,

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings) (fixnum) (not safe))
  (* tmp1 tmp2))
```

sinon, si `Integer`$(x, y) = \mathcal{A}$(`x1`) et `Integer`$(x', y') = \mathcal{A}$(`x2`),

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (if (let () (declare (not safe) (fixnum))
        (and T_Fixnum[[x1]][[tmp1]]
             T_Fixnum[[x2]][[tmp2]]
             (not (< tmp1 -32768))  ; Code tiré de ##*
             (not (< 32767 tmp1))   ; ...
             (< -16384 tmp2)        ; ...
             (not (< 16384 tmp2)))) ; ...
      (let () (declare (fixnum) (not safe))
        (* tmp1 tmp2))
      (* tmp1 tmp2)))
```

sinon

```
(let ((tmp1 x1) (tmp2 x2))
  (declare (standard-bindings))
  (* tmp1 tmp2))
```

Annexe C

CODE SOURCE ANNOTÉ

Je donne ici le code source annoté de la plupart des programmes du banc d'essai de programmes unimodules. Les sites d'appel des primitives Scheme sont annotées de la manière suivante. Si l'appel est complètement optimisé, c'est-à-dire qu'il ne comporte plus aucun test de type dynamique, la primitive apparaît dans une boîte blanche. Voici un exemple, tiré de `fibfp.scm` :

```
6.      (+ (fibfp (- n 1.))
7.         (fibfp (- n 2.)))))
```

Dans ce cas-ci, les optimisations produisent un code équivalent à :

```
6.      (##flonum.+ (fibfp (##flonum.- n 1.))
7.                  (fibfp (##flonum.- n 2.)))))
```

Dans certains cas, des constantes comme `#f` sont entourées d'une telle boîte, ainsi que les appels à `set!`. À cause de la conversion des affectations, certaines références ou initialisations de variables locales sont transformées en des appels à la primitive `##cell-ref`, tandis que les formes spéciales `set!` sont remplacées par des appels à `##cell-set!`. Or ces appels sont traités par GOLD au même titre que les autres appels à des primitives.

GOLD entoure d'une boîte foncée deux types d'appels : les appels pour lesquels il ne peut ni faire l'intégration de la primitive ni ajouter la déclaration `(not safe)`, et les appels aux primitives `+`, `-` et `*` nécessitant des tests de bornes sur le résultat. Ces appels ont l'aspect suivant (code tiré de `fib.scm`) :

```
6.      (+ (fib (- n 1))
7.         (fib (- n 2)))))
```

Dans les autres cas, c'est-à-dire pour les appels qui ne peuvent être complètement optimisés et qui nécessitent au plus un test de type pour chaque argument, la primitive est entourée d'une boîte pâle. Voici un exemple, tiré aussi de `fib.scm` :

```
4.   (if (< n 2)
```

Dans ce cas-ci, le code généré par GOLD ressemble à ceci :

```
4.   (if (let ((t1 n) (t2 2))
5.         (declare (extended-bindings) (not safe))
6.         (if (and (##fixnum? t1) #t)
7.             (##fixnum.< t1 t2)
8.             (##< t1 t2)))
9.       ...
```

