THÈSE

présentée à

L'UNIVERSITÉ PARIS 6

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS 6

Spécialité:

INFORMATIQUE

par

Manuel Serrano

Sujet de la thèse:

Vers une compilation portable *et* performante des langages fonctionnels

Soutenue le 9 Décembre 1994 devant le jury composé de:

MM.	Jean-François Perrot	Président
	Christian Queinnec	Directeur
	Jérôme Chailloux Patrick Sallé	Rapporteurs
	Emmanuel Chailloux Michel Lemaître Michel Mauny	Examinateurs

Remerciements

Je remercie Monsieur Jean-François Perrot d'avoir accepté de présider le jury de cette thèse.

Je remercie Messieurs Jérôme Chailloux, Patrick Sallé, Emmanuel Chailloux, Michel Lemaître et Michel Mauny de participer à ce jury et tout spécialement Messieurs Jérôme Chailloux et Patrick Sallé pour en être les rapporteurs.

Je remercie Monsieur Christian Queinnec, mon directeur de thèse, qui a dirigé mes travaux pendant trois ans. Ses encouragements, son soutien, son aide et ses conseils ont été très précieux. La liberté qu'il m'a laissé pendant ces trois années m'a permis d'aborder des sujets qui me tenaient particulièrement à cœur. Par ailleurs, je souhaite également le remercier pour la qualité de ses cours de DEA sur la sémantique des dialectes de Lisp qui ont rendu mon intérêt pour ces langages beaucoup plus fort.

*Je remercie tout particulièrement Monsieur Pierre Weis, chargé de recherches à l'*INRIA *pour l'aide inestimable qu'il m'a apporté tout au long de mes travaux. Sa patience, sa disponibilité, sa rigueur, et ses dons pédagogiques sont des modèles. Ses remarques sur le manuscrit de cette thèse ont largement contribué à améliorer sa qualité.*

Je remercie l'ensemble des membres du projet ICSLA*, car ils ont contribué à faire de l'*INRIA *un lieu de travail priviligié pour mes recherches. En particulier, je remercie Jean-Marie Geffroy pour nos échanges et nos discussions et Josy Baron car elle est comme une seconde mère pour tout le bâtiment 8.*

J'adresse également un grand merci à ma grand-mère (Claude), qui a, en un temps record, relu le manuscrit et corrigé les deux ou trois erreurs orthographiques qui, par inadvertance de ma part, s'étaient glissées çà et là.

Enfin, une dernière pensée pour Céline qui a supporté sans aucune impatience ni mauvaise humeur mes affres (assez fréquentes) de thésographe.

Table des matières

Présentation

Les langages fonctionnels modernes posent des problèmes inédits dans le domaine de la compilation car ils ont la volonté de réduire au minimum les mécanismes fondamentaux (application, affectation, conditionnelle, liaison valeur/variable). C'est à ce point vrai que plusieurs thèses ont déjà été consacrées à la compilation de Scheme et de ML. Citons par ordre chronologique Guy Steele [Ste78], David Kranz [Kra88] et dans la prolixe année 1991, Olin Shivers [Shi91a], Nitsan Séniak [Sén91] et enfin Emmanuel Chailloux [Cha91]. Chacune de ces cinq thèses traite le problème de façon différente. Guy Steele introduit le CPS et montre que le compilateur doit faire des efforts sur un nombre réduit de cas. David Kranz a consacré la sienne à décrire Orbit [KKR+86], un compilateur du dialecte T [RA82] et, plus précisément, à l'analyse de fermeture qui y est faite. Olin Shivers a exploré l'analyse de contrôle. Nitsan Séniak a eu une optique bien plus pédagogique, il a donné une méthodologie de compilation pour les langages fonctionnels. Emmanuel Chailloux, quant à lui, a brossé un panorama des différentes techniques utilisées pour compiler les langages fonctionnels et a décrit le fonctionnement de son compilateur, CeML. Fort de ces nombreuses expériences, la compilation des langages devient meilleure et les performances se rapprochent de celles des langages itératifs classiques.

Parallèlement à l'évolution des techniques de compilation, les ordinateurs progressent eux aussi. De génération en génération, les architectures sont bouleversées. L'évolution plus rapide des machines que du logiciel n'est pas sans poser de problèmes car l'espérance de vie d'un programme destiné à une machine particulière est devenue bien courte. Ce qui était moins vrai par le passé, est devenu aujourd'hui un problème crucial. Certains systèmes comme, par exemple Le-Lisp [CDD+86, Cha85] ou MIT-Scheme [H*al.*91, Roz84], ont souffert de l'arrivée des machines RISC. Leur modèle d'exécution très proche des processeurs d'alors (MC-68000), les rendent peu adaptés aux machines d'aujourd'hui. Il n'est plus admissible de consacrer un effort important au développement d'un programme si celui-ci doit devenir obsolète peu après sa date d'achèvement ! La portabilité d'un programme complexe, qui lui permettra de bénéficier du progrès des machines devient donc déterminante.

Malheureusement, il n'est pas aisé de concilier portabilité *et* efficacité. Souvent l'une est sacrifiée au profit de l'autre. Par exemple, les interprètes de code octet (*byte-code*) sont portables, car la partie écrite en code natif est petite mais ils ne sont pas très rapides et, à l'inverse, les compilateurs produisant directement du langage machine fonctionnent sur peu de machines (ou fonctionnent efficacement sur peu de machines). L'ambition de cette thèse est d'ouvrir la voie à une alliance entre compilation portable *et* compilation efficace.

La portabilité

La portabilité de notre travail se situe à deux niveaux. La plus évidente est que le compilateur que nous décrivons fonctionne facilement sur différents ordinateurs, en exploitant à chaque fois les capacités de la machine d'accueil (nous nommons ceci *portabilité d'exécution*). Le deuxième, plus subjectif, est que les techniques que nous exposons ne sont pas trop fortement liées à un langage particulier (nous désignons ce second niveau par *portabilité de compilation*). Elles s'appliquent à la famille des langages fonctionnels modernes.

La portabilité d'exécution

Il existe à notre avis deux méthodes pour obtenir la portabilité d'exécution. Nous avons déjà mentionné celle utilisant des interprètes de code octet. Elle ne convient pas à notre travail car elle ne permet pas (ou du moins pas à ce jour) d'obtenir de très bonnes performances. L'autre consiste à ne produire ni code octet

ni langage d'assemblage mais un programme écrit dans un langage évolué. Ce langage de haut niveau doit respecter les critères de portabilité que nous avons déjà énoncés : exister sur la quasi totalité des machines et exploiter correctement les capacités de chaque machine. Au regard de ces critères, le choix du langage n'est (malheureusement) pas difficile. Le candidat le plus sérieux est le langage C [ISO90] (parce qu'il permet une totale manipulation de la mémoire, parce qu'il existe sur toutes les machines, etc.). Choisir ce langage comme cible présente des avantages et des inconvénients :

- L'appel de fonction n'est pas toujours très bien traité dans les compilateurs C. Cela peut constituer un frein important aux performances.
- Les compilateurs C ne reconnaissent pas les cas de récursions terminales[1], alors que la spécification du langage Scheme impose le traitement itératif de toutes les récursions terminales !
- Le code C généré est souvent de taille imposante, or il semble que les compilateurs C ont plus été pensés pour compiler des programmes écrits à la main que des programmes générés. Il en résulte des temps de compilation très longs. Ceci n'est pas un réel inconvénient pour notre recherche, mais peut le devenir pour un potentiel utilisateur.
- Il est difficile en C de simuler des opérateurs de contrôle complexes comme, par exemple, les `catch/throw` munis de `unwind-protect`.
- Le code produit par les compilateurs C ne détecte ni les erreurs arithmétiques (division par 0, débordement des entiers, etc.) ni les débordements de pile.

mais aussi des avantages :

- Les compilateurs C modernes utilisent un arsenal très complet d'optimisations qui leur permet très souvent de générer du langage machine, au moins aussi efficace que celui qu'on aurait pu écrire à la main. Ceci nous a permis de nous consacrer presque exclusivement aux problèmes posés par les caractéristiques évoluées des langages fonctionnels, en supposant connu tout ce qui relève des langages algorithmiques classiques.
- Depuis sa normalisation ISO-C, C est devenu plus portable.
- Les technologies évoluent à très grande vitesse. Il est difficile de prévoir aujourd'hui ce que seront demain les ordinateurs. Compiler vers un langage proche des machines actuelles présente le risque d'être lié à la technologie contemporaine. Ainsi, certains compilateurs qui étaient très efficaces lors des machines CISC, le sont devenus beaucoup moins depuis l'apparition des processeurs RISC. On peut gager que le langage C a un avenir plus long que celui des ordinateurs actuels, le logiciel vivant plus longtemps que le matériel.
- Compiler vers C permet de bénéficier des outils de mise au point qui lui sont destinés. La commande `lint` permet une analyse du code C plus fine que les compilateurs eux-mêmes. Les logiciels `prof` et `gprof` autorisent de mesurer le temps d'exécutions de chaque partie d'un programme. Les débogueurs symboliques (`dbx`, `gdb`, ...) permettent des exécutions "pas à pas". Depuis peu, il existe des outils avec lesquels on peut analyser beaucoup d'erreurs d'allocation de mémoire, ou de références erronées (par exemple, `purify` ou `centerline`).

La portabilité de compilation

Chaque langage fonctionnel possède des caractéristiques qui compliquent ou améliorent sa compilation. Scheme ne possède pas de construction de boucle, aussi bien compiler les appels fonctionnels est déterminant. Il est typé dynamiquement ; les représentations des objets doivent donc permettre des tests rapides. ML autorise l'emploi de fonctions currifiées, pourtant le compilateur ne doit pas produire du code allouant des

[1] Ceci est à modérer par le fait que certains compilateurs tels que `gnu-cc`, génèrent du code qui ne consomme pas de pile pour les cas triviaux de récursions terminales.

fermetures à chaque appel fonctionnel ! Le filtrage étant omniprésent en ML, le compilateur doit faire un effort particulier pour bien le compiler. On pourrait citer d'autres exemples pour d'autres langages.

Bigloo, notre compilateur, est à l'origine un compilateur Scheme mais il offre des possibilités d'extensions qui lui permettent de compiler d'autres langages. À ce jour il peut compiler, en plus de Scheme, le langage Caml-light [WL93] et la couche objet MEROON [Que93]. En étudiant les performances des compilations obtenues par extension du compilateur, nous montrerons que les analyses faites lors de la compilation de Scheme possèdent un caractère général et qu'elles s'appliquent aussi aux autres langages applicatifs.

L'efficacité

En faisant le choix de C comme langage cible, nous ne sommes pas à l'abri de voir les efforts réalisés en début de compilation gâchés en aval (*back-end*) par une compilation mal adaptée. Par exemple, les compilateurs C ne mettent pas l'accent sur la compilation des appels de fonctions, car ils ne sont généralement pas très fréquents dans les programmes qu'ils traitent, alors qu'ils consacrent une part plus grande de leurs efforts dans des optimisations locales (comme les boucles). Les besoins de langages comme Scheme sont un peu opposés : les fonctions sont très nombreuses, c'est dans leur compilation qu'il faut fournir l'effort. La principale contribution de cette thèse est de donner toute une série de techniques d'optimisations de haut niveau (c'est-à-dire s'appliquant lors des premières passes de la compilation) qui permettent de combler le handicap d'une compilation avale pas très bien adaptée *mais* portable. Les efforts de compilation peuvent être divisés en deux catégories :

La compilation du contrôle La compilation du flot de contrôle des langages fonctionnels est particulièrement difficile parce que les fonctions sont des objets de première classe. Une grande partie des analyses de Bigloo est destinée à bien compiler ces fonctions. Le compilateur effectue plusieurs analyses de plus en plus précises pour déterminer quelles sont les fonctions qui nécessitent *réellement* une allocation et quelles sont celles qui peuvent s'en dispenser. Ainsi sur des critères de plus en plus fins, Bigloo parvient, dans la très grande majorité des cas, à ne pas allouer de mémoire pour les fonctions. Notre compilateur va même plus loin dans son effort de bonne compilation des fonctions. Dans la plupart des cas les invocations terminales des fonctions sont traduites par du code qui ne consomme aucune ressource (ni pile ni tas). Étant donné le choix du langage cible actuel (C) nous ne pouvons pas avoir toujours cette propriété mais les cas usuels et importants (boucles) la vérifient.

La compilation des données Il semble que plus les langages de programmations sont de haut niveau plus ils font abstraction des mécanismes d'allocation mémoire. Les langages fonctionnels vont assez loin dans cette direction puisqu'ils incluent la gestion implicite de mémoire à l'aide de glaneurs de cellules. Le programmeur ne se soucie plus de récupérer la mémoire, le système le fait pour lui. Dans le même ordre d'idée, puisque la mémoire est gérée automatiquement, il n'existe plus, dans ces langages, différentes classes d'allocations. Toutes sont, *a priori*, faites dans le tas. Par exemple, il n'existe pas dans ces langages de moyen d'exprimer qu'une allocation doit être faite dans la pile. Ceci est à notre avis une source d'inefficacité car, même si les glaneurs de cellules deviennent de plus en plus performants, les ordinateurs modernes n'en restent pas moins conçus pour fonctionner avec des piles. L'allocation en pile peut vraiment être très efficace. Bigloo supprime donc certaines allocations en tas, les remplaçant soit par des allocations en pile, soit par des allocations en registres.

Puisque les optimisations décrites sont toutes implantées dans notre compilateur, nous pourrons valider leur pertinence et leur efficacité en comparant les performances que nous atteignons avec celles des compilateurs concurrents.

Le langage de Bigloo

Jusqu'au chapitre 10, où nous montrerons que notre méthodologie de compilation s'applique parfaitement à ML, nous considérons que Scheme est le langage source de Bigloo. En fait, Bigloo compile un Scheme largement étendu [Ser94a]. La principale extension du point de vue de la compilation est que Bigloo possède

un langage de modules qui permet la compilation séparée. Nous ne décrirons pas ici ce langage mais nous préciserons quelques points importants.

- Notre but est d'obtenir un compilateur produisant de petits exécutables autonomes et performants. Nous nous plaçons donc en marge des habitudes de Lisp où le langage était étroitement lié à sa boucle d'interaction (*top-level*).
- Le langage de module de Bigloo donne des informations sur les variables du module et sur les exportations. Il permet par exemple l'introduction de constantes. Ceci a un incidence très forte sur la compilation, car Bigloo peut savoir qu'une variable désigne toujours une et une seule fonction.
- Grâce aux modules, Bigloo connaît toutes les variables concernées par une compilation (et quelquefois leurs types). Il peut donc signaler, lors de la compilation, des erreurs introduites par l'emploi de variables indéfinies ou des erreurs de types. Ceci rompt fortement avec la tradition de Lisp.

Par ailleurs, les programmes Scheme, que nous donnons dans ce document, utilisent abondamment le filtrage de Bigloo. Cette extension a été ajoutée par Jean-Marie Geffroy [QG92].

Cette thèse est consacrée aux performances d'un système portable. Tous les points du compilateur présentés seront en liaison avec les performances. Ainsi, nous négligerons certains aspects délicats de la compilation car ils ne s'inscrivent pas dans cette optique (par exemple, la relation entre l'interprète et le code compilé).

Le plan de la thèse

Cette thèse présente une partie des analyses et des optimisations que nous utilisons dans Bigloo, notre compilateur de langages fonctionnels. Nous avons essayé, quand cela était possible, d'isoler chaque type d'optimisation dans un chapitre à part. D'où, en théorie, la lecture de cette thèse n'a pas besoin d'être linéaire. Néanmoins nous n'avons pu éviter totalement les références croisées entre chapitres car il arrive qu'une analyse ou qu'une optimisation nécessite les résultats d'analyses réalisées préalablement.

- Le chapitre 1 présente les divers programmes tests que nous utilisons tout au long de ce document. Nous avons choisi de les décrire très tôt car nous en aurons besoin dès le deuxième chapitre pour comparer différentes solutions techniques.
- Le chapitre 2 contient une description de la bibliothèque d'exécution. Nous considérons qu'elle a une importance primordiale dans les performances globales d'un système.
- Le chapitre 3 présente l'architecture de notre compilateur. Elle permet au lecteur de situer les moments où ont lieu les analyses présentées ultérieurement.
- Le chapitre 4 est consacré à la présentation de l'interface externe. Cette interface n'est pas une simple extension ajoutée au langage source, c'est un mécanisme qui tient un rôle important dans la compilation du noyau du langage cible (par exemple, c'est l'interface qui assure une bonne compilation des appels aux primitives).
- Le chapitre 5 débute la description des optimisations. Nous commençons par les optimisations de plus haut niveau : les transformations source à source.
- Le chapitre 6 expose le mécanisme d'intégration de fonction (*inlining*) utilisé par notre compilateur. Ce chapitre décrit l'algorithme de décision d'intégration que nous avons conçu.
- Le chapitre 7 présente une analyse de flot de contrôle adaptée aux langages d'ordre supérieur. Celle-ci annote l'arbre de syntaxe abstraite pour permettre les optimisations présentées dans le chapitre 8.
- Le chapitre 8 décrit l'analyse de fermeture de notre compilateur. Cette analyse utilise les annotations produite par l'analyse du chapitre 7.
- Le chapitre 9 traite des optimisations classiques de flot de données adaptées au contexte de la compilation de langages fonctionnels.

- Le chapitre 10 décrit le compilateur ML que nous avons conçu comme une simple extension de notre compilateur Scheme. Les performances de ce compilateur ML apportent la preuve que nos optimisations ne sont pas uniquement pertinentes pour la compilation de Lisp.
- Le chapitre 11 contient une étude comparative des performances de différents compilateurs Lisp.

Chapitre 1

Les programmes de tests

Pourquoi commencer par un chapitre sur les mesures de performances? Habituellement il ne se trouve qu'en conclusion d'un document ! Nous avons fait le même choix que les auteurs de [KKR+86], mais pour une raison différente. Il nous faut comparer des solutions techniques à la compilation efficace des langages fonctionnels. Nous avons donc besoin d'instruments de contrôle.

Évaluer la qualité d'un compilateur est chose difficile. Plusieurs problèmes se posent :

- Avec l'importance croissante des mémoires caches dans les performances des processeurs, il est devenu très problématique de prédire l'efficacité d'un programme au seul examen de son code[1].

  Ainsi, la qualité d'un compilateur ne peut pas être évaluée par un simple examen du code produit, il est impératif de faire des mesures de temps d'exécution des codes produits.

- Les machines ont des comportements très chaotiques. Il est toujours frappant de constater que d'une exécution à l'autre, les temps d'exécutions peuvent varier de plus de 10 %. Il est difficile de trouver des explications à ce phénomène et, en particulier, le fait que les machines soient presque toujours connectées dans des réseaux, ne peut pas être la seule raison. Les auteurs de l'article [HBH93] donnent certaines des raisons des comportement étranges des machines équipées de processeurs Sparc.

- Certaines machines sont performantes pour certaines opérations et plus lentes pour d'autres. Ainsi, un compilateur porté sur plusieurs machines peut avoir de bons résultats sur un type de machine et de mauvais par ailleurs.

- Quels programmes mesurer? Que cherche-t-on à évaluer? Faut-il faire des mesures sur des petits programmes qui mettent en exergue un point précis de la compilation ou de la bibliothèque d'exécution (*runtime*)? Ou bien, au contraire, faut-il mesurer des programmes plus conséquents qui permettent d'évaluer les performances globales d'un compilateur en situation réelle.

Ce chapitre va exposer nos solutions et nos choix. Ils nous permettent de parvenir, malgré les difficultés énoncées, à une évaluation significative des différents compilateurs.

[1] Il nous faut relater le cas extrême que nous avons rencontré lors d'une de nos investigation : `scheme-to-c`, le compilateur de Joel Bartlett [Bar89b] possède une option de compilation qui permet d'activer ou de désactiver la production de tests de type et d'intervalle des programmes compilés. Sur un petit programme calculant des décimales, quelle ne fut pas notre surprise de constater que la version compilée avec les tests allait, sur machine Sparc 50 % plus vite que la version compilée sans ces mêmes tests ! Après examen des deux codes C produits; nous avons remarqué que la seule différence était que la première version (la plus rapide) contenait plus de tests et aurait dû logiquement être moins rapide. Nous n'avons pas d'explication sérieuse à apporter à ce résultat. Nous mentionnons seulement ce cas pour montrer la difficulté de faire des mesures

1.1 Les règles minimales

Afin d'éviter les principaux écueils de l'art de la mesure des performances, voici quelques règles que nous avons adoptées.

Règle N^o 1 : La dépendance vis à vis des machines Les machines actuelles possèdent naturellement chacune des caractéristiques particulières. Afin d'apporter plus de crédit à nos mesures et puisque nous prétendons que notre compilateur est bon "en général", tous les bancs d'essais seront mesurés sur deux types de machines.

Règle N^o 2 : Les temps moyens des bancs d'essais Nous pensons qu'un test, pour être significatif doit requérir un temps de calcul suffisamment grand. Arbitrairement, nous avons fixé cette limite à 10 secondes. Ainsi les coûts annexes que nous ne souhaitons pas mesurer (le temps de la fonction de calcul du temps, par exemple) sont rendus négligeables. Nous avons rejeté tous les tests plus courts. En particulier, nous avons refusé le principe qui consiste à répéter un test trop court. Nous refusons cette pratique car elle nous semble introduire une disparité entre les systèmes. En effet, puisque les différentes exécutions ne sont pas reliées entres elles, un glaneur de cellules (GC) copiant, lorsqu'il aura besoin de récupérer de la mémoire, se contentera de parcourir les structures allouées dans une précédente itération alors qu'un GC non copiant sera obligé de faire, pour ces mêmes structures un travail plus lourd. Certains tests n'ont pas du tout pour but de tester le GC. Donc, pour éviter d'avoir des temps parasités inutilement par le gestionnaire mémoire, nous excluons ces tests. Cette règle nous interdit presque la totalité des tests de Gabriel [Gab85]. En effet, ces programmes ont été écrits alors que les compilateurs étaient moins performants et les machines incomparablement moins rapides. Ainsi, par exemple, les temps de Bigloo pour le test `fread` sont de 0 seconde (temps système et temps `cpu`) ! Cet exemple montre bien clairement que ces tests sont aujourd'hui dépassés.

Règle N^o 3 : Le meilleur des temps Il est très surprenant de constater que d'une exécution à l'autre les performances peuvent varier de plus de 10 % et ce, même quand la machine n'est pas chargée. Ainsi, plutôt que de faire la moyenne des exécutions, nous avons trouvé plus juste de prendre la meilleure mesure obtenue sur une série de 3 exécutions (la même décision a été prise dans [DTM94]).

1.2 Les programmes de tests

Une fois ces règles minimales énoncées, se pose le problème du choix des programmes tests. Il existe des programmes presque unanimement utilisés pour relever des mesures de compilateurs Lisp, les programmes de Gabriel. Nous avons choisi de ne les utiliser que très parcimonieusement, car ils nous semblent présenter plusieurs inconvénients majeurs dûs principalement à leur âge (les plus jeunes datent de 1985). Ils ont été conçus pour tester les performances des systèmes Lisp. Ainsi, aucun n'utilise vraiment les possibilités et les styles de programmation des langages fonctionnels modernes. Aucun des tests de Gabriel ne mesure l'efficacité de l'allocation des fermetures, aucun n'utilise de filtrage. De plus, les compilateurs, et surtout les ordinateurs, évoluent beaucoup. Les programmes de Gabriel ne nécessitent maintenant que des temps de calculs trop courts.

Ainsi, voici les programmes que nous avons choisis (tous les codes sont dans l'annexe E) pour faire nos mesures.

Fib (13 lignes)

L'éternel numéro 1 au "hit parade" des tests. Ce programme implante de façon récursive la fonction de fibonacci. En fait, ce test ne présente que peu d'intérêt, il permet juste de fixer les idées sur l'ordre de grandeur des temps atteints par un compilateur. Ce programme ne manipule que des petits entiers, il n'y a pas de structure utilisateur allouée. Ce programme teste principalement la qualité de l'appel fonctionnel simple (l'appel fonctionnel où la fonction est une constante) et l'efficacité de la représentation des petits entiers et de ses primitives associées.

Fibs (12 lignes)

Ce programme implante la fonction de fibonacci en construisant des chaînes de caractères. Le résultat de la fonction est la longueur de la chaîne retournée. Ce programme teste les appels récursifs, l'allocation des chaînes de caractères et leur concaténation.

Tak (15 lignes)

Le "Poulidor" des tests, car presque toujours second derrière `fib`, ce test lui ressemble d'ailleurs beaucoup. Il mesure la performance des appels récursifs. Il ne fait aucune allocation dans le tas.

Queens (126 lignes)

Ce programme est un programme ML traduit en Scheme. Il s'agit de la résolution du problème des placements de 10 reines sur un échiquier de 10x10 cases. Ce programme est presque clos (il n'utilise presque pas de fonctions de la bibliothèque). Il manipule beaucoup de listes (la version compilée par Bigloo en alloue 40 mégaoctet) et alloue plusieurs fermetures. Ce programme teste la qualité de la représentation des petits entiers, la compilation des formes booléennes, la manipulation des listes et, dans une moindre mesure, l'allocation des fermetures.

Dens (277 lignes)

Ce programme est l'implantation en Scheme d'une sémantique dénotationnelle [Sto77] du langage de filtres de Christian Queinnec [Que90]. Il est donc très fonctionnel, il alloue énormément de fermetures. Ce programme teste le comportement des compilateurs pour des programmes écrits utilisant un style très *haute technologie*.

Leval (512 lignes)

Ce programme est un petit compilateur du langage Scheme. Les programmes compilés sont codés par des fermetures. Il s'agit d'un programme allouant principalement listes et fermetures. Ce programme teste donc l'allocation et l'application des fermetures.

Beval (544 lignes)

`Beval` est le même programme que `Leval` mais les programmes compilés sont représentés ici par des structures de données à base de paires et de vecteurs. Ce programme n'alloue donc presque pas de fermetures. Il teste principalement l'allocation et l'accès aux structures de données ainsi que la compilation des formes `case`.

Bague (100 lignes)

Ce programme résout de façon récursive le jeu du baguenaudier. Il teste l'accès dans les vecteurs, l'arithmétique entière et les appels fonctionnels récursifs. Il ne teste pas du tout le glaneur de cellule car il n'alloue qu'un seul petit vecteur, pas de listes et pas de fermetures.

Sievev (50 lignes)

Ce programme calcule le nombre d'entiers premiers par la méthode du crible d'Érastothene. Il ne fait presque pas d'allocations car il utilise un vecteur plutôt que des listes. Ce programme teste la manipulation des vecteurs et l'arithmétique entière.

Church (58 lignes)

Ce tout petit programme implante les entiers de Church. Il est hautement fonctionnel. Peu d'allocations sont faites (seules 17 fermetures sont allouées), il teste la compilation de flots de contrôle très dynamiques et la qualité des appels calculés.

Conform (569 lignes)

Ce programme mélange plusieurs traits de Scheme. Il utilise des vecteurs et des listes. Il contient de nombreuses fonctions locales. Il utilise un peu d'ordre supérieur (principalement les fonctions `map` et `for-each`). Nous l'utilisons plus pour évaluer les performances d'ensemble des compilateurs que pour cerner l'efficacité de quelques points précis.

Earley (661 lignes)

Cette implantation de l'algorithme d'analyse syntaxique d'Earley a été réalisée par M. Feeley. Ce programme nous sert à mesurer la compilation des fonctions locales imbriquées. La structure de données la plus souvent utilisée est la liste. Ce programme nous servira donc à tester la compilation du contrôle et la manipulation des paires.

Peval (632 lignes)

Ce petit évaluateur partiel (écrit par M. Feeley) nous sert pour tester la compilation du contrôle. Il utilise de nombreuses petites fonctions locales utilisant des listes et contient beaucoup d'expressions conditionnelles imbriquées. Par ailleurs, les listes sont utilisées pour représenter les programmes partiellement évalués. La manipulation des listes y joue donc un rôle important.

Pp (527 lignes)

Ce programme est un *pretty-printer* Scheme. Il nous sert à tester les *entrées/sorties* (**Pp** lit sa donnée dans un fichier source et produit son résultat dans un fichier cible). Les programmes étant représentés par des listes, la manipulation des paires a donc une incidence assez forte sur performances. Par ailleurs, ce programme manipule beaucoup de chaînes de caractères.

Pseudoknot (3569 lignes)

Ce programme effectue des calculs flottants. Il est décrit en détail dans l'article [HFA+94]. Il teste l'arithmétique générique (donc le glaneur de cellules (*GC*) pour les systèmes qui allouent les nombres flottants) et la compilation des constantes.

Bigloo (33000 lignes)

Notre compilateur. En effet, puisque Bigloo est autogène (il est compilé par lui-même), les temps de compilations par Bigloo pourront être interprétés comme étant des tests de taille réelle. Le code du compilateur fait 33000 lignes de Scheme qui utilisent presque tout le langage (structure de données en tous genres, gestion d'entrées/sorties, ...). Bien sûr, mesurer les temps de compilation ne sera pertinent que pour comparer différentes versions de Bigloo.

1.3 Les machines

Toutes nos mesures vont être faites sur deux types de machines, des machines à base de processeurs Sparc [Sun87] et des machines à base de processeurs Mips [KH92]. Ce choix repose sur le désir de se servir d'ordinateurs modernes et fréquemment utilisés. Nous n'avons pas pris en compte les qualités de ces machines pour les choisir.

Le premier ordinateur est un serveur SUN `sparc 4/670` (équivalent à un `sparc 2`), disposant de 64 MégaOctet de mémoire centrale, tournant sous le système SunOs 4.1.3. Le SPECint92 de cette machine vaut 21.8. Le deuxième est une station de travail DEC 5000/200 (processeur `Mips R3000`), disposant de 32 MégaOctet de mémoire centrale, tournant sous le système Ultix 4.1. Son SPECint92 est 22.4. Nous avons choisi ces deux machines car elles appartiennent à deux catégories de machines RISC différentes qui nous semblent être représentatives des machines actuelles et parce qu'elles ont des performances proches.

Chapitre 2

La bibliothèque d'exécution

Habituellement ce chapitre est placé après la description du compilateur. En fait, nous pensons que le compilateur est plus dépendant de la bibliothèque d'exécution (*runtime*) que le contraire. Par exemple, le typage dynamique impose des traitements au compilateur. Ainsi, nous trouvons plus logique d'expliquer dans un premier temps la bibliothèque d'exécution, avant de pénétrer dans les méandres de la compilation.

Une bonne bibliothèque d'exécution est capitale pour l'obtention de code performant. Nous allons expliquer ici comment sont représentés les objets de Bigloo, comment sont implantées ses fonctions et quels sont ses protocoles d'appels. Nous allons montrer comment les fonctions difficiles de la bibliothèque Scheme sont implantées (`call/cc` par exemple). Nous montrerons que dans tous les choix que nous avons fait, joindre la portabilité à l'efficacité a toujours été au centre de nos préoccupations. Nous conclurons le chapitre par un paragraphe discutant des avantages et des inconvénients du choix de C comme langage cible.

Mais, à tout seigneur tout honneur, commençons ce chapitre par le gestionnaire mémoire.

2.1 Le glaneur de cellules

Le choix du glaneur de cellules (*GC*) impose des obligations sur la conception du compilateur, et réciproquement, le choix d'un schéma de compilation entraîne des obligations pour le *GC*. Puisque nous avons voulu combattre l'idée que les performances d'un compilateur sont celles de son *GC*, nous avons choisi la deuxième possibilité. Nous avons conçu et réalisé Bigloo comme si le *GC* n'existait pas ou comme s'il ne présentait aucune contrainte. Nous avons concentré une grande part de notre attention sur des optimisations faites pendant la compilation et nous avons délibérément refusé de nous soucier des performances du *GC*.

Dans sa version actuelle Bigloo produit du code C orthodoxe[1], c'est à dire du code utilisant le contrôle du langage C (sa pile, son appel fonctionnel, etc.). Ce choix pose des restrictions très fortes sur le *GC* puisqu'il doit explorer les zones mémoires de la bibliothèque d'exécution C et n'est pas autorisé à en déplacer les objets. Un tel *GC* est dit conservatif à racines ambiguës. Ces *GC* ne sont pas sûrs ; il peut se produire que des optimisations des compilateurs C conduisent à des ramassages intempestifs de cellules. Dans la pratique nous n'avons jamais rencontré ces problèmes[2]. Plusieurs *GC* satisfaisant les contraintes ennoncées sont disponibles dans le domaine public. Citons les *GC* de Joel Bartlett [Bar88], Vincent Delacour [Del91] et Hans Boehm [Boe91]. Principalement pour des raisons de commodités la version courante de Bigloo utilise celui de Boehm. Ce *GC* est dit de type *mark & sweep*, c'est-à-dire qu'il s'agit d'un *GC* laissant tous les objets en place et donc ne compactant pas les données. Cette caractéristique est très importante car les performances des programmes s'exécutant sur les machines actuelles sont liées à la présence du programme et des données dans la mémoire *cache*. A priori des *GC* non copiants entraîneront des performances moins bonnes que les *GC* copiants. La réalité est plus nuancée et en fait, chaque catégorie semble apporter son lot d'avantages et d'inconvénients [Zor90].

[1] Nous mettons dans la section 2.6.2, en opposition, le code C *orthodoxe* de Bigloo et le code C *hétérodoxe* de certains compilateurs comme le compilateur sml2c [TAL91]

[2] E. Chailloux propose même une solution pour rendre le *GC* parfaitement sûr [Cha92a] : maintenir une pile annexe de racines. S'il devient fréquent que les compilateurs réalisent des optimisations dangeureuses nous pourrons utiliser sa technique.

Nous ne nions pas du tout l'impact des *GC* dans les performances d'un système. En particulier nous avons une conscience aiguë de l'amélioration qu'apportent les *GC* dits à générations [LH83]. Certains systèmes parviennent même à obtenir des performances honorables par la simple excellence de leur *GC*. Mais nous prétendons que les performances du code produit par Bigloo ne sont pas aussi fortement conditionnées par le *GC*, car notre compilateur n'alloue pas de mémoire dans le tas pour le contrôle. Le *GC* n'est utilisé que pour gérer les structures de données du programme compilé.

Il est communément reconnu que les langages fonctionnels sont difficiles à implanter. Les raisons invoquées sont toujours centrées sur le même argument : *ces langages possèdent des traits de haut niveau qui facilitent la tâche du programmeur car ils lui permettent un plus grand degré d'abstraction mais, en contre partie, ils éloignent ces langages du modèle de fonctionnement des ordinateurs actuels.* Parmi ces caractéristiques, l'allocation implicite de mémoire est souvent incriminée. Puisque le but avoué de Bigloo est de montrer qu'un compilateur Scheme moderne peut être une alternative valable au langage C, cette section va être consacrée à l'étude d'une comparaison entre notre *GC* et l'allocation explicite à l'aide des fonctions C *malloc* et *free*. Pour cela, nous avons réalisé une expérience qui évalue le coût de l'allocation implicite par rapport à l'allocation explicite. En premier lieu décrivons brièvement le *GC* de Bigloo.

2.1.1 Le *GC* de Boehm

Initialement, pour sa simplicité d'intégration, nous avons choisi pour Bigloo le *GC* de Boehm. Ce *GC* a été conçu pour pouvoir remplacer la fonction `malloc` de la bibliothèque C. Il ne nécessite (en première utilisation) aucune collaboration de la part du programme qui l'emploie. Il offre une fonction `GC_malloc` qui, du point de vue de l'utilisateur, est semblable à la fonction `malloc`. Il est porté sur presque toutes les machines fonctionnant sous Unix. La version 1.6 de Bigloo utilise la version 2.6 du *GC* de Boehm.