C.1 boyer.scm

```
1. ; Generated by CF Analysis Development Version, June 1, 1995
2.
3. (define (pair-expected) (error "pair expected!"))
4.
5. ;(let ()
6. (define (cadr@1 x) (car (cdr x)))
7. (define (caddr@1 x) (car (cdr (cdr x))))
8. (define (cadddr@1 x ) (car (cdr (cdr (cdr x)))))
9.
10. (define member@1
11.   (lambda (x a)
12.     (if (pair? a)
13.         (if (equal? x (car a)) a (member@1 x (cdr a)))
14.         #f)))
15.
16. (define assq@1
17.   (lambda (x a)
18.     (if (pair? a)
19.         (let ((y (car a)))
20.           (if (pair? y)
21.               (if (eq? x (car y)) y (assq@1 x (cdr a)))
22.               (error)))
23.         #f)))
24.
25. (define void (lambda () 'void))
26.
27. (define *namelist* '())
28. (define *lastlook* '(xxx ()))
29. (define nameprop
30.   (lambda (name)
31.     (if (pair? *lastlook*)
32.         (if (eq? name (car *lastlook*))
33.             *lastlook*
34.             (let ([pair (assq@1 name *namelist*)])
35.               (begin (if pair (set! *lastlook* pair) (void)) pair)))
36.         (error))))
37. (define get-null
38.   (lambda (name prop)
39.     (or (let ([r (nameprop name)])
40.           (if (pair? r)
41.               (let ([s (assq@1 prop (cdr r))]) (if (pair? s) (cdr s) #f))
42.               #f))
43.         '())))
44. (define unify-subst 0)
45. (define temp-temp 0)
46. (define add-lemma-lst
47.   (lambda (lst)
48.     (if (pair? lst)
49.         (begin (let ([term (car lst)])
50.                  (if (and (pair? term)
51.                           (eq? (car term) 'equal)
52.                           (pair? (cadr@1 term)))
53.                      (let ([name (car (cadr@1 term))]
54.                            [valu
55.                             (cons term
56.                                   (get-null (car (cadr@1 term)) 'lemmas))])
57.                        (begin (let ([r (nameprop name)])
58.                                 (if (pair? r)
59.                                     (let ([s (assq@1 'lemmas (cdr r))])
60.                                       (if (pair? s)
61.                                           (set-cdr! s valu)
62.                                           (set-cdr!
63.                                            r
64.                                            (cons (cons 'lemmas valu)
65.                                                  (cdr r)))))
66.                                     (set! *namelist*
67.                                       (cons (cons name
68.                                                   (cons (cons 'lemmas valu)
69.                                                         '()))
70.                                             *namelist*))))
71.                               valu))
72.                      (error 'add-lemma
```

```
                        "ADD-LEMMA did not like term:  "
                        term)))
                (add-lemma-lst (cdr lst)))
            #t)))
(define apply-subst
  (lambda (alist term)
    (if (pair? term)
        (cons (car term) (apply-subst-lst alist (cdr term)))
        (if (begin (set! temp-temp (assq@1 term alist)) temp-temp)
            (cdr temp-temp)
            term))))
(define apply-subst-lst
  (lambda (alist lst)
    (if (pair? lst)
        (cons (apply-subst alist (car lst))
              (apply-subst-lst alist (cdr lst)))
        '())))
(define falsep
  (lambda (x lst) (or (equal? x '(f)) (member@1 x lst))))
(define one-way-unify1
  (lambda (term1 term2)
    (if (pair? term2)
        (if (pair? term1)
            (if (eq? (car term1) (car term2))
                (one-way-unify1-lst (cdr term1) (cdr term2))
                #f)
            #f)
        (if (begin (set! temp-temp (assq@1 term2 unify-subst))
                   temp-temp)
            (equal? term1 (cdr temp-temp))
            (begin (set! unify-subst
                     (cons (cons term2 term1) unify-subst))
                   #t)))))
(define one-way-unify1-lst
  (lambda (lst1 lst2)
    (if (pair? lst1)
        (if (one-way-unify1 (car lst1) (car lst2))
            (one-way-unify1-lst (cdr lst1) (cdr lst2))
            #f)
        #t)))
(define rewrite
  (lambda (term)
    (if (pair? term)
        (rewrite-with-lemmas
          (cons (car term) (rewrite-args (cdr term)))
          (get-null (car term) 'lemmas))
        term)))
(define rewrite-args
  (lambda (lst)
    (if (pair? lst)
        (cons (rewrite (car lst)) (rewrite-args (cdr lst)))
        '())))
(define rewrite-with-lemmas
  (lambda (term lst)
    (if (pair? lst)
        (if (let ([term2 (cadr@1 (car lst))])
              (begin (set! unify-subst '()) (one-way-unify1 term term2)))
            (rewrite (apply-subst unify-subst (caddr@1 (car lst))))
            (rewrite-with-lemmas term (cdr lst)))
        term)))
(define tautologyp
  (lambda (x true-lst false-lst)
    (if (truep x true-lst)
        #t
        (if (falsep x false-lst)
            #f
            (if (pair? x)
                (if (eq? (car x) 'if)
                    (if (truep (cadr@1 x) true-lst)
                        (tautologyp (caddr@1 x) true-lst false-lst)
                        (if (falsep (cadr@1 x) false-lst)
                            (tautologyp (cadddr@1 x) true-lst false-lst)
                            (and (tautologyp
                                   (caddr@1 x)
```

```
                                    (cons (cadr@1 x) true-lst)
                                    false-lst)
                                  (tautologyp
                                    (cadddr@1 x)
                                    true-lst
                                    (cons (cadr@1 x) false-lst)))))
                      #f)
                  #f)))))
(define truep
  (lambda (x lst) (or (equal? x '(t)) (member@1 x lst))))

(define run
  (lambda ()
    (set! *namelist* '())
    (set! *lastlook* '(xxx ()))

    (let ((lemmas
           '((equal
              (compile form)
              (reverse
               (codegen (optimize form) (nil))))
             (equal (eqp x y) (equal (fix x) (fix y)))
             (equal (greaterp x y) (lessp y x))
             (equal (lesseqp x y) (not (lessp y x)))
             (equal
              (greatereqp x y)
              (not (lessp x y)))
             (equal
              (boolean x)
              (or (equal x (t)) (equal x (f))))
             (equal
              (iff x y)
              (and (implies x y) (implies y x)))
             (equal
              (even1 x)
              (if (zerop x) (t) (odd (1- x))))
             (equal
              (countps- l pred)
              (countps-loop l pred (zero)))
             (equal (fact- i) (fact-loop i 1))
             (equal
              (reverse- x)
              (reverse-loop x (nil)))
             (equal
              (divides x y)
              (zerop (remainder y x)))
             (equal
              (assume-true var alist)
              (cons (cons var (t)) alist))
             (equal
              (assume-false var alist)
              (cons (cons var (f)) alist))
             (equal
              (tautology-checker x)
              (tautologyp (normalize x) (nil)))
             (equal
              (falsify x)
              (falsify1 (normalize x) (nil)))
             (equal
              (prime x)
              (and (not (zerop x))
                   (not (equal x (add1 (zero))))
                   (prime1 x (1- x))))
             (equal
              (and p q)
              (if p (if q (t) (f)) (f)))
             (equal
              (or p q)
              (if p (t) (if q (t) (f)) (f)))
             (equal (not p) (if p (f) (t)))
             (equal
              (implies p q)
              (if p (if q (t) (f)) (t)))
             (equal (fix x) (if (numberp x) x (zero)))
```

```
(equal
 (if (if a b c) d e)
 (if a (if b d e) (if c d e)))
(equal
 (zerop x)
 (or (equal x (zero))
     (not (numberp x))))
(equal
 (plus (plus x y) z)
 (plus x (plus y z)))
(equal
 (equal (plus a b) (zero))
 (and (zerop a) (zerop b)))
(equal (difference x x) (zero))
(equal
 (equal (plus a b) (plus a c))
 (equal (fix b) (fix c)))
(equal
 (equal (zero) (difference x y))
 (not (lessp y x)))
(equal
 (equal x (difference x y))
 (and (numberp x)
      (or (equal x (zero)) (zerop y))))
(equal
 (meaning (plus-tree (append x y)) a)
 (plus (meaning (plus-tree x) a)
       (meaning (plus-tree y) a)))
(equal
 (meaning (plus-tree (plus-fringe x)) a)
 (fix (meaning x a)))
(equal
 (append (append x y) z)
 (append x (append y z)))
(equal
 (reverse (append a b))
 (append (reverse b) (reverse a)))
(equal
 (times x (plus y z))
 (plus (times x y) (times x z)))
(equal
 (times (times x y) z)
 (times x (times y z)))
(equal
 (equal (times x y) (zero))
 (or (zerop x) (zerop y)))
(equal
 (exec (append x y) pds envrn)
 (exec y (exec x pds envrn) envrn))
(equal
 (mc-flatten x y)
 (append (flatten x) y))
(equal
 (member x (append a b))
 (or (member x a) (member x b)))
(equal
 (member x (reverse y))
 (member x y))
(equal (length (reverse x)) (length x))
(equal
 (member a (intersect b c))
 (and (member a b) (member a c)))
(equal (nth (zero) i) (zero))
(equal
 (exp i (plus j k))
 (times (exp i j) (exp i k)))
(equal
 (exp i (times j k))
 (exp (exp i j) k))
(equal
 (reverse-loop x y)
 (append (reverse x) y))
(equal
 (reverse-loop x (nil))
```

```
295.              (reverse x))
296.             (equal
297.              (count-list z (sort-lp x y))
298.              (plus (count-list z x)
299.                    (count-list z y)))
300.             (equal
301.              (equal (append a b) (append a c))
302.              (equal b c))
303.             (equal
304.              (plus (remainder x y)
305.                    (times y (quotient x y)))
306.              (fix x))
307.             (equal
308.              (power-eval (big-plus1 l i base) base)
309.              (plus (power-eval l base) i))
310.             (equal
311.              (power-eval (big-plus x y i base) base)
312.              (plus i
313.                    (plus (power-eval x base)
314.                          (power-eval y base))))
315.             (equal (remainder y 1) (zero))
316.             (equal
317.              (lessp (remainder x y) y)
318.              (not (zerop y)))
319.             (equal (remainder x x) (zero))
320.             (equal
321.              (lessp (quotient i j) i)
322.              (and (not (zerop i))
323.                   (or (zerop j) (not (equal j 1)))))
324.             (equal
325.              (lessp (remainder x y) x)
326.              (and (not (zerop y))
327.                   (not (zerop x))
328.                   (not (lessp x y))))
329.             (equal
330.              (power-eval (power-rep i base) base)
331.              (fix i))
332.             (equal
333.              (power-eval
334.               (big-plus
335.                (power-rep i base)
336.                (power-rep j base)
337.                (zero)
338.                base)
339.               base)
340.              (plus i j))
341.             (equal (gcd x y) (gcd y x))
342.             (equal
343.              (nth (append a b) i)
344.              (append
345.               (nth a i)
346.               (nth b (difference i (length a)))))
347.             (equal (difference (plus x y) x) (fix y))
348.             (equal (difference (plus y x) x) (fix y))
349.             (equal
350.              (difference (plus x y) (plus x z))
351.              (difference y z))
352.             (equal
353.              (times x (difference c w))
354.              (difference (times c x) (times w x)))
355.             (equal (remainder (times x z) z) (zero))
356.             (equal
357.              (difference (plus b (plus a c)) a)
358.              (plus b c))
359.             (equal
360.              (difference (add1 (plus y z)) z)
361.              (add1 y))
362.             (equal
363.              (lessp (plus x y) (plus x z))
364.              (lessp y z))
365.             (equal
366.              (lessp (times x z) (times y z))
367.              (and (not (zerop z)) (lessp x y)))
368.             (equal
```

```
369.              (lessp y (plus x y))
370.              (not (zerop x)))
371.             (equal
372.              (gcd (times x z) (times y z))
373.              (times z (gcd x y)))
374.             (equal
375.              (value (normalize x) a)
376.              (value x a))
377.             (equal
378.              (equal (flatten x) (cons y (nil)))
379.              (and (nlistp x) (equal x y)))
380.             (equal (listp (gopher x)) (listp x))
381.             (equal
382.              (samefringe x y)
383.              (equal (flatten x) (flatten y)))
384.             (equal
385.              (equal (greatest-factor x y) (zero))
386.              (and (or (zerop y) (equal y 1))
387.                   (equal x (zero))))
388.             (equal
389.              (equal (greatest-factor x y) 1)
390.              (equal x 1))
391.             (equal
392.              (numberp (greatest-factor x y))
393.              (not (and (or (zerop y) (equal y 1))
394.                        (not (numberp x)))))
395.             (equal
396.              (times-list (append x y))
397.              (times (times-list x) (times-list y)))
398.             (equal
399.              (prime-list (append x y))
400.              (and (prime-list x) (prime-list y)))
401.             (equal
402.              (equal z (times w z))
403.              (and (numberp z)
404.                   (or (equal z (zero))
405.                       (equal w 1))))
406.             (equal
407.              (greatereqpr x y)
408.              (not (lessp x y)))
409.             (equal
410.              (equal x (times x y))
411.              (or (equal x (zero))
412.                  (and (numberp x) (equal y 1))))
413.             (equal (remainder (times y x) y) (zero))
414.             (equal
415.              (equal (times a b) 1)
416.              (and (not (equal a (zero)))
417.                   (not (equal b (zero)))
418.                   (numberp a)
419.                   (numberp b)
420.                   (equal (1- a) (zero))
421.                   (equal (1- b) (zero))))
422.             (equal
423.              (lessp
424.               (length (delete x l))
425.               (length l))
426.              (member x l))
427.             (equal
428.              (sort2 (delete x l))
429.              (delete x (sort2 l)))
430.             (equal (dsort x) (sort2 x))
431.             (equal
432.              (length
433.               (cons x1
434.                     (cons x2
435.                           (cons x3
436.                                 (cons x4
437.                                       (cons x5
438.                                             (cons x6
439.                                                   x7)))))))
440.              (plus 6 (length x7)))
441.             (equal
442.              (difference (add1 (add1 x)) 2)
```

```
443.              (fix x))
444.             (equal
445.              (quotient (plus x (plus x y)) 2)
446.              (plus x (quotient y 2)))
447.             (equal
448.              (sigma (zero) i)
449.              (quotient (times i (add1 i)) 2))
450.             (equal
451.              (plus x (add1 y))
452.              (if (numberp y)
453.                  (add1 (plus x y))
454.                  (add1 x)))
455.             (equal
456.              (equal
457.               (difference x y)
458.               (difference z y))
459.              (if (lessp x y)
460.                  (not (lessp y z))
461.                  (if (lessp z y)
462.                      (not (lessp y x))
463.                      (equal (fix x) (fix z)))))
464.             (equal
465.              (meaning (plus-tree (delete x y)) a)
466.              (if (member x y)
467.                  (difference
468.                   (meaning (plus-tree y) a)
469.                   (meaning x a))
470.                  (meaning (plus-tree y) a)))
471.             (equal
472.              (times x (add1 y))
473.              (if (numberp y)
474.                  (plus x (times x y))
475.                  (fix x)))
476.             (equal
477.              (nth (nil) i)
478.              (if (zerop i) (nil) (zero)))
479.             (equal
480.              (last (append a b))
481.              (if (listp b)
482.                  (last b)
483.                  (if (listp a)
484.                      (cons (car (last a)) b)
485.                      b)))
486.             (equal
487.              (equal (lessp x y) z)
488.              (if (lessp x y)
489.                  (equal t z)
490.                  (equal f z)))
491.             (equal
492.              (assignment x (append a b))
493.              (if (assignedp x a)
494.                  (assignment x a)
495.                  (assignment x b)))
496.             (equal
497.              (car (gopher x))
498.              (if (listp x)
499.                  (car (flatten x))
500.                  (zero)))
501.             (equal
502.              (flatten (cdr (gopher x)))
503.              (if (listp x)
504.                  (cdr (flatten x))
505.                  (cons (zero) (nil))))
506.             (equal
507.              (quotient (times y x) y)
508.              (if (zerop y) (zero) (fix x)))
509.             (equal
510.              (get j (set i val mem))
511.              (if (eqp j i) val (get j mem))))))
512.
513.       (add-lemma-lst lemmas))
514.
515.
516.      (let ([ans #f] [term #f])
```

```
517.        (begin (set! term
518.                 (apply-subst
519.                   '((x f
520.                       (plus (plus a b)
521.                             (plus c (zero))))
522.                     (y f
523.                        (times
524.                          (times a b)
525.                          (plus c d)))
526.                     (z f
527.                        (reverse
528.                          (append
529.                            (append a b)
530.                            (nil))))
531.                     (u equal
532.                        (plus a b)
533.                        (difference x y))
534.                     (w lessp
535.                        (remainder a b)
536.                        (member a (length b))))
537.                   '(implies
538.                      (and (implies x y)
539.                           (and (implies y z)
540.                                (and (implies z u)
541.                                     (implies
542.                                       u
543.                                       w))))
544.                           (implies x w))))
545.               (car term)
546.               (set! ans
547.                 (tautologyp (rewrite term) '() '()))
548.               ans))))
549. ;)
550.
551. (run-benchmark 10 (run))
```

C.2 browse.scm

```
;;; BROWSE -- Benchmark to create and browse through
;;; an AI-like data base of units.

(define (lookup key table)
  (let loop ((x table))
    (if (null? x)
      #f
      (let ((pair (car x)))
        (if (eq? (car pair) key)
          pair
          (loop (cdr x)))))))

(define properties '())

(define (get key1 key2)
  (let ((x (lookup key1 properties)))
    (if x
      (let ((y (lookup key2 (cdr x))))
        (if y
          (cdr y)
          #f))
      #f)))

(define (put key1 key2 val)
  (let ((x (lookup key1 properties)))
    (if x
      (let ((y (lookup key2 (cdr x))))
        (if y
          (set-cdr! y val)
          (set-cdr! x (cons (cons key2 val) (cdr x)))))
      (set! properties
        (cons (list key1 (cons key2 val)) properties)))))

(define *current-gensym* 0)

(define (generate-symbol)
  (set! *current-gensym* (+ *current-gensym* 1))
  (string->symbol (number->string *current-gensym*)))

(define (append-to-tail! x y)
  (if (null? x)
      y
      (do ((a x b)
           (b (cdr x) (cdr b)))
          ((null? b)
           (set-cdr! a y)
           x))))

(define (tree-copy x)
  (if (not (pair? x))
      x
      (cons (tree-copy (car x))
            (tree-copy (cdr x)))))

;;; n is # of symbols
;;; m is maximum amount of stuff on the plist
;;; npats is the number of basic patterns on the unit
;;; ipats is the instantiated copies of the patterns

(define *rand* 21)

(define (init n m npats ipats)
  (let ((ipats (tree-copy ipats)))
    (do ((p ipats (cdr p)))
        ((null? (cdr p)) (set-cdr! p ipats)))
    (do ((n n (- n 1)))
        (i m (cond ((zero? i) m)
                   (else (- i 1))))
        (name (generate-symbol) (generate-symbol))
        (a '()))
        ((= n 0) a)
        (set! a (cons name a))
```

```
        (do ((i i (- i 1)))
            ((zero? i))
            (put name (generate-symbol) #f))
        (put name
             'pattern
             (do ((i npats (- i 1))
                  (ipats ipats (cdr ipats))
                  (a '()))
                 ((zero? i) a)
                 (set! a (cons (car ipats) a))))
        (do ((j (- m i) (- j 1)))
            ((zero? j))
            (put name (generate-symbol) #f)))))

(define (browse-random)
  (set! *rand* (remainder (* *rand* 17) 251))
  *rand*)

(define (randomize l)
  (do ((a '()))
      ((null? l) a)
      (let ((n (remainder (browse-random) (length l))))
        (cond ((zero? n)
               (set! a (cons (car l) a))
               (set! l (cdr l))
               l)
              (else
               (do ((n n (- n 1))
                    (x l (cdr x)))
                   ((= n 1)
                    (set! a (cons (cadr x) a))
                    (set-cdr! x (cddr x))
                    x)))))))

(define (match pat dat alist)
  (cond ((null? pat)
         (null? dat))
        ((null? dat) '())
        ((or (eq? (car pat) '?)
             (eq? (car pat)
                  (car dat)))
         (match (cdr pat) (cdr dat) alist))
        ((eq? (car pat) '*)
         (or (match (cdr pat) dat alist)
             (match (cdr pat) (cdr dat) alist)
             (match pat (cdr dat) alist)))
        (else (cond ((not (pair? (car pat)))
                     (cond ((eq? (string-ref (symbol->string (car pat)) 0)
                                 #\?)
                            (let ((val (assq (car pat) alist)))
                              (cond (val (match (cons (cdr val)
                                                      (cdr pat))
                                                dat alist))
                                    (else (match (cdr pat)
                                                 (cdr dat)
                                                 (cons (cons (car pat)
                                                             (car dat))
                                                       alist))))))
                           ((eq? (string-ref (symbol->string (car pat)) 0)
                                 #\*)
                            (let ((val (assq (car pat) alist)))
                              (cond (val (match (append (cdr val)
                                                        (cdr pat))
                                                dat alist))
                                    (else
                                     (do ((l '()
                                             (append-to-tail!
                                               l
                                               (cons (if (null? d)
                                                         '()
                                                         (car d))
                                                     '()))))
                                         (e (cons '() dat) (cdr e))
                                         (d dat (if (null? d) '() (cdr d))))
```

```
147.                                        ((or (null? e)
148.                                             (match (cdr pat)
149.                                                    d
150.                                                    (cons
151.                                                     (cons (car pat) l)
152.                                                     alist)))
153.                                         (if (null? e) #f #t)))))))))
154.                      (else (and
155.                             (pair? (car dat))
156.                             (match (car pat)
157.                                    (car dat) alist)
158.                             (match (cdr pat)
159.                                    (cdr dat) alist)))))))
160.
161. (define database
162.    (randomize
163.     (init 100 10 4 '((a a a b b b b a a a a a b b a a a)
164.                      (a a b b b b a a
165.                                    (a a)(b b))
166.                      (a a a b (b a) b a b a)))))
167.
168. (define (browse)
169.   (investigate
170.     database
171.     '((*a ?b *b ?b a *a a *b *a)
172.       (*a *b *b *a (*a) (*b))
173.       (? ? * (b a) * ? ?))))
174.
175. (define (investigate units pats)
176.   (do ((units units (cdr units)))
177.       ((null? units))
178.       (do ((pats pats (cdr pats)))
179.           ((null? pats))
180.           (do ((p (get (car units) 'pattern)
181.                   (cdr p)))
182.               ((null? p))
183.               (match (car pats) (car p) '())))))
184.
185. (run-benchmark 500 (browse))
```

C.3 cpstak.scm

```
1. ;;; CPSTAK -- A continuation-passing version of the TAK benchmark.
2. ;;; A good test of first class procedures and tail recursion.
3.
4. (define (cpstak x y z)
5.
6.   (define (tak x y z k)
7.     (if (not (< y x))
8.         (k z)
9.         (tak (- x 1)
10.             y
11.             z
12.             (lambda (v1)
13.               (tak (- y 1)
14.                    z
15.                    x
16.                    (lambda (v2)
17.                      (tak (- z 1)
18.                           x
19.                           y
20.                           (lambda (v3)
21.                             (tak v1 v2 v3 k)))))))))
22.
23.   (tak x y z (lambda (a) a)))
24.
25. (run-benchmark 1000 (cpstak 18 12 6))
```