Puisque le glaneur ne nécessite pas que les pointeurs soient étiquetés, il ne peut pas garantir la récupération de tous les objets inaccessibles. Néanmoins, l'expérience montre qu'en pratique le taux de récupération est très grand. Ce *GC* trouve ses racines dans la pile et les variables de C. Les zones mémoires allouées au moyen de `malloc` ne sont, par défaut, pas considérées comme faisant partie de l'ensemble des racines.

2.1.2 Remplacement du *GC* par de l'allocation explicite

Afin de cerner le coût d'un *GC*, nous avons réalisé l'expérience qui consiste à compiler une version de Bigloo qui utilise une gestion explicite des allocations et à la comparer avec la version standard. Puisque Bigloo a été en partie conçu pour pouvoir réaliser ce genre d'expérience, les modifications à apporter à la version standard sont mineures. Pour obtenir la nouvelle version, il suffit de remplacer tous les appels à la fonction d'allocation du *GC* par un appel à une nouvelle fonction qui va se charger d'allouer la mémoire dans le tas. Plusieurs stratégies peuvent être mise en œuvre :

- Au lancement d'une application, allouer une grande zone initiale. Chaque allocation consistera dans le cas général à incrémenter un pointeur. Lorsque cette zone sera pleine, on en allouera une nouvelle et ainsi de suite. La figure 2.1 contient le code d'initialisation de la zone et la fonction d'allocation. La variable globale `nb_octet_allocated` (lignes *3* et *36*) mémorise le nombre d'octets alloués. Elle est uniquement utile pour l'affichage de statistiques concernant les allocations. Par ailleurs, il faut remarquer que l'allocation de ligne *28* n'est pas testée ! Nous nous le permettons car il ne s'agit ici que d'une maquette d'allocateur servant à une expérience.

- On n'alloue pas de grande zone mais pour chaque appel à notre fonction d'allocation, on invoque la fonction de la bibliothèque C `malloc` qui alloue la petite zone voulue.

Cette seconde stratégie serait lourdement pénalisée par les piètres performances de la fonction `malloc`. En effet, de nombreux programmes de test construisent beaucoup de listes et donc de paires. Ces paires sont des petits objets (8 octets). Ces programmes vont donc invoquer très souvent la fonction d'allocation et on risque de ne plus mesurer que les performances de cette fonction. Afin d'être plus objectif, nous allons comparer les temps du *GC* avec les temps de la première stratégie sans `GC_malloc`.

Pour ces deux versions de Bigloo (version standard, version utilisant le déplacement de pointeur dans `heap_alloc`), nous allons mesurer les temps de compilation et les temps d'exécution de quelques programmes.

```
 1:  long init_heap( long octet )
 2:  {
 3:     nb_octet_allocated = 0;
 4:
 5:     heap_hd = malloc( octet );
 6:
 7:     heap_tl = heap_hd + octet - 1;
 8:     bzero( heap_hd, octet );
 9:
10:     return 1;
11:  }
12:
13:  obj_t heap_alloc( long octet )
14:  {
15:     void *aux = heap_hd;
16:
17:     heap_hd += octet;
18:
19:     if( (long)heap_hd & 7 )
20:        heap_hd += (8 - ((long)heap_hd & 7));
21:
22:     if( heap_hd > heap_tl )
23:     {
24:        long size = (octet < ADD_OCTET_SIZE) ?
25:                       ADD_OCTET_SIZE :
26:                       octet * 2;
27:
28:        heap_hd = malloc( size );
29:
30:        heap_tl = heap_hd + size - 1;
31:        bzero( heap_hd, size );
32:
33:        return heap_alloc( octet );
34:     }
35:
36:     nb_octet_allocated += octet;
37:
38:     return (obj_t)aux;
39:  }
```

Programme 2.1: allocation explicite par déplacement de pointeurs

Le temps de compilation est intéressant car comme Bigloo est autogène (*bootstrapped*), le temps de compilation est en fait un test *grandeur nature*. Néanmoins les temps de compilation ont tendance à être un peu courts pour être vraiment révélateurs.

Nous avons choisi cinq programmes de test très différents. Ils recouvrent l'ensemble des allocations qui peuvent être faites (listes, fermetures et vecteurs et chaîne de caractères).

fib Ce programme n'alloue aucune structure de donnée. Il indique que le code produit par Bigloo n'alloue pas de mémoire dans le tas pour le contrôle.

queens Allocation de listes.

dens Allocation de fermetures.

beval Allocation de vecteurs.

fibs Allocation de chaînes de caractères. Ceci est intéressant pour notre test car les chaînes sont des zones mémoires ne contenant pas de pointeurs.

Le tableau suivant donne pour chacun de ces programmes la taille des zones mémoires allouées pour compiler et pour exécuter les tests.

	fib	**queens**	**dens**	**beval**	**fibs**
compilation	1437 k	2731 k	23736 k	17907 k	1502 k
exécution	16 k	15816 k	934 k	3157 k	27520 k

Nous avons choisi pour les deux versions la configuration qui avait les meilleurs résultats. Pour la version avec *GC*, le tas initial est de 4 Mo alors que pour la version sans, le tas initial est vide.

Les tableaux ci-dessous donnent pour chacun des cinq programmes de tests, le temps de compilation (stoppée après la production de code C) et le temps d'exécution.

Tout d'abord, voici les temps de compilation :

	fib	**queens**	**dens**	**beval**	**fibs**
avec *GC*	1.9 s	2.9 s	25.2 s	15.5 s	2.2 s
sans *GC*	1.7 s	2.6 s	23.4 s	14.9 s	1.8 s
sans/avec	10 %	10 %	7 %	3 %	18 %

Les temps de compilation sur Sparc.

	fib	**queens**	**dens**	**beval**	**fibs**
avec *GC*	1.3 s	1.8 s	16.4 s	10.0 s	1.3 s
sans *GC*	1.2 s	1.7 s	14.2 s	8.0 s	1.2 s
sans/avec	1 %	5 %	13 %	20 %	1 %

Les temps de compilation sur Mips.

Et les temps d'exécutions :

	fib	**queens**	**dens**	**beval**	**fibs**
avec *GC*	10.4 s	16.0 s	28.4 s	11.0 s	10.2 s
sans *GC*	10.4 s	17.1 s	28.8 s	9.6 s	26.9 s
sans/avec	0 %	-6 %	1 %	12 %	-164 %

Les temps d'exécution sur Sparc.

	fib	**queens**	**dens**	**beval**	**fibs**
avec *GC*	12.7 s	8.6 s	9.0 s	9.6 s	9.7 s
sans *GC*	12.7 s	8.2 s	9.0 s	9.6 s	12.0 s
sans/avec	0 %	4 %	0 %	0 %	-23 %

Les temps d'exécution sur Mips.

Avant de tirer les conclusions de cette expérience, faisons quelques remarques :

1. Ces mesures sont difficiles à interpréter car les temps varient beaucoup d'un test à l'autre et surtout, le rapport entre la version avec *GC* et la version sans, est très différent sur les deux types de machines testées.

2. Le rapport entre les deux versions donne l'avantage à celle qui utilise une gestion mémoire explicite (sans *GC*) pour les temps de compilation par rapport aux temps d'exécution (rapports moyens de 10 % et 8 % en temps de compilation contre des rapports de -31 % et -4 % en temps d'exécution).

3. Le rapport entre les deux versions n'est pas uniquement lié à la taille totale allouée car la compilation du test **beval** alloue quasiment autant que l'exécution du test **queens** et pourtant le rapport est très différent sur Sparc comme sur Mips. Le rapport varie en fonction de la localité des données. Ainsi pour les temps de compilation, la localité est assez importante car le compilateur alloue principalement passe par passe, alors que pour les exécutions, la localité est beaucoup moins constante, elle varie d'un test à l'autre.

Malgré les difficultés que nous avons à expliquer les comportements des deux versions, nous pouvons tout de même tirer une conclusion majeure. Le but de cette expérience est de cerner le coût de la gestion mémoire implicite : il dépend des machines et semble être de l'ordre de 10 %. Cet écart sans être négligeable, n'est pas un gouffre. L'utilisation de *GC* n'est donc pas la raison pour laquelle les langages fonctionnels sont moins performants que les langages itératifs. S'ils le sont, c'est peut-être parce que souvent le style fonctionnel requiert beaucoup plus d'allocations que le style itératif.

Remarque : Il ne faudrait pas essayer de tirer plus de conclusions de cette expérience car elle présente quelques faiblesses. Par exemple, afin de vraiment mesurer le coût réel de la gestion mémoire implicite, il aurait fallu modifier plus profondément notre compilateur pour que nous puissions, dans la version à gestion explicite, libérer de la mémoire. Pour cela, il aurait fallu dans la bibliothèque d'exécution avoir une version plus complexe de l'allocateur et surtout, que nous modifions nos programmes sources pour y inclure des ordres de libération. Ces modifications auraient probablement augmenté les temps d'exécution de la version sans GC. Nous n'avons donc mesuré qu'une approximation optimiste (défavorable aux GCs) de la version à gestion explicite. De plus, notre expérience n'est pas assez générale pour permettre de tirer des conclusions sur les GC en général. En effet, nous n'avons testé que l'allocateur de Boehm qui n'est pas forcément celui qui permet d'obtenir les meilleures performances. D'après des mesures que nous avons faites dans un autre contexte, il semblerait que l'allocateur de Vincent Delacour soit légèrement plus performant que celui utilisé ici. Mais comme les performances ne semblent pas très différentes, nous ne pensons pas que cela présente

un défaut majeur. Et enfin, nous ne sommes capables ici de ne mesurer que des GCs à racines ambiguës examinant la pile d'exécution de C et c'est peut-être là que se trouve le principal défaut de la compilation vers C orthodoxe. En effet, les allocateurs modernes (à générations) sont beaucoup plus performants, mais interdits par la méthode. Pour pouvoir les mesurer, il nous faudrait réécrire les dernières passes du compilateur Bigloo pour qu'il génère du langage machine ou du code octet.

2.1.3 La part du *GC*

Nous avons expliqué dans le préambule de cette section que Bigloo a été conçu sans préoccupation à l'égard du *GC*. Le chapitre 11 montrant que Bigloo se compare favorablement aux autres compilateurs prouve que cette approche ne nous a pas été fatale. Néanmoins, il semble que la très bonne qualité du code produit par Bigloo est un peu dévalorisée par les performances moyennes de son *GC*. Pour quantifier cela, nous avons étudié sur plusieurs tests quelle est la part du temps d'exécution utilisée par le *GC* et quelle est la part du temps utilisée par le reste du programme. Pour cela nous allons nous servir, à l'exception de **fib** qui a été remplacé par `leval`, des programmes de la section précédente. Il s'agit des programmes **queens**, **dens**, **beval** et **fibs**.

	queens	**dens**	**beval**	**leval**	**fibs**
allocation	30 %	2.9 %	6.4 %	9.4 %	35 %
récupération	20 %	0.5 %	0.7 %	1.2 %	6 %
cumul	50 %	3.4 %	7.1 %	10.6 %	41 %

Le pourcentage du temps *cpu* requis par l'allocateur sur Mips

La première ligne de ce tableau indique le pourcentage du temps passé lors des exécutions pour réaliser les allocations. Nous avons inclu pour ces mesures les temps de création des paires avec les remplissages des `car` et des `cdr`. La deuxième ligne du tableau indique le pourcentage du temps passé lors de la récupération (lors des déclenchements du *GC*).

Pour faire ces mesures, le choix des programmes tests est particulièrement crucial ! Faut-il utiliser des programmes qui n'allouent pas (comme **fib**)? Cela n'a bien sûr pas d'intérêt pour tester le temps passé dans le *GC*, mais a le mérite de prouver que le compilateur n'alloue pas pour le contrôle. Des programmes qui ne font qu'allouer (comme **fibs** ou **queens**)? Ceux-ci auront une grande part de leur temps d'exécution dans le *GC* mais correspondent-ils à une réalité? Écrit-on réellement des programmes qui allouent 27 Mega pour deux lignes de codes? Ceux qui nous semblent les plus significatifs pour cette expérience sont ceux de taille réelle. C'est pourquoi nous pensons ici que les deux programmes les plus intéressants sont les deux évaluateurs (**leval** et **beval**). Sur ces deux programmes, on s'aperçoit que les temps de *GC* sont très raisonnables (moins de 1 %), en revanche les temps d'allocations sont assez importants. Cette tendance semble être d'ailleurs vraie pour tous les tests (excepté **queens** qui ne semble tester que l'allocation et la récupération). Ainsi donc la lacune de notre *GC* semble être sa faible vitesse d'allocation.

2.2 La représentation des objets de la bibliothèque d'exécution

Le choix de la représentation des objets est cruciale pour les performances du système. Une représentation astucieuse peut très sensiblement améliorer les performances d'ensemble. Cette section contient la description des choix que nous avons fait pour Bigloo. Voici en premier lieu, les contraintes imposées par Scheme qui sont parfois différentes des autres langages fonctionnels comme Lisp ou ML[3].

- Scheme possède deux types fonctionnels. Les fonctions à arité fixe et les fonctions à arité variable. La bibliothèque d'exécution doit donc posséder elle aussi deux types de fonctions ou bien n'avoir qu'un seul type mais suffisamment général pour pouvoir servir aux deux sortes de fonctions de Scheme.
- Scheme possède un type booléen mais tous les objets peuvent être utilisés pour faire des tests. En position de test, seul l'objet noté `#f` doit être considéré comme faux. Tous les autres sont vrais.

[3] Les contraintes que nous mentionnons ici nous sont seulement imposées par Scheme. Notre *GC* n'impose aucune représentation des objets, en particulier, il n'exige aucun *bit* de marquage.

- Pour finir, Scheme est un dialecte de Lisp [IEE91], donc typé dynamiquement. Cela signifie que les types ne sont pas vérifiés pendant la compilation mais pendant l'exécution. Si cela présente l'évident inconvénient de ralentir les exécutions (car il faut faire des tests) cela permet aussi d'avoir un plus grand pouvoir d'expression (certains programmes courants comme les macro-expanseurs utilisant une technique nommée EPS [DFH86] ne sont pas typables par les typeurs actuels). Cette caractéristique exige des contraintes très fortes car elle impose au système qu'il reconnaisse le type de n'importe quel d'objet à l'exécution.

Nous n'allons pas nous livrer à une taxinomie complète des représentations des langages typés dynamiquement car une étude approfondie des différentes solutions et de leur coût peut être trouvée dans [Gud93] et dans les deux articles jumeaux [Ste91, SH87]. Nous n'allons pas non plus tenter d'évaluer dans ce chapitre le coût du typage dynamique car c'est un problème qui n'est pas uniquement dépendant de la représentation des objets. En effet le compilateur réalise plusieurs analyses pour réduire les tests dynamiques obligatoires.

Contentons nous ici d'examiner le codage des données de Bigloo.

2.2.1 Les données de Bigloo

Toutes les machines actuelles exigent des pointeurs alignés sur au moins 4 octets (sur Sparc les compilateurs C utilisent même des pointeurs alignés sur 8 octets). Ainsi, les deux bits de poids faible sont exploitables pour mémoriser des informations de type. L'utilisation des ces bits permet donc de distinguer quatre sortes d'objets. Leur combinaison qui permet de reconnaître les différentes catégories est appelée étiquette (*tag*). Pour Bigloo nous avons fait le choix suivant :

1. La première configuration distingue les pointeurs. Nous nommons l'étiquette `TAG_STRUCT`. Le *GC* de Boehm n'impose pas que `TAG_STRUCT` soit `0x00`. Il est possible de prévenir le *GC* que les pointeurs sont déplacés par rapport aux alignements standard.

2. La deuxième configuration distingue les petits entiers (*fixnum*). Nous nommerons cette étiquette `TAG_INT`.

3. La troisième configuration (étiquette `TAG_CNST`) distingue les constantes en tout genre (`#t`, `#f`, `()`, ...).

4. Enfin, historiquement, un système Lisp se devant de manipuler très efficacement les listes, nous avons décidé de leur consacrer la dernière configuration (étiquette `TAG_PAIR`).

Le Programme 2.2 contient le code principal pour mettre en place (*étiquetage*) ou retirer (*désétiquetage*) les étiquettes pour des objets sans alignement (i.e. des objets qui ne sont donc pas des pointeurs). Examinons, cas par cas, les différents codages.

```
#define TAG_SHIFT     PTR_ALIGNMENT
#define TAG_MASK      ((1 << PTR_ALIGNMENT) - 1)

#define TAG( val, shift, tag )   ((long)(((long)(val) << shift) | tag))
#define UNTAG( val, shift, tag ) ((long)((long)(val) >> shift))
```

Programme 2.2: L'étiquetage/désétiquetage générique

La valeur de `PTR_ALIGNMENT` est dépendant de la taille des pointeurs. Sur les machines 32 bits elle est égale à 2, sur les machines 64 bits, elle est égale à 3.

2.2.2 Les petits entiers

Le codage par des valeurs immédiates des entiers implique qu'ils aient au maximum la taille des pointeurs de la machine moins 2 bits. Sur les Sparc comme sur les Mips ils ont donc une longueur de 30 bits. Le programme 2.3 contient le code qui convertit des petits entiers C et des petits entiers Bigloo. Notons ici la convention que les macros traduisant des objets C en objet Bigloo sont préfixées de la majuscule `B` alors que les opérations inverses (de Bigloo vers C) sont préfixées de `C`.

```
#define BINT( i )          (obj_t)TAG( i, TAG_SHIFT, TAG_INT )
#define CINT( i )          (long)UNTAG( i, TAG_SHIFT, TAG_INT )
#define INTEGERP( o )      ((((long)o) & TAG_MASK) == TAG_INT)
```

Programme 2.3: L'étiquetage/désétiquetage des entiers

Nous n'avons pas encore précisé quelles sont les valeurs des différentes étiquettes. Si les manipulations des autres objets sont indépendants du choix de ces étiquettes, en revanche les petits entiers en dépendent fortement. Il y a deux possibilité pour les entiers : `TAG_INT` est égal à `0` ou pas. La première solution est plus simple, car les opérations arithmétiques sont plus facile à coder. Le Programme 2.4 contient les deux implantations possibles en fonction de la valeur de `TAG_INT`.

```
#if( !TAG_INT )
#   define ADD_I( a, b ) ((obj_t)((long)( a ) + (long)( b )))
#   define SUB_I( a, b ) ((obj_t)((long)( a ) - (long)( b )))
#   define MUL_I( a, b ) ((obj_t)((CINT( a ) * (long)b)))
#   define DIV_I( a, b ) (BINT( (CINT( a )    / CINT( b )) ))
#else
#   define ADD_I( a, b ) ((obj_t)(((long)( a ) - TAG_INT) + (long)( b )))
#   define SUB_I( a, b ) ((obj_t)(((long)( a ) - (long)( b )) | TAG_INT))
#   define MUL_I( a, b ) (BINT( (CINT( a )    * CINT( b )) ))
#   define DIV_I( a, b ) (BINT( (CINT( a )    / CINT( b )) ))
#endif
```

Programme 2.4: Les opérations arithmétiques

L'impression, largement répandue, que fixer `TAG_INT` à `0` apportera de meilleures performances semble confortée par les codes donnés dans le programme 2.4. S'il est vrai qu'en faisant ce choix les opérations arithmétiques sont plus rapides, les performances d'ensemble restent cependant très proches. Cela s'explique simplement : les surcoûts pour l'addition et la soustraction sont seulement d'une opération logique (soit 1 cycle) et, bien souvent, un des deux arguments d'une opération arithmétique est une constante sur laquelle il est possible de faire dès la compilation tous les calculs d'étiquette. Le code du programme 2.5 montre les versions de l'addition et de la soustraction quand le second argument est constant.

```
#if( !TAG_INT )
#   define ADD_I_PTAG( a, b ) (ADD_I( a, b ))
#   define SUB_I_PTAG( a, b ) (SUB_I( a, b ))
#   define PSUB_TAG( a ) BINT( a )
#   define PADD_TAG( a ) BINT( a )
#else
#   define ADD_I_PTAG( a, b ) ((obj_t)((long)(a) + (long)(b)))
#   define SUB_I_PTAG( a, b ) ((obj_t)((long)(a) - (long)(b)))
#   define PADD_TAG( a ) ((obj_t)((long)BINT( a ) + (long)TAG_INT))
#   define PSUB_TAG( a ) ((obj_t)((long)BINT( a ) - (long)TAG_INT))
#endif
```

Programme 2.5: Les opérations arithmétiques pré-étiquetées

Afin d'estimer précisément le coût du choix de `TAG_INT` non nul nous allons comparer les temps de compilation et les temps d'exécution avec deux versions de Bigloo : l'une utilisant `00` et l'autre `01`. Pour cela nous allons utiliser quatre de nos tests :

fib Ce programme fait des additions entières sur des arguments qui ne sont pas tous constants.

tak Contrairement a `fib`, les opérations entières ne concernent que des constantes.

bague Ce programme manipule des vecteurs et fait des opérations arithmétiques entières. L'efficacité de l'accès aux vecteurs est conditionnée par une bonne représentation des petits entiers. Avoir des entiers

décalés de deux bits sur la gauche permet d'avoir un accès plus efficace. Ceci est nettement visible dans le codage de la fonction Scheme `vector-ref`:

```
#if( (PTR_ALIGNMENT == TAG_SHIFT) )
#   define VECTOR_REF( v, i ) \
       (*((obj_t *)((long)CREF( v ) + (VECTOR_SIZE - TAG_INT) + ((long)i))))
#else
#   define VECTOR_REF( v, i ) \
       (*((obj_t *)(((long)CREF( v )) + VECTOR_SIZE + (OBJ_SIZE * CINT( i )))))
#endif
```

Afin d'être sûr de ne pas négliger ces opérations nous incluons ici des tests sur les tableaux.

sievev En plus des manipulations de vecteurs, ce programme fait des opérations arithmétiques entières plus complexes que les précédentes.

Tout d'abord, voici les temps de compilation:

	fib	tak	bague	sievev
TAG_INT = 01	1.9 s	1.9 s	2.8 s	2.3 s
TAG_INT = 00	1.9 s	1.9 s	2.8 s	2.3 s
01/00	0 %	0 %	0 %	0 %

Les temps de compilation sur Sparc.

	fib	tak	bague	sievev
TAG_INT = 01	1.3 s	1.3 s	1.8 s	1.5 s
TAG_INT = 00	1.3 s	1.3 s	1.8 s	1.5 s
01/00	0 %	0 %	0 %	0 %

Les temps de compilation sur Mips.

Ensuite, voici les temps d'exécution:

	fib	tak	bague	sievev
TAG_INT = 01	10.4 s	11.2 s	26.4 s	16.5 s
TAG_INT = 00	10.2 s	11.2 s	26.3 s	16.5 s
01/00	2 %	0 %	0 %	0 % s

Les temps d'exécution sur Sparc.

	fib	tak	bague	sievev
TAG_INT = 01	12.7 s	12.4 s	35.0 s	18.1 s
TAG_INT = 00	11.9 s	12.3 s	35.0 s	17.3 s
01/00	6 %	0 %	0 %	4 %

Les temps d'exécution sur Mips.

On voit donc que choisir de mettre des étiquette s valant `00` pour les entiers a un impact presque imperceptible sur les temps d'exécution car, même des programmes presque spécialisés dans la manipulation d'entiers, ont sensiblement les mêmes performances que si l'étiquette est non nulle. Bien que cela puisse sembler paradoxal, cela est parfaitement normal, car le coût d'une opération arithmétique sur une machine moderne est presque imperceptible par rapport au coût d'opérations complexes comme les accès mémoire ou les appels de fonctions. Pour Bigloo, nous sommes libres de choisir les étiquettes des entiers qui nous conviennent le plus, mais nous avons tenu à donner les résultats de cette expérience pour bien montrer qu'une implantation qui aurait la combinaison `00` interdite, ne serait pas pénalisée par rapport à celle qui pourrait la choisir.

Nous avons déjà vu que la représentation des petits entiers permet d'avoir des fonctions d'accès aux tableaux performantes. Il nous faut également mentionner ici que cette représentation permet également de coder très efficacement les tests d'intervalles pour ces mêmes accès. Puisque l'étiquette des entiers est sur les bits de poids faible, on peut utiliser les comparaisons non signées de C. Voici le code réalisant des tests d'intervalle dans Bigloo:

```
#if( TAG_SHIFT <= LONG_MAX )
#   define BOUND_CHECK( o, v ) ((unsigned long)o < (unsigned long)v)
#else
#   define BOUND_CHECK( o, v ) (((long)o >= 0) && ((long)o < (long)v))
#endif
```

Puisque `TAG_SHIFT` vaut `2`, c'est donc la première définition de `BOUND_CHECK` qui est utilisée. Elle économise un test par rapport à la deuxième.

```
#define BREF( r )          ((obj_t)((long)r | TAG_STRUCT))
#define CREF( r )          ((obj_t)((long)r - TAG_STRUCT))

#if( TAG_STRUCT != 0 )
#   define POINTERP( o )    ((((long)o) & TAG_MASK) == TAG_STRUCT)
#else
#   define POINTERP( o )    (o && ((((long)o) & TAG_MASK) == TAG_STRUCT))
#endif
```

Programme 2.6: L'étiquetage/désétiquetage des structures

2.2.3 Les structures

Par structure, nous désignons ici tous les objets qui sont codés par des structures allouées. Le programme 2.6 contient le code chargé de mettre en place ou de supprimer les étiquettes permettant de reconnaître qu'un objet est une structure allouée et le prédicat retournant vrai si son argument est une structure.
Il faut distinguer le cas `TAG_STRUCT == 0` pour être sûr que la valeur C `0L` ne répondra pas vrai à ce prédicat.

Tous les objets alloués possèdent une représentation uniforme. Le premier mot des structures est un en-tête permettant de reconnaître le type de l'objet (vecteur, chaîne de caractères, procédure, ...) ; les mots suivants sont l'objet lui-même. Les nombres flottants, puisqu'ils sont alloués, constituent un des exemples les plus simples pour illustrer notre représentation. Au vu de ce que nous venons d'exposer, ils sont représentés par la structure :

```
struct {                        /*  Les nombres flottants            */
   header_t        header;
   double          real;
} real_t;
```

Le prédicat retournant vrai si l'argument est un nombre flottant est codé de la façon suivante :

```
#define HEADER( o )    (CREF( o )->header)
#define HEADER_REAL    ((header_t)BINT( 16 ))

#define REALP( o )     (POINTERP( o ) && (HEADER( o ) == HEADER_REAL))
```

Les chaînes de caractères, les procédures, les vecteurs et quelques autres objets que nous négligerons ici ont en commun de ne pas avoir de taille fixe mais une taille dépendant de leur valeur. Puisque tous sont programmés de la même façon, nous allons seulement examiner le codage de ces vecteurs. Voici la structure `vector_t` :

```
   struct {                     /*  Les vecteurs, un header et une    */
      header_t        header;   /*  taille (ATTENTION: sur 22 bits,   */
      union object   *length;   /*  voir la macro vector-length).     */
   } vector_t;
```

Le champs `length` contiendra la taille des vecteurs (information codée en utilisant un petit entier au format vu précédemment). Le vecteur lui-même, ou plus exactement les différents champs du vecteur, n'apparaissent pas dans la structure pas plus que la structure ne contient de pointeur sur ces champs. Ainsi, l'allocation d'un vecteur alloue une zone mémoire de la taille de la structure additionnée à la taille du vecteur à allouer. Les champs du vecteur sont donc stockés "derrière" la structure. Ce codage évite l'indirection qu'imposerait la solution d'un pointeur sur les champs car la fonction `VECTOR_REF` ressemblerait obligatoirement à : `#define VECTOR_REF( o, i ) ((obj_t)(BREF( o ).fields)[ i ])`

2.2.4 Les listes

Il n'est pas obligatoire de faire un cas particulier pour les paires. On peut les coder en les considérant comme des objets alloués (structures) normaux mais cela présente deux intérêts majeurs :

- La taille des paires est plus petite. Avec cette solution, elle a exactement la taille de deux pointeurs. La solution générale aurait conduit à une taille de trois pointeurs. La gestion mémoire des listes aurait

```
#define BPAIR( p )          ((obj_t)((long)p | TAG_PAIR))
#define CPAIR( p )          ((obj_t)((long)p - TAG_PAIR))
#define PAIRP( c )          ((c && ((((long)c) & TAG_MASK) == TAG_PAIR)))
```

Programme 2.7: L'étiquetage/désétiquetage des paires

donc été 2/3 plus lourde[4].

- Tester qu'un objet est une paire est beaucoup plus rapide, car on n'a pas besoin d'effectuer une déréférenciation, il suffit d'examiner le pointeur lui même.

Le Programme 2.7 contient le code permettant de manipuler les paires Bigloo. La dissymétrie qui existe entre les macros `BPAIR` et `CPAIR` (la première utilise l'opérateur | et la deuxième une soustraction) n'est pas fruit du hasard. En utilisant un `-` plutôt qu'une opération booléenne on permet au compilateur C de mieux optimiser l'accès au `cdr`. Compiler un accès au `cdr` nécessite en effet deux opérations : convertir la paire Bigloo en paire C au moyen de la macro `CPAIR`, puis référencer le deuxième champ de la structure. Ainsi, calculer l'adresse du `cdr` consiste à ajouter à l'adresse de la structure la taille du `car`. Puisque cette taille et la valeur `TAG_PAIR` sont connues pendant la compilation, le compilateur C réalise la soustraction de la taille du `car` et de la valeur de `TAG_PAIR` pendant la compilation. Ainsi le code machine produit est optimal. Le Programme 2.8 le montre[5].

```
obj_t foo( obj_t p )
{
   return CDR( CPAIR( p ) );
}
```

```
_foo:
        !#PROLOGUE# 0
        save %sp,-112,%sp
        !#PROLOGUE# 1
        ld [%i0-2],%i0
        ret
        restore
```

Programme 2.8: cdr sur Sparc

Remarque : Même si ce n'est pas ici notre propos, nous faisons remarquer que le typage dynamique n'entraîne aucun surcoût pour le codage des fonctions `car` *et* `cdr` *car les opérations* d'étiquetage/désétiquetage *disparaissent à la compilation.*

2.2.5 Les constantes

Nous avons vu jusqu'à présent les cas des entiers, des objets alloués et des listes. Il nous reste à examiner les représentations de certains objets qui sont représentés par des valeurs immédiates. C'est toujours dans un souci de performance qu'est prise la décision de ne pas allouer de structure pour coder un objet. Certains objets jouent des rôles tellement importants que ce choix peut être déterminant, pour les performances globales. Nous avons, par exemple, mesuré, pour l'interprète de Bigloo, des temps environ 5 % plus longs lorsque les constantes sont codées par des valeurs allouées plutôt que par des constantes immédiates (sur machine à base de processeurs Sparc). Ces objets sont les deux booléens (`#t` et `#f`)[6] et la liste vide `'()`.

Le Programme 2.9 contient le code utilisé pour tester qu'une expression est vraie ou bien qu'un objet est la liste vide. Si ce codage est indépendant de la façon dont sont représentés la liste vide et l'objet `#f`, le code produit par le compilateur C en dépend fortement. Voici les codes en langage d'assemblage Sparc et Mips, résultat de la compilation de la fonction suivante :

[4] Sur les processeurs Sparc, pour des raisons d'alignement sur 8 octets, la solution générale aurait même conduit à avoir des paires de la taille de 4 pointeurs.

[5] L'extrême faiblesse du coût du `cdr` rend totalement sans effet l'astuce de naguère qui, partant de la constatation empirique que les `cdr` sont plus fréquemment utilisés que les `car`, consistait à les placer en $1^{ère}$ position dans la structure représentant les paires. Ainsi, il est ici permis d'adopter en toute tranquillité le codage qui place le `car` en $1^{ère}$ position et le `cdr` en $2^{ème}$.

[6] En fait c'est surtout `#f` qui est important, `#t` ne jouant qu'un rôle plus mineur.

```
#define NULLP( c )          ((long)(c) == (long)BNIL)
#define TRUEP( c )          ((bool_t)(c != BFALSE))
```

Programme 2.9: Les tests essentiels

```
obj_t foo( obj_t x )
{
   return NULLP( x );
}
```

Voici en premier les codes Sparc:

```
_foo:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        xor %o0,2,%o0
        subcc %g0,%o0,%g0
        retl
        subx %g0,-1,%o0
```

```
_foo:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        sethi %hi(_nil_object),%g2
        ld [%g2+%lo(_nil_object)],%g2
        xor %o0,%g2,%o0
        subcc %g0,%o0,%g0
        retl
        subx %g0,-1,%o0
```

Programme 2.10: La différence entre valeur immédiate (à gauche) et valeur allouée (à droite) (sur Sparc)

Le programme 2.11 contient la compilation de la fonction `foo` sur Mips.

```
foo:
        .frame  $sp,0,$31
        .mask   0x00000000,0
        .fmask  0x00000000,0
        xori    $2,$4,0x0002
        .set    noreorder
        .set    nomacro
        j       $31
        sltu    $2,$2,1
        .set    macro
        .set    reorder

        .end    foo
```

```
foo:
        .frame  $sp,0,$31
        .mask   0x00000000,0
        .fmask  0x00000000,0
        lw      $2,nil_object
        #nop
        xor     $2,$4,$2
        .set    noreorder
        .set    nomacro
        j       $31
        sltu    $2,$2,1
        .set    macro
        .set    reorder

        .end    foo
```

Programme 2.11: La différence entre valeur immédiate (à gauche) et valeur allouée (à droite) (sur Mips)

Il apparaît très clairement que sur les deux machines, la version qui utilise des constantes immédiates produit du code beaucoup plus efficace que celle qui utilise des constantes allouées. Sur Sparc la différence est un peu plus nette que sur Mips mais dans les deux cas une instruction de lecture (`load`) est économisée. Étant donné que ce type de tests (tester la liste vide ou l'objet faux) apparaît très souvent dans les programmes, il est crucial de choisir la meilleure représentation possible des constantes. Il est donc naturel que nous ayons choisi la version qui utilise les valeurs immédiates.

D'autres objets sont implantés avec des valeurs immédiates : l'objet "fin de fichier", l'objet "non initialisé"[7] et enfin les caractères. Le Programme 2.12 montre toutes les constantes et leur codage dans Bigloo.
La valeur `BEAO` est employée par les fonctions à arité variable de Scheme qui utilisent le protocole C `stdarg`; elle sert à repérer la fin de la liste des paramètres effectifs. La valeur `BCHARH` n'est utilisée que dans le prédicat

[7] Cette valeur est utile pour caractériser toutes les valeurs qui ne sont pas bien établies (comme, par exemple, les valeurs d'initialisation des liaisons d'un `letrec`).

```
#define BCNST( c )          (obj_t)BOX( c, TAG_SHIFT, TAG_CNST )
#define CCNST( c )          (long)UNBOX( c, TAG_SHIFT, TAG_CNST )

#define BNIL                ((obj_t)BCNST( 0 ))
#define BFALSE              ((obj_t)BCNST( 1 ))
#define BTRUE               ((obj_t)BCNST( 2 ))
#define BUNSPEC             ((obj_t)BCNST( 3 ))
#define BEOF                ((obj_t)BCNST( 4 ))
#define BCHARH              ((obj_t)BCNST( 5 ))
#define BEOA                ((obj_t)BCNST( 6 ))

#define TRUEP( c )          ((unsigned char)(c != BFALSE))

#define BCHAR( i )          ((obj_t)((long)BCHARH +
                             ((long)((unsigned char)(i) << 8))))
#define CCHAR( i )          (long)((long)(i)>>8)
#define CHARP( o )          (((long)(o) & (long)(BCHARH)) == (long)BCHARH)
```

Programme 2.12: Toutes les constantes

`CHARP`.