C.4 dderiv.scm

```
1. ;;; DDERIV -- Table-driven symbolic derivation.
2.
3. ;;; Returns the wrong answer for quotients.
4. ;;; Fortunately these aren't used in the benchmark.
5.
6. (define (caddr x) (car (cdr (cdr x))))
7.
8. (define (lookup key table)
9.   (let loop ((x table))
10.     (if (null? x)
11.       #f
12.       (let ((pair (car x)))
13.         (if (eq? (car pair) key)
14.           pair
15.           (loop (cdr x)))))))
16.
17. (define properties '())
18.
19. (define (get key1 key2)
20.   (let ((x (lookup key1 properties)))
21.     (if x
22.       (let ((y (lookup key2 (cdr x))))
23.         (if y
24.           (cdr y)
25.           #f))
26.       #f)))
27.
28. (define (put key1 key2 val)
29.   (let ((x (lookup key1 properties)))
30.     (if x
31.       (let ((y (lookup key2 (cdr x))))
32.         (if y
33.           (set-cdr! y val)
34.           (set-cdr! x (cons (cons key2 val) (cdr x)))))
35.       (set! properties
36.         (cons (list key1 (cons key2 val)) properties)))))
37.
38. (define (+dderiv a)
39.   (cons '+
40.         (map dderiv (cdr a))))
41.
42. (define (-dderiv a)
43.   (cons '-
44.         (map dderiv (cdr a))))
45.
46. (define (*dderiv a)
47.   (list '*
48.         a
49.         (cons '+
50.               (map (lambda (a) (list '/ (dderiv a) a)) (cdr a)))))
51.
52. (define (/dderiv a)
53.   (list '-
54.         (list '/
55.               (dderiv (cadr a))
56.               (caddr a))
57.         (list '/
58.               (cadr a)
59.               (list '*
60.                     (caddr a)
61.                     (caddr a)
62.                     (dderiv (caddr a))))))
63.
64. (put '+ 'dderiv +dderiv)
65. (put '- 'dderiv -dderiv)
66. (put '* 'dderiv *dderiv)
67. (put '/ 'dderiv /dderiv)
68.
69. (define (dderiv a)
70.   (if (not (pair? a))
71.     (if (eq? a 'x) 1 0)
72.     (let ((f (get (car a) 'dderiv)))
```

```
73.       (if f
74.         (f a)
75.         (error "No derivation method available")))))
76.
77. (define (run)
78.   (dderiv '(+ (* 3 x x) (* a x x) (* b x) 5)))
79.
80. (run-benchmark 100000 (run))
```

C.5 *earley.scm*

```
;;; EARLEY -- Earley's parser, written by Marc Feeley.

(define (make-parser grammar lexer)

  (define (non-terminals grammar) ; return vector of non-terminals in grammar

    (define (add-nt nt nts)
      (if (member nt nts) nts (cons nt nts))) ; use equal? for equality tests

    (let def-loop ((defs grammar) (nts '()))
      (if (pair? defs)
        (let* ((def (car defs))
               (head (car def)))
          (let rule-loop ((rules (cdr def))
                          (nts (add-nt head nts)))
            (if (pair? rules)
              (let ((rule (car rules)))
                (let loop ((l rule) (nts nts))
                  (if (pair? l)
                    (let ((nt (car l)))
                      (loop (cdr l) (add-nt nt nts)))
                    (rule-loop (cdr rules) nts))))
              (def-loop (cdr defs) nts))))
        (list->vector (reverse nts))))) ; goal non-terminal must be at index 0

  (define (ind nt nts) ; return index of non-terminal 'nt' in 'nts'
    (let loop ((i (- (vector-length nts) 1)))
      (if (>= i 0)
        (if (equal? (vector-ref nts i) nt) i (loop (- i 1)))
        #f)))

  (define (nb-configurations grammar) ; return nb of configurations in grammar
    (let def-loop ((defs grammar) (nb-confs 0))
      (if (pair? defs)
        (let ((def (car defs)))
          (let rule-loop ((rules (cdr def)) (nb-confs nb-confs))
            (if (pair? rules)
              (let ((rule (car rules)))
                (let loop ((l rule) (nb-confs nb-confs))
                  (if (pair? l)
                    (loop (cdr l) (+ nb-confs 1))
                    (rule-loop (cdr rules) (+ nb-confs 1)))))
              (def-loop (cdr defs) nb-confs))))
        nb-confs)))

; First, associate a numeric identifier to every non-terminal in the
; grammar (with the goal non-terminal associated with 0).
;
; So, for the grammar given above we get:
;
; s -> 0   x -> 1   = -> 4   e ->3   + -> 4   v -> 5   y -> 6

  (let* ((nts (non-terminals grammar))          ; id map = list of non-terms
         (nb-nts (vector-length nts))           ; the number of non-terms
         (nb-confs (+ (nb-configurations grammar) nb-nts)) ; the nb of confs
         (starters (make-vector nb-nts '()))    ; starters for every non-term
         (enders (make-vector nb-nts '()))      ; enders for every non-term
         (predictors (make-vector nb-nts '()))  ; predictors for every non-term
         (steps (make-vector nb-confs #f))      ; what to do in a given conf
         (names (make-vector nb-confs #f)))     ; name of rules

    (define (setup-tables grammar nts starters enders predictors steps names)

      (define (add-conf conf nt nts class)
        (let ((i (ind nt nts)))
          (vector-set! class i (cons conf (vector-ref class i)))))

      (let ((nb-nts (vector-length nts)))

        (let nt-loop ((i (- nb-nts 1)))
          (if (>= i 0)
            (begin
```

```
73.               (vector-set! steps i (- i nb-nts))
74.               (vector-set! names i (list (vector-ref nts i) 0))
75.               (vector-set! enders i (list i))
76.               (nt-loop (- i 1)))))
77.
78.         (let def-loop ((defs grammar) (conf (vector-length nts)))
79.           (if (pair? defs)
80.             (let* ((def (car defs))
81.                    (head (car def)))
82.               (let rule-loop ((rules (cdr def)) (conf conf) (rule-num 1))
83.                 (if (pair? rules)
84.                   (let ((rule (car rules)))
85.                     (vector-set! names conf (list head rule-num))
86.                     (add-conf conf head nts starters)
87.                     (let loop ((l rule) (conf conf))
88.                       (if (pair? l)
89.                         (let ((nt (car l)))
90.                           (vector-set! steps conf (ind nt nts))
91.                           (add-conf conf nt nts predictors)
92.                           (loop (cdr l) (+ conf 1)))
93.                         (begin
94.                           (vector-set! steps conf (- (ind head nts) nb-nts))
95.                           (add-conf conf head nts enders)
96.                           (rule-loop (cdr rules) (+ conf 1) (+ rule-num 1))))))
97.                   (def-loop (cdr defs) conf))))))))
98.
99.  ; Now, for each non-terminal, compute the starters, enders and predictors and
100. ; the names and steps tables.
101.
102.    (setup-tables grammar nts starters enders predictors steps names)
103.
104. ; Build the parser description
105.
106.    (let ((parser-descr (vector lexer
107.                               nts
108.                               starters
109.                               enders
110.                               predictors
111.                               steps
112.                               names)))
113.      (lambda (input)
114.
115.        (define (ind nt nts) ; return index of non-terminal 'nt' in 'nts'
116.          (let loop ((i (- (vector-length nts) 1)))
117.            (if (>= i 0)
118.              (if (equal? (vector-ref nts i) nt) i (loop (- i 1)))
119.              #f)))
120.
121.        (define (comp-tok tok nts) ; transform token to parsing format
122.          (let loop ((l1 (cdr tok)) (l2 '()))
123.            (if (pair? l1)
124.              (let ((i (ind (car l1) nts)))
125.                (if i
126.                  (loop (cdr l1) (cons i l2))
127.                  (loop (cdr l1) l2)))
128.              (cons (car tok) (reverse l2)))))
129.
130.        (define (input->tokens input lexer nts)
131.          (list->vector (map (lambda (tok) (comp-tok tok nts)) (lexer input))))
132.
133.        (define (make-states nb-toks nb-confs)
134.          (let ((states (make-vector (+ nb-toks 1) #f)))
135.            (let loop ((i nb-toks))
136.              (if (>= i 0)
137.                (let ((v (make-vector (+ nb-confs 1) #f)))
138.                  (vector-set! v 0 -1)
139.                  (vector-set! states i v)
140.                  (loop (- i 1)))
141.                states))))
142.
143.        (define (conf-set-get state conf)
144.          (vector-ref state (+ conf 1)))
145.
146.        (define (conf-set-get* state state-num conf)
```

```
147.          (let ((conf-set (conf-set-get state conf)))
148.            (if conf-set
149.              conf-set
150.              (let ((conf-set (make-vector (+ state-num 6) #f)))
151.                (vector-set! conf-set 1 -3) ; old elems tail (points to head)
152.                (vector-set! conf-set 2 -1) ; old elems head
153.                (vector-set! conf-set 3 -1) ; new elems tail (points to head)
154.                (vector-set! conf-set 4 -1) ; new elems head
155.                (vector-set! state (+ conf 1) conf-set)
156.                conf-set))))
157.
158.        (define (conf-set-merge-new! conf-set)
159.          (vector-set! conf-set
160.            (+ (vector-ref conf-set 1) 5)
161.            (vector-ref conf-set 4))
162.          (vector-set! conf-set 1 (vector-ref conf-set 3))
163.          (vector-set! conf-set 3 -1)
164.          (vector-set! conf-set 4 -1))
165.
166.        (define (conf-set-head conf-set)
167.          (vector-ref conf-set 2))
168.
169.        (define (conf-set-next conf-set i)
170.          (vector-ref conf-set (+ i 5)))
171.
172.        (define (conf-set-member? state conf i)
173.          (let ((conf-set (vector-ref state (+ conf 1))))
174.            (if conf-set
175.              (conf-set-next conf-set i)
176.              #f)))
177.
178.        (define (conf-set-adjoin state conf-set conf i)
179.          (let ((tail (vector-ref conf-set 3))) ; put new element at tail
180.            (vector-set! conf-set (+ i 5) -1)
181.            (vector-set! conf-set (+ tail 5) i)
182.            (vector-set! conf-set 3 i)
183.            (if (< tail 0)
184.              (begin
185.                (vector-set! conf-set 0 (vector-ref state 0))
186.                (vector-set! state 0 conf)))))
187.
188.        (define (conf-set-adjoin* states state-num l i)
189.          (let ((state (vector-ref states state-num)))
190.            (let loop ((l1 l))
191.              (if (pair? l1)
192.                (let* ((conf (car l1))
193.                       (conf-set (conf-set-get* state state-num conf)))
194.                  (if (not (conf-set-next conf-set i))
195.                    (begin
196.                      (conf-set-adjoin state conf-set conf i)
197.                      (loop (cdr l1)))
198.                    (loop (cdr l1))))))))
199.
200.        (define (conf-set-adjoin** states states* state-num conf i)
201.          (let ((state (vector-ref states state-num)))
202.            (if (conf-set-member? state conf i)
203.              (let* ((state* (vector-ref states* state-num))
204.                     (conf-set* (conf-set-get* state* state-num conf)))
205.                (if (not (conf-set-next conf-set* i))
206.                  (conf-set-adjoin state* conf-set* conf i))
207.                #t)
208.              #f)))
209.
210.        (define (conf-set-union state conf-set conf other-set)
211.          (let loop ((i (conf-set-head other-set)))
212.            (if (>= i 0)
213.              (if (not (conf-set-next conf-set i))
214.                (begin
215.                  (conf-set-adjoin state conf-set conf i)
216.                  (loop (conf-set-next other-set i)))
217.                (loop (conf-set-next other-set i))))))
218.
219.        (define (forw states state-num starters enders predictors steps nts)
220.
```

```
221.          (define (predict state state-num conf-set conf nt starters enders)
222.
223.            ; add configurations which start the non-terminal 'nt' to the
224.            ; right of the dot
225.
226.            (let loop1 ((l (vector-ref starters nt)))
227.              (if (pair? l)
228.                (let* ((starter (car l))
229.                       (starter-set (conf-set-get* state state-num starter)))
230.                  (if (not (conf-set-next starter-set state-num))
231.                    (begin
232.                      (conf-set-adjoin state starter-set starter state-num)
233.                      (loop1 (cdr l)))
234.                    (loop1 (cdr l))))))
235.
236.            ; check for possible completion of the non-terminal 'nt' to the
237.            ; right of the dot
238.
239.            (let loop2 ((l (vector-ref enders nt)))
240.              (if (pair? l)
241.                (let ((ender (car l)))
242.                  (if (conf-set-member? state ender state-num)
243.                    (let* ((next (+ conf 1))
244.                           (next-set (conf-set-get* state state-num next)))
245.                      (conf-set-union state next-set next conf-set)
246.                      (loop2 (cdr l)))
247.                    (loop2 (cdr l)))))))
248.
249.          (define (reduce states state state-num conf-set head preds)
250.
251.            ; a non-terminal is now completed so check for reductions that
252.            ; are now possible at the configurations 'preds'
253.
254.            (let loop1 ((l preds))
255.              (if (pair? l)
256.                (let ((pred (car l)))
257.                  (let loop2 ((i head))
258.                    (if (>= i 0)
259.                      (let ((pred-set (conf-set-get (vector-ref states i) pred)))
260.                        (if pred-set
261.                          (let* ((next (+ pred 1))
262.                                 (next-set (conf-set-get* state state-num next)))
263.                            (conf-set-union state next-set next pred-set)))
264.                        (loop2 (conf-set-next conf-set i)))
265.                      (loop1 (cdr l))))))))
266.
267.            (let ((state (vector-ref states state-num))
268.                  (nb-nts (vector-length nts)))
269.              (let loop ()
270.                (let ((conf (vector-ref state 0)))
271.                  (if (>= conf 0)
272.                    (let* ((step (vector-ref steps conf))
273.                           (conf-set (vector-ref state (+ conf 1)))
274.                           (head (vector-ref conf-set 4)))
275.                      (vector-set! state 0 (vector-ref conf-set 0))
276.                      (conf-set-merge-new! conf-set)
277.                      (if (>= step 0)
278.                        (predict state state-num conf-set conf step starters enders)
279.                        (let ((preds (vector-ref predictors (+ step nb-nts))))
280.                          (reduce states state state-num conf-set head preds)))
281.                      (loop)))))))
282.
283.          (define (forward starters enders predictors steps nts toks)
284.            (let* ((nb-toks (vector-length toks))
285.                   (nb-confs (vector-length steps))
286.                   (states (make-states nb-toks nb-confs))
287.                   (goal-starters (vector-ref starters 0)))
288.              (conf-set-adjoin* states 0 goal-starters 0) ; predict goal
289.              (forw states 0 starters enders predictors steps nts)
290.              (let loop ((i 0))
291.                (if (< i nb-toks)
292.                  (let ((tok-nts (cdr (vector-ref toks i))))
293.                    (conf-set-adjoin* states (+ i 1) tok-nts i) ; scan token
294.                    (forw states (+ i 1) starters enders predictors steps nts)
```

```
295.                 (loop (+ i 1)))))
296.           states))
297.
298.       (define (produce conf i j enders steps toks states states* nb-nts)
299.         (let ((prev (- conf 1)))
300.           (if (and (>= conf nb-nts) (>= (vector-ref steps prev) 0))
301.             (let loop1 ((l (vector-ref enders (vector-ref steps prev))))
302.               (if (pair? l)
303.                 (let* ((ender (car l))
304.                        (ender-set (conf-set-get (vector-ref states j)
305.                                                 ender)))
306.                   (if ender-set
307.                     (let loop2 ((k (conf-set-head ender-set)))
308.                       (if (>= k 0)
309.                         (begin
310.                           (and (>= k i)
311.                                (conf-set-adjoin** states states* k prev i)
312.                                (conf-set-adjoin** states states* j ender k))
313.                           (loop2 (conf-set-next ender-set k)))
314.                         (loop1 (cdr l))))
315.                     (loop1 (cdr l)))))))))
316.
317.       (define (back states states* state-num enders steps nb-nts toks)
318.         (let ((state* (vector-ref states* state-num)))
319.           (let loop1 ()
320.             (let ((conf (vector-ref state* 0)))
321.               (if (>= conf 0)
322.                 (let* ((conf-set (vector-ref state* (+ conf 1)))
323.                        (head (vector-ref conf-set 4)))
324.                   (vector-set! state* 0 (vector-ref conf-set 0))
325.                   (conf-set-merge-new! conf-set)
326.                   (let loop2 ((i head))
327.                     (if (>= i 0)
328.                       (begin
329.                         (produce conf i state-num enders steps
330.                                  toks states states* nb-nts)
331.                         (loop2 (conf-set-next conf-set i)))
332.                       (loop1)))))))))
333.
334.       (define (backward states enders steps nts toks)
335.         (let* ((nb-toks (vector-length toks))
336.                (nb-confs (vector-length steps))
337.                (nb-nts (vector-length nts))
338.                (states* (make-states nb-toks nb-confs))
339.                (goal-enders (vector-ref enders 0)))
340.           (let loop1 ((l goal-enders))
341.             (if (pair? l)
342.               (let ((conf (car l)))
343.                 (conf-set-adjoin** states states* nb-toks conf 0)
344.                 (loop1 (cdr l)))))
345.           (let loop2 ((i nb-toks))
346.             (if (>= i 0)
347.               (begin
348.                 (back states states* i enders steps nb-nts toks)
349.                 (loop2 (- i 1)))))
350.           states*))
351.
352.       (define (parsed? nt i j nts enders states)
353.         (let ((nt* (ind nt nts)))
354.           (if nt*
355.             (let ((nb-nts (vector-length nts)))
356.               (let loop ((l (vector-ref enders nt*)))
357.                 (if (pair? l)
358.                   (let ((conf (car l)))
359.                     (if (conf-set-member? (vector-ref states j) conf i)
360.                       #t
361.                       (loop (cdr l))))
362.                   #f)))
363.             #f)))
364.
365.       (define (deriv-trees conf i j enders steps names toks states nb-nts)
366.         (let ((name (vector-ref names conf)))
367.
368.           (if name ; 'conf' is at the start of a rule (either special or not)
```

```
            (if (< conf nb-nts)
              (list (list name (car (vector-ref toks i))))
              (list (list name)))

            (let ((prev (- conf 1)))
              (let loop1 ((l1 (vector-ref enders (vector-ref steps prev)))
                          (l2 '()))
                (if (pair? l1)
                  (let* ((ender (car l1))
                         (ender-set (conf-set-get (vector-ref states j)
                                                  ender)))
                    (if ender-set
                      (let loop2 ((k (conf-set-head ender-set)) (l2 l2))
                        (if (>= k 0)
                          (if (and (>= k i)
                                   (conf-set-member? (vector-ref states k)
                                                     prev i))
                            (let ((prev-trees
                                   (deriv-trees prev i k enders steps names
                                                toks states nb-nts))
                                  (ender-trees
                                    (deriv-trees ender k j enders steps names
                                                 toks states nb-nts)))
                              (let loop3 ((l3 ender-trees) (l2 l2))
                                (if (pair? l3)
                                  (let ((ender-tree (list (car l3))))
                                    (let loop4 ((l4 prev-trees) (l2 l2))
                                      (if (pair? l4)
                                        (loop4 (cdr l4)
                                               (cons (append (car l4)
                                                             ender-tree)
                                                     l2))
                                        (loop3 (cdr l3) l2))))
                                  (loop2 (conf-set-next ender-set k) l2))))
                            (loop2 (conf-set-next ender-set k) l2))
                          (loop1 (cdr l1) l2)))
                      (loop1 (cdr l1) l2)))
                  l2))))))

        (define (deriv-trees* nt i j nts enders steps names toks states)
          (let ((nt* (ind nt nts)))
            (if nt*
              (let ((nb-nts (vector-length nts)))
                (let loop ((l (vector-ref enders nt*)) (trees '()))
                  (if (pair? l)
                    (let ((conf (car l)))
                      (if (conf-set-member? (vector-ref states j) conf i)
                        (loop (cdr l)
                              (append (deriv-trees conf i j enders steps names
                                                   toks states nb-nts)
                                      trees))
                        (loop (cdr l) trees)))
                    trees)))
              #f)))

        (define (nb-deriv-trees conf i j enders steps toks states nb-nts)
          (let ((prev (- conf 1)))
            (if (or (< conf nb-nts) (< (vector-ref steps prev) 0))
              1
              (let loop1 ((l (vector-ref enders (vector-ref steps prev)))
                          (n 0))
                (if (pair? l)
                  (let* ((ender (car l))
                         (ender-set (conf-set-get (vector-ref states j)
                                                  ender)))
                    (if ender-set
                      (let loop2 ((k (conf-set-head ender-set)) (n n))
                        (if (>= k 0)
                          (if (and (>= k i)
                                   (conf-set-member? (vector-ref states k)
                                                     prev i))
                            (let ((nb-prev-trees
                                   (nb-deriv-trees prev i k enders steps
                                                   toks states nb-nts))
```

```
                                    (nb-ender-trees
                                      (nb-deriv-trees ender k j enders steps
                                                      toks states nb-nts)))
                                  (loop2 (conf-set-next ender-set k)
                                         (+ n (* nb-prev-trees nb-ender-trees))))
                                (loop2 (conf-set-next ender-set k) n))
                              (loop1 (cdr l) n)))
                            (loop1 (cdr l) n)))
                        n)))))

        (define (nb-deriv-trees* nt i j nts enders steps toks states)
          (let ((nt* (ind nt nts)))
            (if nt*
              (let ((nb-nts (vector-length nts)))
                (let loop ((l (vector-ref enders nt*)) (nb-trees 0))
                  (if (pair? l)
                    (let ((conf (car l)))
                      (if (conf-set-member? (vector-ref states j) conf i)
                        (loop (cdr l)
                              (+ (nb-deriv-trees conf i j enders steps
                                                 toks states nb-nts)
                                 nb-trees))
                        (loop (cdr l) nb-trees)))
                    nb-trees)))
              #f)))

        (let* ((lexer      (vector-ref parser-descr 0))
               (nts        (vector-ref parser-descr 1))
               (starters   (vector-ref parser-descr 2))
               (enders     (vector-ref parser-descr 3))
               (predictors (vector-ref parser-descr 4))
               (steps      (vector-ref parser-descr 5))
               (names      (vector-ref parser-descr 6))
               (toks       (input->tokens input lexer nts)))

          (vector nts
                  starters
                  enders
                  predictors
                  steps
                  names
                  toks
                  (backward (forward starters enders predictors steps nts toks)
                            enders steps nts toks)
                  parsed?
                  deriv-trees*
                  nb-deriv-trees*))))))

(define (parse->parsed? parse nt i j)
  (let* ((nts     (vector-ref parse 0))
         (enders  (vector-ref parse 2))
         (states  (vector-ref parse 7))
         (parsed? (vector-ref parse 8)))
    (parsed? nt i j nts enders states)))

(define (parse->trees parse nt i j)
  (let* ((nts          (vector-ref parse 0))
         (enders       (vector-ref parse 2))
         (steps        (vector-ref parse 4))
         (names        (vector-ref parse 5))
         (toks         (vector-ref parse 6))
         (states       (vector-ref parse 7))
         (deriv-trees* (vector-ref parse 9)))
    (deriv-trees* nt i j nts enders steps names toks states)))

(define (parse->nb-trees parse nt i j)
  (let* ((nts             (vector-ref parse 0))
         (enders          (vector-ref parse 2))
         (steps           (vector-ref parse 4))
         (toks            (vector-ref parse 6))
         (states          (vector-ref parse 7))
         (nb-deriv-trees* (vector-ref parse 10)))
    (nb-deriv-trees* nt i j nts enders steps toks states)))
```

```
517. (define (test)
518.   (let ((p (make-parser '( (s (a) (s s)) )
519.                        (lambda (l) (map (lambda (x) (list x x)) l)))))
520.     (let ((x (p '(a a a a a a a a a))))
521.       (length (parse->trees x 's 0 9)))))
522.
523. (run-benchmark 100 (test))
```