2.2.6 Le type `obj_t`

Nous avons examiné les quatre familles d'objet de Bigloo, il ne nous reste donc plus qu'à exposer le type générique `obj_t` qui est déjà apparu dans les codes donnés plus avant. Quelle que soit leur famille, les objets de Bigloo ne sont que d'un seul type C, `obj_t`. Deux possibilités existent pour le représenter :

- Fixer que ce type est le même que le type `long` (ou n'importe quel autre type C servant usuellement à représenter des objets génériques, par exemple, en C ISO [ISO90], on pourrait choisir le type `void *`) puis faire des conversions de type (*cast*) à chaque accès à un objet.
- Représenter le type `obj_t` comme une union C.

Ces deux solutions sont strictement équivalentes, notamment en matière de portabilité. C'est uniquement parce qu'elle nous semblait plus élégante que nous avons choisi la deuxième :

```
 1:  typedef long  int_t;
 2:  typedef int_t header_t;
 3:
 4:  typedef union object {
 5:     int_t              integer;
 6:
 7:     header_t           header;
 8:
 9:     struct {
10:  #if( !(defined( TAG_PAIR ) ) )
11:        header_t        header;
12:  #endif
13:        union object   *car;
14:        union object   *cdr;
15:     } pair_t;
16:
17:  #if( defined( ALLOCATE_CONSTANT ) )
18:     struct {
19:        header_t        header;
20:     } boolean_t;
21:  #endif
22:
23:     struct {
24:        header_t        header;
25:        union object   *length;
26:     } string_t;
27:
28:     struct {
29:        header_t        header;
30:        union object   *length;
31:     } vector_t;
32:
33:     struct {
34:        header_t        header;
35:        char           *name;
36:        union object   *cval;
37:     } symbol_t;
38:  ...
39:  } *obj_t;
40:
```

Nous présenterons dans la suite de ce chapitre quelques unes des autres « branches » de cette union (comme par exemple le type `procedure_t` qui représente les fonctions ou bien le type `stack_t` qui désigne les piles d'exécution).

2.3 Les fonctions

Les langages fonctionnels mettent en avant un style de programmation où les fonctions sont des objets centraux. Les boucles des langages impératifs n'existent pas obligatoirement dans ces langages[8], car elles ne sont que des cas particuliers des fonctions. Ainsi l'efficacité d'un système fonctionnel est suspendue à l'efficacité de son traitement des fonctions. Cette section présente la façon dont Bigloo les manipule.

Nous nous sommes aperçus que la terminologie employée pour décrire les fonctions est dépendante des communautés. Par exemple, le terme de la norme de Scheme [IEE91] *procédure* n'est pas du tout à prendre dans le même sens que dans le langage Pascal ! Ainsi, pour désigner les fonctions (ou λ-expressions) du langage source nous emploierons le terme *fonctions* ou éventuellement la lettre λ. Quand une fonction ne pourra pas être seulement représentée par un pointeur de code nous dirons qu'il y a création d'une fermeture (ou éventuellement, création d'une *procédure* pour suivre la norme [IEE91]). Autrement dit, dans notre terminologie, toutes les fonctions du langage source ne nécessitent pas de création de fermetures.

2.3.1 L'objet "fermeture" (ou *procédure*)

Les fonctions des langages fonctionnels modernes (Scheme, ML, ...) sont des objets de première classe. Elles peuvent être manipulées de la même façon que les autres données. Il peut donc y avoir obligation pour le compilateur de créer un objet manipulable représentant cette fonction. De plus, la liaison statique impose aux fonctions de garder en mémoire de leur environnement de définition. Ce sont ces deux traits qui conditionnent le plus fortement leur représentation. Ainsi, les procédures de Bigloo sont définies par la structure suivante :[9].

```
struct {                        /*  Les fermetures                    */
   header_t        header;
   union object *(*entry)();
   union object *(*va_entry)();
   long             arity;
   char            *env;        /*  Ce champ est utilise pour etre    */
} procedure_t;                  /*  sur que l'alignement est correct. */
```

Avant d'expliquer ces cinq champs, nous devons faire une remarque très importante. *Bigloo ne construit des fermetures que lorsqu'il y est contraint. Cela signifie que la plupart des fonctions ne nécessite* aucune allocation*. Seules, celles qui sont manipulées comme valeurs sont représentées par des objets alloués. Les autres ne sont seulement présentes qu'à l'état de code dans l'exécutable.*

header Ici est rangé l'information de type indispensable au prédicat `procedure?` qui teste si son argument est une fermeture :

```
#define PROCEDUREP( fun ) \
   (POINTERP( fun ) && (HEADER( fun ) == HEADER_PROCEDURE))
```

entry Une fermeture est un couple composé d'un environnement et d'un code à exécuter. Ce champ est un pointeur sur le code. Cela signifie que lorsqu'une fermeture sera invoquée, le contrôle se branchera à l'adresse pointée par `entry` (qui est une fonction C).

va_entry Les fonctions Scheme peuvent être d'arité variable ou d'arité fixe. Dans le premier cas, ce champ est le pointeur sur le code de la fonction. Le champ `entry` est recyclé ; il est un pointeur vers une petite fonction qui se charge d'invoquer convenablement le code pointé par `va_entry`. Ce protocole d'appel sera décrit ultérieurement. Dans le deuxième cas (arité fixe), ce champ est inutilisé. Puisque les fonctions d'arité fixe sont majoritaires, dans le cas le plus fréquent ce champ est perdu. Il est regrettable que la

[8] C'est le cas de Scheme qui ne possède aucune construction de boucle comme les `while`, `for` ou autres

[9] Nous verrons par la suite dans le chapitre 7 qu'il existe dans Bigloo trois façons de représenter les fermetures.

minorité des fonctions à arité variable alourdissent l'implantation de la majorité mais nous avons accepté cette perte (relative puisqu'elle n'est que d'un mot mémoire) car ainsi l'implantation du protocole d'appel est plus simple et plus performant dans le cas général (c'est-à-dire dans le cas des fonctions d'arité fixe) [10].

arity Pour les langages typés dynamiquement, il faut pouvoir tester que la fonction invoquée l'est avec un nombre correct d'argument. Ce champ contient donc un entier qui est l'arité de la fonction. Nous avons choisi la convention suivante : les fonctions d'arité fixe ont un champ `arity` qui est un entier positif, ce nombre est directement le nombre d'arguments que la fonction doit recevoir ; les fonctions d'arité variable ont un champ `arity` qui est un nombre négatif. Pour qu'une fonction à arité variable soit invoquée avec un nombre correct d'arguments, il faut que pour chacun de ses sites d'appels, il y ait au moins $-\texttt{arity}+1$ arguments.

env Ce pointeur n'est utile que pour assurer un alignement correct pour le premier champ de l'environnement. Les environnements des fermetures sont codés à plat, c'est-à-dire au moyen de simples vecteurs. Comme l'explique le Dr Séniak dans sa thèse [Sén91], cette représentation est plus simple, mais probablement moins efficace qu'un codage utilisant la technique de *display* [ASU86]. Il faut toutefois préciser que cette inefficacité n'apparaît que pour des programmes très *higher order* utilisant beaucoup de fermetures. Pour des programmes plus algorithmiques la représentation plate n'est pas pénalisante. Son inefficacité ne provient que de ce qu'elle n'utilise pas de partage physique d'environnement (elle alloue donc plus) et que les variables capturées affectées (par `set!`) ne peuvent pas être mémorisées telles quelles dans les environnements. Il faut avoir recours à des cellules d'indirection pour que les modifications physiques des variables soient correctement répercutées et vues par toutes les fonctions qui les capturent. Par ailleurs, il faut préciser que la représentation utilisant des *displays* doit être utilisée avec précaution car elle peut occasionner des fuites de mémoires [Bak92b]. Puisque les fermetures sont chaînées les unes aux autres, il est possible que le *GC* ne parvienne pas à récupérer des zones mémoires qui sont en fait inutilisées. Le vrai problème est que ce phénomène est incontrôlable et imprévisible depuis le langage source, puisqu'il dépend uniquement du choix du compilateur. En d'autres termes, un programme peut échouer à l'exécution faute de mémoire, sans que le programme lui-même soit en cause.

Nous avons expliqué l'utilité de chaque champ des fermetures, examinons maintenant sur un exemple le code produit par le compilateur lorsqu'il crée des fermetures. Pour cela, voici un petit programme Scheme qui n'a pas de sens, mais qui utilise une fonction d'arité fixe `bar` et une fonction d'arité variable `gee` :

```
(letrec ((gee (lambda (x . y) a))
         (bar (lambda (x y) a)))
   gee
   bar)
```

le produit de compilation en C de cette expression est :

```
{
   obj_t GEE_ENV, BAR_ENV;
   GEE_ENV = make_va_procedure( GEE, -2, 1 );
   BAR_ENV = make_fx_procedure( BAR, 2, 1 );
   {
      PROCEDURE_ENV_SET( BAR_ENV, 0, A );
      PROCEDURE_ENV_SET( GEE_ENV, 0, A );
      BAR_ENV;
      GEE_ENV;
   }
}
```

Les deux procédures sont créées par deux fonctions différentes suivant qu'elles ont une arité fixe ou variable (`make_fx_procedure` et `make_va_procedure`). Ces deux fonctions admettent les mêmes arguments, le point d'entrée de la fonction, l'arité et la taille de l'environnement. Les deux procédures ont chacune une variable

[10] Nous aurions probablement pu trouver une solution où ce mot mémoire n'aurait pas été perdu mais comme le mécanisme C utilisé (`stdargs`, voir [HS91, pages 266-268]) pour les fonctions d'arité variable ne permet pas de manipuler l'*ensemble* des arguments facultatifs (il ne permet que de récupérer successivement les valeurs de tous les paramètres) cette solution aurait risqué d'entraîner une augmentation importante de la taille du code produit.

libre (on dit aussi que chacune *capture* une variable, ici la variable `A`), l'environnement est donc de taille un. Les deux fonctions créatrices de fermetures se contentent d'allouer un espace mémoire et d'initialiser les quatre premiers champs des fermetures ; l'initialisation de l'environnement apparaît dans le bloc lexical suivant. Voici le code de la fonction `PROCEDURE_ENV_SET` :

```
#define PROCEDURE( o ) CREF( o )->procedure_t

#define PROCEDURE_ENV_REF( p, i ) \
   (*((obj_t *)(((long)&(PROCEDURE( p ).env)) + (OBJ_SIZE * i))))

#define PROCEDURE_ENV_SET( p, i, o ) ((PROCEDURE_ENV_REF( p, i ) = o), p)
```

Nous avons vu comment les fermetures sont créées, nous allons voir maintenant comment elles sont utilisées. Pour cela, examinons le produit de compilation de la fonction suivante :

```
(define (foo f x) (f x))
```

Cette fonction (`foo`) se contente d'invoquer son premier argument avec son second. Dans ce cas, on ne sait rien sur le type de `f`. Le code produit en C est donné dans le programme 2.13.

```
 1:  obj_t
 2:  FOO( obj_t F, obj_t X )
 3:  {
 4:  _FOO:
 5:
 6:     if( PROCEDUREP( F ) )
 7:     {
 8:        if( PROCEDURE_CORRECT_ARITYP( F, 1 ) )
 9:           return PROCEDURE_ENTRY( F )( F, X, BEOA );
10:        else
11:           FAILURE( ... , ..., F );
12:     }
13:     else
14:         FAILURE( ..., ..., F );
15:  }
```

Programme 2.13: allocation dans le tas: produit de compilation

Ce code C est le résultat d'une compilation où tous les tests de type ne sont pas effectués pendant la compilation. Le premier de ces tests consiste à s'assurer que le premier argument est une fonction. Voici le code de la fonction `PROCEDUREP` :

```
#define PROCEDUREP( fun ) \
   (POINTERP( fun ) && (HEADER( fun ) == HEADER_PROCEDURE))
```

Le deuxième test effectué est un test d'arité. Le code de la fonction `PROCEDURE_CORRECT_ARITYP` est :

```
#define PROCEDURE_ARITY( fun ) (PROCEDURE( fun ).arity)

#define VA_PROCEDUREP( fun ) ( PROCEDURE_ARITY( fun ) < 0 )

#define PROCEDURE_CORRECT_ARITYP( fun, num )      \
   ( (PROCEDURE_ARITY( fun ) == num) ||           \
     (VA_PROCEDUREP( fun ) &&                     \
       ((-num - 1) <= (PROCEDURE_ARITY( fun )))) )
```

Si l'un des tests échoue, la fonction d'erreur `FAILURE` est invoquée. En revanche, si les deux tests sont positifs, la fonction est invoquée au moyen de la fonction `PROCEDURE_ENTRY` dont le code simplissime est :

```
#define PROCEDURE_ENTRY( fun )    (obj_t)(PROCEDURE( fun ).entry)
#define PROCEDURE_VA_ENTRY( fun ) (obj_t)(PROCEDURE( fun ).va_entry)
```

Le nombre d'opérations et de tests pour effectuer un appel de fonction peut sembler important, apportons quelques précisions. Le type d'invocation utilisé ici n'est pas très fréquent dans le code produit par Bigloo.

Dans la plupart des cas, le compilateur possède des informations sur les fonctions invoquées qui lui permettent d'utiliser l'un de ses deux autres protocoles d'appel (voir la section 2.3.2). Le cas présent est le moins efficace. D'autre part, la très grande majorité des fonctions sont d'arité fixe, ainsi seulement deux tests (le test de type et le premier cas du test d'arité) et un appel de fonction C sont exécutés. Il est raisonnable que le cas rare (fonction d'arité variable) soit défavorisé par rapport au cas le plus fréquent.

2.3.2 Les protocoles d'appels

Bigloo utilise trois protocoles d'appel aux fonctions. Le choix du protocole est lié à la nature de la fonction invoquée (la fonction est-elle compilée en une fermeture ou pas) et au site d'appel lui-même (est-ce, par exemple, un site terminal). Nous allons utiliser une terminologie proche de celle de D. Kranz [Kra88] pour cet exposé. Il ne s'agit pas ici d'expliquer quand et comment le compilateur choisit l'un des protocoles, cela sera vu dans les chapitres consacrés au compilateur, nous nous contenterons ici de montrer ce qu'ils sont. Nous les présentons dans un ordre décroissant d'efficacité.

Arguments passés dans des registres La fonction n'a pas d'environnement alloué et ses arguments sont passés dans des registres. Il n'y a pas eu de création de fermeture. Ce mode d'appel est le plus fréquent (et le plus performant) car ce sont les boucles qui l'utilisent. Le choix de ce protocole pour invoquer une fonction dépend uniquement de la position de l'invocation elle-même (la position terminale l'autorisant), et pas de la fonction appelée. Voici, sur un exemple en quoi il consiste :

```
1:  (letrec ((for (lambda (i j)
2:                  (if (= i j)
3:                      ...
4:                      (begin
5:                         ...
6:                         (for (+ i 1) (- j 1)))))))
7:     (for 0 10))
```

Tous les sites d'invocation de la fonction `for` (lignes *6* et *7*) sont en position terminale. Il n'y a donc pas de nécessité d'utiliser une pile et l'utilisation du protocole utilisant des registres est ainsi permis. Le code C produit est :

```
1:  {
2:     obj_t  I, J;
3:     {
4:        I = BINT( 0 );
5:        J = BINT( 10 );
6:        {
7:  _FOR:
8:
9:           if( EQ_I( I, J ) )
10:             ...
11:          else
12:          {
13:             ...
14:             {
15:                obj_t J2, I2;
16:                J2 = J;
17:                I2 = I;
18:                J = SUB_I( J2, BINT( 1 ) );
19:                I = ADD_I( I2, BINT( 1 ) );
20:                goto _FOR;
21:             }
22:          }
23:       }
24:    }
25: }
```

La fonction `for` est expansée en ligne sur le site d'invocation de la ligne *7* du code source (ligne *1* dans le code généré). Le compilateur alloue deux registres pour représenter les paramètres formels de `for` (ligne *2*). Le résultat de compilation de la ligne *6* du code source est constitué des lignes *14* à *21*. Ces lignes sont des affectations des registres (lignes *15* à *19*) et un saut direct (ligne *20*).

Ce protocole d'appel est le plus performant car il ne consomme aucune ressource (ni pile, ni tas) ; il préserve la propriété de récursivité terminale. Dès que le compilateur le pourra, il choisira d'allouer les fermetures dans les registres. Ce choix dépend uniquement du site d'application. Une fonction peut être invoquée au sein d'un même programme par plusieurs protocoles différents dont seulement quelques uns utilisent des registres.

Ce protocole n'est pas restreint aux simples boucles, des fonctions mutuellement récursives peuvent l'utiliser. Ainsi, le programme :

```
(letrec ((odd? (lambda (n)
                  (if (= n 0)
                      #f
                      (even? (- n 1)))))
         (even? (lambda (m)
                  (if (= m 0)
                      #t
                      (odd? (- m 1))))))
   (odd? 10))
```

est compilé par Bigloo en :

```
obj_t N, M;
{
   N = BINT( 10 );
_ODD:
   if( N == BINT( 0 ) )
      return BFALSE;
   else
   {
      M = SUB_I( N, BINT( 1 ) );
   _EVEN:
      if( M == BINT( 0 ) )
         return BTRUE;
      else
      {
         N = SUB_I( M, BINT( 1 ) );
         goto _ODD;
      }
   }
}
```

Ce protocole n'est utilisable que si tous les sites d'appels sont en position terminale. Si ce n'est pas le cas, il faut utiliser une autre technique.

paramètres passés dans la pile La fonction n'a pas d'environnement alloué, elle est compilée en une fonction C, le protocole d'appel est donc celui de C. Voici un exemple de son utilisation :

```
1:  (letrec ((copy-list (lambda (l)
2:                        (if (null? l)
3:                            '()
4:                            (cons (car l) (copy-list (cdr l)))))))
5:     (copy-list l))
```

L'invocation de `copy-list` de la ligne *4* n'est pas récursive terminale. À moins de dérécursiver cette fonction, une pile est nécessaire pour son exécution. Ainsi le protocole d'appel utilisant les registres n'est pas utilisable. Bigloo va donc faire correspondre à la fonction `copy-list` une fonction C et le protocole d'appel sera celui de C. Voici le code produit :

```
1:  {
2:     return COPY_LIST( L );
3:  }
4:
5:  static obj_t
6:  COPY_LIST( obj_t L )
7:  {
8:  _COPY_LIST:
9:
10:    if( NULLP( L ) )
11:       return BNIL;
12:    else
13:    {
14:       obj_t OBJ1, OBJ2;
15:       OBJ1 = CAR( L );
16:       OBJ2 = COPY_LIST( CDR( L ) );
17:       return MAKE_PAIR( OBJ1, OBJ2 );
18:    }
19: }
```

Le fait que Bigloo s'en remette au compilateur C pour choisir la stratégie d'allocation des paramètres effectifs et du bloc d'activation [ASU86] et que presque tous ces compilateurs, sur presque toutes les machines, les allouent dans la pile, explique que nous nommions ce protocole *allocation dans la pile*. Il est très important de noter que ce protocole d'appel ne nécessite pas d'allocation mémoire préalable (ni dans le tas, ni dans la pile) pour accueillir l'environnement de définition de la fonction appelée ; il n'y pas eu de création de fermeture. En effet, la fonction étant connue, une transformation de programme appelée λ-lifting [Joh85] permet de rajouter les variables capturées comme paramètres supplémentaires de la fonction. Ce protocole peut préserver la propriété de récursion terminale uniquement si le compilateur C fait cette optimisation[11]. Tous les appels aux fonctions qui n'ont pu utiliser le protocole utilisant des registres et qui sont invoquées directement, se servent de ce nouveau protocole. Les seules fonctions qui ne sont pas dans ce cas sont les fonctions invoquées par des appels calculés (des sites d'invocation où la fonction appelée n'est pas une constante connue du compilateur). Ce protocole d'appel a l'efficacité du protocole d'appel de C.

La fonction possède un environnement alloué dans le tas Ce dernier cas est le moins efficace et un exemple a déjà été vu précédemment (programme 2.13). C'est le protocole utilisé quand la fonction invoquée n'est pas connue du compilateur. Il y a création d'une fermeture qui est allouée dans le tas et les arguments sont alloués dans la pile. La fonction invoquée n'est pas connue, le compilateur ignore donc si elle a des variables capturées et si elle attend un nombre fixe ou variable d'arguments. Le schéma d'invocation doit donc permettre d'appeler ces quatre classes de fonctions. Voici les règles que nous avons adoptées :

- La représentation des fonctions susceptibles d'être invoquées par ce protocole (fonctions exportées ou passées en argument ou encore retournées comme résultat, dont, par la suite nous dirons qu'elles *s'échappent* ou *s'enfuient*) est différente suivant que ces fonctions sont globales ou locales. Le cas des fonctions globales est plus simple car leur environnement de définition est toujours vide. Pour elles, on se contente de créer un deuxième point d'entrée qui respecte le protocole décrit et qui invoque la vraie fonction après avoir rétabli les paramètres effectifs et filtré les arguments inutiles (comme l'environnement).
- Les fonctions locales qui s'échappent sont toutes affublées d'un paramètre supplémentaire qui est un pointeur sur la fermeture elle-même. Ce paramètre est ajouté en tête d'arguments et sert à retrouver l'environnement de définition.
- Que la fonction soit d'arité fixe ou variable, c'est toujours la fonction pointée par le champ `entry` des procédures qui est invoquée (voir la section 2.3.1). Pour coder les fonctions d'arité variable,

[11] Le compilateur `gnu-cc`, [Sta89] contrairement à l'idée reçue ne fait cette optimisation que pour les fonctions auto récursives terminales; il ne la fait pas dans le cas général.

nous avons utilisé les `stdarg` de C ISO. Les *stdarg* sont un mécanisme fourni par les compilateurs C pour utiliser des fonctions à nombre variable d'arguments. Bien sûr celles ci ne sont pas les mêmes que celles de Scheme mais, néanmoins, ce système est relativement bien adapté pour notre implantation. Nous ne rappelons pas ici comment fonctionne ce mécanisme (cela peut-être trouvé dans [HS91]) mais mentionnons tout de même que le programmeur a le choix de la méthode pour déterminer combien d'arguments sont passés. Puisque la façon classique de faire cela en C est d'ajouter un dernier argument servant de butée, nous ajoutons systématiquement un deuxième argument supplémentaire lors d'un appel calculé en dernière position. Cette butée est nommée `BEAO`, c'est une constante Bigloo(au sens de la section 2.2.5). Puisque cet argument est ajouté systématiquement pour toutes les fonctions, il l'est donc également pour les fonctions à nombre fixe d'arguments. Ces fonctions ne l'utiliseront pas mais comme C a un schéma de compilation où c'est l'appelant qui empile et qui dépile les arguments, les paramètres supplémentaires non consommés ne sont pas interdits. En guise d'exemple, examinons le produit de compilation déjà donné dans le programme 2.13. L'appel a lieu ligne *9*. La fonction `f` est invoquée avec un argument `x` mais comme nous l'avons expliqué, le premier paramètre est la fermeture elle-même et le dernier argument est la butée.

2.3.3 Les fonctions à arité variable de Scheme

Voici comment sont implantées les fonctions à arité variable de Scheme. Ce propos ne s'applique qu'à ce langage. L'utilisation de la notation pointée (voir [IEE91]) fait que le dernier paramètre formel d'une fonction sera lié à la liste des paramètres effectifs encore non consommés. Il est explicité dans la norme de Scheme qu'une liste accueillant tous les paramètres effectifs doit être construite à chaque invocation de la fonction. Deux cas différents se présentent :

- La fonction à argument variable est invoquée directement (le compilateur la connaît) la construction de la liste est alors intégrée dans l'appel à la fonction. C'est possible car le compilateur connaissant la fonction, connaît les arguments facultatifs et peut donc construire la liste.
- La fonction à argument variable n'est invoquée qu'après avoir été passée en argument ou retournée comme résultat, le compilateur ne la connaît pas, il utilise le protocole d'appel avec allocation dans le tas. C'est donc le point d'entrée `entry` de la procédure qui est utilisé. Ce point d'entrée se charge alors de construire une liste et d'invoquer le second point d'entrée `va_entry` une fois que la liste est construite. Une unique fonction C est pointée par `entry` (elle se nomme `va_generic_entry`) pour toutes ces fonctions. Une autre solution avait été imaginée ; elle consistait pour chaque fonction à arité variable à construire un point d'entrée chargé de construire la liste et d'invoquer le deuxième point d'entrée. Cette solution était probablement un tout petit peu plus efficace mais l'efficacité des appels calculés pour les fonctions à arité variable n'est pas déterminant car ils sont très peu nombreux. Cette solution présentant l'inconvénient de produire plus de code (car chaque fonction nécessitait ces deux points), nous l'avons donc abandonnée.

2.3.4 La fonction de la bibliothèque : `apply`

Pour en terminer avec les fonctions, il nous faut encore expliquer comment est implantée la fonction de la bibliothèque Scheme `apply`. Cette fonction qui existe dans tout les Lisp prend en argument, dans sa version binaire, deux valeurs, une fonction et une liste. Elle permet d'invoquer la fonction argument définie classiquement non pas avec n arguments (où n serait l'arité de la fonction) mais avec une liste de longueur n. La fonction `apply` se charge alors d'invoquer correctement la fonction argument en retrouvant les arguments dans la liste. Comme toujours, deux cas se présentent :

- La fonction fournie à `apply` est une constante connue du compilateur. Ce dernier peut alors produire le code qui discrimine la liste et invoque convenablement la fonction argument. C'est le cas de l'exemple :

```
(define (foo x y) ...)

(define (bar l) (apply foo l))
```

qui est compilé (en mode où les tests de type ne sont pas émis) en :

```
{
   obj_t Y, X, AUX;

   Y   = BUNSPEC;
   X   = BUNSPEC;
   AUX = BUNSPEC;

   {
      AUX = L;
      X   = CAR( AUX );
      AUX = CDR( AUX );
      Y   = CAR( AUX );

      return FOO( X, Y );
   }
}
```

- La fonction fournie en argument n'est pas une constante connue du compilateur, on utilise alors une fonction (nommée également `apply`) de la libraire, écrite en C qui utilise les `vararg`. Cette fonction est le pendant de la fonction `va_generic_entry` mais au lieu de construire une liste, elle se charge de la discriminer. Là aussi, nous avons un peu sacrifié l'efficacité de ce type d'appel au profit d'un code généré plus petit. Ceci n'a aucun impact visible sur les performances d'ensemble du compilateur car ce cas est très marginal.

2.4 L'implantation de la fonction `call/cc`

Abordons maintenant un morceau de bravoure : l'implantation en C de la fonction de la bibliothèque Scheme `call-with-current-continuation` (cette fonction est souvent désignée par son diminutif : `call/cc`). Le défi est de taille car il s'agit ici de progresser dans un terrain où tout est hostile ! Nous allons combattre la pile, affronter les fenêtres de registres, nous prémunir contre les affectations[12]. Précisons bien entendu que cette section ne concerne que les lecteurs intéressés par l'implantation des continuations. Les autres (par exemple les membres de la communauté ML) peuvent, sans risque de perdre quelques éléments indispensables, s'abstenir de sa lecture ! Comme cette fonction est pour Scheme un élément essentiel[13] il était impensable que Bigloo ne la possède pas. Quel est donc le contrat ? Implanter en C (puisque jusqu'ici nous n'avons pas écrit une seule ligne d'assembleur) une fonction qui enfreint toutes les règles de la programmation classique utilisant des piles. Nous allons devoir programmer dans ce langage des opérations qui ne sont a priori pas implantables, car elles sont « méta ». Il faut changer le contrôle des exécutions des programmes C à leur insu.

2.4.1 Rappels sur `call/cc`

La fonction `call/cc` est un opérateur de capture de continuations. Sa forme d'appel est `(call/cc` *proc* `)`. Son évaluation consiste à capturer la continuation courante et à la passer à *proc* sous forme d'une procédure à un argument. Cette continuation a une durée de vie indéfinie. Elle peut donc être sauvée dans une variable puis invoquée à tout moment. Voici un exemple extrait du rapport [IEE91] utilisant `call/cc` :

[12] Nous espérons que le lecteur sera indulgent envers cette petite envolée lyrique, elle est probablement dûe au terrible souvenir des difficultés rencontrées par l'auteur lorsqu'il s'acharnait sur cette implantation.

[13] Peut-être parce que c'est ce qui fait une grande partie de sa différence avec les autres langages.

```
(define list-length
   (lambda (obj)
      (call/cc
       (lambda (return)
          (letrec ((r (lambda (obj)
                        (cond ((null? obj) 0)
                              ((pair? obj)
                               (+ (r (cdr obj)) 1))
                              (else (return #f))))))
             (r obj))))))

:=> (list-length '(1 2 3 4))
4

:=> (list-length '(a b . c))
#f
```

Cet exemple est particulièrement simple car la continuation n'est utilisée que dans la portée dynamique du `call/cc`. Un exemple plus complexe sera présenté ultérieurement. Par ailleurs, une présentation complète de `call/cc` peut-être trouvée dans [Que94, chapitre 3].

2.4.2 Le principe général

La fonction `call/cc` capture tout le contexte de calcul et peut le restaurer à tout moment. Ainsi, tout est permis ; sortir plusieurs fois d'une fonction où pourtant on n'est entré qu'une seule fois, repartir à un endroit de la pile (si on en utilise une) qui n'existe plus, etc. Voici le schéma de fonctionnement de `call/cc`. Lors de la capture de la continuation, on effectue les opérations suivantes :

1. On pose une marque dans la pile pour savoir où il faudra reprendre le calcul courant quand la continuation sera invoquée.
2. On sauve la pile d'exécution.
3. On construit une fonction Bigloo qui sera la continuation réifiée par `call/cc`. Cette fonction a dans son environnement la pile sauvegardée à l'étape précédente, l'adresse de la marque posée dans la première opération et quelques autres valeurs indispensables à l'implantation. Elle attend un argument et son point d'entrée est une fonction qui restaurera la pile (`apply_continuation`) et reprendra le calcul actuel.
4. Invoquer la fonction qui était l'argument de `call/cc`.

Lors de l'invocation d'une continuation les opérations effectuées sont :

5. Faire grandir la pile pour qu'elle ait une taille suffisante afin de pouvoir restaurer celle qui a été sauvée lors de la capture de continuation.
6. Restaurer la pile.
7. Aller au point de contrôle qui a été mémorisé en première étape des opérations effectuées lors de la capture.

Ceci est un schéma général qui est indépendant du langage utilisé[14]. Voyons comment nous parvenons à les implanter en C.

2.4.3 Les détails techniques de l'implantation en C

La première préoccupation pour l'implantation en C est de savoir dans quel sens la pile d'exécution évolue. Croît-elle ou décroît-elle ? La réponse à cette question n'est pas aussi évidente qu'il y paraît, et est très dépendante des machines. Nous sommes obligés d'exécuter sur chaque nouvelle machine le programme 2.14.

```
 1:  main( argc, argv )                    13:
 2:  int   argc;                           14:     if( !flag )
 3:  char *argv[];                         15:     {
 4:  {                                     16:        old_addr = &new_addr;
 5:     direction( 1 );                    17:        flag = 1;
 6:  }                                     18:        direction( 2 );
 7:                                        19:     }
 8:  direction( new_addr )                 20:     else
 9:  long new_addr;                        21:        old_addr > &new_addr ?
10:  {                                     22:           puts( "stack grows DOWN" ) :
11:     static long *old_addr;             23:           puts( "stack grows UP" );
12:     static int flag = 0;               24:  }
```

Programme 2.14: Le sens de la pile C

Il compare les adresses d'un même paramètre effectif d'une fonction lors de deux invocations consécutives. Si la deuxième adresse est plus petite que la première la pile décroît, sinon elle croît.

1. Poser une marque dans la pile pour être par la suite capable de revenir à ce point est réalisé au moyen d'un `setjmp`.

2. Sauver la pile d'exécution demande plusieurs opérations :

   (a) Il faut calculer l'adresse actuelle du sommet de pile. Ceci est réalisé par la fonction `get_top_of_stack` qui retourne l'adresse d'une variable locale :

   ```
   char *get_top_of_stack()
   {
      long *dummy;

      return (char *)(&(dummy));
   }
   ```

   (b) Au début de toute exécution, l'adresse du bas de pile est mémorisée. Dans notre implantation, c'est la variable `stack_bottom` qui contient cette valeur ; dans la première fonction de C (`main`) on l'initialise par l'affectation :

   ```
   stack_bottom = ((char *)&argc);
   ```

   Ainsi, nous pouvons calculer la taille de la pile :

   ```
   #if( STACK_GROWS == DOWN )
         stack_size = (unsigned long)stack_bottom - (unsigned long)stack_top;
   #else
         stack_size = (unsigned long)stack_top - (unsigned long)stack_bottom;
   #endif
   ```

   (c) Les piles sont des objets de type `stack_t` :

   ```
      struct {                        /*  Les piles de 'call/cc'          */
         header_t        header;      /*  sont:                           */
         union object   *self;        /*        - un ptr sur soi meme     */
         union object   *size;        /*        - une taille              */
         char           *stack;       /*        - un espace memoire       */
      } stack_t;
   ```

   Leurs allocations sont réalisées au moyen de la fonction[15] :

   ```
   #define MAKE_STACK( _size_, aux ) \
      (BREF( MAKE_OBJECT( STACK_SIZE + (long)_size_, HEADER_STACK, aux )))
   ```

[14] pour autant que ce langage possède des fonctions et utilise une pile.
[15] Il faut remarquer que cette allocation est bien entendu à ajouter à l'ensemble des racines du GC.

et leur initialisation par les deux affectations :

```
STACK( stack ).size = BINT( (long)stack_size );
STACK( stack ).self = CREF( stack );
```

(d) Il faut maintenant dupliquer la pile ; ceci est réalisée par l'expression :

```
#if( STACK_GROWS == DOWN )
      memcpy( &(STACK( stack ).stack), (char *)stack_top, stack_size );
#else
      memcpy( &(STACK( stack ).stack), (char *)stack_bottom, stack_size);
#endif
```

3. Il ne reste plus qu'à allouer et initialiser la fonction continuation :

```
/* on construit la continuation */
continuation = make_fx_procedure( &apply_continuation, 1, 5 );
PROCEDURE_ENV_SET( continuation, 0, stack );
PROCEDURE_ENV_SET( continuation, 1, BREF( stack_top ) );
PROCEDURE_ENV_SET( continuation, 2, BREF( jmpbuf ) );
PROCEDURE_ENV_SET( continuation, 3, BUNSPEC );
PROCEDURE_ENV_SET( continuation, 4, BREF( (obj_t)top_of_frame ) );
```

4. et à invoquer l'argument de `call/cc` au moyen du protocole utilisant le tas :

```
if( !PROCEDURE_CORRECT_ARITYP( proc, 1 ) )
   the_failure( c_constant_string_to_string( "call/cc" ),
                c_constant_string_to_string( "illegal arity" ),
                BINT( PROCEDURE_ARITY( proc ) ) );
else
   return PROCEDURE_ENTRY( proc )( proc, continuation, BEOA );
```

Jusqu'ici il n'y a rien de très inattendu ni de très inefficace. On a alloué et initialisé. C'est lors d'une invocation que les choses se compliquent :

5. Avant de pouvoir restaurer la pile, il faut la faire grandir pour que sa taille redevienne suffisante pour pouvoir accueillir son ancienne valeur. Puisque nous ne nous permettons pas l'utilisation de la fonction `alloca` (n'étant pas dans la norme ISO [ISO90], elle est réputée peu portable [DLKR92]), nous n'avons trouvé qu'un seul moyen portable de le faire. Le code est donné dans le programme 2.15.

   Le principe est donc de faire des appels récursifs en empilant des paramètres effectifs (ligne *20*). Afin de nous prémunir contre des compilateurs trop malins qui s'apercevraient que ces paramètres ne sont jamais utilisés, nous les faisons pointer par des variables globales (ligne *19*).

6. Toutes les informations nécessaires à la restauration de la pile ont été sauvées, soit dans l'environnement de la fonction construite, soit dans la pile elle-même. Le code ré-installant l'ancienne pile est donné dans le programme 2.16.

7. Il ne reste donc plus qu'à faire faire un `longjmp` pour revenir après le `setjmp` qui a été réalisé lors de la capture de la continuation :

   ```
   longjmp( (JMP_BUF *)jmpbuf, (JMP_VAL)s_proc );
   ```

   La valeur avec laquelle est invoquée la continuation a été rangée dans la fermeture (ligne *17* du programme 2.16), il ne reste donc plus qu'à retourner cette valeur.

Le code que nous venons d'exposer est déjà assez complexe et inefficace mais il nous faut encore ajouter quelques traitements particuliers indispensables quand la machine utilisée est à base de processeurs Sparc.

```
 1:  obj_t
 2:  apply_continuation( proc, value )
 3:  obj_t proc, value;
 4:  {
 5:     char *stack_top, *actual_stack_top;
 6:
 7:     actual_stack_top = get_top_of_stack();
 8:     stack_top        = (char *)CREF( PROCEDURE_ENV_REF( proc, 1 ) );
 9:
10:     /* on fait grandir la pile jusqu'a ce qu'elle depasse stack_top */
11:  #if( STACK_GROWS == DOWN )
12:     if( ((unsigned long)stack_top) <= (unsigned long)actual_stack_top)
13:  #else
14:     if( ((unsigned long)stack_top) >= (unsigned long)actual_stack_top )
15:  #endif
16:     {
17:        char *dummy[ BLOCK_SIZE ];
18:
19:        glob_dummy = (long)dummy;
20:        apply_continuation( proc, value, dummy );
21:     }
22:     else
23:        ...
24:  }
```

Programme 2.15: L'application d'une continuation

2.4.4 Le cas retors des machines à base de processeurs Sparc.

Les Sparc présentent cette particularité d'utiliser des fenêtres de registres. Sans entrer dans les détails, cela signifie que, pour les compilateurs les utilisant, la pile de C est à la fois représentée par une pile (au sens classique) et par des registres. Ainsi donc, avant de sauver la pile, il faut être sûr que ces registres ont été copiés dans la zone qu'on va mémoriser. De même façon, lorsqu'on va restaurer la pile, il faut que les fenêtres de registres soient elles aussi, rétablies. Nous parvenons à assurer cela en provoquant des débordements de fenêtres qui forcent leur copie ou leur restauration. Ces débordements sont obtenus par une simple fonction C qui se contente de faire plus d'appels récursifs qu'il n'y a de fenêtres. Bien entendu, le coût de ces appels est à ajouter au coût global de `call/cc` sur Sparc !

2.4.5 La fonction `call/cc` coûte même si l'on ne s'en sert pas !

Examinons le programme 2.17. Une exécution conforme à la sémantique de Scheme donnée dans [IEE91] doit produire les affichages : `0`, `1`, `2`, `3` et `4`. Cela signifie que la variable `x` qui est affectée ligne *9* doit conserver sa valeur lors de l'invocation d'une continuation ! La seule façon de parvenir à l'obtenir est que `x` ne soit pas dans la pile mais dans le tas. Une cellule pointant sur `x` sera elle placée dans la pile. Comme il n'y a aucun moyen statique de savoir si un module utilise ou non `call/cc`, toutes les variables affectées doivent être placées dans des cellules et ce, même si `call/cc` n'est pas utilisée !

2.4.6 `call/cc` et son implantation

Nous avons tenu à ne rien cacher de l'implantation de `call/cc` afin de bien faire prendre au lecteur la mesure des difficultés rencontrées. De plus, nous devons ajouter que la difficulté principale n'apparaît plus dans le code final car c'est la mise au point ! Le problème est de manipuler la pile de C. Si quelque chose se passe mal, immanquablement on va être tenté d'examiner cette pile. On a alors principalement deux solutions, utiliser un débogueur symbolique (comme `gdb`) ou bien, de façon plus rustique, placer des affichages dans le code de `call/cc`. Malheureusement ces deux possibilités échouent pour la même raison : utiliser un débogueur ou rajouter des appels de fonctions (appels à `printf` par exemple) changent l'aspect

```
 1:  {
 2:     static obj_t  stack;
 3:     static obj_t  jmpbuf;
 4:     static obj_t  s_value;
 5:     static obj_t  s_proc;
 6:     static char  *stack_top;
 7:
 8:     stack        = PROCEDURE_ENV_REF( proc, 0 );
 9:     stack_top    = (char *)CREF( PROCEDURE_ENV_REF( proc, 1 ) );
10:     jmpbuf       = CREF( PROCEDURE_ENV_REF( proc, 2 ) );
11:     top_of_frame = (struct dframe *)CREF( PROCEDURE_ENV_REF( proc, 4 ) );
12:
13:     s_value      = value;
14:     s_proc       = proc;
15:
16:     /* on sauve la valeur de retour */
17:     PROCEDURE_ENV_SET( s_proc, 3, s_value );
18:
19:     /* on verifie que c'est bien une pile qu'on va restaurer */
20:     if( (!STACKP( stack )) || (!EQP( CREF( stack ), STACK( stack ).self )) )
21:        the_failure( c_constant_string_to_string( "apply_continuation" ),
22:                     c_constant_string_to_string( "not a continuation" ),
23:                     proc );
24:     else
25:     {
26:        /* on restaure la pile */
27:  #if( STACK_GROWS == DOWN )
28:        memcpy( (char *)stack_top,
29:                &(STACK( stack ).stack),
30:                CINT( STACK( stack ).size ));
31:  #else
32:        memcpy( (char *)stack_bottom,
33:                &(STACK( stack ).stack),
34:                CINT( STACK( stack ).size ) );
35:  #endif
36:        ...
37:     }
38:  }
```

Programme 2.16: re-installation d'une pile

```
 1:  (define cont 'nothing-yet)
 2:  (define keep 0)
 3:
 4:  (define (foo x)
 5:     (call/cc (lambda (f) (set! cont f)))
 6:     (if (< keep 3)
 7:         (begin
 8:            (print x)
 9:            (set! x (+ x 1))
10:            (set! keep (+ 1 keep)))
11:         (begin
12:            (print x)
13:            (set! cont (lambda (x) x))))
14:     (cont 4))
15:
16:  (foo 0)
```

Programme 2.17: retours multiples

de la pile (notamment il y aura de fortes chances que la pile soit plus grande en mode de débogage). Ainsi ce comportement en mode de débogage ne sera pas le même qu'en mode normal[16].

Indépendamment de C, l'utilisation d'une pile rend l'implantation de `call/cc` très difficile (des arguments peuvent être trouvés dans [MB93]). N'en déplaise à A. Appel [App92, App87, AS94] les machines sont toutes conçues pour utiliser des piles et nous ne pouvons nous empêcher de penser que c'est en les utilisant qu'on atteint les meilleures performances (cette idée n'est pas seulement fondée sur une intuition mais sur la lecture des travaux [DTM94, WLM92]). Par ailleurs, nous espérons que cette section aura clairement montré que `call/cc` et C ne font pas très bon ménage. Le lecteur pourra toutefois se demander si l'auteur de Bigloo n'a pas été empreint d'une légère mauvaise foi en choisissant systématiquement pour résoudre chacun de ces problèmes, une implantation dramatiquement inefficace. Nous répondrons simplement à cette remarque que nous n'avons jamais eu la possibilité de choisir plusieurs solutions à partir du moment ou nous souhaitions impérativement écrire du C portable. Par ailleurs, puisque nous sommes dans le monde des extrêmes, continuons le raisonnement jusqu'au bout. Puisque `call/cc` n'est pas implantable raisonnablement et que cette fonction coûte même si l'on ne s'en sert pas, supprimons-la de Bigloo ! C'est ce que nous avons partiellement choisi. Par défaut `call/cc` est inconnue de Bigloo (ou plus exactement, par défaut, Bigloo oublie cette fonction juste après en avoir pris connaissance). Pour pouvoir l'utiliser, il faut compiler les modules avec l'option `-callcc`. Il n'est cependant pas possible de supprimer simplement `call/cc` de Scheme. Puisque cette fonction de manipulation de contrôle permet de réaliser toutes les opérations permises par les autres fonctions classiques du même domaine (`block/return-from` et `catch/throw` en Common Lisp [Ste84], `try/raise` en Caml [W*al*.91, WL93], etc.), Scheme ne possède pas d'autre fonction de manipulation de contrôle. Ainsi nous proposons de lui ajouter la forme inspirée (anciennement) de Dylan [ACT92] `bind-exit`[17].

2.5 La forme spéciale `bind-exit`

Il n'est pas possible de supprimer de Scheme la fonction `call/cc` sans proposer un remplaçant permettant de réaliser des ruptures de contrôle ! La forme `bind-exit` permet ces ruptures. Du point de vue de l'exécution, cette forme n'exige pas de longues explications. C est connu pour ne pas implanter efficacement les opérateurs de rupture de contrôle. Malheureusement nous ne ferons que confirmer cette idée. Le seul moyen d'exprimer des saut non locaux en C est d'utiliser les fonctions `setjmp` et `longjmp`. Examinons leur utilisation sur la compilation du petit exemple du programme 2.18

```
 1:  (define (sum-of-integer-elements l)
 2:     (bind-exit (stop)
 3:                (let loop ((l l))
 4:                   (cond
 5:                      ((null? l)
 6:                       0)
 7:                      ((not (integer? (car l)))
 8:                       (stop #f))
 9:                      (else
10:                       (+ (car l) (loop (cdr l))))))))
```

Programme 2.18: Une classique utilisation d'échappements

La fonction `loop` de la ligne *3* n'est pas récursive terminale. Ainsi donc, elle sera compilée en une fonction C. Par là même, l'échappement `stop`, introduit ligne *2*, sera utilisé dans la fonction `loop` alors qu'il aura été défini dans la fonction `sum-of-integer-elements`. Il y aura donc un saut non local à effectuer et le couple `setjmp/longjmp` devra être utilisé. Le programme 2.19 contient le produit de compilation de la fonction `sum-of-integer-elements`.

[16] et c'est d'ailleurs toujours le mode de débogage qui fonctionne le mieux se rendant ainsi totalement inutile.

[17] Cette section quelque peu ironique ne doit pas laisser penser que nous souhaitons purement et simplement voir `call/cc` être éradiquée du prochain rapport Scheme. Nous souhaitons seulement mettre l'accent sur le fait que dans l'état actuel, elle est très difficile à implanter. Nous ne nions en aucun cas son intérêt comme fabuleux outil d'expérimentation.

```
 1:  static obj_t
 2:  sum_of_integer_elements( obj_t x )
 3:  {
 4:     return stop( x );
 5:  }
 6:
 7:  static obj_t
 8:  stop( obj_t x )
 9:  {
10:     jmp_buf stop;
11:     if( SETJMP( (JMP_BUF *)stop ))
12:        return __ContinueValue;
13:     else
14:        return loop( stop, x );
15:  }
16:
17:  static obj_t
18:  loop( jmp_buf stop, obj_t x )
19:  {
20:     if( NULLP( x ) )
21:        return BINT( 0 );
22:     else
23:     {
24:        obj_t obj;
25:        obj = CAR( x );
26:        if( INTEGERP( obj ) )
27:        {
28:           obj_t aux;
29:           aux = loop( stop, CDR( x ) );
30:           return ADD_I( CAR( x ) , aux );
31:        }
32:        else
33:        {
34:           __ContinueValue = BFALSE;
35:           LONGJMP( (JMP_BUF *)stop, (JMP_VAL)1 );
36:        }
37:     }
38:  }
```

Programme 2.19: Somme d'entiers avec rupture de contrôle

La forme `(bind-exit (stop) ...)` a donné lieu à la création d'une fonction `stop` (ligne *8* dans le résultat de compilation). L'argument du `bind-exit` est placé dans la variable globale `__ContinueValue`.

Bien que la fonction `stop` se contente d'invoquer `setjmp`, elle est indispensable. Lors du saut non local `longjmp`, le contexte du `setjmp` doit être complètement restauré. Malheureusement C n'apporte que des garanties minimum quant à la restauration effectuée lors d'un `longjmp` et notamment les variables locales ne le sont pas nécessairement. C'est pour contourner cet obstacle que nous créons systématiquement des fonctions quand un `setjmp` doit être émis par Bigloo. En faisant de la sorte, le `setjmp` est toujours en début de fonction (avant toutes déclarations locales) et puisqu'il n'y a plus de variables locales à restaurer lors du `longjmp`, le problème est résolu.

Nous parvenons donc de façon portable à implanter la forme spéciale `bind-exit`, mais l'efficacité de cette implantation est liée à celle de `setjmp/longjmp` qui bien souvent n'est pas très brillante. Ainsi, pour avoir une meilleure implantation de `bind-exit`, Bigloo réalise une analyse qui lui permet parfois de ne pas produire de saut non locaux quand l'échappement n'est utilisé que dans la portée lexicale du `bind-exit`. Reprenons l'exemple du programme 2.18. Puisque la fonction `loop` (ligne *3* du programme source) est récursive terminale (donc compilée en une boucle C) et que l'échappement `stop` n'est utilisé que dans la portée lexicale du `bind-exit` Bigloo n'utilise pas de `setjmp/longjmp` mais un saut local, à savoir un `goto`. Ainsi donc l'invocation de `stop` (ligne *8* du programme source) est compilée en un saut C (ligne *35* du produit de compilation). La variable globale `__ContinueValue` contient ici aussi le paramètre effectif de l'échappement.

```
 1:  static obj_t
 2:  SUM_OF_INTEGER_ELEMENTS( obj_t X )
 3:  {
 4:  _SUM_OF_INTEGER_ELEMENTS:
 5:
 6:     {
 7:        obj_t X2, ACC;
 8:        {
 9:           X2 = X;
10:           ACC = BINT( 0 );
11:           {
12:  _LOOP:
13:
14:              if( NULLP( X2 ) )
15:              {
16:                 __ContinueValue = BINT( 0 );
17:                 goto _STOP;
18:              }
19:              else
20:              {
21:                 obj_t OBJ;
22:                 OBJ = CAR( X2 );
23:                 if( INTEGERP( OBJ ) )
24:                 {
25:                    obj_t AUX;
26:                    AUX = ADD_I( CAR( X2 ),
27:                                 ACC );
28:                    {
29:                       obj_t X3;
30:                       X3 = X3;
31:                       {
32:                          ACC = AUX;
33:                          X2 = CDR( X3 );
34:                          goto _LOOP;
35:                       }
36:                    }
37:                 }
38:                 else
39:                 {
40:                    __ContinueValue = BFALSE;
41:                    goto _STOP;
42:                 }
43:              }
44:           }
45:        }
46:     }
47:  _STOP:
48:     return __ContinueValue;
49:  }
```

Programme 2.20: Un échappement lexical: Le produit de compilation

Hélas, cette optimisation ne s'applique que dans des cas assez rares et donc `bind-exit` conserve des performances très moyennes. Si ce n'est pas trop gênant en Scheme, car le style de programmation n'est pas d'utiliser abondamment des échappements, ça l'est en revanche beaucoup plus en ML (ou plus particulièrement en Caml [W*al.*91, WL93]), où les exceptions sont monnaie courante.[18]

2.6 C est-il un bon langage cible?

Cette question a déjà été posée dans le chapitre d'introduction. Nous avions alors apporté quelques éléments de réponse mais les arguments invoqués étaient assez généraux. Ils relevaient d'orientations méthodologiques et de principes (comme par exemple l'acceptation d'une éventuelle perte d'efficacité au profit d'une plus grande portabilité). Maintenant que nous avons expliqué comment fonctionne la bibliothèque d'exécution pour C, nous sommes plus à même d'évaluer les difficultés et les avantages que présente ce langage quand il est choisi comme cible.

L'argumentation ne va porter ici que sur l'implantation du corps principal de Scheme. Il ne s'agit donc en aucun cas de la réponse définitive à la question. Il existe d'autres éléments de réponse importants liés à la bibliothèque d'exécution : citons la facilité d'implanter une interface complète entre Scheme et le monde extérieur (voir chapitre 4).

Intuitivement, choisir pour cible un langage de haut niveau[19] entraîne une perte d'efficacité. Cette perte s'imagine facilement par le fait que le langage source et le langage cible n'étant pas de même nature, il n'existera pas forcément d'équivalent de chaque construction du premier dans le second. Les absences devront donc être implantées par un mécanisme plus ou moins lourd et performant dans le second. C présente cette particularité qu'on peut y faire presque tout. C'est malheureusement à cause de ce *presque* que des difficultés se présentent. Afin de le disculper là où il est innocent, nous allons comparer la bibliothèque d'exécution avec ce qu'elle pourrait être si la cible était un langage d'assemblage.

[18] Les fonctions `setjmp/longjmp` ont une efficacité qui dépend du type de machine. Sur celles équipées de processeurs Sparc leur inefficacité est dramatique (due encore une fois aux fenêtres de registres utilisées par ce processeur). Ainsi, pour avoir un `try/raise` ayant des performances décentes, nous avons dû écrire des remplaçants de `setjmp` et `longjmp` en assembleur. Nous les verrons dans le chapitre consacré à la passe de pré-compilation (*front-end*) Caml.

[19] Nous n'entrerons pas dans la polémique qui consiste à savoir si C est ou n'est pas de haut niveau.

2.6.1 Langage C contre langage d'assemblage

Étudions, au cas par cas, les différences entre la bibliothèque d'exécution décrite et ce qu'elle pourrait être si le langage cible était un assembleur.

Les opérations d'étiquetage et d'encapsulation C possède une arithmétique sur les pointeurs qui permet de réaliser presque toutes les opérations désirées. Ce langage a d'ailleurs été conçu pour remplacer l'assembleur dans l'écriture de programmes nécessitant ces opérations. Les compilateurs les optimisent très bien et c'est donc sans surprise qu'elles sont très rapidement performantes. Sur ce point il n'y a pas de différence entre lui et l'assembleur. Ici le choix de C n'est pas pénalisant.

Le *GC* Si le langage C permet de tout faire, il n'en n'est pas pour autant dépourvu de contraintes. Sa bibliothèque d'exécution ne comporte pas de *GC*. En plus, les compilateurs produisent du code tel qu'il n'est pas facile d'en ajouter un (parce que les compilateurs acceptent que les pointeurs sur les objets soient déplacés, ou bien parce qu'il n'y a aucune norme sur le mode de fonctionnement de la pile du langage. Pour de plus amples explications voir [BW88, Bar89a]). C'est toutefois possible mais à un coût difficilement évaluable. Les *GC* modernes sont interdits à C mais si l'on s'en réfère à la section 2.1 et à l'article de Zorn [Zor90], ce n'est pas ici que l'écart avec les assembleurs est le plus important. De plus, les programmes très impératifs (avec beaucoup d'effets de bords sur des variables, par exemple) seront même désavantagés avec des *GC* copiants.

Les appels fonctionnels La principale faiblesse des compilateurs de langages fonctionnels utilisant C comme cible est probablement l'inefficacité (relative) de leurs appels fonctionnels. Elle apparaît sur les points suivants :

1. L'impossibilité de garantir totalement que les récursions terminales ne consommeront pas de ressource (pile).
2. Pour la plupart des compilateurs, sur la majorité des machines, les paramètres des fonctions ne sont pas placés dans des registres (mais sur la pile) alors que D. Kranz montre dans sa thèse [Kra88] que pour Scheme, cette stratégie est préférable.
3. Certaines optimisations classiques faites par les compilateurs qui produisent du code natif sont impossibles, car C n'offre pas d'outils pour utiliser sa pile. Examinons sur un exemple l'optimisation la plus fréquente. Voici une petite fonction :

```
(define (foo x)
   (letrec ((gee (lambda (a) (if ...
                                 ...
                                 (+fx a (gee x))))))
      gee))
```

Étudions la compilation de la fonction `gee`. Puisqu'elle s'enfuit, elle doit être invoquée par le protocole utilisant le tas. Le produit de compilation est :

```
 1:  static obj_t
 2:  GEE( obj_t ENV, obj_t A )
 3:  {
 4:  _GEE:
 5:
 6:     {
 7:        obj_t X;
 8:
 9:        X = PROCEDURE_ENV_REF( ENV, 0 );
10:        if( ... )
11:           ...
12:        else
13:        {
14:           obj_t Z2;
15:           Z2 = GEE( ENV, X );
16:           return ADD_I( A, Z2 );
17:        }
18:     }
19:  }
```

L'appel auto-récursif de la ligne *15* n'est pas terminal, il utilise la pile C. La fonction `gee` est ré-invoquée en lui fournissant son environnement comme premier argument. Cette invocation va conduire à l'exécution de la ligne *9* qui déréférence l'environnement pour trouver la valeur de `x`, bien que cette variable ait déjà été obtenue. L'optimisation consiste à ne pas empiler l'environnement et à ne pas se brancher au début de la fonction mais juste après la collecte des variables libres. Ceci n'est pas faisable en C. En revanche, ce qui est permis, c'est d'utiliser une troisième fonction et de produire le code :

```
 1:  static obj_t
 2:  GEE( obj_t ENV, obj_t A )
 3:  {
 4:  _GEE:
 5:
 6:     {
 7:        obj_t X;
 8:
 9:        X = PROCEDURE_ENV_REF( ENV, 0 );
10:        return GEE_LIFTED( X, A );
11:     }
12:  }
13:
14:  static obj_t
15:  GEE_LIFTED( obj_t X, obj_t A )
16:  {
17:     if( ... )
18:        ...
19:     else
20:     {
21:        obj_t Z2;
22:        Z2 = GEE_LIFTED( X, X );
23:        return ADD_I( A, Z2 );
24:     }
25:  }
```

Cette solution, si elle réalise l'optimisation, est plus lourde et moins efficace que ce qui pourrait être obtenu en produisant du code natif.

Des trois points mentionnés, ce sont nettement les deux premiers qui sont les plus importants. Nous devons donc faire une remarque essentielle : *ce qui est relevé ici comme étant source d'inefficacité n'est pas dû au langage C mais aux implantations actuelles de ses compilateurs.* Rien ne leur interdit, a priori, de réaliser les optimisations des récursions terminales (du moins dans de nombreux cas) et d'allocations des paramètres dans des registres. Ces deux obstacles principaux à l'efficacité peuvent donc, en quelque sorte, être qualifiés de conjoncturels. Si le langage C se développe en tant que langage cible pour des compilations de langages de plus haut niveau, il est envisageable que les implanteurs de compilateurs songent à les incorporer à leurs nouvelles réalisations. Cette situation est quelque peu paradoxale car la principale raison d'inefficacité quand C est le langage cible, ne peut être imputée au langage lui-même mais à ses implantations. Il ne nous reste donc qu'à déplorer qu'à ce jour aucun compilateur (à notre connaissance) ne fasse ces optimisations, car leur absence peut à elle seule remettre en cause le choix de ce langage comme cible.

La fonction `call_cc` Bien entendu les performances de l'implantation de la fonction `call_cc` ne sont pas aussi cruciales que l'efficacité des protocoles d'appels fonctionnels, mais néanmoins les difficultés rencontrées ici nous font penser que nous sommes aux limites des possibilités de C. La phrase presque proverbiale, *C peut tout faire*, est presque prise en défaut ! La fonction Scheme `call_cc` n'est pratiquement pas implantable dans ce langage. Il est très difficile de l'implanter efficacement avec un langage utilisant une pile [MB93], mais quand, en plus, ce langage n'offre pas de possibilité de la manipuler, les problèmes deviennent très ardus ! Les problèmes dûs à C sont liés à cette absence. De plus `call_cc` est très loin d'avoir un équivalent en C. Ce ne sont pas les maigres `setjmp` et `longjmp` qui comblent cet écart. Cessons les griefs (qui s'appliquent aussi bien à C pour ses faiblesses qu'à Scheme pour la trop grande richesse de cette fonction) et résumons-nous en disant que `call_cc` est un problème de taille pour les implanteurs Scheme et en particulier ceux qui compilent vers C. Si les performances de cette

fonction sont importantes aux yeux du concepteur, il sera alors préférable de renoncer à C.

2.6.2 C orthodoxe/C hétérodoxe

Bigloo associe à chaque fonction Scheme une fonction ou une boucle C. De façon similaire, à chaque variable Scheme correspond une variable C. Plus généralement, les constructions Scheme qui possèdent un équivalent en C sont compilées par des projections de Scheme vers C. L'intérêt principal de cette méthode est d'éviter les encodages des constructions Scheme en C. Le code C produit conserve alors la même structure que le programme source et utilise au maximum le contrôle de C : il a des allures de code écrit *à la main* (il utilise des fonctions et des variables C, il utilise des instructions plus que des expressions, etc.). Nous appelons le style de code produit du *C orthodoxe*. Plusieurs compilateurs choisissent ce style. Citons, en plus de Bigloo, Scheme-to-C [Bar89b], Camlot [Cri92] et d2c [SK93][20].

Par opposition, nous nommons les autres styles de code C produit, du C hétérodoxe. Parmi les compilateurs choisissant ce style, il existe plusieurs catégories.

La première catégorie contient des compilateurs comme sml2c [TAL91] et plus récemment, le compilateur de H.Baker [Bak94]. Tous deux produisent du C utilisant une technique dite à base de *trampolines*. Leurs principes diffèrent toutefois légèrement. Celui du compilateur sml2c est le suivant : une fonction principale, le trampoline, appelle des fonctions qui, chacune, lui retourne l'adresse de la prochaine fonction à invoquer. Bien sûr, il n'est plus possible d'utiliser la pile C pour passer les arguments des fonctions, ils doivent être placés dans des zones statiques. Ici, les possibilités de contrôle de C ne sont presque plus utilisées. L'intérêt majeur de cette méthode est de garantir que les récursions terminales ne consommeront pas de ressources, ce qui n'est pas le cas avec du C orthodoxe. La technique de H.Baker est différente. Le principe est que les fonctions ne retournent jamais. Il s'agit d'une cascade en avant. Cela présente l'avantage, par rapport à sml2c, de permettre d'utiliser la pile de C.

Il existe une troisième catégorie de compilateurs : ceux qui utilisent C comme un langage d'assemblage [FMRW94, Hof93, YH88]. Ils n'utilisent pas ou presque pas les structures de contrôle de ce langage. Par exemple, ces compilateurs produisent du code qui n'utilise pas la pile d'exécution de C. Ils utilisent des variables C comme des registres et à ce titre, ils réalisent une allocation de registres comme s'ils produisaient du langage d'assemblage. S'il est vrai que ces compilateurs en choisissant de produire du C sont plus portables qu'en produisant du langage d'assemblage, il nous semble néanmoins que la technique employée n'offre pas les avantages du C orthodoxe, car le compilateur C n'optimisera pas leur code et produira du code tout juste moyen.

Enfin, une dernière classe de compilateurs se situe à mi-chemin entre le C orthodoxe et le C hétérodoxe. C'est le cas, par exemple, du compilateur CeML [Cha92b, Cha92a]. Celui-ci utilise certaines des structures de contrôle de C mais aussi des constructions idoines. CeML utilise la pile C mais aussi une pile propre pour empiler les racines du *GC*. Il faut d'ailleurs noter que cette technique présente un intérêt indiscutable. Si elle sacrifie un peu à la performance (car il y a gestion de deux piles lors des exécutions), le *GC* est sûr (il n'y a plus de risque de ramasser illégalement des racines).

Chaque solution présente des avantages et des inconvénients et bien souvent les avantages de l'un sont les inconvénients de l'autre. Mais il nous semble que quels que soient les atouts du C hétérodoxe la première solution est préférable. Produire du C orthodoxe (c'est-à-dire produire du C qui ressemble le plus possible à du C écrit à la main) offre trois possibilités :

- Puisque le C produit ressemble à du C écrit à la main, tous les outils existants d'aide à la programmation C vont pouvoir être utilisés (débogueur, profileur, ...).
- Puisque le protocole d'appel est celui de C, l'interface externe du langage de haut niveau est simple à réaliser.
- Comme le bon sens le dicte, un compilateur ne réalise une optimisation que si elle a des chances de pouvoir s'appliquer suffisamment ! Pour cette raison, un compilateur (quel que soit le langage) compilera mieux ceux utilisant des schémas classiques que des programmes utilisant des techniques rarement

[20] Dans le chapitre 10 qui décrit la compilation de Caml-light, nous utilisons le terme de *projection naturelle* pour désigner un schéma de compilation qui minimise les encodages en tirant profit des similitudes des constructions du langage source et du langage cible.

employées. Il est donc préférable de produire du code qui ressemble le plus possible à celui que le compilateur s'attend à trouver. Ainsi, les performances de notre compilateur suivront celles du compilateur C, progressant avec lui. Le code produit par les compilateurs C usuels sera uniformément « assez bon », alors qu'au contraire, les C hétérodoxes risquent d'être compilés de façon très inégale rendant ainsi les performances peu prédictibles.

Finalement les avantages du C orthodoxe sont plus importants que ses inconvénients. C'est pourquoi nous l'avons choisi pour Bigloo.

2.7 Récapitulatif

Ce chapitre contient un exposé des choix fondamentaux que nous avons pris pour la bibliothèque d'exécution de Bigloo. Ces choix s'étendent du *GC* aux protocoles d'appels, en passant par la représentation des différents types de données en Scheme.

Nous avons montré que notre approche qui a consisté à concevoir notre compilateur en ignorant parfaitement le *GC*, ne nous pénalise pas, car les temps obtenus avec ou sans *GC* n'ont pas plus de 10 % de différence. Pour la représentation des objets, nous avons examiné systématiquement les solutions qui se présentaient. Pour motiver nos choix nous avons, dans chaque cas, ou bien fait des mesures de performances, ou bien examiné les codes en langage d'assemblage produit par les compilateurs C. Nous avons comparé les différents styles de C que produisent les compilateurs de langages fonctionnels qui le choisissent comme cible. Les éléments de réponse apportés dans ce chapitre à la question *faut-il compiler vers C ou faut-il compiler vers du code natif ?* montrent que le choix, s'il augmente la portabilité, n'interdit pas de bonnes performances.

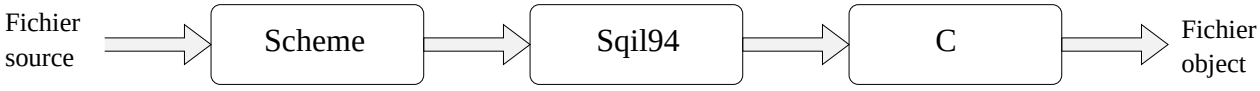
Chapitre 3

L'architecture de Bigloo

Bigloo rassemble toute une série d'optimisations et d'analyses. Avant de les présenter dans les chapitres suivants, nous allons exposer l'architecture du compilateur. Le compilateur n'engendre pas directement du C en partant de code Scheme. Il utilise un langage intermédiaire que nous allons présenter dans la première section. Nous ne pouvons pas exposer l'architecture du compilateur sans présenter sa syntaxe abstraite. Une section lui sera consacrée. Enfin, nous montrerons toutes les passes qui forment le compilateur.

3.1 Le langage intermédiaire Sqil94

Partant d'un code source Scheme, Bigloo ne produit pas directement du code C car ces langages sont trop éloignés l'un de l'autre. Bigloo commence par réaliser diverses études et transformations du code source qui le conduisent à l'obtention d'un code intermédiaire écrit dans un langage nommé Sqil94. Une fois que ce code est obtenu la compilation vers C proprement dite commence. Les optimisations sont réalisées avant le code Sqil, elles sont toutes de haut niveau et totalement indépendantes du langage destination. Cette décomposition en deux étapes impose une répartition nette des tâches. La compilation de Bigloo d'un point de vue macroscopique est donc[1]:



Même si Sqil est décrit par N. Séniak dans [Sén91], nous allons en faire ici une présentation rapide car, d'une part nous lui avons apporté quelques petites retouches (ce qui justifie le sobriquet affectueux de Sqil94), d'autre part il constitue l'ossature de Bigloo. Bien sûr, ce langage ne possède pas "d'incarnation" dans le monde réel, il n'existe que sous forme d'une syntaxe abstraite, mais pour communiquer une intuition de ce qu'il est, nous donnons ici sa "pseudo-syntaxe concrète".

Les mots clés de la grammaire, empruntés à Common Lisp [Ste84], ne nécessitent pas d'explication car ils sont à prendre dans le même sens. Par ailleurs $\mathcal{VAL}$ désigne les valeurs du langage (les nombres, les chaînes, les vecteurs, les listes, ...) et `failure` est une construction qui évalue ses arguments et termine l'exécution courante.

Sqil est un langage qui se situe à mi-chemin entre Scheme et un langage de moins haut niveau comme C. Il ressemble à un langage évolué mais n'en est pas un, car :

- Il ne possède pas de gestion automatique d'erreurs.
- Il est bivalué, c'est-à-dire que les fonctions sont dans un espace différent des autres valeurs.
- Les fonctions sont toutes d'arité fixe.
- Les fonctions ne sont pas des valeurs de première classe.

[1] Les autres langages sources de Bigloo (ML ou Meroon [Que93]) utilisent d'autres étapes qui interviennent en amont (*front-end*) de la compilation présentée ici.

Catégorie syntaxique :

V	$\in$	VarId	(Identifcateurs de variables)
F	$\in$	FunId	(Identificateurs de fonctions)
C	$\in$	FunId	(Identificateurs de continuations)
$\mathcal{C}$	$\in$	Cnst	(Valeurs constantes)
Π	$\in$	Prgm	(Programme)
Γ	$\in$	Def	(Définitions)
Σ	$\in$	Exp	(Expressions)

Syntaxe :

Π	::=		$\Gamma \cdots \Gamma$
Γ	::=		(defun F ($V \cdots V$) Σ)
		|	(defvar V)
Σ	::=		V
		|	(quote $\mathcal{VAL}$)
		|	(setq V Σ)
		|	(function V)
		|	(if Σ Σ Σ)
		|	(failure Σ Σ Σ)
		|	(progn $\Sigma \cdots \Sigma$)
		|	(case Σ ($\mathcal{C}$ Σ) $\cdots$ ($\mathcal{C}$ Σ))
		|	(let ((V Σ) $\cdots$ (V Σ)) Σ)
		|	(labels ((F ($V \cdots V$) Σ) $\cdots$ (F ($V \cdots V$) Σ)) Σ)
		|	(block C Σ)
		|	(return-from C Σ)
		|	(funcall Σ $\Sigma \cdots \Sigma$)
		|	(apply Σ $\Sigma \cdots \Sigma$)
		|	(F $\Sigma \cdots \Sigma$)

FIG. 3.1 - La pseudo syntaxe concrète de Sqil94

3.2 La syntaxe abstraite de Bigloo

La syntaxe abstraite de Bigloo ressemble de très près à la pseudo-syntaxe de Sqil (figure 3.1). Elle est constituée d'arbres Scheme (donc de listes). Les nœuds de ces arbres sont des symboles (représentant les opérateurs) et les feuilles sont des structures (pour représenter les variables) ou bien les constantes elles-mêmes. Les programmes 3.2 et 3.1 présentent ces structures. Elles sont consacrées respectivement aux variables globales et locales. Leurs très petites tailles s'expliquent par le fait qu'elles ne retiennent que les informations globales à toute la compilation. Évidemment chaque passe nécessite de mémoriser plus d'informations sur les variables ; pour cela elle alloue une structure supplémentaire qui va être rattachée aux variables au moyen des champs `info`. Le squelette des passes sera donc :

```
(define-struct local
   name          ;; symbol      : le nom de la variable sans pretty-print
   key           ;; integer     : une cle d'identification
   class         ;; { variable, function, return }
   value         ;;             : une eventuelle info sur la valeur
   info          ;;             : ce champ est utilise par toutes les passes
   access        ;; { read, write }
   type)         ;; type        : le type des variables (defaut: *bobj*)
```

Programme 3.1: Les variables locales dans l'arbre de syntaxe

```
(define-struct global
   name          ;; symbol      : le nom de la variable sans pretty-print
   module        ;; symbol      : le nom du module d'appartenance.
   c-name        ;; string      : le nom C de la variable.
   import        ;; { static, export, import, top-level, foreign }
   class         ;; { variable, function, foreign }
   library?      ;; { t, f }    : est-ce une fonction de la bibliothèque ?
   value         ;;             : une eventuelle info sur la valeur
   pragma        ;;             : des infos pour les fonctions de la lib.
   info)         ;;             : ce champ est utilise par toutes les passes
```

Programme 3.2: Les variables globales dans l'arbre de syntaxe

1. *Pour chaque variable du programme allouer la structure nécessaire à la passe. Cette nouvelle structure est rattachée à la variable par le champ info.*

2. *Faire le traitement de la passe.*

3. *Pour chaque variable du programme, défaire le pointeur* `info`.