C.6 fib.scm

```
1. ;;; FIB -- A classic benchmark, computes fib(25) inefficiently.
2.
3. (define (fib n)
4.   (if (< n 2)
5.     n
6.     (+ (fib (- n 1))
7.        (fib (- n 2)))))
8.
9. (run-benchmark 250 (fib 25))
10.
```

C.7 fibfp.scm

```
1. ;;; FIBFP -- Computes fib(25) using floating point
2.
3. (define (fibfp n)
4.   (if (< n 2.)
5.     n
6.     (+ (fibfp (- n 1.))
7.        (fibfp (- n 2.)))))
8.
9.
10. (run-benchmark 25 (fibfp 25.))
11.
```

C.8 mbrot.scm

```
1. ;;; MBROT -- Generation of Mandelbrot set fractal.
2.
3. (define (count r i step x y)
4.
5.   (let ((max-count 64)
6.         (radius^2  16.0))
7.
8.     (let ((cr (+ r (* (exact->inexact x) step)))
9.           (ci (+ i (* (exact->inexact y) step))))
10.
11.       (let loop ((zr cr)
12.                  (zi ci)
13.                  (c 0))
14.         (if (= c max-count)
15.           c
16.           (let ((zr^2 (* zr zr))
17.                 (zi^2 (* zi zi)))
18.             (if (> (+ zr^2 zi^2) radius^2)
19.               c
20.               (let ((new-zr (+ (- zr^2 zi^2) cr))
21.                     (new-zi (+ (* 2.0 (* zr zi)) ci)))
22.                 (loop new-zr new-zi (+ c 1)))))))))))
23.
24. (define (mbrot matrix r i step n)
25.   (let loop1 ((y (- n 1)))
26.     (if (>= y 0)
27.       (let loop2 ((x (- n 1)))
28.         (if (>= x 0)
29.           (begin
30.             (vector-set! (vector-ref matrix x) y (count r i step x y))
31.             (loop2 (- x 1)))
32.           (loop1 (- y 1)))))))
33.
34. (define (test)
35.   (let ((n 75))
36.     (let ((matrix (make-vector n #f)))
37.       (let loop ((i (- n 1)))
38.         (if (>= i 0)
39.           (begin
40.             (vector-set! matrix i (make-vector n 0.0))
41.             (loop (- i 1)))))
42.       (mbrot matrix -1.0 -0.5 0.005 n)
43.       (vector-ref (vector-ref matrix 0) 0))))
44.
45.
46. (run-benchmark 100 (test))
```

C.9 nqueens.scm

```
1. ;;; NQUEENS -- Compute number of solutions to 8-queens problem.
2.
3. (define trace? #f)
4.
5. (define (nqueens n)
6.
7.   (define (1-to n)
8.     (let loop ((i n) (l '()))
9.       (if (= i 0) l (loop (- i 1) (cons i l)))))
10.
11.  (define (try x y z)
12.    (if (null? x)
13.      (if (null? y)
14.        (begin (if trace? (begin (write z) (newline))) 1)
15.        0)
16.      (+ (if (ok? (car x) 1 z)
17.             (try (append (cdr x) y) '() (cons (car x) z))
18.             0)
19.         (try (cdr x) (cons (car x) y) z))))
20.
21.  (define (ok? row dist placed)
22.    (if (null? placed)
23.      #t
24.      (and (not (= (car placed) (+ row dist)))
25.           (not (= (car placed) (- row dist)))
26.           (ok? row (+ dist 1) (cdr placed)))))
27.
28.  (try (1-to n) '() '()))
29.
30. (run-benchmark 750 (nqueens 8))
```