Les deux grands avantages de ce principe sont :

1. Une diminution de la taille de l'arbre et un meilleur comportement vis à vis du *GC*. Cela est évident car d'une passe sur l'autre le *GC* peut glaner une grande partie de l'arbre.

2. D'un point de vue méthodologique, ce principe renforce la séparation entre les passes. En effet, il est difficile de les faire se chevaucher puisqu'elles n'ont en commun que des informations minimales.

Sans décrire en détail l'utilisation de tous les champs des structures `local` et `global`, précisons que leur champ `value` pointe sur d'autres structures quand la variable est une fonction, un échappement ou une variable externe[2]. Ces structures sont données dans le programme 3.3 Elles contiennent des informations globales utilisées par toute la compilation.

```
(define-struct function
   inline?       ;; { t, f }    : est-ce une inline ?
   property      ;; property*   : une liste de proprietes
   arity         ;; integer     : l'arite de la fonction
   args          ;; local*      : les arguments d'une fonction
   body          ;; s-exp       : le corps.
   info          ;;             : des infos dependants des passes.
   escape?       ;; { t, f }    : cette fonction s'enfuie-t-elle ?
   the-closure   ;; variable    : la variable fermeture
   type-res      ;; type        : le type du resultat (defaut: *bobj*)
   invocations)  ;; integer     : le nombre d'invocation

(define-struct return
   args          ;; local*      : les arguments d'un return
   body          ;; s-exp       : le corps.
   escape?       ;; { t, f }    : ce return s'enfuie-t-il ?
   the-continue) ;; variable    : la variable de continuation

(define-struct foreign
   class         ;; { function, variable, macro-function, macro-cnst }
   type)         ;; type        : le type de cette foreign
```

Programme 3.3: Les valeurs des variables

Nous avons illustré dans la figure 3.2 un arbre de syntaxe abstraite pour la fonction `(labels ((id (x) x)) id)`. Les partages des structures sont apparents.

Afin de fixer les idées nous donnons dans le programme 3.4 un exemple complet de fonction parcourant l'arbre de syntaxe. Bien entendu ce code est fictif et en particulier il n'a pas été extrait du code du compilateur

[2] une variable C.

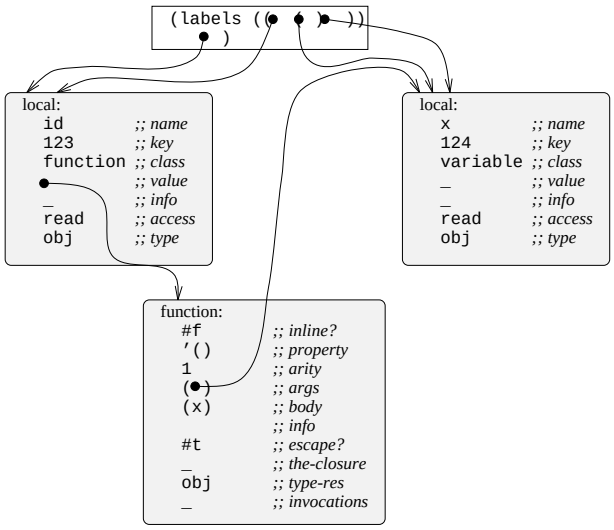


FIG. 3.2 - Un arbre de syntaxe abstraite

lui-même. Néanmoins, il illustre bien le style qu'on rencontrera lors d'une aventure dans la lecture des sources de Bigloo. Comme on peut le voir l'arbre est réellement composé de listes, la fonction de parcours procède par filtrage. Tous les nœuds sont des symboles sauf pour les variables qui sont des structures (testées lignes 9 et 11).

Ce code est par ailleurs révélateur du choix qui a été fait pour Bigloo : plutôt que d'allouer un nouvel arbre à chaque passe on préférera toujours faire des effets de bords sur celui alloué au début de la compilation (on voit apparaître cette caractéristique pour la première fois en ligne 17). Même les passes qui réorganisent totalement l'arbre le font sur place (comme la passe `hoist` par exemple). Nous avons choisi cette technique pour réduire au maximum le temps d'un parcours de l'arbre. Dans les faits, ce temps est toujours négligeable devant le temps du traitement qui y est effectué. Ceci nous a permis de choisir une architecture globale qui favorise le nombre de passes de compilation (la version actuelle ne compte pas moins de 21 passes). Empiriquement, cette approche où l'on multiplie le nombre de passes nous semble être plus performante que l'approche opposée. Celle-ci consiste à faire effectuer plusieurs traitements à chaque parcours de l'arbre ou pire, à effectuer des calculs lors d'une passe qui ne seront utilisés que dans les passes suivantes car :

Elle conduit à du code plus facile à maintenir : les calculs sont plus nettement isolés et surtout, la modification d'une passe ne nécessite pas de connaître les passes suivantes.

Elle permet la réalisation d'un compilateur plus rapide : puisque chaque calcul global sur l'arbre est isolé dans une passe, il sera beaucoup plus facile de voir où et quand il a été fait ; on pourra donc éviter les calculs redondants[3].

Elle conduit à du code plus facile à faire évoluer : puisque les passes n'ont pas (ou presque) de liens entre elles, il est facile d'en supprimer, d'en déplacer ou même d'en ajouter ! Ainsi, ajouter une nouvelle optimisation n'impose pas de se souvenir (ou de connaître) la structure complète du compilateur.

3.3 Toutes les passes de Bigloo

On peut repérer dans le compilateur plusieurs blocs fondamentaux correspondant chacun à une transformation majeure de l'arbre de syntaxe. La figure 3.3 donne une séparation possible.

Nous allons détailler dans cette section chacun de ces blocs.

[3] Ceci peut sembler anodin, mais sur un gros programme complexe comme un compilateur, il n'est pas toujours facile de se souvenir qu'un calcul a déjà été effectué, surtout quand il est perdu au milieu d'autres.

```
(define (walk exp)
   (match-case exp
;*--- nil ------------------------------*/
      (()
       exp)
;*--- atom -----------------------------*/
      ((atom ?-)
       (cond
          ((global? exp)
           ...)
          ((local? exp)
           ...)
          (else
           exp)))
;*--- setq -----------------------------*/
      ((setq . ?-)
       (set-car! (cddr exp)
                 (walk (caddr exp)))
       exp)
;*--- function -------------------------*/
      ((function ?var)
       exp)
;*--- quote ----------------------------*/
      ((quote ?-)
       (walk-quote (cadr exp)))
;*--- failure --------------------------*/
      ((failure . ?-)
       (set-car! (cdr exp)
                 (walk (cadr exp)))
       (set-car! (cddr exp)
                 (walk (caddr exp)))
       (set-car! (cdddr exp)
                 (walk (cadddr exp)))
       exp)
;*--- if -------------------------------*/
      ((if . ?-)
       (set-car! (cdr exp)
                 (walk (cadr exp)))
       (set-car! (cddr exp)
                 (walk (caddr exp)))
       (set-car! (cdddr exp)
                 (walk (cadddr exp)))
       exp)))
;*--- case -----------------------------*/
      ((case ?test . ?clauses)
       (set-car! (cdr exp) (walk test))
       (let loop ((hook clauses))
          (if (null? hook)
              exp
              (begin
                 (set-car!
                  (cdr (car hook))
                  (walk (cadr (car hook))))
                 (loop (cdr hook))))))
;*--- progn ----------------------------*/
      ((progn . ?body)
       (let loop ((hook body))
          (if (null? (cdr hook))
              (begin
                 (set-car!
                  hook
                  (walk (car hook)))
                 exp)
              (begin
                 (set-car!
                  hook
                  (walk (car hook)))
                 (loop (cdr hook))))))
;*--- let ------------------------------*/
      ((let . ?-)
       (let loop ((hook (cadr exp)))
          (if (null? hook)
              (begin
                 (set-car!
                  (cddr exp)
                  (walk (caddr exp)))
                 exp)
              (begin
                 (set-car!
                  (cdar hook)
                  (walk (cadr (car hook))))
                 (loop (cdr hook))))))
;*--- labels ---------------------------*/
      ((labels . ?-)
       (let loop ((hook (cadr exp)))
          (if (null? hook)
              (begin
                 (set-car!
                  (cddr exp)
                  (walk (caddr exp)))
                 exp)
              (begin
                 (set-car!
                  (cddar hook)
                  (walk (caddr (car hook))))
                 (loop (cdr hook))))))
;*--- block ----------------------------*/
      ((block . ?-)
       (set-car! (cddr exp)
                 (walk (caddr exp)))
       exp)
;*--- return-from ----------------------*/
      ((return-from . ?-)
       (set-car! (cddr exp)
                 (walk (caddr exp)))
       exp)
;*--- apply ----------------------------*/
      ((apply . ?-)
       (let loop ((hook (cdr exp)))
          (if (null? hook)
              exp
              (begin
                 (set-car!
                  hook
                  (walk (car hook)))
                 (loop (cdr hook))))))
;*--- funcall --------------------------*/
      ((funcall . ?-)
       (let loop ((hook (cdr exp)))
          (if (null? hook)
              exp
              (begin
                 (set-car!
                  hook
                  (walk (car hook)))
                 (loop (cdr hook))))))
;*--- application ----------------------*/
      (else
       (let loop ((hook exp))
          (if (null? hook)
              exp
              (begin
                 (set-car!
                  hook
                  (walk (car hook)))
                 (loop (cdr hook)))))))
```

Programme 3.4: Une fonction de parcours

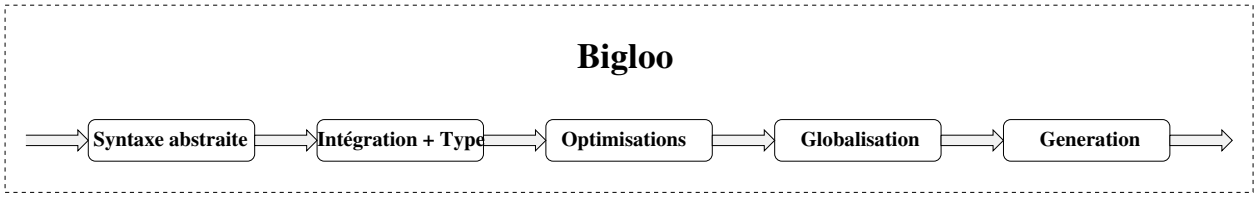


FIG. 3.3 - L'architecture globale de Bigloo

3.3.1 Construction de l'arbre de syntaxe abstraite

Puisque nous allons faire ici une étude chronologique (les passes vont être présentées dans l'ordre où elles se succèdent lors d'une compilation) il est naturel que nous commencions par examiner le bloc qui produit l'arbre de syntaxe. La figure 3.4 présente les passes qui le constituent :

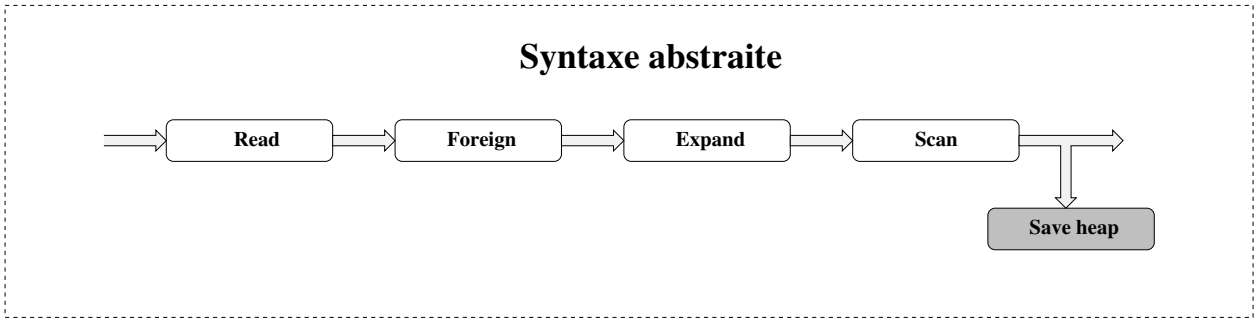


FIG. 3.4 - La construction de l'arbre de syntaxe abstraite

Ce premier bloc (figure 3.4) contient quatre passes plus la sauvegarde de l'image mémoire (c'est parce que ce n'est pas un parcours de l'arbre de syntaxe abstraite que le rectangle symbolisant cette sauvegarde (**save heap**) possède un fond très grisé). Voici les tâches dont s'acquittent ces passes :

Read La syntaxe de Scheme est tellement simple qu'elle ne requiert quasiment aucune analyse syntaxique. Une analyse lexicale est suffisante. Le lecteur Scheme lit le programme source et construit un arbre formé des S-expressions lues. La déclaration de module est analysée, les modules importés sont lus.

Foreign Les clauses **foreign** (voir chapitre 4) ont été lues, Bigloo connaît donc l'intégralité des types externes utilisés dans le module à compiler. Il faut alors construire les accesseurs et les convertisseurs externes. Cette passe se charge de ce travail.

Expand Avant de construire l'arbre de syntaxe abstraite, on applique une passe de macro expansion. Le compilateur utilise ses propres macros plus celles que l'utilisateur introduit. Nous profitons de cette macro expansion pour effectuer quelques vérification syntaxiques et quelques optimisations évidentes.

Scan Cette passe est la construction de l'arbre de syntaxe abstraite. On termine les quelques vérifications syntaxiques indispensables. On vérifie les concordances entre les déclarations du module et les définitions parcourues. Après cette passe, Bigloo détient dans ses tables, toutes les déclarations de variables (exportées, locales, importées) et leur signatures.

Save heap Après la passe **Scan**, Bigloo possède des informations sur toutes les variables importées. Quand il compile la bibliothèque d'exécution, il produit un fichier (nommé *tas*) contenant toutes les informations sur toutes les fonctions et variables de la bibliothèque. Ainsi, quand Bigloo a en charge la compilation d'un fichier, il commence par charger ce fichier *tas* pour récupérer les informations sur la bibliothèque.

3.3.2 Intégration et typage

Une fois que l'arbre de syntaxe abstraite est construit, Bigloo commence réellement son travail de compilation.

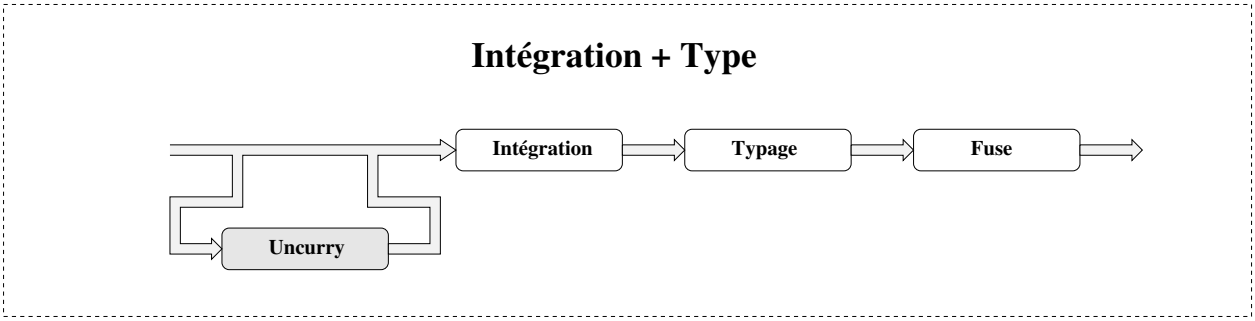


FIG. 3.5 - Le typage et l'inlining

Le bloc **Intégration + Typage** (figure 3.5) contient d'ailleurs la première optimisation : **Uncurry**. Les optimisations sont symbolisées par des boîtes dont le fond est légèrement grisé, elles ne se situent jamais sur l'axe principal mais de part et d'autre. En plus de cette optimisation (passe donc facultative) le bloc contient trois autres passes.

Uncurry Cette passe réalise une *décurrification* des fonctions currifiées. En Scheme il est peu fréquent d'en rencontrer mais en revanche en ML elles sont monnaie courante. Cette optimisation a été ajoutée au moment où le *front-end* Camloo (voir le chapitre 10) a été réalisé.

Inlining Cette passe se charge de remplacer des appels fonctionnels par le corps des fonctions appelées. Toutes les fonctions déclarées `inline` sont systématiquement remplacées. En plus, Bigloo, suivant certains critères expliqués ultérieurement, intègre des fonctions classiques. Cette passe pourrait être considérée comme une passe d'optimisation mais sa position très centrale dans le processus de compilation la rend presque obligatoire. Ainsi les appels aux primitives sont en partie traités par cette passe.

Typage C'est un peu par provocation que nous avons choisit le nom équivoque de **typage** pour cette passe ! Puisque Scheme est un langage typé dynamiquement, lors d'une exécution incorrecte, le contrôle doit être détourné et une erreur signalée. Cette passe sert à poser ces tests de type et à poser les conversions requises par d'éventuels appels à des fonctions externes. De plus cette passe se charge d'effectuer la transformation que Nitsan Séniak nomme multi-valuation. C'est-à-dire que les valeurs de Scheme (variables et fonctions) sont projetées dans des espaces de valeurs Sqil différents.

Fuse Cette petite passe sert à nettoyer le code. L'intégration (la passe **Inlining**) peut avoir rendu des définitions de fonctions inutiles (par exemple si une fonction est statique et qu'elle a été intégrée sur tous ses sites d'invocation et qu'elle n'est jamais utilisée en valeur). Cette passe se charge de supprimer leur définition de l'arbre de syntaxe abstraite.

3.3.3 Quelques optimisations

Exception faite de la passe **Assert**, ce bloc (figure 3.6) ne contient que des optimisations.

Assert Cette passe facultative n'est utilisée que si le compilateur est invoqué en mode de déboguage. Cette passe active les assertions placées par l'utilisateur dans le code source.

Effect L'exploitation des approximations de l'analyse de flot de contrôle nécessite de savoir quelles sont les fonctions qui réalisent des effets de bords. Cette passe fait cette étude en annotant les nœuds *fonctions* de l'arbre de syntaxe abstraite.

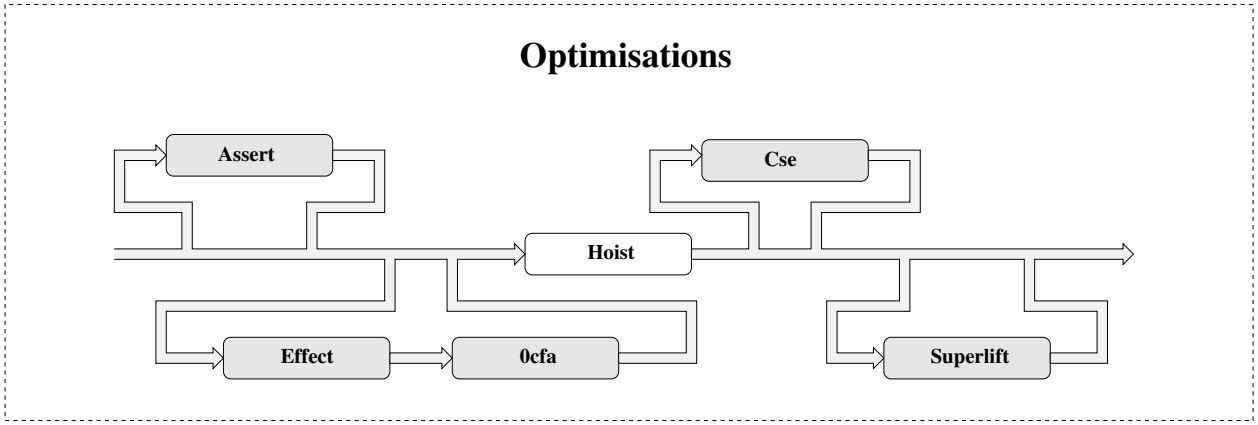


FIG. 3.6 - Quelques unes des optimisations

0cfa 0cfa signifie *0th-order Control Flow Analysis*. Il s'agit d'une analyse statique qui calcule des sur-ensembles des valeurs possibles des variables d'un programme. Elle est exploitée dans Bigloo pour réduire le nombre de tests de type et pour réduire le nombre de procédures allouées dans le tas.

Hoist Cette passe dont le nom complet devrait être *Failure hoisting* réorganise l'arbre de syntaxe abstraite de telle sorte que les tests débouchant sur des arrêts du contrôle soient "remontés" dans l'arbre de syntaxe. Le but de cette opération est de faciliter la détection et l'élimination de sous-expressions communes.

Cse Cse signifie *Common sub-expression elimination*. Cette optimisation consiste à supprimer les calculs redondant (ceux déjà effectués) d'un programme.

Superlift La transformation que réalise cette optimisation a été très rapidement vue dans la section 2.6.1. Elle a pour but d'éviter les déréférenciations redondantes des variables libres des fermetures.

3.3.4 Globalisation

La globalisation est une phase essentielle de la compilation. C'est en grande partie elle qui conditionne les performances d'ensemble du compilateur.

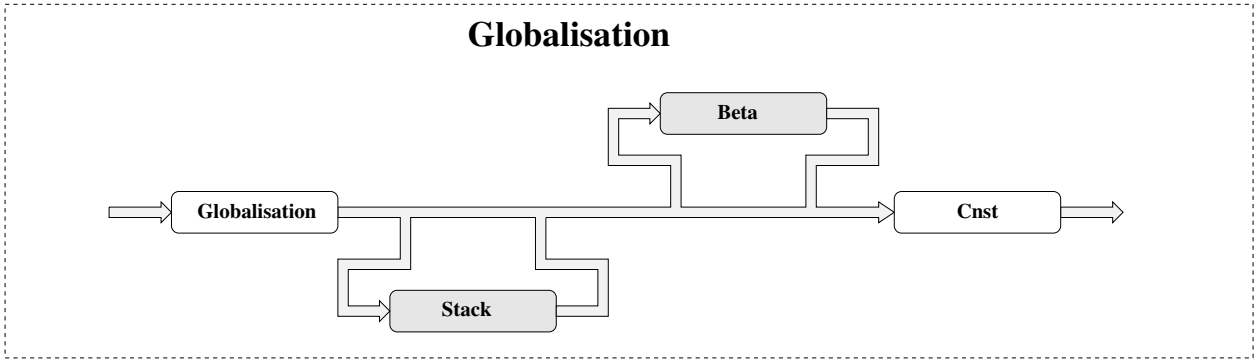


FIG. 3.7 - La globalisation et la compilation des constantes

Globalisation Cette passe transforme les définitions de fonctions locales en définitions globales. Les fonctions ainsi transformées sont celles qui sont utilisées en valeur (les fonctions qui s'enfuient). C'est après cette passe que l'arbre de syntaxe ressemble le plus à du Sqil94.

Stack Cette passe fait migrer certaines allocations du tas vers la pile.

Beta En partie à cause des passes précédentes (notamment la passe d'intégration (**Inlining**)), il peut arriver que l'arbre de syntaxe abstraite contienne des fragments constants non triviaux (par exemple des expressions arithmétiques qui ne portent que sur des constantes). Cette passe calcule (donc pendant la compilation) les valeurs de ces expressions constantes.

Cnst Toutes les passes précédentes ont introduit (en plus, bien sûr, de celle du programme initial) des constantes (chaînes de caractères, λ-expressions closes, symboles, ...). Cette passe prend en charge leur compilation.

3.3.5 La génération de code

La compilation touche à sa fin, il ne reste plus que le bout de la chaîne à effectuer, c'est-à-dire à produire le code destination.

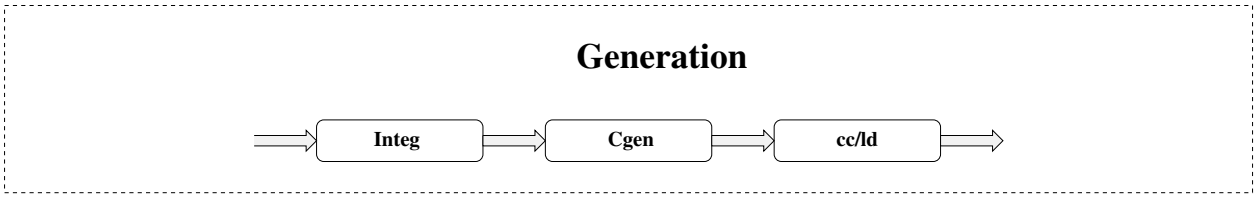


FIG. 3.8 - La generation de code

Integ Puisque dans sa version actuelle Bigloo produit du langage C, il faut finir la projection des constructions Scheme sur celles de C. À ce stade de la compilation, l'arbre de syntaxe abstraite contient encore des définitions de fonctions locales. Comme C ne les admet pas, il faut les remplacer, quand c'est possible, par des boucles C ou bien par des fonctions globales C. **Integ** réalise cette transformation. Il est à noter que c'est la première fois que le choix du langage cible conditionne la compilation. Toutes les passes précédentes étaient ignorantes de ce choix. Cela montre qu'il serait facilement possible de changer Bigloo pour qu'il produise du code natif !

Cgen Le code C peut maintenant être émis. **Cgen** le fait.

cc/ld La compilation Scheme est terminée, il ne reste plus qu'à invoquer le compilateur C sur le fichier C produit par **Cgen**. Pour produire des exécutables, il faut en plus invoquer l'éditeur de liens d'Unix `ld`.

3.3.6 L'ordre des passes

Si certaines passes ne peuvent être appliquées qu'à des moments précis de la compilation il n'en est pas de même pour d'autres. En particulier les passes d'optimisations peuvent souvent être déplacées. Éventuellement comme l'écrit Appel dans [App92] certaines optimisations peuvent être appliquées à diverses étapes de la compilation. Comment alors sommes-nous parvenu à l'ordre présenté dans la section précédente? Il existe des contraintes entre les passes qui conduisent assez naturellement à un ordre global.

Expand < Scan **Expand** n'est pas à proprement dit une phase d'optimisation. Néanmoins comme nous l'avons juste mentionné, elle réalise certaines optimisations triviales. Puisqu'il s'agit de transformation source à source cette passe doit prendre place avant la construction de l'arbre de syntaxe abstraite !

Uncurry < Inlining La passe **Uncurry** remplace des invocations à des fonctions d'ordre supérieur par des fonctions de premier ordre. Par exemple, elle remplace deux appels à une addition currifiée par un appel à l'addition classique à deux arguments. Il est normal qu'après cette transformation, toutes les optimisations de Bigloo puissent s'appliquer, notamment l'intégration.

Inlining < Fuse Comme nous l'avons déjà signalé, l'intégration de fonctions peut rendre des définitions globales inutiles. Il faut donc que la passe chargée de les supprimer prenne place après.

Inlining < Beta La passe d'intégration produit beaucoup de bloc lexicaux inutiles. La passe **Beta** réduit ces constructions inutiles. Il faut donc qu'elle soit exécutée après l'intégration.

Typage < Hoist La passe de typage introduit les tests de type et place dans l'arbre de syntaxe des nœuds d'erreurs (forme `failure`). Puisque la passe **hoist** se sert principalement de ces nœuds pour réorganiser l'arbre, elle ne peut avoir lieu qu'après le typage.

Hoist < Cse Cet ordre est évident car une des missions de **Hoist** est de préparer l'arbre pour faciliter le travail de la passe **Cse**.

Cse < Beta La passe **Cse** travaille sur les expressions qui sont placées dans des liaisons locales (formes `let`). La passe **Beta** supprime bon nombre de ces liaisons. L'efficacité de l'optimisation de suppression des sous-expressions communes ne peut donc être garantie que si cette passe intervient avant la β-réduction.

Effect < 0cfa L'analyse de flot de contrôle utilise directement les résultats de l'analyse d'effets de bord. L'ordre respectives de ces deux passes est donc imposé.

0cfa < Globalisation La passe **0cfa** réduit le nombre de fermetures à allouer. Il faut donc qu'elle soit exécutée avant la production de ces fermetures.

0cfa < Superlift L'optimisation **Superlift** ne doit être appliquée qu'aux fonctions qui vont donner lieu à des créations de fermetures. Ainsi il ne faut pas l'appliquer aux fonctions optimisées par l'analyse de flot de contrôle.

Typage < 0cfa L'analyse de flot de contrôle supprime des tests de types de l'arbre de syntaxe, il faut donc évidemment que ces tests aient été posés !

Globalisation < Cnst La globalisation produit des constantes (λ-expressions closes). La compilation des constantes ne peut donc être effectuée qu'ultérieurement.

Stack < Beta Car la passe de β-réduction a principalement pour but de supprimer de l'arbre certaines variables intermédiaires, or la passe **Stack** travaille sur ces variables.

Lift < Stack Car les allocations de fermetures ne sont rendues explicites que par la passe **Lift**. Pour que **Stack** optimise toutes les allocations, il faut qu'elle n'intervienne qu'après **Lift**.

3.4 Le moteur de Bigloo

Pour conclure ce chapitre, nous allons donner le fragment de code qui réalise l'enchaînement des passes de la compilation.

```
 1: (define (compiler)
 2:    ;; on commence par lire le fichier d'access
 3:    (read-access-file)
 4:    ;; On creer la table de hash pour
 5:    ;; les macros et les globales
 6:    (init-global-environment!)
 7:    ;; on installe les macros initiales
 8:    (install-initial-expander)
 9:    ;; on prepare l'interface etrangere
10:    (init-foreign-interface!)
11:    (let ((code (read-src)))
12:       ;; on termine la partie concernant
13:       ;; l'interface etrangere
14:       (set! code (make-foreign-access! code))
15:       (make-foreign-casting!)
16:       ;; on fait la macro expansion du code.
17:       (set! code (expand-code code))
18:       (stop-on-pass 'expand
19:                     (lambda ()
20:                        (write-expanded code)))
21:       ;; on scan le code pour trouver
22:       ;; les definitions manquantes,
23:       ;; verifier les autres et construire
24:       ;; l'arbre de syntaxe abstraite.
25:       (let ((tree (scan-walk code)))
26:          ;; on sauve le tas
27:          (stop-on-pass 'make-heap
28:                        (lambda ()
29:                           (make-heap)))
30:          ;; on fait (si optim) la passe
31:          ;; de 'de-curryfication'
32:          (if (>fx *optim* 1)
33:              (set! tree (curry-walk tree)))
34:          ;; on inline le code
35:          (set! tree (inline-walk tree))
36:          (stop-on-pass 'inline
37:                        (lambda ()
38:                           (write-tree tree)))
39:          ;; on pose les tests de type et
40:          ;; les fonctions de conversions
41:          (set! tree (type-walk tree))
42:          (stop-on-pass
43:           'type
44:           (lambda ()
45:              (write-tree tree)))
46:          ;; on eliminie les fonctions
47:          ;; globales jamais invoquee
48:          (set! tree (fuse-walk tree))
49:          (stop-on-pass
50:           'fuse
51:           (lambda ()
52:              (write-tree tree)))
53:          ;; on pose les assertions
54:          (set! tree (assert-walk tree))
55:          (stop-on-pass
56:           'assert
57:           (lambda ()
58:              (write-tree tree)))
59:          ;; on fait l'analyse de
60:          ;; control (en mode -O2)
61:          (if (>fx *optim* 1)
62:              (begin
63:                 (effect-walk tree)
64:                 (set! tree (0cfa tree))
65:                 (stop-on-pass
66:                  '0cfa
67:                  (lambda ()
68:                     (write-tree tree)))))
69:          ;; passes omises
70:          ...
71:          (let ((c-prefix (cgen-walk tree)))
72:             (stop-on-pass
73:              'cgen
74:              (lambda () 'done))
75:             (stop-on-pass
76:              'distrib
77:              (lambda () 'done))
78:             ;; on indente (eventuellement)
79:             (if (or (eq? *pass* 'cindent)
80:                     *c-debug*)
81:                 (indent c-prefix))
82:             (stop-on-pass
83:              'cindent
84:              (lambda () 'done))
85:             ;; on compile
86:             (cc c-prefix)
87:             (stop-on-pass
88:              'cc
89:              (lambda () 'done))
90:             ;; on effectue l'edition de liens
91:             (ld c-prefix)
92:           'done))))
```

Programme 3.5: Le "moteur" de Bigloo

Bien sûr, le code donné ici a été un peu simplifié (toutes les passes intervenant lors d'une compilation ne sont pas présentées). La fonction `stop-on-pass` applique son deuxième argument et arrête éventuellement la compilation. On peut remarquer que l'arbre de syntaxe est obtenu par la passe `scan` (ligne 25). Ensuite, toutes les passes font dessus des modifications physiques.

3.5 Récapitulatif

Nous avons présenté dans ce chapitre la structure de Bigloo. Nous avons pour cela présenté le langage Sqil. Ce langage intermédiaire, issu de la thèse de N. Séniak, est à mi-chemin entre un langage d'ordre supérieur comme Scheme et un langage d'assez bas niveau comme C. Utiliser un langage intermédiaire comme celui-ci, nous permet d'ignorer le langage cible dans la majeure partie du compilateur. Ce n'est que dans les toutes dernières passes que C intervient.

Par ailleurs, nous avons exposé notre choix méthodologique qui consiste à multiplier le nombre de passes et à isoler chaque traitement et chaque analyse. Nous l'avons justifié en expliquant que ce choix permet selon

nous :

- d'être plus rapide car :
  - on évite plus facilement les calculs redondants.
  - l'arbre de syntaxe est plus petit (on n'est pas obligé d'avoir simultanément les informations de plusieurs passes), le compilateur se comporte donc mieux vis à vis de son *GC*.
- d'être plus facile à maintenir. Car les différentes composantes du compilateur sont mieux isolées les unes des autres.

Enfin, nous avons présenté les différentes passes de Bigloo. Nous avons montré leur agencement et nous les avons brièvement décrites. Dans les chapitres suivants nous les présenterons plus en détail.

Chapitre 4

L'interface externe

Nous nommons *interface externe* (*foreign interface* en anglais) la partie du langage source de Bigloo qui permet de mélanger des programmes Scheme et des programmes écrits dans d'autres langages. Un compilateur moderne ne peut plus être centré sur lui-même et fermé au monde extérieur. Il doit permettre d'utiliser des programmes (ou des fragments de programmes) écrits dans d'autres langages de programmation. Ce besoin est d'autant plus important que le compilateur a pour source un langage de haut niveau comme Scheme. En effet, généralement, plus un langage est de haut niveau plus il abstrait les caractéristiques des machines: il ne permet pas de manipulation de la mémoire *bit par bit*, sa norme n'inclut qu'un très petit nombre d'opérations système, etc. Néanmoins, pour qu'un langage soit réellement utilisable, il doit impérativement permettre de réaliser toutes les opérations système, il doit pouvoir permettre d'utiliser les capacités particulières des machines comme, par exemple, leurs systèmes de fenêtrages. L'interface externe est donc une pièce essentielle d'un système de programmation. Dans la section 4.1 nous allons présenter celle de Bigloo et les motivations qui nous ont conduit à sa conception. La principale difficulté de l'interface est la manipulation des objets de type externe. Nous montrerons ce que permet notre interface dans ce domaine et nous la comparerons avec d'autres systèmes. Nous montrerons comment sont implantés les objets externes dans la bibliothèque d'exécution.