C.10 pnpoly.scm

```
1. ;;; PNPOLY - Test if a point is contained in a 2D polygon.
2.
3. (define (pt-in-poly2 xp yp x y)
4.   (let loop ((c #f) (i (- (vector-length xp) 1)) (j 0))
5.     (if (< i 0)
6.       c
7.       (if (or (and (or (> (vector-ref yp i) y)
8.                        (>= y (vector-ref yp j)))
9.                    (or (> (vector-ref yp j) y)
10.                       (>= y (vector-ref yp i))))
11.               (>= x
12.                   (+ (vector-ref xp i)
13.                      (/ (*
14.                           (- (vector-ref xp j)
15.                              (vector-ref xp i))
16.                           (- y (vector-ref yp i)))
17.                         (- (vector-ref yp j)
18.                            (vector-ref yp i))))))
19.           (loop c (- i 1) i)
20.           (loop (not c) (- i 1) i)))))
21.
22. (define (run)
23.   (let ((count 0)
24.         (xp (vector 0. 1. 1. 0. 0. 1. -.5 -1. -1. -2. -2.5
25.                                -2. -1.5 -.5 1. 1. 0. -.5 -1. -.5))
26.         (yp (vector 0. 0. 1. 1. 2. 3. 2. 3. 0. -.5 -1.
27.                                -1.5 -2. -2. -1.5 -1. -.5 -1. -1. -.5)))
28.     (if (pt-in-poly2 xp yp .5 .5) (set! count (+ count 1)))
29.     (if (pt-in-poly2 xp yp .5 1.5) (set! count (+ count 1)))
30.     (if (pt-in-poly2 xp yp -.5 1.5) (set! count (+ count 1)))
31.     (if (pt-in-poly2 xp yp .75 2.25) (set! count (+ count 1)))
32.     (if (pt-in-poly2 xp yp 0. 2.01) (set! count (+ count 1)))
33.     (if (pt-in-poly2 xp yp -.5 2.5) (set! count (+ count 1)))
34.     (if (pt-in-poly2 xp yp -1. -.5) (set! count (+ count 1)))
35.     (if (pt-in-poly2 xp yp -1.5 .5) (set! count (+ count 1)))
36.     (if (pt-in-poly2 xp yp -2.25 -1.) (set! count (+ count 1)))
37.     (if (pt-in-poly2 xp yp .5 -.25) (set! count (+ count 1)))
38.     (if (pt-in-poly2 xp yp .5 -1.25) (set! count (+ count 1)))
39.     (if (pt-in-poly2 xp yp -.5 -2.5) (set! count (+ count 1)))
40.     count))
41.
42. (run-benchmark 50000 (run))
```

C.11 puzzle.scm

```
;;; PUZZLE -- Forest Baskett's Puzzle benchmark, originally written in Pascal.

(define (iota n)
  (do ((n n (- n 1))
       (list '() (cons (- n 1) list)))
      ((zero? n) list)))

(define size 511)
(define classmax 3)
(define typemax 12)

(define *iii* 0)
(define *kount* 0)
(define *d* 8)

(define *piececount* (make-vector (+ classmax 1) 0))
(define *class* (make-vector (+ typemax 1) 0))
(define *piecemax* (make-vector (+ typemax 1) 0))
(define *puzzle* (make-vector (+ size 1) #f))
(define *p* (make-vector (+ typemax 1) #f))

(define (fit i j)
  (let ((end (vector-ref *piecemax* i)))
    (do ((k 0 (+ k 1)))
        ((or (> k end)
             (and (vector-ref (vector-ref *p* i) k)
                  (vector-ref *puzzle* (+ j k))))
         (if (> k end) #t #f)))))

(define (place i j)
  (let ((end (vector-ref *piecemax* i)))
    (do ((k 0 (+ k 1)))
        ((> k end))
        (cond ((vector-ref (vector-ref *p* i) k)
               (vector-set! *puzzle* (+ j k) #t)
               #t)))
    (vector-set! *piececount*
                 (vector-ref *class* i)
                 (- (vector-ref *piececount* (vector-ref *class* i)) 1))
    (do ((k j (+ k 1)))
        ((or (> k size) (not (vector-ref *puzzle* k)))
         (if (> k size) 0 k)))))

(define (puzzle-remove i j)
  (let ((end (vector-ref *piecemax* i)))
    (do ((k 0 (+ k 1)))
        ((> k end))
        (cond ((vector-ref (vector-ref *p* i) k)
               (vector-set! *puzzle* (+ j k) #f)
               #f)))
    (vector-set! *piececount*
                 (vector-ref *class* i)
                 (+ (vector-ref *piececount* (vector-ref *class* i)) 1))))

(define (trial j)
  (let ((k 0))
    (call-with-current-continuation
     (lambda (return)
       (do ((i 0 (+ i 1)))
           ((> i typemax) (set! *kount* (+ *kount* 1)) #f)
           (cond
            ((not
              (zero?
               (vector-ref *piececount* (vector-ref *class* i))))
             (cond
              ((fit i j)
               (set! k (place i j))
               (cond
                ((or (trial k) (zero? k))
                 (set! *kount* (+ *kount* 1))
                 (return #t))
                (else (puzzle-remove i j))))))))))))
```

```
73.
74. (define (definePiece iclass ii jj kk)
75.   (let ((index 0))
76.     (do ((i 0 (+ i 1)))
77.         ((> i ii))
78.         (do ((j 0 (+ j 1)))
79.             ((> j jj))
80.             (do ((k 0 (+ k 1)))
81.                 ((> k kk))
82.                 (set! index (+ i (* *d* (+ j (* *d* k)))))
83.                 (vector-set! (vector-ref *p* *iii*) index  #t))))
84.     (vector-set! *class* *iii* iclass)
85.     (vector-set! *piecemax* *iii* index)
86.     (cond ((not (= *iii* typemax))
87.            (set! *iii* (+ *iii* 1))))))
88.
89. (define (start)
90.   (set! *kount* 0)
91.   (do ((m 0 (+ m 1)))
92.       ((> m size))
93.       (vector-set! *puzzle* m #t))
94.   (do ((i 1 (+ i 1)))
95.       ((> i 5))
96.       (do ((j 1 (+ j 1)))
97.           ((> j 5))
98.           (do ((k 1 (+ k 1)))
99.               ((> k 5))
100.              (vector-set! *puzzle* (+ i (* *d* (+ j (* *d* k)))) #f))))
101.  (do ((i 0 (+ i 1)))
102.      ((> i typemax))
103.      (do ((m 0 (+ m 1)))
104.          ((> m size))
105.          (vector-set! (vector-ref *p* i) m #f)))
106.  (set! *iii* 0)
107.  (definePiece 0 3 1 0)
108.  (definePiece 0 1 0 3)
109.  (definePiece 0 0 3 1)
110.  (definePiece 0 1 3 0)
111.  (definePiece 0 3 0 1)
112.  (definePiece 0 0 1 3)
113.
114.  (definePiece 1 2 0 0)
115.  (definePiece 1 0 2 0)
116.  (definePiece 1 0 0 2)
117.
118.  (definePiece 2 1 1 0)
119.  (definePiece 2 1 0 1)
120.  (definePiece 2 0 1 1)
121.
122.  (definePiece 3 1 1 1)
123.
124.  (vector-set! *piececount* 0 13)
125.  (vector-set! *piececount* 1 3)
126.  (vector-set! *piececount* 2 1)
127.  (vector-set! *piececount* 3 1)
128.  (let ((m (+ (* *d* (+ *d* 1)) 1))
129.        (n 0))
130.    (cond ((fit 0 m) (set! n (place 0 m)))
131.          (else (begin (newline) (display "Error."))))
132.    (if (trial n)
133.      *kount*
134.      #f)))
135.
136. (for-each (lambda (i) (vector-set! *p* i (make-vector (+ size 1) #f)))
137.           (iota (+ typemax 1)))
138.
139. (run-benchmark 100 (start))
```

C.12 simplex.scm

```
;;; SIMPLEX -- Simplex algorithm.

(define (matrix-rows a)       (vector-length a))
(define (matrix-columns a)    (vector-length (vector-ref a 0)))
(define (matrix-ref a i j)    (vector-ref (vector-ref a i) j))
(define (matrix-set! a i j x) (vector-set! (vector-ref a i) j x))

(define (fuck-up)
  (error "This shouldn't happen"))

(define (simplex a m1 m2 m3)
 (define *epsilon* 1e-6)
 (if (not (and (>= m1 0)
               (>= m2 0)
               (>= m3 0)
               (= (matrix-rows a) (+ m1 m2 m3 2))))
  (fuck-up))
 (let* ((m12 (+ (+ m1 m2) 1))
        (m (- (matrix-rows a) 2))
        (n (- (matrix-columns a) 1))
        (l1 (make-vector n 0))
        (l2 (make-vector m 0))
        (l3 (make-vector m2 0))
        (nl1 n)
        (iposv (make-vector m 0))
        (izrov (make-vector n 0))
        (ip 0)
        (kp 0)
        (bmax 0.0)
        (one? #f)
        (pass2? #t))
  (define (simp1 mm abs?)
   (set! kp (vector-ref l1 0))
   (set! bmax (matrix-ref a mm kp))
   (do ((k 1 (+ k 1))) ((>= k nl1))
    (if (positive?
         (if abs?
             (- (abs (matrix-ref a mm (vector-ref l1 k)))
                (abs bmax))
             (- (matrix-ref a mm (vector-ref l1 k)) bmax)))
        (begin
         (set! kp (vector-ref l1 k))
         (set! bmax (matrix-ref a mm (vector-ref l1 k)))))))
  (define (simp2)
   (set! ip 0)
   (let ((q1 0.0)
         (flag? #f))
    (do ((i 0 (+ i 1))) ((= i m))
     (if flag?
         (if (< (matrix-ref a (vector-ref l2 i) kp) (- 0. *epsilon*))
             (begin
              (let ((q (/ (- 0. (matrix-ref a (vector-ref l2 i) 0))
                          (matrix-ref a (vector-ref l2 i) kp))))
               (cond
                ((< q q1)
                 (set! ip (vector-ref l2 i))
                 (set! q1 q))
                ((= q q1)
                 (let ((qp 0.0)
                       (q0 0.0))
                  (let loop ((k 1))
                   (if (<= k n)
                       (begin
                        (set! qp (/ (- 0. (matrix-ref a ip k))
                                    (matrix-ref a ip kp)))
                        (set! q0 (/ (-
                                     0. (matrix-ref a (vector-ref l2 i) k))
                                    (matrix-ref a (vector-ref l2 i) kp)))
                        (if (= q0 qp)
                            (loop (+ k 1))))))
                  (if (< q0 qp)
                      (set! ip (vector-ref l2 i)))))))))))))
```

```
73.          (if (< (matrix-ref a (vector-ref l2 i) kp) (- 0. *epsilon*))
74.              (begin
75.               (set! q1 (/ (- 0. (matrix-ref a (vector-ref l2 i) 0))
76.                           (matrix-ref a (vector-ref l2 i) kp)))
77.               (set! ip (vector-ref l2 i))
78.               (set! flag? #t)))))))
79.    (define (simp3 one?)
80.     (let ((piv (/ 1. (matrix-ref a ip kp))))
81.      (do ((ii 0 (+ ii 1))) ((= ii (+ m (if one? 2 1))))
82.       (if (not (= ii ip))
83.           (begin
84.            (matrix-set! a ii kp (* piv (matrix-ref a ii kp)))
85.            (do ((kk 0 (+ kk 1))) ((= kk (+ n 1)))
86.             (if (not (= kk kp))
87.                 (matrix-set! a ii kk (- (matrix-ref a ii kk)
88.                                         (* (matrix-ref a ip kk)
89.                                            (matrix-ref a ii kp))))))))))
90.      (do ((kk 0 (+ kk 1))) ((= kk (+ n 1)))
91.       (if (not (= kk kp))
92.           (matrix-set! a ip kk (* (- 0. piv) (matrix-ref a ip kk)))))
93.      (matrix-set! a ip kp piv)))
94.    (do ((k 0 (+ k 1))) ((= k n))
95.     (vector-set! l1 k (+ k 1))
96.     (vector-set! izrov k k))
97.    (do ((i 0 (+ i 1))) ((= i m))
98.     (if (negative? (matrix-ref a (+ i 1) 0))
99.         (fuck-up))
100.    (vector-set! l2 i (+ i 1))
101.    (vector-set! iposv i (+ n i)))
102.   (do ((i 0 (+ i 1))) ((= i m2)) (vector-set! l3 i #t))
103.   (if (positive? (+ m2 m3))
104.       (begin
105.        (do ((k 0 (+ k 1))) ((= k (+ n 1)))
106.         (do ((i (+ m1 1) (+ i 1)) (sum 0.0 (+ sum (matrix-ref a i k))))
107.           ((> i m) (matrix-set! a (+ m 1) k (- 0. sum)))))
108.        (let loop ()
109.         (simp1 (+ m 1) #f)
110.         (cond
111.          ((<= bmax *epsilon*)
112.           (cond ((< (matrix-ref a (+ m 1) 0) (- 0. *epsilon*))
113.                  (set! pass2? #f))
114.                 ((<= (matrix-ref a (+ m 1) 0) *epsilon*)
115.                  (let loop ((ip1 m12))
116.                   (if (<= ip1 m)
117.                       (cond ((= (vector-ref iposv (- ip1 1)) (+ ip (- n 1)))
118.                              (simp1 ip1 #t)
119.                              (cond ((positive? bmax)
120.                                     (set! ip ip1)
121.                                     (set! one? #t))
122.                                    (else
123.                                     (loop (+ ip1 1)))))
124.                             (else
125.                              (loop (+ ip1 1))))
126.                       (do ((i (+ m1 1) (+ i 1))) ((>= i m12))
127.                        (if (vector-ref l3 (- i (+ m1 1)))
128.                            (do ((k 0 (+ k 1))) ((= k (+ n 1)))
129.                             (matrix-set! a i k (- 0. (matrix-ref a i k)))))))))
130.                 (else (simp2) (if (zero? ip) (set! pass2? #f) (set! one? #t)))))
131.          (else (simp2) (if (zero? ip) (set! pass2? #f) (set! one? #t))))
132.         (if one?
133.             (begin
134.              (set! one? #f)
135.              (simp3 #t)
136.              (cond
137.               ((>= (vector-ref iposv (- ip 1)) (+ n (- m12 1)))
138.                (let loop ((k 0))
139.                 (cond
140.                  ((and (< k nl1) (not (= kp (vector-ref l1 k))))
141.                   (loop (+ k 1)))
142.                  (else
143.                   (set! nl1 (- nl1 1))
144.                   (do ((is k (+ is 1))) ((>= is nl1))
145.                    (vector-set! l1 is (vector-ref l1 (+ is 1))))
146.                   (matrix-set! a (+ m 1) kp (+ (matrix-ref a (+ m 1) kp) 1.0))
```

```
147.                    (do ((i 0 (+ i 1))) ((= i (+ m 2)))
148.                     (matrix-set! a i kp (- 0. (matrix-ref a i kp))))))))
149.                ((and (>= (vector-ref iposv (- ip 1)) (+ n m1))
150.                      (vector-ref l3 (- (vector-ref iposv (- ip 1)) (+ m1 n))))
151.                 (vector-set! l3 (- (vector-ref iposv (- ip 1)) (+ m1 n)) #f)
152.                 (matrix-set! a (+ m 1) kp (+ (matrix-ref a (+ m 1) kp) 1.0))
153.                 (do ((i 0 (+ i 1))) ((= i (+ m 2)))
154.                  (matrix-set! a i kp (- 0. (matrix-ref a i kp))))))
155.                (let ((t (vector-ref izrov (- kp 1))))
156.                 (vector-set! izrov (- kp 1) (vector-ref iposv (- ip 1)))
157.                 (vector-set! iposv (- ip 1) t))
158.                (loop))))))
159.   (and pass2?
160.        (let loop ()
161.         (simp1 0 #f)
162.         (cond
163.          ((positive? bmax)
164.           (simp2)
165.           (cond ((zero? ip) #t)
166.                 (else (simp3 #f)
167.                       (let ((t (vector-ref izrov (- kp 1))))
168.                        (vector-set! izrov (- kp 1) (vector-ref iposv (- ip 1)))
169.                        (vector-set! iposv (- ip 1) t))
170.                       (loop))))
171.          (else (list iposv izrov)))))))
172.
173. (define (test)
174.  (simplex (vector (vector 0.0 1.0 1.0 3.0 -0.5)
175.                   (vector 740.0 -1.0 0.0 -2.0 0.0)
176.                   (vector 0.0 0.0 -2.0 0.0 7.0)
177.                   (vector 0.5 0.0 -1.0 1.0 -2.0)
178.                   (vector 9.0 -1.0 -1.0 -1.0 -1.0)
179.                   (vector 0.0 0.0 0.0 0.0 0.0))
180.           2 1 1))
181.
182. (run-benchmark 100000 (test))
```

C.13 tak.scm

```
1. ;;; TAK -- A vanilla version of the TAKeuchi function.
2.
3. (define (tak x y z)
4.   (if (not (< y x))
5.       z
6.       (tak (tak (- x 1) y z)
7.            (tak (- y 1) z x)
8.            (tak (- z 1) x y))))
9.
10. (run-benchmark 1000 (tak 18 12 6))
```

C.14 takl.scm

```
1. ;;; TAKL -- The TAKeuchi function using lists as counters.
2.
3. (define (listn n)
4.   (if (= n 0)
5.     '()
6.     (cons n (listn (- n 1)))))
7.
8. (define l18 (listn 18))
9. (define l12 (listn 12))
10. (define  l6 (listn 6))
11.
12. (define (mas x y z)
13.   (if (not (shorterp y x))
14.       z
15.       (mas (mas (cdr x) y z)
16.            (mas (cdr y) z x)
17.            (mas (cdr z) x y))))
18.
19. (define (shorterp x y)
20.   (and (not (null? y))
21.        (or (null? x)
22.            (shorterp (cdr x)
23.                      (cdr y)))))
24.
25. (run-benchmark 100 (mas l18 l12 l6))
```