Le mélange de deux espaces de valeurs (l'espace des valeurs Scheme et l'espace des valeurs externes) impose la présence d'un mécanisme de conversion. Nous le présenterons dans la section 4.2 en donnant l'algorithme que Bigloo utilise pour placer les conversions dans l'arbre de syntaxe abstraite et en montrant comment sont gérées les projections d'un espace vers l'autre au sein du compilateur et de la bibliothèque d'exécution.

Par ailleurs, bien qu'un compilateur manie trois langages: le langage source, le langage cible et le langage de la bibliothèque d'exécution[1], la plupart des compilateurs n'ont qu'une connaissance très faible de ces deux derniers. Généralement la seule opération qu'ils savent réaliser est l'invocation de fonctions externes de la bibliothèque (utilisée dans la compilation des appels aux primitives). C'est par exemple le cas du compilateur Caml-light [Ler90] et du système Le-Lisp [Cha85]. La présence de l'interface externe semble pratiquement se réduire, dans ces compilateurs, à l'existence d'un type de nœuds particuliers dans les arbres de syntaxe abstraite chargés de désigner les appels externes. Nous pensons que c'est insuffisant. Bigloo se situe à l'opposé de cette approche car son interface externe est une partie essentielle du compilateur : elle est élevée au rang de méthodologie de compilation. Ses mécanismes fondamentaux sont utilisés pour des tâches aussi diverses que la compilation des appels aux primitives, la compilation des expressions conditionnelles ou encore la compilation du typage dynamique. Nous présenterons en détail dans la section 4.3 le rôle que tient l'interface externe dans Bigloo pour la compilation du noyau Scheme. Nous présenterons successivement la compilation des primitives, des expressions conditionnelles, des formes fonctionnelles et enfin, la compilation du typage dynamique. Pour conclure ce chapitre, nous donnerons des exemples de code produit tant sur la compilation du noyau Scheme que sur la manipulation des valeurs externes en Scheme.

L'interface externe est une pièce maîtresse de Bigloo. C'est pourquoi nous lui avons consacré un chapitre entier.

[1] Dans le cas de Bigloo, le langage de la bibliothèque d'exécution est en partie le langage source (Scheme) et en partie le langage cible (C).

4.1 L'interface externe

Il existe plusieurs degrés possibles d'interface entre deux langages. En bas de l'échelle, l'interface peut seulement permettre à un langage d'invoquer des fonctions d'un autre langage. En haut de l'échelle l'interface peut permettre à deux langages d'appeler de façon croisée leurs fonctions, d'échanger des données et surtout elle peut permettre aux deux langages de manipuler des données qui n'appartiennent pas à leur espace de valeurs mais à celui de l'autre langage. Ces extrémités sont séparées par un long dégradé. Plus l'interface offre de possibilités, plus elle est ancrée dans le compilateur. Pour Bigloo nous avons voulu une interface aussi évoluée que possible.

4.1.1 Les exigences de notre interface externe

Nous présentons les motivations qui nous ont conduit au choix de notre interface. Dans la suite, nous désignerons par $\mathcal{S}$ le langage source auquel nous souhaitons adjoindre une interface externe. Nous utiliserons la lettre $\mathcal{E}$ pour désigner un langage externe quelconque.

Que pouvons nous souhaiter de l'interface entre le langage $\mathcal{S}$ et le langage $\mathcal{E}$? Énumérons quelques points fondamentaux :

1. La première exigence (*l'exigence minimale*): nous voulons impérativement être capable d'invoquer depuis $\mathcal{S}$ des fonctions du langage $\mathcal{E}$ (pour autant que le langage $\mathcal{E}$ possède des fonctions).

2. Avec les systèmes actuels (comme les gestionnaire de fenêtres qui utilisent une mécanique dite de *call back*) il est indispensable que le langage $\mathcal{E}$ soit également capable d'invoquer des fonctions de $\mathcal{S}$.

3. Une des raisons d'être de l'interface est la réutilisation des programmes, écrits dans d'autres langages (C par exemple). Ces programmes écrits en $\mathcal{E}$ utilisent des valeurs de $\mathcal{E}$. Nous voulons donc invoquer depuis $\mathcal{S}$ des fonctions de $\mathcal{E}$ en transmettant des objets valides dans l'espace des valeurs de $\mathcal{E}$.

4. Pour la même raison que le point précédent, nous voulons invoquer des fonctions de $\mathcal{S}$ depuis $\mathcal{E}$ en leur fournissant des objets de l'espace de valeurs de $\mathcal{E}$. Ceci montre que notre approche impose des efforts de la part du compilateur car ce dernier doit savoir construire et manipuler des données de $\mathcal{E}$.

5. Bien que dans la pratique l'interface la plus utile soit l'interface avec C, nous ne voulons pas nous restreindre à ce langage. Notre interface ne doit pas être dépendante de C.

Les points [3,4] imposent qu'il soit permis de manipuler (donc de mentionner) depuis le langage $\mathcal{S}$ des valeurs du langage $\mathcal{E}$. C'est à notre avis ceci qui conditionne totalement la réalisation de l'interface. Il ne sera pas suffisant d'avoir dans l'arbre de syntaxe un nœud désignant l'appel de fonctions externes. Le compilateur devra être capable de manipuler pendant toute la compilation des valeurs qui ne sont pas dans l'espace de valeurs de son langage source !

4.1.2 La conception de l'interface externe

L'utilité immédiate de l'interface externe a été de réaliser l'interface entre Scheme, le langage source, et C, le langage cible de Bigloo. Pour cette raison, l'implantation actuelle est complètement orientée vers C. C'est cette interface que nous présentons (sans trop de détails syntaxiques car ils peuvent être trouvés dans [Ser94a]) dans cette section. Les déclarations externes sont introduites par une clause spéciale dans la définition des modules :

$$\texttt{(foreign} < clause_1 > \ldots < clause_n > \texttt{)}$$

Les $< clause_i >$ introduisent des importations (de C vers Scheme) ou des exportations (de Scheme vers C) et sont toujours construites sur le même modèle :

($[classe] < type_{\mathcal{E}} > < nom_{Scheme} > < type_{\mathcal{E}} > < nom_C >$) ou, ($< type_{\mathcal{E}} > < nom_{Scheme} > < nom_C >$)

Il est obligatoire d'indiquer au compilateur les $< nom_{Scheme} >$ et $< nom_C >$ car l'espace des identificateurs n'est pas le même dans les deux langages (Scheme utilise un ensemble de caractères différent de C). Le $< type_{\mathcal{E}} >$ permet au compilateur de réaliser la projection des valeurs Scheme vers les valeurs C. L'argument optionnel *classe* permet d'indiquer la nature de l'objet externe référencé (par exemple de préciser que ce n'est pas une fonction mais une macro). Examinons quelques déclarations standard de la bibliothèque d'exécution.

`(double cos (double) "cos")` Cette clause introduit la fonction `cos`. Son argument et son résultat sont des `double` C.

`(define bpair cons (obj obj) "MAKE_PAIR")` Cette clause introduit la fonction `cons`. Le mot clé `define` indique que `"MAKE_PAIR"` n'est pas une fonction mais une macro C. Les arguments sont des valeurs Scheme quelconques (désignées par le type `obj`) alors que le résultat est une paire Scheme (type `bpair`).

`(define obj car (bpair) "CAR")` La fonction `car` est le pendant de la fonction `cons`, elle destructure une paire. Cette fonction s'applique sur des paires (type `bpair`) son résultat est un objet quelconque Scheme (type `obj`).

`(export obj error (obj obj obj) "__error")` Le mot clé `export` précise qu'il s'agit là d'une exportation d'une fonction Scheme pour C. Le but de cette exportation est simple: certaines primitives de Bigloo sont écrites en C. Depuis ce code il peut être utile de déclencher une erreur au moyen de la fonction usuelle `error`. Cette clause d'exportation précise que la fonction nommée en Scheme `error` sera utilisée sous le nom `__error` en C et que C devra lui fournir des arguments au format de Bigloo (type `obj`).

Plusieurs types externes sont utilisés dans ces exemples (`double`, `obj`, `bpair`). Nous allons montrer dans la section suivante à quoi ils correspondent et comment l'utilisateur peut en définir de nouveaux.

4.1.3 Les types externes

Dans les exemples que nous avons donné précédemment, il faut distinguer deux catégories de types :

- Les types externes construits dans l'ensemble des types du langage externe.
- Les types désignant dans le langage de la bibliothèque d'exécution de Bigloo les représentations des objets Scheme.

Le langage de la bibliothèque d'exécution de Bigloo est le même que son langage cible qui est également le même que le langage externe pour lequel on a développé notre interface. Cela peut entraîner quelques confusions. Les types désignant des objets Bigloo ne sont pas forcément des types construits à partir de l'ensemble des types C (par exemple, comme c'est montré dans le chapitre 2, les paires sont codées par des manipulations de pointeurs et non par un type C). En particulier, l'interface ne suppose pas que les types désignant des objets Bigloo soient des types C. Pour cette raison, l'interface n'utilise pas la propriété que le langage cible est le même que le langage avec lequel on s'interface.

Commençons par décrire les types désignant les objets Bigloo.

Les types externes désignant des objets Bigloo

Chaque objet Scheme possède une implantation dans le langage de la bibliothèque d'exécution. Est donc associé un type externe à chaque type Scheme, dans le langage de désignation des types externes de Bigloo. La convention que nous avons utilisée est de préfixer les noms de ces types par la lettre `b` (excepté le type désignant tout objet Scheme, le type `obj`). Ainsi, les paires sont désignées par le type `bpair`, les entiers par le type `bint`, les booléens par le type `bbool`, etc.

Les types externes construit dans l'ensemble de type du langage externe

Puisque notre interface est actuellement destinée à C, les types externes (qui ne désignent pas les implantations des objets Bigloo) sont construits dans l'ensemble des types C. C'est-à-dire que les types de base (`int`, `double`, `char`, `bool`, etc.) sont prédéfinis et qu'il est possible de construire les mêmes types composés qu'en C (union, somme, pointeur, etc.). Nous allons illustrer les constructions de types composés par un programme manipulant une liste chaînée C. En C, cette liste aura le type donné dans le programme 4.1.

```
struct el {
        int value;
        struct el *next;
};
```

Programme 4.1: La structure C

Dans l'interface externe de Bigloo la syntaxe pour définir les types composés n'est pas la même qu'en C. Il nous faut ici définir deux types. L'équivalent du type `el` et le type *pointeur sur* `el` (programme 4.2).

```
(foreign (type el (struct ((int "value")
                           (el* "next"))
                  "struct el"))
         (type el* (pointer el)))
```

Programme 4.2: Un type externe en Bigloo

La définition d'un type externe crée automatiquement des fonctions de manipulation sur les objets de ce type. Dans notre exemple, ces fonctions sont :

`make-el` Pour allouer un objet externe de type `el`.

`el?` Le prédicat associé au type `el`.

`el-value` & `el-value-set!` Pour accéder et modifier le champ `value`.

`el-next` & `el-next-set!` Pour accéder et modifier le champ `next`.

Il est alors possible d'utiliser ces fonctions dans un programme Scheme. Par exemple, voici un programme allouant une liste de type `el`:

```
(define (list->el* l)
   (let ((head (make-el)))
      (let loop ((l l)
                 (c head))
         (if (null? l)
             head
             (let ((new (make-el)))
                (el-value-set! c (car l))
                (el-next-set! c new)
                (loop (cdr l) new))))))
```

Comme on peut le voir dans le code produit pour l'exemple de la section 4.4.6, l'écriture des fonctions d'accès aux champs des structures exige qu'il existe une interface entre le langage cible et le langage externe. Il n'existe pas de moyen en général de connaître l'adresse d'un champ d'une structure connaissant l'adresse de la structure elle-même. Le seul moyen de réaliser une interface portable est donc de produire un morceau de code dans le langage externe accédant aux champs voulus.

4.1.4 Les implantations des objets de types externes

Nous étudions dans cette section les implantations des objets externes dans la bibliothèque d'exécution.

Les objets externes atomiques

Les objets externes atomiques de l'ensemble des types du langage externe (entiers, doubles, caractères, chaînes de caractères, ...) n'ont pas de représentation en Scheme. On ne peut pas les manipuler depuis le code Scheme, mais ils possèdent des équivalents en Scheme (les entiers, flottants, caractères, chaînes de caractères Scheme). On peut donc manipuler en Scheme leurs équivalents. Ceci s'effectue de façon transparente car Bigloo se charge automatiquement de réaliser les conversions. En effet, le compilateur connaît les fonctions de conversion à appliquer pour, par exemple, transformer une chaîne Scheme en chaîne C et réciproquement. Ainsi lors d'un appel à une fonction externe utilisant des types externes atomiques, Bigloo insère des fonctions de conversion à appliquer aux arguments ainsi qu'aux résultats de la fonction.

Les objets externes composés

Les objets externes composés n'ont pas d'équivalents en Scheme. La partie Scheme de la bibliothèque d'exécution n'utilise donc pas des transpositions, mais des encapsulations au dessus des valeurs C. Les encapsulations ont pour but de permettre à Bigloo de reconnaître la nature des objets externes qu'il manipule. Ainsi il est capable de vérifier leur type, de les imprimer, etc. Pour désigner les encapsulations, un type Bigloo atomique a été ajouté à ceux déjà décrits : le type `foreign`. Ce type est implanté par la structure C donnée dans le programme 4.3. Le champ `header` est standard dans l'implantation des objets Bigloo (voir le chapitre 2). Il sert à réaliser le prédicat `foreign?`. Le champ `id` mémorise l'identificateur du type externe de l'objet. Ce champ est utilisé pour implanter les prédicats spécifiques à chaque type externe (le prédicat `el?` dans l'exemple 4.2). Une prochaine version utilisera probablement les noms C comme identificateur de type, pour que deux types Scheme représentant le même type C vérifient le même prédicat. Ce n'est pas le cas dans notre implantation actuelle.

Et enfin, le champ `value` est un pointeur vers l'objet externe lui-même.

```
struct foreign {
   header_t        header;
   union object   *id;
   void           *value;
} foreign_t
```

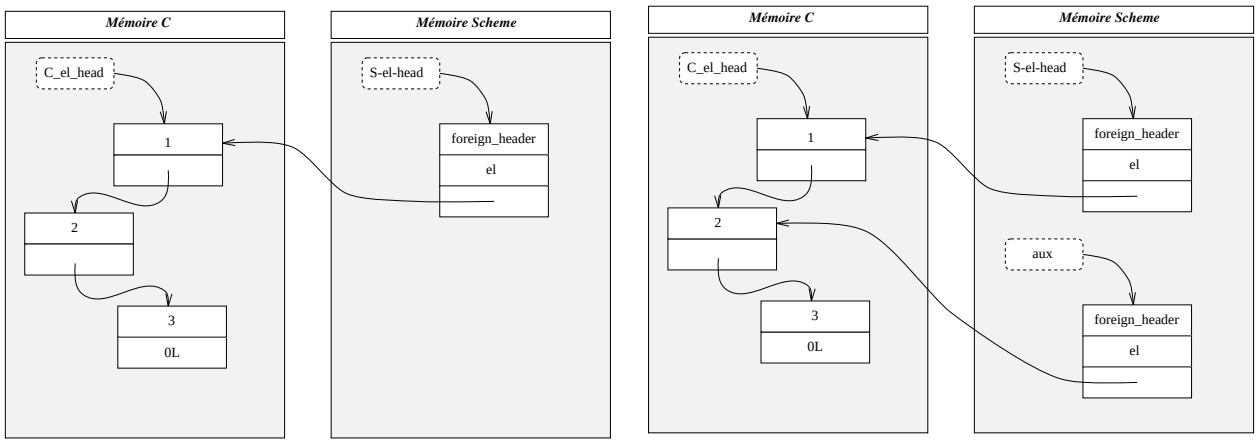
Programme 4.3: foreign-t

Les encapsulations des objets externes sont effectuées de manière paresseuse. C'est-à-dire que Bigloo n'encapsule une valeur externe que lorsque la valeur est référencée en Scheme. Examinons la vision Scheme d'une liste chaînée allouée par C. Reprenons la structure donnée dans le programme 4.1. Examinons les

représentations mémoire C et Scheme.

Le programme C a alloué une liste chaînée basée sur le type `el` qui contient trois éléments. La tête de cette liste est pointée par la variable nommée `C_el_head`. Une clause externe a permis à Bigloo de récupérer un pointeur sur cette liste. Seul le premier élément est donc visible depuis Bigloo. Il est donc le seul à être encapsulé.

Imaginons maintenant qu'une variable Scheme (nommée `aux` dans la figure) soit liée au résultat de l'expression `(el-next S-head-el)`. À ce moment de l'exécution il s'agit du deuxième élément de la liste. C'est lorsque le pointeur sur ce deuxième élément est récupéré par Bigloo que la deuxième encapsulation est faite.



L'implantation du prédicat `eq?`

Les encapsulations sont allouées à la demande, c'est-à-dire quand un objet externe est référencé par Scheme. Pour être capable d'implanter correctement le prédicat `eq?`, il faut garantir que si l'objet externe est référencé plus d'une fois, le même emplacement mémoire sera toujours utilisé pour supporter l'encapsulation. Ceci est réalisé au moyen de tables de hachage. Dès qu'une adresse d'un objet externe parvient à Bigloo (qu'elle soit récupérée depuis C ou bien allouée par la bibliothèque d'exécution Scheme), elle est placée dans une table où une encapsulation lui est associée. Ensuite, avant chaque allocation d'encapsulation, il suffit de consulter cette table pour savoir si l'adresse externe est déjà en service.

La zone d'allocation des objets externes

Les objets externes, lorsqu'ils sont alloués depuis Scheme sont alloués comme tous les autres objets. Ils peuvent être ramassés par le *GC* s'ils ne sont plus pointés. Ceci n'est pas sans poser de problèmes car il se pourrait qu'un objet externe ne soit plus pointé par un autre objet Scheme mais qu'il le soit par un objet C. Le *GC* pourrait ramasser cet objet bien qu'il soit encore vivant ! Ce problème est très complexe et il n'existe pas de solutions parfaites. Voici celles qui sont envisageables :

1. Les objets externes sont alloués dans le tas de C et ils ne sont plus visibles du *GC*; ils doivent être explicitement désalloués. Cette solution ne nous semble pas s'allier très harmonieusement avec la gestion implicite de la mémoire en vigueur dans le monde fonctionnel.

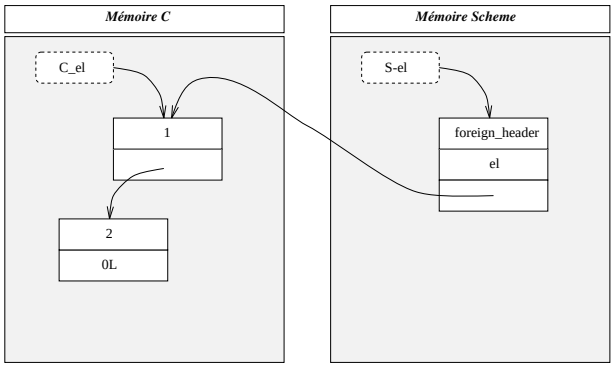
2. On ajoute le tas C comme ensemble de racines. Cette solution ne règle pas tous les problèmes. Il n'en reste pas moins que les pointeurs fournis à C par Scheme ne sont pas des pointeurs comme les autres. Par exemple ils ne peuvent être passés à la fonction de désallocation `free`.

3. Les objets externes sont alloués dans le tas de Scheme. Ils ne peuvent être ramassés par le *GC* que quand ils sont explicitement marqués comme inutilisés pour C. Ceci se fait au moyen d'une fonction de la bibliothèque. Cette solution a l'avantage sur la précédente que les pointeurs vers des objets Scheme éventuellement contenus dans l'objet externe seront tracés par le *GC*. Il ne risque donc pas d'y avoir de récupérations inopinées.

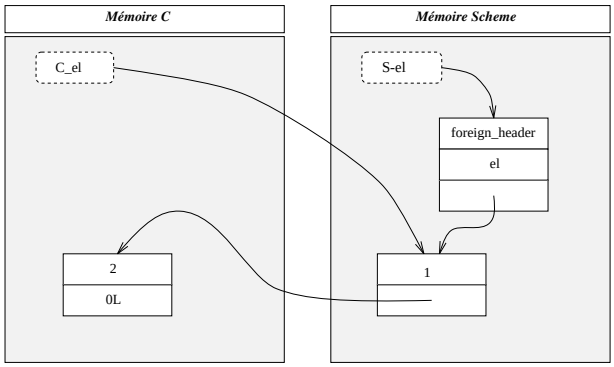
Aucune solution n'est parfaite. La version actuelle de Bigloo utilise la troisième, mais ce choix n'est pas définitif. Il n'est pas à exclure que les prochaines versions choisissent plutôt la première solution.

Voici, toujours sur l'exemple de la structure `el`, quelles peuvent être les occupations mémoires si une structure est allouée par C ou par Scheme.

Si la structure pointée est allouée par C puis est passée à Scheme l'occupation mémoire sera :



En revanche si la structure est allouée par Scheme puis passée à C, l'occupation mémoire sera :



4.1.5 Comparaison avec d'autres interfaces

Tous les compilateurs (ou presque) offrent des interfaces externes. Malheureusement ces interfaces sont généralement très peu documentées. À notre connaissance peu de travaux ont fait l'objet de publication sur ce sujet.

- L'article de Harlan Sexton [Sex88] compare trois interfaces externes, celle de Franz ExCL, de DEC Vax Lisp et enfin, celle de Lucid CL. L'auteur examine les trois points cruciaux de l'interface.
  - La capacité qu'offrent ces interfaces pour invoquer du code externe. Les trois interfaces semblent disposer d'un mécanisme semblable à celui de Bigloo pour invoquer des fonctions externes : une déclaration définit une fonction avec son prototype, les conversions et les tests sont assurés par les compilateurs. Ces interfaces ne sont pas conçues que pour C, elles offrent donc plus de protocoles d'appels que le simple appel par valeur : certaines permettent l'appel par référence et même l'appel dit par valeur de retour (*call by return value*). Toutefois, ces interfaces ne semble pas posséder l'appel par nom (qui correspond par exemple aux macros de C) et il n'apparaît pas clairement qu'on puisse utiliser dans la partie Lisp des variables externes. En outre, le problème des fonctions externes d'arité variable n'est pas mentionné.
  - La possibilité qu'ont ces interfaces de charger du code externe et de fournir au monde externe des fonctions Lisp. Les trois systèmes permettent le chargement dynamique de fichiers objets. Cette facilité n'existe pas actuellement dans Bigloo mais elle est en cours de développement ; elle ne semble pas poser de difficulté majeure. Enfin, les trois interfaces permettent d'exporter des fonctions suivant le même mécanisme que Bigloo.
  - La possibilité de définir et manipuler des types externes en Lisp. Ceci constitue le point le plus sensible et le plus difficile.

    Bien que l'interface de Franz ExCL semble avoir été conçue dans la même optique que celle de Bigloo, elle offre beaucoup moins de possibilités. Il ne semble pas être possible de définir d'autres types que des structures C (pas de tableaux, énumération, pointeur, etc.). Et pour ces structures, on ne peut accéder aux champs qui n'ont pas un type simple !

    Les deux autres interfaces ont une vision des types externes très différente puisqu'elle est basée sur la représentation des objets en mémoire. C'est-à-dire qu'on définit un type en donnant sa structure octet après octet. Cette approche nous semble nettement moins portable que la nôtre car elle impose d'inscrire dans le programme des informations trop précises (dépendantes des machines)

sur les représentations mémoire. Cette solution ne permet pas de résoudre le problème de l'accès aux champs des structures sans interface entre le langage cible et le langage externe. Il ne sert à rien d'être capable d'accéder au $i^{ème}$ octet d'une structure s'il n'existe aucun moyen de savoir où le compilateur place le $n^{ème}$ champ par rapport à l'adresse de la structure elle-même (comme c'est, par exemple, le cas avec les compilateurs Ada).

Après cette comparaison, H. Sexton énonce ce que pourrait être une interface externe standard. Les importations et exportations de fonctions sont visiblement consensuelles, en revanche la manipulation des types externes est divergente. La solution proposée ici n'est pas basée comme la nôtre sur les constructions de type des langages externes mais sur les représentations mémoire des objets externes. Nous pensons que cette solution est moins portable et moins lisible que la nôtre.

- L'article [RM92] contient la description de l'interface externe qui est à la base de *esh* (*Embeddable SHell*). Ce projet consiste à faire de Scheme une sorte de *shell*. L'accent est donc mis sur Scheme comme langage glue. L'objectif principal de cette interface est de rendre les types C accessibles depuis Scheme. En revanche, rendre accessible à C tous les types Scheme n'est pas primordial dans ce cas. Ce système étant conçu pour son interface, la représentation des objets coïncide le plus souvent dans les deux mondes. Les opérations permises sont très proches des nôtres mais les types externes sont des objets de première classe et des opérations de conversion explicites sont possibles. Cet article n'insiste pas sur l'implantation de l'interface. En particulier, il n'est pas aisé de comprendre comment sont représentés les objets Scheme.

- H. Davis, P. Parquier et N. Séniak présentent dans l'article [DPS94] une interface très poussée entre Ilog Talk et C++. L'intégration est axée sur le côté objet des deux langages. Néanmoins cette interface présente les mêmes possibilités que la nôtre. La distinction se fait plus au niveau de l'implantation. L'approche utilise du code d'interface (*stub code*) au lieu d'intégrer les accesseurs. Si ce n'est pas gênant pour Talk, ce serait dramatique pour Bigloo car les performances du compilateur dépendent en partie de la qualité de notre interface.

- G. Attardi présente dans l'article [Att94] une implantation de Common Lisp permettant de mélanger des programmes Lisp et des programmes C. Le compilateur exposé produit du C qui est presque orthodoxe (voir la section 2.6.2). L'interface externe proposée dans cet article est basée sur le partage d'une bibliothèque d'exécution commune à différents langages. C'est-à-dire que la solution présentée ne consiste pas à introduire dans le compilateur des connaissances sur les langages externes mais à rendre compatible la bibliothèque d'exécution. L'optique de ce travail est donc assez éloignée de la nôtre.

Quelques compilateurs décrivent dans leur documentation leur interface externe.

Scheme-to-C [Bar89b]. Ce compilateur dispose d'une interface qui semble proche de la nôtre. Il est possible d'importer des fonctions, de définir des types composés. Il n'est apparemment pas possible d'exporter des fonctions pour C. Il est également difficile de savoir comment sont alloués les objets externes (dans le tas de Scheme ou celui de C? Sont-ils tracés par le *GC*?).

T La distribution de ce compilateur [RA82] contient une documentation de son interface externe sous Unix [Pat92]. L'interface décrite est assez pauvre, elle permet seulement d'importer des fonctions C depuis Lisp et de charger dynamiquement du code. Elle ne permet ni d'exporter des fonctions pour C, ni de définir des types composés.

Caml-light [Ler90]. Ce compilateur a une interface minimale. Elle ne permet principalement que d'invoquer depuis le code Caml-light des fonctions externes et dans une moindre mesure, elle donne le moyen à un programme C d'invoquer une fonction Caml-light. Les fonctions C qui peuvent être invoquées depuis Caml-light ont toutes le même prototype. C'est-à-dire que ce sont toutes des fonctions prenant en argument et retournant en résultat des objets Caml-light. Ceci facilite l'écriture du compilateur car ce dernier ne manipule que des valeurs ML mais en revanche l'écriture de la bibliothèque d'exécution est difficile car il n'est pas permis (à moins que le type de l'objet ML ait une représentation qui coïncide avec la représentation dans le langage externe de l'objet) d'invoquer des fonctions externes quelconques : elles doivent avoir été écrites spécialement. Étudions les possibilités pour implanter dans la bibliothèque

d'exécution une fonction classique comme la fonction calculant le cosinus d'un nombre. Deux possibilités peuvent être envisagées :

- La fonction cosinus est implantée par un algorithme écrit en Caml-light. Cette solution est parfaitement possible mais on peut supposer qu'elle sera moins efficace qu'une solution utilisant un langage mieux adapté au calcul flottant.
- La fonction ML cosinus invoque une fonction d'interface qui transforme son argument en flottant C, invoque la fonction C `cos`, puis retourne le résultat de la transformation du nombre obtenu en valeur ML. Cette deuxième solution présente deux inconvénients : *(i)* Écrire la bibliothèque est long et fastidieux car elle comporte beaucoup de fonctions comme cosinus et il faudra donc écrire de nombreuses fonctions d'interface (cf. la `libunix` de la distribution de Caml-light). *(ii)* Elle n'est pas très efficace (du moins elle est nettement moins efficace que le programme C faisant le même calcul) car avant d'appeler la fonction C, il a été indispensable d'appeler une autre fonction servant d'intermédiaire.

Parce qu'aucune de ces deux solutions n'est satisfaisante, Caml-light en utilise une troisième ! Dans le compilateur est inscrite la liste des fonctions (liste de primitives) qui sont invoquées de façon particulière. Cette liste contient toutes les fonctions telle que celle calculant le cosinus (les fonctions de manipulation des nombres, des chaînes, etc.). Cette solution permet d'avoir des appels aux primitives efficaces car le compilateur effectuant un traitement particulier est à même d'éviter un appel auxiliaire avant d'invoquer la fonction C `cos`. Néanmoins, elle ne nous semble pas satisfaisante car elle fige le nombre de primitives dans le compilateur. Il est difficile d'en ajouter et surtout il faudrait pour cela changer le compilateur, ce qui n'est pas acceptable du point de vue du programmeur.

expressions	**notation**	**signification**
fun une fonction	$fun \downarrow_{tres}$	le type du résultat de la fonction fun
fun une fonction	$fun \downarrow_{tformal_i}$	le type du $i^{éme}$ paramètre formel de la fonction fun

FIG. 4.1 - Les extensions du pseudo-code

4.2 Le typage (conversions et vérifications)

Nous avons montré dans la section 4.1.4 que l'interface externe est présente dans toute la compilation car le compilateur jongle avec deux espaces de valeurs : les valeurs du langage source et les valeurs du langage externe. Pour cela, le compilateur se charge de gérer les conversions indispensables entre les deux mondes. Nous allons étudier dans cette section ce mécanisme de conversion.

4.2.1 L'algorithme de typage

Que ce soit pour les objets externes atomiques ou les objets externes composés, la bibliothèque d'exécution Scheme n'a pas les mêmes vues que la bibliothèque d'exécution C. Le compilateur doit donc transformer les valeurs suivant leurs utilisations dans le langage source ou dans le langage externe. Ces conversions sont effectuées lors de l'invocation de fonctions externes ou lors de la manipulation de variables externes. Pour reprendre l'exemple de la fonction cosinus (dont nous avons donné le prototype dans la section 4.1.2), lors de l'invocation de cette fonction, l'argument fourni doit, s'il s'agit d'un nombre flottant Scheme, être converti en nombre double C. Le résultat qui est un double C doit être converti en flottant Scheme. Toutes les valeurs Scheme ne peuvent pas être converties en double C. Seuls les nombres flottants Scheme peuvent l'être. Mais comme Scheme est typé dynamiquement, bien souvent le compilateur n'a aucun moyen de savoir si l'argument qu'il doit convertir est réellement un nombre flottant. Pour se prémunir des exécutions incorrectes, le compilateur doit donc insérer des tests de type dynamiques avant d'invoquer les fonctions de conversion. Pour cela il utilise l'algorithme de typage 4.1.

Le langage algorithmique utilisé est celui décrit dans la figure 6.1 du chapitre 6 auquel nous avons ajouté quelques extensions (voir figure 4.1) et le filtrage. L'expression $[\![\mathcal{A}]\!]$ désigne n'importe quelle expression atomique (nombre, constante, ...) et l'expression $[\![\mathcal{V}]\!]$ désigne une variable quelconque.

```
type-atree( tree, τ1 )=
   selon tree
        [[ A ]]:
             convert( tree, type-of( tree ), τ1 )
        [[ (quote ...) ]]:
             convert( tree, obj, τ1 )
        [[ (begin tree1 ... treen) ]]:
             [[ (begin type-atree( tree1, obj ) ... type-atree( treen, τ1 )) ]]
        [[ (set! var val) ]]:
             [[ (set! var type-atree( val, type-of( var ) )) ]]
        [[ (let ((var1 val1) ...) body) ]]:
             [[ (let ((var1 type-atree( val1, obj )) ...) type-atree( body, τ1 )) ]]
        [[ (labels ((fun1 (arg1 ...) body1) ...) body) ]]:
             [[ (labels ((fun1 (arg1 ...) type-atree( body1, obj )) ...) type-atree( body, τ1 )) ]]
              ⋮
        [[ (fun arg1 ...) ]]:
             si fun est une fonction connue du compilateur
                  alors si Le nombre de paramètre formels est incompatible avec l'arité de fun
                             alors error( ... )
                             sinon soit τr=fun↓tres,
                                   soit type-f1=fun↓tformal1,
                                      ⋮
                                      convert( [[ (fun type-atree( arg1, type-f1 ) ...) ]], τr, τ1 )
                  sinon ...
   fin
```

Algorithme 4.1: L'algorithme de typage

Voici quelques explications et remarques sur la fonction *type-atree* :

- La fonction *type-atree* n'est pas une inférence de type. La signature de cette fonction est :

$$type\text{-}atree : atree \times type \mapsto atree$$

  La fonction *type-atree* réalise un parcours d'un arbre de syntaxe abstraite. Ses arguments sont : *(i)* L'arbre de syntaxe abstraite à convertir. *(ii)* Le type dans lequel l'arbre doit être converti. La fonction *type-atree* est invoquée sur les définitions des fonctions globales Scheme. Le type initial τ_1 est donc soit `obj`, soit un type externe si la fonction Scheme est exportée pour un langage externe, (voir la section 4.1.2).

- La fonction `type-of` retourne le type d'une constante (pour un entier, elle retourne `int`, pour un booléen elle retourne `bool`, etc.).

- Les fonctions locales (introduites par des constructions `labels`) ne peuvent pas prendre en argument ou retourner en résultat des valeurs des types externes car la fonction *type-atree* leur impose de ne manipuler que des objets Scheme. C'est un choix de réalisation, nous aurions parfaitement pu modifier la fonction *type-atree* pour que les fonctions locales puissent manipuler des objets externes. La même remarque s'applique également aux variables locales introduites par les constructions `let` (permettre les types externes dans les fonctions locales pourra constituer un prochain travail).

- Dans le typage des appels fonctionnels de l'algorithme 4.1 ont peut constater que l'arité des fonctions connues du compilateur est vérifiée dès la compilation.

La fonction *type-atree* utilise une fonction nommée *convert*. C'est cette dernière qui installe dans l'arbre de syntaxe les opérations de conversion et de vérification de type. Sa définition est donnée dans l'algorithme 4.2.

```
convert( atree, τs, τd )=
   si τs = τd
        alors atree
        sinon soit converter=find-converter( τs, τd ),
                  si converter?( converter )
                       alors ⊥
                       sinon soit check-op=converter↓checkop,
                             soit convert-op=converter↓convertop,
                                  convert-op( check-op( atree ) )
```

Algorithme 4.2: La fonction de conversion

La signature de cette fonction est : $convert : atree \times type \times type \mapsto atree$. Elle prend en argument un arbre de syntaxe abstraite dont le résultat actuel est de type τ_s. Elle crée un nouvel arbre dont le type est τ_d. Expliquons le fonctionnement de cette fonction :

1. Si les deux types sont identiques l'arbre est inchangé.
2. Sinon, cette fonction consulte les tables du compilateur au moyen de la fonction `find-converter` pour savoir si la transformation $\tau_s \mapsto \tau_d$ existe. Si elle n'existe pas, alors le programme ne peut être compilé, le compilateur déclenche une erreur (ce qui est marqué dans l'algorithme par la valeur $\perp$).
3. Il existe une conversion entre τ_s et τ_d. Cette conversion nécessite une vérification de type et un appel à une fonction de conversion.
4. Pour simplifier l'exposé, nous n'avons pas considéré le typage des appels fonctionnels où les fonctions reçoivent un nombre variable d'arguments. Ce typage est très ressemblant au typage des appels de fonctions donné dans l'algorithme 4.1, si ce n'est que les arguments sont regroupés en une liste.

4.2.2 Les conversions

La fonction *convert* consulte les tables du compilateur pour savoir s'il existe une conversion entre deux types et quels sont les appels de fonctions à insérer dans l'arbre de syntaxe abstraite pour l'effectuer.

Les conversions entre les types de bases Bigloo

Tous les types de Scheme sont des sous-types du type `obj`, le type général. C'est-à-dire qu'un entier, une chaîne de caractère, un booléen Scheme sont des objets de type `obj`. Tous les sous-types peuvent donc être convertis en type `obj`. La réciproque n'est évidemment pas vraie. Par exemple tout objet Scheme ne peut pas être converti en vecteur (type `bvector`). Ainsi les conversions depuis `obj` vers des sous-types requièrent des vérifications : un objet de type `obj` ne peut être converti en vecteur que si l'objet vérifie le prédicat `vector?`. Dans la définition des types de base Scheme sont indiqués les conversions possibles et les opérations de vérification à insérer dans l'arbre de syntaxe.

Les conversions entre les types de bases Bigloo et les types atomiques externes

Comme nous l'avons expliqué dans la section 4.1.4, les objets externes atomiques n'ont pas de représentation en Scheme, mais le compilateur connaît les fonctions de conversion à invoquer pour passer d'un monde à l'autre. Ces conversions sont indiquées au compilateur lors de la définition des types atomiques externes. Par exemple, il y est déclaré qu'un entier Scheme peut être transformé en un entier C en utilisant la fonction `bint->cint`. Les fonctions de conversion peuvent se composer. Un objet Scheme quelconque peut être transformé en entier Scheme s'il vérifie le prédicat `integer?`. Un entier Scheme peut se transformer en un entier C en lui appliquant la conversion `bint->cint`. Un objet Scheme quelconque peut donc être transformé

τ_{source}	τ_{cible}	checkop	convertop
obj	bint	integer?	_
bint	obj	_	_
bint	int	_	bint->cint
obj	int	integer?	bint->cint
obj	bpair	pair?	_
bpair	obj	_	_

FIG. 4.2 - Quelques exemples de conversions de bases

en un entier C s'il vérifie le prédicat `integer?` et en lui appliquant la fonction de conversion `bint->cint`. La fonction `find-converter` de l'algorithme 4.2, invoquée sur les arguments `obj` et `int` retourne une structure où la projection *checkop* est une fonction insérant dans un arbre de syntaxe abstraite le test `integer?` alors que la projection *convertop* est une fonction qui insère dans un arbre de syntaxe l'appel à la fonction `bint->cint`. La figure 4.2 contient à titre d'exemple quelques unes des nombreuses conversions permettant de convertir des types Scheme et des types externes atomiques.

Les conversions entre les types Bigloo et les types externes composés

Pour chaque type composé présent dans les clauses « foreign », Bigloo crée deux types. Le premier pour désigner la vision C des objets externes, le second pour la vision Scheme de ces mêmes objets. Le compilateur met en place des conversions entre ces deux types et les autres types déjà existants. Le mécanisme mis en œuvre est toujours le même quelque soit le type composé défini. Pour illustrer les créations de types, nous allons donc seulement montrer ce qui se passe pour les structures. Ce cas est plus complexe que les autres car Bigloo produit pour les structures un troisième type qui est un pointeur sur les structures. Une fois que les trois types ont été définis il ne reste plus qu'à mettre en place les conversions. Ces conversions utilisent un certain nombre de fonctions de la bibliothèque d'exécution que nous présentons ici :

c-cforeign->bforeign Cette fonction convertit un objet C vers un objet externe Scheme. C'est cette fonction qui alloue des objets de type `foreign_t`. La fonction `c-cforeign->bforeign` prend en argument l'objet à encapsuler ainsi que le nom de son type. Cela permet l'implantation de la fonction `c-foreign-is?`.

c-foreign-is? Ce prédicat prend en argument, un objet et un nom de type externe et retourne vrai si l'objet est un externe dont le type a pour nom le deuxième paramètre effectif. Cette fonction s'implante très facilement par quelques macros C :

```
#define FOREIGN( o )          CREF( o)->foreign_t
#define FOREIGN_ID( o )       FOREIGN( o ).id
#define FOREIGN_VALUE( o )    FOREIGN( o ).value

#define FOREIGN_ISP( o, key ) (FOREIGNP( o ) && (EQP( FOREIGN_ID( o ), key)))
```

De plus, Bigloo construit une fonction d'allocation externe dont le nom est la juxtaposition du préfixe "`make-`" et du nom du type. Supposons que la structure porte le nom `struct`, voici le prototype de la fonction créée par Bigloo :

```
(define bstruct (make-struct bobj int) "allocate_foreign")
```

Cette fonction retourne donc un objet externe de Scheme (`bstruct` et non pas `struct`), qui est obtenu en invoquant la fonction de la bibliothèque `allocate_foreign` avec deux arguments : l'identificateur du type (dans notre exemple, le symbole `struct`) et la taille de la structure C. Cette définition donne à Bigloo une vision correcte vis-à-vis des types de la fonction `allocate_foreign` pour la structure qu'on est en train de définir.

Les fonctions de conversion entre le type désignant l'objet C et le type désignant l'objet Scheme doivent être créées par le compilateur. Comme pour la fonction de création, il va s'agir ici de donner des prototypes corrects, à des fonctions génériques déjà existantes. Examinons en premier lieu la conversion de Scheme vers C. Il s'agit juste ici de suivre un pointeur. La macro C `BFOREIGN_TO_CFOREIGN` réalise cette opération. On définit cette macro pour Scheme à l'aidre de la clause :

```
(define foreign bforeign->cforeign (bforeign) "BFOREIGN_TO_CFOREIGN")
```

Pour notre exemple de la structure `struct`, il faut donner un autre prototype à cette fonction :

```
(define struct bstruct->struct (bstruct) "BFOREIGN_TO_CFOREIGN")
```

La conversion dans l'autre sens (de C vers Scheme) est faite sur le même moule. Elle s'appuie sur une fonction de la bibliothèque qui a le prototype suivant :

```
(bforeign c-cforeign->bforeign (bsymbol foreign) "cforeign_to_bforeign")
```

Le premier argument est un symbole désignant le nom du type tandis que le deuxième est un pointeur sur la valeur C.

Il ne reste plus alors qu'à mettre en place les conversions. Désignons par τ_C le type C de la structure, par τ_{Scheme} le type Scheme de la structure et enfin, $\tau_{Scheme*}$ le type qui désigne un pointeur sur la structure. Les conversions sont donc[2] :

$\tau_{Scheme} \mapsto$ `obj` Cette conversion ne nécessite aucune opération car le type τ_{Scheme} est un sous-type de `obj`.

`obj` $\mapsto \tau_{Scheme}$ Cette conversion ne nécessite pas de transformation physique mais elle exige une vérification de type. Elle utilise la fonction `c-foreign-is?` en lui passant en arguments l'objet à tester et l'identificateur du type τ_{Scheme}.

$\tau_C \mapsto \tau_{Scheme}$ Cette conversion est la plus complexe. Elle n'exige pas de vérification mais elle doit allouer (ou récupérer) une encapsulation sur l'objet externe. Ceci est réalisé par la fonction de la bibliothèque `c-foreign->bforeign` qui est invoquée avec l'identificateur du type τ_{Scheme} et l'adresse de l'objet externe à convertir.

$\tau_{Scheme} \mapsto \tau_C$ Cette conversion n'exige pas de vérification. L'opération de conversion est assurée par une fonction qui retourne la valeur du champ `value` (cf. le programme 4.3) des objets externes de Bigloo.

$\tau_{Scheme} \mapsto \tau_{Scheme*}$ Cette conversion est simple à réaliser car elle se contente de changer d'encapsulation pour le même objet externe. Elle n'exige pas de conversion.

$\tau_{Scheme*} \mapsto \tau_{Scheme}$ Cette conversion est aussi simple que la précédente. Elle réalise les mêmes opérations.

4.3 L'interface externe et la compilation du noyau Scheme

Nous avons jusqu'ici décrit l'interface externe et les mécanismes mis en œuvre dans le compilateur pour son implantation. Nous allons étudier maintenant son utilité dans la compilation du noyau Scheme.

4.3.1 La compilation des primitives

La vocation de Bigloo est d'offrir des performances d'exécution aussi proches que possible des langages impératifs traditionnels. Cela implique que des opérations aussi banales que les additions entières soient aussi bien compilées par Bigloo que par un compilateur C. Le code produit par la compilation de `(+fx x 1)` doit donc être `x + 1` en C. En aucun cas, il ne faut appeler une fonction de la bibliothèque qui utiliserait l'opérateur `+` de C. L'interface externe de Bigloo permet d'appeler des fonctions et des macros C depuis le code Scheme : l'addition de la bibliothèque d'exécution est donc simplement une macro C. Il en va de même

[2] Nous avons caché ici quelques conversions qui ne sont pas essentielles comme la conversion des structures externes vers le type `bool` qui permet d'utiliser les structures externes dans les expressions conditionnelles Scheme.

pour toutes les primitives simples: les fonctions d'accès `car`, `cdr`, `vector-ref`, `vector-set!`, ...sont de simples macros et ceci nous garantit de bonnes performances.

L'autre qualité de cette approche pour la compilation des primitives est de ne pas figer l'ensemble des primitives connues du compilateur. Les primitives ne subissent aucun traitement particulier ; ce sont des fonctions externes comme les autres. Il est donc aisé d'en ajouter ou de changer leur implantation. Le compilateur exige que certaines fonctions soient implantées dans la bibliothèque (par exemple, les fonctions de conversion et les fonctions de test de types) mais il n'a pas besoin de connaître les détails d'implantation de ces fonctions. Il lui suffit de savoir qu'elles existent et quels sont leurs prototypes. Les prototypes et les implantations des primitives peuvent changer sans que le compilateur ait besoin d'être recompilé.

4.3.2 La compilation des expressions conditionnelles

L'algorithme 4.1 n'est pas complet car le typage de certaines expressions n'a pas été présenté. C'est notamment le cas des expressions conditionnelles. La compilation des expressions conditionnelles Scheme s'inscrit dans le cadre d'une projection naturelle des constructions Scheme vers les constructions équivalentes C. C'est ce principe de projection naturelle qui nous permet d'obtenir du C orthodoxe. La projection des expressions conditionnelles se réduit à un simple problème de conversion de type. Pour pouvoir traduire une conditionnelle Scheme en une conditionnelle C, il suffit que le test soit traduit en un booléen C. Pour cela on étend l'algorithme de typage en l'algorithme 4.3.

```
type-atree( atree, τ1 )=
   selon atree
                ⋮
        ⟦ (if si alors sinon) ⟧:
             ⟦ (if type-atree( si, bool ) type-atree( alors, τ1 ) type-atree( sinon, τ1 )) ⟧
                ⋮
   fin
```

Algorithme 4.3: Le typage des expressions booléennes

La partie *test* d'une expression conditionnelle est convertie en une valeur booléenne C (type `bool`). Ceci nous permet de compiler très efficacement les prédicats primitifs. En effet, comme ils sont, pour la plupart, écrits en C, le type de leur résultat est souvent `bool`. Lorsque ces prédicats sont utilisés en position de test, il est donc inutile de construire une valeur Scheme booléenne. Ceci permet de supprimer un test. En revanche, quand ces prédicats ne sont pas utilisés dans des tests alors il y a conversion des types `bool` en `bbool`. Les exemples de la section 4.4 illustrent ces cas de compilations. Cette optimisation est également réalisée par le compilateur Screme [Ple91] mais nécessite un traitement *ad hoc* (nommé *type tracking phase*) dans ce compilateur. Dans Bigloo elle a été obtenue sans aucun effort particulier grâce à l'interface externe.

Cette compilation des expressions conditionnelles est simple mais, comme l'indique Steele [Ste78], pas suffisante pour obtenir de bonnes performances. Examinons la compilation de l'expression : `(if (or a b) x y)`. Voici le résultat de l'expansion du `or` et le résultat du typage de cette expression.

```
(if (let ((ortest a))                    (if (let ((ortest a))
      (if ortest                               (if (bbool->cbool ortest)
          ortest                                   (bbool->cbool ortest)
          b))                                      (bbool->cbool b)))
    x                                        x
    y)                                       y)
```

Il apparaît nettement que l'expression `(bbool->cbool ortest)` est évaluée deux fois. Ceci est source d'inefficacité. Steele ne propose pas de solution à ce problème. Nous verrons que la solution s'inscrit naturellement dans une optimisation présentée ultérieurement : l'élimination de sous-expressions communes 9.1.

4.3.3 La compilation des formes fonctionnelles

Dans l'algorithme 4.1 nous n'avons donné que le typage des appels fonctionnels où la fonction invoquée était une constante connue du compilateur. Nous allons maintenant étudier la compilation des appels fonctionnels

quand la fonction n'est pas une constante.

```
type-atree( atree, τ1 )=
   selon atree
                ⋮
         [[ (fun arg1 ... argn) ]]:
              si fun est une fonction connue du compilateur
                   alors ...
                   sinon soit fun=enforce-procedure( exp, n ),
                            convert( [[ (funcall fun type-atree( arg1, obj ) ...) ]], obj, τ1 )
   fin
```

Algorithme 4.4: Le typage des appels de fonctions calculées

La fonction `enforce-procedure` insère dans un arbre de syntaxe abstraite les vérifications indispensables pour s'assurer que le premier argument est une fonction Scheme et, si c'est le cas, que cette fonction est compatible avec le nombre d'argument (n) fourni sur le site d'appel. C'est le typage de ces expressions qui insère la construction `funcall` (voir la section 3.1) exigée par Sqil.

4.3.4 La compilation du typage dynamique

Toutes les vérifications dynamiques de type sont introduites dans l'arbre de syntaxe abstraite grâce à l'algorithme de typage 4.1. Les accesseurs et constructeurs (`car`, `vector-ref`, `+fx`, ...) prennent en argument des sous-types de `obj`. Les appels aux fonctions calculées sont traitées comme indiqué dans la section 4.3.3. Il n'y a donc pas d'autre passe de Bigloo qui se charge de la compilation du typage dynamique. Après la passe de typage, tous les tests de types sont explicités et les optimisations chargées de supprimer les calculs redondants s'appliqueront sur les opérations de vérification de type.

4.4 Des exemples complets de typage

Maintenant que nous avons exposé la mécanique utilisée pour convertir et tester les objets, nous allons examiner quelques exemples complets de typage.

4.4.1 Les listes

Commençons par de la manipulation de listes. Le programme:

```
1:  (define (copy-list l)
2:     (if (null? l)
3:         '()
4:         (cons (car l)
5:               (copy-list (cdr l)))))
```

est compilé en (sans optimisation des tests redondants) :

```
1:  obj_t COPY_LIST( obj_t L )
2:  {
3:     if( NULLP( L ) )
4:        return BNIL;
5:     else
6:     if( PAIRP( L ) )
7:     {
8:        obj_t OBJ2;
9:        OBJ2 = COPY_LIST( CDR( L ) );
10:       if( PAIRP( L ) )
11:         return MAKE_PAIR( CAR( L ), OBJ2 );
12:       else F
13:         FAILURE( ..., L );
14:    }
15:    else
16:       FAILURE( ..., L );
17: }
```

Plusieurs éléments sont significatifs dans cet exemple. Le premier est la compilation de l'expression conditionnelle de la ligne *2* du code source. Le test est `(null? l)`. La fonction `null?` est définie par la clause externe suivante :

```
(define bool null? (obj) "NULLP")
```

La fonction de la bibliothèque retourne donc une valeur booléenne C (`bool`) et non pas Scheme (`bbool`), le mot-clé `define` en tête de clause signifie que la fonction introduite est en fait une macro C. La partie *test* d'une expression conditionnelle doit être convertie en type `bool` (voir l'algorithme 4.3), dans le cas présent il n'y a donc rien à faire, c'est pourquoi la ligne *2* du code source a donné lieu à la compilation canonique de la ligne *3* du code objet.

L'expression de la ligne *4* du programme source invoque les fonctions `car` et `cdr` de la bibliothèque. Ces deux fonctions sont appliquées à l'argument `l`. Leur compilation est quasi-similaire, nous n'examinerons donc que celle concernant le `cdr`. Cette fonction est définie par la clause externe suivante :

```
(define obj cdr (bpair) "CDR")
```

L'argument doit donc être de type paire Scheme. Comme le compilateur voit l'argument `l` de `copy-list` comme étant de type `obj`, il faut donc appliquer la conversion `obj` $\mapsto$ `bpair` à `l` avant de pouvoir lui appliquer la fonction `cdr`. Cette conversion ne requiert pas de modifications physiques de l'argument mais exige un test : c'est la ligne *5* du code produit. Si l'invocation de la fonction `pair?` échoue on produit un échec (ligne *12*). Le test `pair?` n'a exigé ni conversion ni test car ce prédicat est défini par la clause :

```
(define bool pair? (obj) "PAIRP")
```

4.4.2 Les opérations arithmétiques entières

Notre deuxième exemple concerne l'arithmétique entière et concerne la très traditionnelle fonction de Fibonacci. Ainsi le programme :

```
1:  (define (fib x)
2:     (if (<fx x 2)
3:         1
4:         (+fx (fib (-fx x 1)) (fib (-fx x 2)))))
```

est compilé en :

```
1:  obj_t
2:  fib( obj_t x )
3:  {
4:     if( INTEGERP( x ) )
5:        if( LT_I( x, BINT( 2 ) ) )
6:           return BINT( 1 );
7:        else
8:        if( INTEGERP( x ) )
9:        {
10:          obj_t z2;
11:
12:          z2 = FIB( SUB_I( x, BINT( 2 ) ) );
13:
14:          if( INTEGERP( x ) )
15:          {
16:             obj_t z1;
17:
18:             z1 = FIB( SUB_I( x, BINT( 1 ) ) );
19:
20:             if( INTEGERP( z1 ) )
21:                if( INTEGERP( z2 ) )
22:                   return ADD_I( z1, z2 );
23:                else
24:                   FAILURE( ..., z2 );
25:             else
26:                FAILURE( ..., z1 );
27:          }
28:          else
29:             FAILURE( ..., x );
30:       }
31:       else
32:          FAILURE( ..., x );
33:    else
34:       FAILURE( ..., x );
35: }
```

Pour mettre particulièrement en évidence l'introduction des tests de type, nous avons volontairement désactiver toutes les optimisations utilisées normalement par Bigloo qui réduisent en proportion très importante le nombre de ces tests. La fonction <`fx` est définie par la clause :

```
(define bool <fx (bint bint) "LT_I")
```

C'est pourquoi la compilation du test de la ligne *2* a introduit une vérification (ligne *3* du code produit) mais pas de conversion. Toutes les opérations arithmétiques attendent des arguments de type `bint`, c'est pourquoi toutes utilisent le prédicat `integer?` avant d'effectuer leur calcul (lignes *3*, *6*, *9*, *14* et *15*). Le lecteur ne doit surtout pas s'affoler à la lecture de ces tests de type redondants car il pourra voir dans les chapitres suivants que presque tous disparaissent.

4.4.3 Les chaînes et les symboles

Examinons pour terminer un exemple manipulant des chaînes et des symboles :

```
1:  (define (is-dummy? x)
2:     (cond
3:        ((string? x)
4:         (string-ci=? x "dummy"))
5:        ((symbol? x)
6:         (eq? x 'dummy))))
```

Le résultat de la compilation est :

```
 1:  obj_t IS_DUMMY_( obj_t X )
 2:  {
 3:     if( STRINGP( X ) )
 4:        if( STRINGP( X ) )
 5:        {
 6:           bool_t AUX;
 7:           AUX=strcicmp( BSTRING_TO_CSTRING( X ), "dummy" );
 8:           return CBOOL_TO_BBOOL( AUX );
 9:        }
10:        else FAILURE( ..., X );
11:     else
12:        if( SYMBOLP( X ) )
13:           return CBOOL_TO_BBOOL( EQP( X, 'dummy ) );
14:        else return BFALSE;
15:  }
```

L'implantation de la fonction `string-ci?` est basée sur une fonction C `strcicmp` qui est une pure fonction C, c'est-à-dire qu'elle prend en argument deux pointeurs sur des chaînes et retourne un booléen C. La fonction `string-ci?` est définie par la clause :

```
(bool string-ci? (char* char*) "strcicmp")
```

L'usage de cette fonction peut donc produire des tests de type pour vérifier que les arguments sont des chaînes et des conversions de Scheme vers C. Dans notre programme (ligne *4*), le premier argument est une valeur Scheme et le second une chaîne constante. Le compilateur n'a donc besoin de faire un test et une conversion que sur le premier argument (`x`). Le test apparaît ligne *4* dans le programme source, il peut déboucher sur la construction `failure` de la ligne *10*. Ce test de type ne doit pas être confondu avec le test de la ligne *3* qui est le test explicite du programme source (ligne *3*). La conversion de `x` est effectuée ligne *7* par la macro `BSTRING_TO_CSTRING`.

Le résultat de la fonction `is-dummy?` peut être la valeur `#f` (si l'argument n'est ni une chaîne ni un symbole), ou bien le résultat de l'invocation de `eq?` ou encore de l'invocation de `string-ci?`. La clause définissant `eq?` est :

```
(define bool eq? (obj obj) "EQP")
```

Les deux fonctions (`strcicmp` et `EQP`) invoquées pour obtenir le résultat retournent des valeurs C. Ces valeurs doivent être converties en valeurs Scheme. D'où les conversions des lignes *8* et *13*. Lorsque ces deux fonctions sont utilisées en positions de test elles ne donnent pas lieu à des conversions. Lorsque leur résultat est utilisé comme valeur, il faut avoir recours à la conversion `CBOOL_TO_BBOOL`. De façon symétrique, tous les tests que nous avons vus concernaient des invocations de fonctions externes retournant des booléens C. Si ce n'est pas le cas le test doit être converti au moyen de la fonction `BBOOL_TO_CBOOL`. Par exemple :

```
(define (foo f x)
   (if (f x)
       Exp1
       Exp2))
```

est compilé en :

```
obj_t foo( obj_t f, obj_t x )
{
   obj_t aux;
   aux = PROCEDURE_ENTRY( f )( f, x, BEOA );
   if( BBOOL_TO_CBOOL( aux ) )
      CExp1
   else
      CExp2
}
```

4.4.4 Les appels de fonctions calculés

Montrons la compilation de la fonction Scheme suivante :

```
(define (foo f x) (f x))
```

La pose de tests de type et de conversions produit le code suivant :

```
obj_t
FOO( obj_t F, obj_t X )
{
   if( PROCEDUREP( F ) )
   {
      if( PROCEDURE_CORRECT_ARITYP( F, 1 ) )
         ...
      else
         FAILURE( ... );
   }
   else
      FAILURE( ... );
}
```

4.4.5 Les fonctions C utilisées comme valeur

Pour conclure voici un petit programme utilisant des fonctions externes comme objets de première classe. Notre interface externe autorise ce traitement extrême des valeurs étrangères au langage :

```
(module extern-example
   (foreign (int square (int) "square"))
   (export  (mk-square l)))

(define (mk-square l)
   (map square l))
```

La fonction externe `square` est utilisée comme valeur en ligne *6* du programme source et voici le code compilé :

```
 1:  extern int square();
 2:  static obj_t PROCEDURE = BUNSPEC;
 3:  static obj_t FUN_1();
 4:
 5:  INITIALISATION()
 6:  {
 7:     PROCEDURE = make_fx_procedure( FUN_1, 1, 0 );
 8:  }
 9:
10:  static obj_t
11:  FUN_1( obj_t FUN_ENV, obj_t A0 )
12:  {
13:     if( INTEGERP( A0 ) )
14:     {
15:        int AUX;
16:        AUX = square( CINT( A0 ) );
17:        return BINT( AUX );
18:     }
19:     else
20:        FAILURE( ..., A0 );
21:  }
22:
23:  obj_t MK_SQUARE( obj_t L )
24:  {
25:     return MAP( PROCEDURE, L );
26:  }
```

Bigloo a produit une fonction statique (`FUN_1`) pour encapsuler la fonction externe. Ainsi la compilation de la référence de `square` se compile de la même façon que s'il s'agissait d'une variable Scheme.

4.4.6 Les structures externes

Le programme 4.4 déclare un type externe (`el`) et implante quelques opérations sur ce type. Nous allons examiner les produits de compilation de chacune des fonctions.

```
 1:  (module liste_chainee
 2:     (foreign (type el (struct ((int "key")
 3:                                (el* "next"))
 4:                               "struct el"))
 5:              (int sum-els-in-C (el*)
 6:                   "sum_els_in_C"))
 7:     (export (test n)
 8:             (sum-els-in-Scheme l)))
 9:
10:  (define (create-el next int)
11:     (let ((new (make-el)))
12:        (el-key-set!  new int)
13:        (el-next-set! new next)
14:        new))
15:
16:  (define (test n)
17:     (let ((head (make-el)))
18:        (el-key-set! head -1)
19:        (let loop ((n  n)
20:                   (c  head))
21:           (if (=fx n 0)
22:               (+fx (sum-els-in-Scheme c)
23:                    (sum-els-in-C c))
24:               (let ((new (create-el n c)))
25:                  (loop (- n 1) new))))))
26:
27:  (define (sum-els-in-Scheme head)
28:     (let loop ((el  head)
29:                (acc 0))
30:        (if (foreign-null? el)
31:            acc
32:            (loop (el-next el)
33:                  (+fx (el-key el) acc)))))
```

Programme 4.4: Les structures externes

La déclaration de type a donné lieu à la définition automatique de l'allocateur `make-el`. Cette fonction alloue un objet mais ne remplit pas les champs de la structure. La fonction `create-el` comble cette lacune. Le résultat de la compilation de cette fonction est donné dans le programme 4.5.

```
 1:  static obj_t
 2:  DEFINE_EL(obj_t NEXT, obj_t INT)
 3:  {
 4:     obj_t NEW;
 5:
 6:     {
 7:        long AUX;
 8:        AUX = sizeof(struct el);
 9:        NEW = allocate_foreign('el, AUX);
10:     }
11:
12:     if (FOREIGN_ISP(NEW, 'el))
13:        if (INTEGERP(INT))
14:           FOREIGN_STRUCT_SET(NEW, struct el *, key, CINT(INT));
15:        else
16:           FAILURE(..., C);
17:     else
18:        FAILURE(..., NEW);
19:
20:     if (FOREIGN_ISP(NEW, 'el))
21:        if (FOREIGN_ISP(NEXT, 'el))
22:           FOREIGN_STRUCT_SET(NEW, struct el *, new, NEXT);
23:        else
24:           FAILURE(..., NEXT);
25:     else
26:        FAILURE(..., NEW);
27:
28:     return NEW;
29:  }
```

Programme 4.5: La compilation de la fonction "create-el"

Les lignes *7* à *10* correspondent à la compilation de l'appel à la fonction `make-el`. Il s'agit d'une invocation de la fonction `allocate_foreign`. Le premier argument est le nom du type (ici le symbole `el`). Le deuxième argument est la taille de l'objet à allouer. Cette taille n'a pas à être calculée car C fournit la fonction `sizeof` qui s'acquitte de cette tâche. Elle est utilisée en ligne *8* du code produit. De la ligne *11* à la ligne *21* on trouve la compilation de l'expression ligne *13* du code source. Tester qu'un objet est de type `el` se fait au moyen de la fonction `FOREIGN_ISP` à qui l'on fournit le nom du type (le symbol `el`). L'utilisation de la macro `FOREIGN_STRUCT_SET` (ligne *14*) correspond à l'affectation du champ `key` à la ligne *12* dans le code source. Voici la définition de cette macro :

```
#define FOREIGN_STRUCT_REF( o, tname, slot )        \
   (((tname)(FOREIGN_VALUE( o )))->slot)

#define FOREIGN_STRUCT_SET( o, tname, slot, value ) \
   (FOREIGN_STRUCT_REF( o, tname, slot ) = value, BUNSPEC)
```

Le premier argument est l'objet, le deuxième est le nom du type, le troisième est le nom du champ qu'on affecte et enfin le quatrième est la valeur d'affectation. Ainsi l'appel de la ligne *13* sera expansé par cpp (le macro-processeur de C) en :

```
( (struct el *)(NEW.foreign_t->value)->next = NEXT, BUNSPEC )
```

Examinons maintenant le fragment du résultat de la compilation de la fonction `test` qui se situe ligne *22* et ligne *23* du programme source :

```
 1:  {
 2:     obj_t z2;
 3:
 4:     {
 5:        int aux;
 6:
 7:        {
 8:           struct el *aux_2;
 9:
10:           aux_2 = BFOREIGN_TO_CFOREIGN( c );
11:           aux = sum_els_in_C( aux_2 );
12:        }
13:
14:        z2 = BINT( aux );
15:     }
16:
17:     {
18:        obj_t z1;
19:
20:        z1 = sum_els_in_scheme( c );
21:
22:        if( INTEGERP( z1 ) )
23:           if( INTEGERP( z2 ) )
24:              ...
25:     }
26:  }
```

Ce code nous permet de voir que la tête de la liste chaînée est une valeur Scheme puisqu'il faut la convertir avant l'appel à `sum_els_in_C`.

Passons directement à la compilation de la fonction `sum-els-in-Scheme` en nous focalisant sur les lignes *32* et *33* du programme source.

```
1:  ...
2:  obj_t Z1;
3:  {
4:     int AUX;
5:     AUX = FOREIGN_STRUCT_REF( EL, struct el *, key );
6:     Z1 = BINT( AUX );
7:  }
8:
9:  if( INTEGERP( Z1 ) )
10:    if( INTEGERP( ACC ) )
11:    {
12:       obj_t AUX;
13:       struct el *AUX2;
14:
15:       AUX2 = FOREIGN_STRUCT_REF( EL, struct el *, next );
16:       AUX = cforeign_to_bforeign( 'el, AUX2 );
17:
18:       ACC = ADD_I(Z1, ACC );
19:       EL = AUX;
20:       goto LOOP;
21:    }
22:    ...
```

La variable pointant sur la tête de la liste chaînée pointe sur une valeur Scheme, comme nous l'avons déjà dit pour la compilation de la fonction précédente. En revanche, la liste elle-même est une liste de valeurs C. Si ce n'était pas le cas, on ne pourrait pas passer la liste telle quelle à C, il faudrait allouer une nouvelle structure. Ainsi, dans le code Scheme, lors du parcours de cette liste, faire la récursion sur la valeur pointée par `el-next` en ligne *32* du code source impose la construction d'une encapsulation Scheme. C'est ce qui est fait aux lignes *20-21* du code produit.

4.5 Conclusion

Nous avons présenté l'interface externe de Bigloo. Cette interface étant actuellement orientée vers C, elle permet d'utiliser dans du code Scheme toutes les constructions de ce langage. C'est-à-dire qu'elle permet d'utiliser des variables, des fonctions mais aussi des macros C dans du code Scheme ! Tous les types C, même les types composés comme les structures, unions et tableaux, sont utilisables depuis Scheme puisque l'interface est suffisamment puissante pour permettre leur définition. Inversement, cette interface permet d'utiliser dans du code C des fonctions écrites en Scheme.

L'interface entre deux langages ayant deux espaces de valeurs distincts nécessite des tests et des vérifications de type. Nous avons présenté l'algorithme se chargeant de les introduire dans l'arbre de syntaxe abstraite de Bigloo.

Tous les compilateurs manipulent au moins trois langages : le langage source, le langage cible et le langage dans lequel est écrit la bibliothèque d'exécution. Tout compilateur est donc confronté à des problèmes d'interface entre langages (par exemple pour compiler les invocations des fonctions primitives). Souvent ces compilations sont faites de façon *ad hoc*. L'interface externe a constitué un modèle général parfaitement adapté à la manipulation, dans le compilateur, des objets externes. Ainsi les algorithmes développés pour l'interface se chargent maintenant dans le compilateur de tâches aussi diverses que les invocations de primitives, les compilations des formes conditionnelles ou la compilation du typage dynamique. L'interface externe est donc presque élevée au rang de paradigme de compilation dans Bigloo.

Chapitre 5

Les transformations source à source

Les passes rassemblées dans le bloc de compilation **Syntaxe abstraite** (voir la section 3.3.1) débouchent sur la construction de l'arbre de syntaxe abstraite. Ces passes sont obligatoires, elles constituent un cheminement obligé (lecture du code source, macro expansion, construction de l'arbre). Mais en marge de leur simple rôle utilitaire, certaines de ces passes constituent de judicieux moments pour effectuer les optimisations de très haut niveau que sont les transformations source à source.

Les optimisations que nous allons présenter dans ce chapitre sont toutes simples. Elles ne reposent pas sur des théories complexes puisqu'elles sont toutes syntaxiques ! Néanmoins, malgré leur nature évidente, elles améliorent les performances du code produit car elles s'appliquent très souvent. Par exemple, ces transformations sont appliquées pour mieux compiler les appels à certaines fonctions de la bibliothèque d'exécution qui sont souvent utilisées. Elles servent dans Bigloo à des transformations aussi variées que :

- Remplacer un appel à une fonction d'arité variable par une série d'appels à des fonctions d'arité fixe produisant un résultat équivalent (par exemple, remplacer des appels à la fonction générale d'addition `+` par des appels à la fonction binaire `2+`).
- Compiler efficacement des constructions comme le `case` Scheme.
- Améliorer la compilation des expressions booléennes.
- Remplacer quelques appels calculés par des appels directs.
- Éliminer quelques cas de récursion par des programmes itératifs équivalents.

...

L'intérêt des transformations source à source peut être discuté: quelle est l'utilité d'incorporer des transformations que le programmeur peut lui-même faire dans ses programmes source? Le risque est qu'une subtile modification du texte du programme rendant impossible l'application de ces transformations, le compilateur risque alors d'avoir aux yeux du programmeur un comportement non déterministe. Toutefois plusieurs arguments nous semblent peser en leur faveur.

- La puissance du système de macros de Scheme favorise leur emploi. Une partie du programme source n'est pas visible par le programmeur puisqu'il est engendré par la macro expansion. Le programmeur ne peut donc pas toujours appliquer lui-même les transformations source à source !
- Même si l'on connaît les transformations à appliquer pour rendre plus efficace un programme, on est tenté de ne pas les appliquer. Par exemple, par économie de développement ou pour laisser le code plus lisible. Nous nous rallions donc pleinement aux arguments présentés en faveur des optimisations sources qui sont présentées dans le rapport technique décrivant le compilateur Common Lisp de CMU [Mac92a].
- Certaines des transformations que nous faisons dans Bigloo font intervenir des fonctions cachées au programmeur. Seul le compilateur peut donc faire les ré-écritures.

Nos transformations ont lieu soit dans la passe de macro expansion (**Expand**), soit dans la passe de construction de l'arbre (**Scan**). Commençons par celles de la macro expansion.

5.1 Les transformations faites lors de la macro expansion

La macro expansion est un lieu de prédilection pour insérer des transformations source à source puisque les macros prennent en argument des programmes source et retournent comme résultat de nouveaux programmes. Autrement dit, l'objet des macros est justement de faire des transformations source à source ! Nous allons les utiliser pour mieux compiler l'appel à certaines fonctions de la bibliothèque d'exécution. Il s'agit principalement des fonctions d'arité variable. Les fonctions que nous souhaitons mieux compiler sont surchargées par des macros que nous nommons *O-macro* (nommées *compiler-macro* en Le-Lisp 15). Ces macros sont différentes des macros normales car elles sont définies dans le même espace de noms que les fonctions. La définition d'une fonction utilisateur peut donc cacher une *O-macro*. Prenons l'exemple de la fonction de la bibliothèque `map`. Elle est surchargée par une *O-macro* du même nom. Ainsi la macro expansion va substituer un appel à `map` par le code produit par la macro `map`. Pourtant rien n'interdit à un programme de redéfinir la fonction `map`. Dans ce cas, il ne faut plus que la macro expansion de `map` ne s'applique. C'est pourquoi les *O-macros* sont particulières, elles ne sont pas définies dans le même espace que les autres macros.

5.1.1 Les fonctions de concaténation

En Scheme (et Lisp) presque toutes les fonctions de concaténation sont d'arité variable. Néanmoins, l'expérience montre qu'elles sont presque toujours invoquées avec seulement deux arguments. Il est dommageable de payer le coût d'une invocation de fonction d'arité variable pour deux arguments ! Le coût est inhérent à la sémantique des fonctions d'arité variable Lisp. Les arguments optionnels *doivent* être placés dans des listes. L'appel de ces fonctions fait donc de l'allocation mémoire dans le tas. Pour éviter ce désagrément pour les fonctions classiques (`append`, `string-append`, `vector-append`, ...), on les surcharge par des *O-macros* qui s'expanseront des deux façons : *(i)* il y a deux arguments, l'invocation de la fonction générale de la bibliothèque sera alors remplacée par une invocation à une fonction spécialisée prenant exactement deux arguments, *(ii)* il y a plus de deux arguments, l'invocation sera laissée telle quelle.

5.1.2 Les fonctions arithmétiques

Les fonctions arithmétiques (entière ou flottante) constituent un cas extrême des fonctions d'arité variable. Sans surcharge par des *O-macro* leurs coûts seraient très importants. Toutes les opérations numériques alloueraient de la mémoire du tas ! Les appels à ces fonctions sont donc systématiquement remplacés par des cascades d'appels à des fonctions binaires.

En plus des améliorations sur les fonctions numériques génériques, on améliore également la compilation des fonctions arithmétiques entières. Ces fonctions sont binaires, ce ne sont donc pas les invocations qui vont progresser ! Souvent un des deux arguments d'appel est constant (par exemple, lors d'un test de fin de boucle, lors d'un incrément ou d'un décrément, etc.), le principe de la transformation est alors d'effectuer lors de la compilation plutôt que lors de l'exécution, les opérations d'étiquetage. Prenons le cas de l'incrément : `(+fx x 1)`. Cette expression va être compilée en C : `ADD_I( x, BINT( 1 ))`. Cette expression est elle-même traduite par le macro processeur de C en : `((obj_t)(((long)( x ) - TAG_INT) + (long)( BINT( 1 ) )))`. C'est-à-dire qu'on supprime le tag d'un des arguments, puis on fait une véritable addition. Dans notre exemple, le deuxième argument (`1`) est une constante. On calcule donc lors de la compilation la valeur du nombre Scheme `1` moins le tag. Ainsi, lors de l'exécution, il ne reste plus qu'à faire l'addition. La *O-macro* `+fx`, se charge de faire cette transformation lorsqu'un argument est un entier. Ainsi le résultat de compilation de `(+fx x 1)` est : `ADD_I_PTAG(x, PSUB_TAG(1))`. Les macros `ADD_I_PTAG` et `PSUB_TAG` ont été vues dans le chapitre 2. Le code engendré par cpp est : `x + (1 - TAG_INT)`. La partie droite de cette addition étant constante, elle est effectuée lors de la compilation C et non pas lors de l'exécution.

5.1.3 Les fonctions `map` et `for-each`

Ces deux fonctions, `map` et `for-each`, sont tellement fréquemment utilisées qu'elles méritent un effort particulier. Elles ont un comportement assez semblable, nous n'allons donc étudier que la première : `map`. Cette fonction prend en argument une fonction f et un nombre variable de listes. Le résultat de `map` est une nouvelle liste composée des applications successives de f sur les $i^{ème}$ éléments des listes. Une implantation classique peut-être trouvée dans le programme 5.1.

```
 1:  (define (map-2 f l)
 2:     (let loop ((l l))
 3:        (if (null? l)
 4:            '()
 5:            (cons (f (car l))
 6:                  (loop (cdr l))))))
 7:
 8:  (define (map f . l)
 9:     (cond
10:        ((null? l)
11:         '())
12:        ((null? (cdr l))
13:         (map-2 f (car l)))
14:        (else
15:         (let loop ((l l))
16:            (if (null? (car l))
17:                '()
18:                (cons (apply f (map-2 car l))
19:                      (loop (map-2 cdr l))))))))
```

Programme 5.1: L'implantation classique de la fonction map

Comme on peut facilement s'en convaincre, la fonction `map` est onéreuse ! Puisque la fonction `map-2` est plus légère à invoquer, on pourrait se contenter de mettre en place une *O-macro* détournant les appels à `map` au profit de `map-2`. Nous avons adopté la solution plus efficace encore de Le-Lisp où l'appel à la fonction `map` est remplacé par l'expression calculée par la *O-macro* donnée dans le programme 5.2.

```
(define (expand-map x e)
   (match-case x
      ((?- ?fun ?list)
       (let ((l (gensym)))
          (e `(let ((,l ,list))
                (if (null? ,l)
                    '()
                    (let ((head (cons '() '())))
                       (let ,lname ((,l    ,l)
                                    (tail head))
                            (if (null? ,l)
                                (cdr head)
                                (let ((new-tail (cons (,fun (car ,l)) '())))
                                   (set-cdr! tail new-tail)
                                   (,lname (cdr ,l) new-tail)))))))
             e)))
      ((?- ?fun . ?lists)
       `(map ,(e fun e) ,@(map (lambda (l) (e l e)) lists)))
      (else
       (partial-error "map" "Illegal `map' form" x))))
```

Programme 5.2: La O-macro map

L'objet de cette transformation est de supprimer la récursion non terminale de la fonction `map`. Le principe est le maintien d'un pointeur sur le dernier élément de la liste en construction, de manière à pouvoir faire les concaténations directement, sans construire la liste dans l'ordre inverse (dans la littérature francophone cette technique est nommée *accrochage de bord* [SJ93]). L'autre intérêt de la transformation est de faire une propagation de constante sur les arguments de `map`. Ainsi la fonction passée normalement en argument n'apparaît plus que sur des sites d'appels (elle n'est plus utilisée comme argument). Cela peut économiser une fermeture. Voyons sur l'exemple `(map negfx l)` l'effet de cette macro expansion :

```
 1:  (let ((:-d.1000 l))
 2:     (if (null? :-d.1000)
 3:         '()
 4:         (let ((head (cons '() '())))
 5:            (labels ((_loop_ (:-d.1000 tail)
 6:                        (if (null? :-d.1000)
 7:                            (cdr head)
 8:                            (let ((new-tail (cons (negfx (car :-d.1000)) '())))
 9:                               (begin
10:                                  (set-cdr! tail new-tail)
11:                                  (_loop_ (cdr :-d.1000) new-tail))))))
12:              (_loop_ :-d.1000 head)))))
```

On peut mieux se convaincre sur une expansion que sur le code de la macro elle-même que tous les appels récursifs sont maintenant terminaux (ligne *11* et ligne *12*). On voit aussi que la fonction `negfx` n'est plus

utilisée en valeur mais est directement appliquée (ligne *8*). Bien sûr l'inconvénient de cette transformation est le grossissement important de la taille du programme.

5.1.4 La fonction `vector`

La fonction **`vector`** constitue elle aussi un cas à part. Elle est plus complexe que les autres. Une implantation naïve de cette fonction n'est pas du tout satisfaisante. Cette fonction prend un nombre n d'arguments et construit un vecteur de taille n rempli avec les arguments optionnels. L'implantation naïve de cette fonction est donc :

```
(define (vector . n) (list->vector n))
```

Chaque invocation de cette fonction alloue une liste de taille n puis un vecteur de taille n. La taille totale des allocations est donc $3 * n$ (car une paire contient deux mots). Ceci n'est pas acceptable ! Nous faisons donc la transformation suivante :

$$\mathcal{EXPAND}_{vector}\ [\![\texttt{(vector } arg_0 \ \ldots \ arg_n \texttt{)}]\!] = \left[\!\!\left[\begin{array}{l} \texttt{(let ((vec (create-vector } n \texttt{)))} \\ \quad \texttt{(vector-set! vec 0 } arg_0 \texttt{)} \\ \quad \vdots \\ \quad \texttt{(vector-set! vec } n\ arg_n \texttt{)} \\ \quad \texttt{vec)} \end{array} \right]\!\!\right]$$

La fonction introduite par la transformation $\mathcal{EXPAND}_{vector}$ (**`create-vector`**) ne peut pas être visible du programmeur car elle alloue un vecteur sans initialiser ses champs. Une utilisation maladroite pourrait donc conduire à un programme ayant des comportements inacceptables.

5.1.5 La forme spéciale `case`

La forme spéciale **`case`** est le dernier point particulier que nous souhaitions aborder ici. Une implantation « facile » de cette construction est de la projeter sur des cascades de **`if`**. Ceci est néanmoins dommage car, dans certain cas, cette construction pourrait être compilée vers C comme un **`switch`**, les compilateurs C sachant bien compiler ces constructions[1]. Pour profiter de cette compétence du compilateur, **`case`** est une macro qui étudie les types de ces constantes. Si toutes sont des entiers ou des caractères alors la construction sera compilée en un **`switch`** C. Autrement elle est traduite en une cascade de **`if`**.

Maintenant que nous avons vu les transformations réalisées lors de la macro expansion, examinons celles faites lors de la construction de l'arbre de syntaxe abstraite.

5.2 Les transformations de la passe Scan

Les macros prennent en argument des expressions représentant des appels de fonction et construisent de nouvelles expressions. La passe **Scan** construit l'arbre de syntaxe abstraite. C'est-à-dire que les fonctions appliquées lors de cette passe prennent des morceaux de définition. Ainsi elles ont donc une «vision» plus vaste que celles des macros. Cette passe permet donc des transformations qu'on ne peut exprimer avec des macros.

5.2.1 Les expressions conditionnelles

Les expressions conditionnelles sont compilées par "court-circuit" [ASU86, Ste78] mais la compilation de la négation pose problème. Examinons la compilation de l'expression **`(if (not`** *test*`)`** *alors sinon*`)`. La

[1] *Sachant partiellement bien* devrait-on écrire, car, même si un **`switch`** est compilé avec une table de saut, aucun compilateur, à notre connaissance, ne parvient à omettre de produire du code, vérifiant que la valeur discriminée est située dans l'intervalle du **`switch`**. On devrait pourtant y échapper en utilisant des types **`enum`** plutôt que des entiers ou des caractères!

négation est expansée en ligne. On doit donc en fait compiler l'expression `(if (if` *test* `#t #f)` *alors sinon*`)`. Il apparaît clairement que deux tests sont effectués. Comme nous l'avons déjà mentionné dans la section 4.3.2, les tests redondants seront supprimés, en revanche, les tests introduits par des négations doivent subir un traitement particulier. C'est ce qu'effectue la passe **Scan** en appliquant les deux transformations évidentes suivantes :

$$\mathcal{SCAN}_{not_1} \left[\!\!\left[\begin{array}{l} \texttt{(if (not } si\texttt{)} \\ \quad alors \\ \quad sinon\texttt{)} \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(if } si \\ \quad sinon \\ \quad alors\texttt{)} \end{array} \right]\!\!\right]$$

et

$$\mathcal{SCAN}_{not_2} \left[\!\!\left[\begin{array}{l} \texttt{(if (if } si \texttt{ \#f \#t)} \\ \quad alors \\ \quad sinon\texttt{)} \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(if } si \\ \quad sinon \\ \quad alors\texttt{)} \end{array} \right]\!\!\right]$$

Grâce à ces deux transformations un seul test subsiste. Les négations sont donc mieux compilées.

5.2.2 Les formes `let` et `letrec`

Les constructions `let` et `letrec` qui introduisent des fonctions locales sont, quand c'est légal, transformées en des constructions `labels`. Cette transformation est autorisée quand la variable liée à une λ-expression n'est jamais affectée (n'est jamais le premier argument d'un `set!`) car les formes `labels` n'introduisent que des constantes. Ainsi, pour toute liaison concernant une variable non affectée et une fonction, Bigloo applique la transformation $\mathcal{SCAN}_{let}$:

$$\mathcal{SCAN}_{let} \left[\!\!\left[\begin{array}{l} \texttt{(let (...} \\ \quad \texttt{(}F\ \texttt{(lambda } args\ body\texttt{))} \\ \quad \texttt{...)} \\ \quad \ddots \\ \quad \texttt{(}F\ \texttt{...)} \\ \quad \vdots \\ \quad F \\ \quad \quad \ddots\texttt{)} \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(labels ((}F\ args\ body\texttt{))} \\ \quad \texttt{(let (...)} \\ \quad \quad \ddots \\ \quad \quad \texttt{(}F\ \texttt{...)} \\ \quad \quad \vdots \\ \quad \quad F \\ \quad \quad \quad \ddots\texttt{)} \\ \texttt{)} \end{array} \right]\!\!\right]$$

La forme `lambda` n'existe pas dans l'arbre de syntaxe abstraite. Elle doit donc totalement disparaître lors de la construction de l'arbre de syntaxe. La transformation $\mathcal{SCAN}_{let}$ fait disparaître des `lambda` au profit des constructions `labels`. Les formes `lambda` qui subsistent subissent la transformation $\mathcal{SCAN}_{\lambda}$ suivante :

$$\mathcal{SCAN}_{\lambda} \left[\!\!\left[\texttt{(lambda } \boxed{args\ body} \texttt{)} \right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(labels ((}fun\ \boxed{args\ body}\texttt{))} \\ \quad fun\texttt{)} \end{array} \right]\!\!\right]$$

La transformation $\mathcal{SCAN}_{\lambda}$ est de bien moins bonne qualité que $\mathcal{SCAN}_{let}$ car elle introduit une fonction qui est utilisée comme valeur (la fonction *fun* est le résultat du `labels`). Ainsi, lors des exécutions, il y aura créations dynamiques de fonctions. C'est pourquoi la transformation $\mathcal{SCAN}_{let}$ joue un rôle important dans la qualité du code produit.

5.2.3 Les applications calculées

Les autres transformations que nous effectuons ont pour but de transformer des appels calculés en appels directs. Ces transformations s'appliquent principalement sur des programmes générés par macro expansion. Elles sont au nombre de trois. La première s'applique sur des applications dont la fonction est obtenue par évaluation d'une forme `begin`.

$$\mathcal{SCAN}_{begin}\left[\!\left[\begin{array}{l}\texttt{(}\\ \boxed{\begin{array}{l}\texttt{(begin}\\ \quad exp_0\\ \quad \vdots\\ \quad exp_n\texttt{)}\end{array}}\\ \quad actual_0\ \ldots\ actual_n\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\boxed{\begin{array}{l}\texttt{(begin}\\ \quad exp_0\\ \quad \vdots\\ \quad (exp_n\ actual_0\ \ldots\ actual_n))\end{array}}\right]\!\right]$$

La deuxième est une simple β-réduction.

$$\mathcal{SCAN}_{\beta}\left[\!\left[\begin{array}{l}\texttt{((lambda (}formal_0\ \ldots\ formal_n\texttt{)}\\ \quad \boxed{Exp}\texttt{)}\\ \quad actual_0\\ \quad \vdots\\ \quad actual_n\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\texttt{(let ((}formal_0\ actual_0\texttt{)}\\ \qquad \vdots\\ \qquad (formal_n\ actual_n))\\ \quad \boxed{Exp}\texttt{)}\end{array}\right]\!\right]$$

et la troisième est l'application d'une forme conditionnelle.

$$\mathcal{SCAN}_{if}\left[\!\left[\begin{array}{l}\texttt{(}\\ \boxed{\begin{array}{l}\texttt{(if }si\\ \quad alors\\ \quad sinon\texttt{)}\end{array}}\\ \quad actual_0\\ \quad \vdots\\ \quad actual_n\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\texttt{(let ((}actual'_0\ actual_0\texttt{)}\\ \qquad \vdots\\ \qquad (actual'_n\ actual_n))\\ \boxed{\begin{array}{l}\texttt{(if }si\\ \quad (alors\ actual'_0\ \ldots\ actual'_n)\\ \quad (sinon\ actual'_0\ \ldots\ actual'_n))\end{array}}\\ \texttt{)}\end{array}\right]\!\right]$$

Les bénéfices de ces transformations sont évidents : dans les trois transformations un appel calculé est remplacé par un appel direct (voir le chapitre 2 et plus précisément la section 2.3.2 pour connaître les différences de coût entre les appels directs et les appels calculés).

5.3 Les mesures de performances

Il n'est pas possible de désactiver l'application des transformations énoncées dans ce chapitre. Ainsi il ne nous est pas permis de comparer les performances du code produit avec et sans les optimisations. En revanche nous avons pu compter, sur nos programmes tests, le nombre de fois que les transformations s'appliquent. Ce sont ces chiffres que nous présentons dans les figures 5.1 et 5.2.

En plus des programmes tests habituels, nous avons également mesuré le nombre d'application des transformations sur la passe **Scan** du compilateur lui-même. Cette passe contient 1543 lignes de Scheme. Les mesures appellent certaines remarques :

- Les transformations s'appliquent de façon très inégale et très dépendante du type du programme. Ceci était prévisible. Par exemple la transformation $\mathcal{EXPAND}_{vector}$ ne s'applique que sur les programmes utilisant la fonction `vector` (de nos programmes test, seul **Earley** l'utilise).

	$\mathcal{EXPAND}_{arith}$	$\mathcal{EXPAND}_{append}$	$\mathcal{EXPAND}_{map}$	$\mathcal{EXPAND}_{vector}$
peval	36	0	12	0
leval	5	0	6	0
earley	66	2	2	2
conform	20	1	15	0
passe scan	10	16	19	0

FIG. 5.1 - Le nombre d'occurrences des transformations source à source $\mathcal{EXPAND}$

	$\mathcal{SCAN}_{not_1}$	$\mathcal{SCAN}_{not_2}$	$\mathcal{SCAN}_{let}$	$\mathcal{SCAN}_{\lambda}$	$\mathcal{SCAN}_{begin}$	$\mathcal{SCAN}_{\beta}$	$\mathcal{SCAN}_{if}$
peval	0	3	10	28	0	14	7
leval	0	12	2	60	0	11	0
earley	0	4	32	34	0	4	0
conform	0	5	20	29	0	13	1
passe scan	0	80	3	7	0	26	0

FIG. 5.2 - Le nombre d'occurrences des transformations source à source $\mathcal{SCAN}$

- La transformation $\mathcal{SCAN}_{not_1}$ semble ne jamais s'appliquer. Ceci s'explique simplement. La fonction de la bibliothèque d'exécution `not` est une fonction intégrable. La passe d'intégration survient après les transformations source à source. Des intégrations de fonctions peuvent nécessiter un parcours local de l'arbre de syntaxe par la passe **Scan**. Autrement dit, le processus de la passe **Scan** est réutilisé dans Bigloo après la passe d'intégration. Dans ce cas la transformation $\mathcal{SCAN}_{not_1}$ peut s'appliquer. Dans nos mesures, nous avons stoppé la compilation avant l'intégration. C'est pourquoi cette transformation semble ne jamais s'appliquer.
- Les transformations $\mathcal{EXPAND}_{arith}$, $\mathcal{EXPAND}_{map}$ et $\mathcal{SCAN}_{not_2}$ s'appliquent très souvent. Pour nous, cette fréquence justifie à elle seule la présence des transformations source à source dans Bigloo.

5.4 Récapitulatif et extensions

Les transformations syntaxiques présentées sont très simples à mettre en œuvre. Néanmoins, elles améliorent sensiblement la qualité du code produit puisqu'elles s'appliquent à des programmes usuels (manipulation de nombres, conditionnelle, etc.). On pourrait envisager de faire plus de transformations source à source et plus particulièrement des dérécursivations. Plusieurs recherches ont été menées sur le traitement itératif de programmes récursifs, soit par transformations de programme à l'exécution [SJ90, SJ84] soit par réécriture [Dur85]. La première approche est pour nous sans avenir car elle suppose un moyen de contrôle sur l'interprète qui évalue le programme. La seconde, en revanche, pourrait être utile. Malheureusement les schémas de dérécursivations connus utilisent souvent des propriétés sémantiques des fonctions comme l'associativité, la "quasi-associativité"[2] ou la connaissance d'éléments neutres. Or, ces propriétés sont extrêmement difficiles à calculer automatiquement. Il en résulte que les seuls programmes que le compilateur pourra dérécursiver sont les récursions mettant en jeu les quatre opérations arithmétiques élémentaires et les récursions portant sur des allocations (par exemple `copy-list`). Ceci donne un nombre d'applications potentiellement faible[3].

[2] La quasi-associativité est une propriété moins forte que l'associativité. Sa définition formelle est :

$$\forall \theta : Z \times Y \mapsto Y; \zeta : T \times Y \mapsto Y; \theta^* : T \times Z \mapsto T$$

θ est quasi-associative avec θ^* via ζ si et seulement si :

$$\zeta(t, \theta(z, y)) = \zeta(\theta^*(t, z), y)$$

Par exemple, l'*ajout d'éléments* à gauche est quasi-associatif avec l'*ajout d'éléments à droite* via la concaténation.

[3] Nous avons fait une recherche dans les sources complètes du compilateur pour déterminer si cette transformation serait profitable. Nous n'avons trouvé que 4 cas d'application!

C'est pourquoi notre compilateur n'inclut pas à ce jour de telles transformations.

Chapitre 6

L'intégration fonctionnelle

L'intégration des fonctions (*inline expansion* ou *open-coding* en anglais) est une opération qui consiste à remplacer un appel de fonction par son corps. Cette optimisation améliore les performances du code produit pour plusieurs raisons : *(i)* Elle supprime le coût de l'invocation de fonctions (passage des arguments, sauvegarde du contexte, déplacement de registre, rupture du flot de contrôle). *(ii)* Les optimisations locales du compilateur seront plus efficaces car les morceaux de code qu'elles traiteront seront plus significatifs. *(iii)* L'intégration de fonction récursives peut contribuer à diminuer le nombre de fermetures allouées. A. Appel mentionne [App92] que dans le compilateur Sml/NJ, cette optimisation est la plus efficace. C'est elle qui est susceptible d'apporter le plus d'améliorations aux performances du code produit. Ainsi elle permet d'écrire des programmes lisibles et efficaces. Par exemple, abstraire des structures conduit généralement à la création d'une multitude de petites fonctions où le coût de l'invocation est plus lourd que celui du calcul effectué. Ce surcoût disparaît totalement grâce à l'optimisation d'intégration en ligne.

Cette optimisation ne doit être appliquée que prudemment car elle peut faire augmenter la taille des programmes dans des proportions importantes. Le corps des fonctions intégrées étant dupliqué, le compilateur doit adopter un compromis raisonnable entre gain de performance et respect d'une taille décente.

L'intégration est parfois confondue avec la macro expansion, ou plus exactement la macro expansion est parfois utilisée à des fins d'intégration. Ces mécanismes sont pourtant très différents. L'appel à une macro est un appel par nom, alors que l'appel à une fonction *expansée* a le même protocole d'appel que celui des fonctions classiques. Les macros ne sont pas des valeurs de première classe, alors que les fonctions intégrées le sont. Utiliser des macros pour singer le comportement des fonctions intégrées en ligne, alors qu'elles ne sont pas conçues pour ça, révèlent les faiblesses du compilateur.

On ne peut pas s'en remettre aux compilateurs C pour réaliser l'optimisation d'intégration. Tout d'abord, peu nombreux sont ceux qui font réellement cette transformation (parfois ils offrent une déclaration de fonction *inline* mais rarement, ils prennent seuls l'initiative d'expanser des fonctions quelconques). Ensuite parce que nous profitons de l'intégration pour augmenter la portée de certaines de nos optimisations comme par exemple les optimisations du flot de contrôle.

6.1 La construction `define-inline`

Le rôle de l'intégration est double dans Bigloo. C'est à la fois une optimisation très efficace (parce qu'elle s'applique souvent et qu'elle améliore notablement le code produit) et c'est aussi le paradigme de compilation des primitives.

Dans le chapitre sur l'interface externe (chapitre 4) on sous-entendait que certaines primitives étaient implantées par des fonctions externes (`car`, `cons`, `+fx`). Ceci n'est pas exact, car les fonctions externes ne satisfont pas toutes les exigences des primitives. Par exemple, les primitives doivent pouvoir être redéfinies ; or les fonctions externes qui ont un comportement particulier vis-à-vis du langage de module ne le peuvent pas. Cependant, la compilation de ces primitives doit être aussi efficace que si elles étaient effectivement implantées par des fonctions externes. Nous avons recours à la technique d'intégration pour résoudre ce problème. Il nous faut ainsi disposer d'une construction forçant le compilateur à faire de l'intégration. C'est

le rôle de la construction `define-inline` (voir [Ser94a]).

La seule différence entre la forme spéciale `define-inline` et la classique `define` est que notre nouvelle forme ne peut introduire que des fonctions. En revanche, il n'y a pas de restriction sur la nature de ces fonctions ; en particulier elles peuvent être récursives (mutuellement récursives ou simplement récursives).

Comme nous le montrerons par la suite, le compilateur prend l'initiative d'intégrer certaines fonctions classiques. La construction `define-inline` est néanmoins utile car elle permet au compilateur de posséder plus d'informations qu'avec la forme `define`. Bigloo réalise de la compilation séparée, c'est-à-dire qu'il compile des modules indépendamment les uns des autres. Cela signifie qu'il ne connaît pas le corps des fonctions importées. Il ne peut donc pas prendre la décision de les intégrer. Par définition toutes les fonctions de la bibliothèque sont importées ; sans `define-inline` aucune ne pourrait donc être intégrée ! La construction `define-inline` *traverse* les modules. Elle force le compilateur à lire des morceaux de certains modules importés.

6.2 Quand faire de l'intégration?

Le principe de l'intégration est de dupliquer le corps des fonctions. Ceci entraîne une augmentation de la taille du code produit. La difficulté majeure de l'intégration est de contrôler ce facteur d'expansion. Quand doit-on intégrer des appels de fonctions et quand doit-on les laisser subsister?

Plusieurs types de solutions ont été proposés :

- Le premier est le plus simple. Seules les fonctions explicitement déclarées *intégrables* sont intégrées ! Cette solution est largement répandue, par exemple, dans le compilateur Common Lisp Python [Mac92b] et dans la plupart des compilateurs C. Ainsi, nous ne sommes pas parvenus à trouver un seul compilateur C qui intègre la fonction `make_cons`, sur le programme suivant :

```
 1:  struct cons {                                18:
 2:     int car;                                  19:
 3:     int cdr;                                  20:  int bar( x, y )
 4:  };                                           21:  {
 5:                                               22:     struct cons *p;
 6:  extern struct cons *malloc();                23:     p = make_cons( x, y );
 7:                                               24:
 8:  static                                       25:     return (p->car + p->cdr);
 9:  struct cons *make_cons( car , cdr )          26:  }
10:  {                                            27:
11:     struct cons *p;                           28:  int foo( x, y )
12:     p = malloc( sizeof( struct cons ) );      29:  {
13:     p->car = car;                             30:     struct cons *p;
14:     p->cdr = cdr;                             31:     p = make_cons( x, y );
15:                                               32:
16:     return p;                                 33:     return (p->car - p->cdr);
17:  }                                            34:  }
```

  Même gcc ne fait pas l'intégration. Il faut dire que la documentation de gcc fait état de l'intégration des *fonctions suffisamment simples* (*"simple enough"* dans le texte) mais il est difficile de savoir ce que cela signifie[1] !

- La deuxième catégorie [Sch77, HC89] repose sur des études statistiques lors des exécutions. On compile un programme puis on l'exécute dans un mode particulier où l'on trace les appels de fonctions. On obtient ainsi des informations sur la fréquence des appels, le temps passé dans le corps des fonctions, etc. Ensuite, on recompile le programme en se servant des informations récoltées pour choisir d'intégrer telle ou telle fonction. Cette méthode nous semble très lourde à mettre en œuvre. Elle demande trop de collaboration de la part du programmeur.

[1] Cet exemple montre clairement, qu'on ne peut pas s'en remettre au compilateur C pour faire l'intégration. Dans le programme que nous avons donné en exemple, on souhaite tout particulièrement que la fonction `make_cons` soit intégrée.

- La troisième méthode, repose sur une étude entièrement statique [App92] des programmes. Elle consiste à étudier pour chaque appel de fonction le gain que peut représenter l'intégration de l'invocation. Ce gain ne provient que des autres optimisations (propagation de constantes par exemple) que le compilateur pourra appliquer s'il expanse l'invocation. Ainsi le calcul du gain se fait en fonction des utilisations des paramètres formels au sein de la fonction et des approximations des paramètres effectifs. Par exemple, si une fonction F prend un argument et fait un branchement indexé sur cet argument et si le paramètre effectif est un nombre (une constante représentant un nombre), alors l'optimisation de propagation de constantes pourra totalement supprimer le branchement. Le gain sera donc important. Enfin, la croissance de la taille du code est bornée par une limite dépendante de la taille du programme initiale.

  Cette approche ne convient pas dans le cadre de Bigloo car elle repose sur l'axiome que l'appel de fonction a un coût très faible (comme l'explique A. Appel, grâce au CPS et à une bonne allocation de registres, l'appel fonctionnel ne coûte qu'un branchement direct (`goto`)). Dans cette optique, le compilateur ne va décider d'intégrer des fonctions que s'il parvient à montrer que des optimisations ultérieures s'appliqueront dans le corps des fonctions intégrées. Pour Bigloo, l'appel de fonction est onéreux ; nous voulons donc le supprimer, même si aucune autre optimisation ne s'applique. De plus, nous pensons que l'intégration de fonction permet aux optimisations de mieux s'appliquer non pas dans le corps de la fonction intégrée mais dans le corps de la fonction qui reçoit l'intégration. Examinons le petit programme suivant :

  ```
  (define (foo x)
     (cons x x))

  (define (bar x)
     (car (foo x))
     ...)
  ```

  La fonction `foo` est un parfait exemple de fonction que nous cherchons à intégrer. Le compilateur Sml/NJ n'intègrera pas `foo` dans `bar` car il ne connaît pas les valeurs des paramètres formels. Or pour Bigloo, intégrer `foo` dans `bar` permettra de supprimer le test de type, autrement obligatoire de la fonction `car`.

- L'approche présentée dans [DC94] s'apparente à celle d'A. Appel. Elle part de la constatation qu'on ne peut prendre la décision de faire l'intégration que lorsqu'on connaît les résultats des optimisations ultérieures sur le corps des fonctions intégrées. Les deux méthodes se distinguent car Appel estime les gains des optimisations alors que Dean & Chambers appliquent les optimisations, puis examinent ce qu'elles ont amélioré. Les auteurs montrent qu'en gérant une base de données des fonctions, les coûts de ces « essais de compilations » ne sont pas trop importants. Ceci n'est pas possible pour Bigloo, car les optimisations appliquées ne sont pas toutes locales. Certaines nécessitent un parcours de tout l'arbre (comme l'analyse de flot de contrôle). Une telle technique de choix d'intégration prendrait trop de temps à la compilation !

- D'autres critères d'intégrations reposent principalement sur la taille des fonctions invoquées. La mesure de la taille dépend des compilateurs. Dans SELF [Cha92c], une fonction est intégrée si elle comporte moins d'envois de message qu'une certaine constante du compilateur. Le compilateur gcc [Sta89] n'intègre une fonction que si elle ne comporte qu'un petit nombre d'instructions RTL. Cette approche nous semble assez intéressante mais elle n'est pas suffisante, car elle ne permet pas à elle seule, d'éviter de tomber dans des boucles infinies lors d'intégration de fonctions récursives.

Aucune des méthodes proposées pour décider d'intégrer une fonction ne nous satisfait pleinement. Nous allons donc présenter celle que nous avons conçue pour Bigloo.

6.2.1 Quand Bigloo intègre-t-il une fonction ?

Notre algorithme de décision est simple mais empiriquement nous avons constaté qu'il donnait des résultats satisfaisants. Voici la liste des cas où Bigloo peut choisir d'intégrer une application fonctionnelle. Dans tous les cas, l'argument en position fonctionnelle (la fonction invoquée) doit être une constante connue statiquement

expressions	notation	signification
d la définition d'une fonction	$d \downarrow_{fun}$	le corps de la fonction définie
exp une application	$exp \downarrow_{fun}$	l'objet en position fonctionnelle de l'expression
exp une application	$exp \downarrow_{actuals}$	les arguments de l'application
exp une application	$\forall e \in exp$	$\{exp \downarrow_{fun}\} \bigcup exp \downarrow_{actuals}$
exp une expression	$\forall e \in exp$	les arguments de exp
fun une fonction	$fun \downarrow_{arity}$	l'arité de la fonction fun
fun une fonction	$fun \downarrow_{formals}$	les paramètres formels de la fonction fun
Soit fun une fonction	$fun \downarrow_{body}$	le corps de la fonction fun.
Soit $pred$ un prédicat	$\overline{pred}$	la négation du prédicat $pred$.

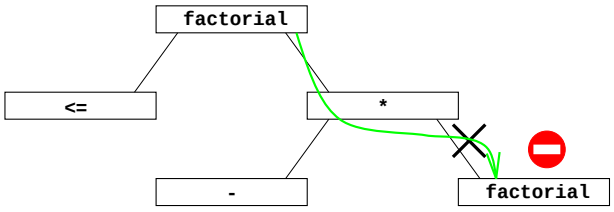
FIG. 6.1 - Le pseudo-code

du compilateur :

- La fonction a été déclarée comme intégrable (au moyen de la primitive `define-inline`). Le compilateur respecte les consignes du programmeur.
- La fonction est petite (c'est-à-dire que la taille de l'arbre de syntaxe abstraite correspondant à son code a un nombre de feuilles qui est plus petit qu'une certaine constante[2]). Nous reviendrons ultérieurement sur la notion de « petite ».
- La fonction n'est invoquée qu'une seule fois dans tout le programme. Dans ce cas, même si la fonction est très grosse, il n'y aura bien sûr pas d'augmentation de la taille du code produit si on l'intégre. L'application de ce critère peut tout de même ralentir la compilation car, en augmentant la taille des fonctions, certaines optimisations locales (CSE par exemple), dont la complexité dépend directement de cette taille ralentissent de façon perceptible.

Les critères que nous venons de donner s'appliquent aux fonctions invoquées. Ils ne requièrent aucune information sur les sites où elles sont appelées. Certains contextes interdisent pourtant l'intégration :

- Pour que le processus d'intégration ne puisse pas tomber dans des boucles infinies, nous interdisons au compilateur d'intégrer l'appel à une fonction s'il est déjà en train d'en intégrer un. Ce critère garantit que traitement correct des récursions simples *et* croisées.



- Le critère de « taille des arbres » que nous appliquons sert à contrôler l'expansion du code à compiler. Ce critère n'est pas suffisant. En effet imaginons un appel à une petite fonction, qui invoque elle-même une autre petite fonction, qui invoque elle-même un autre petite fonction, etc. Bien que chaque appel à une petite fonction n'augmente que dans une faible proportion la taille du programme, si le compilateur intègre tous les appels, l'augmentation devient tout de même trop importante. C'est pour éviter ce phénomène que nous interdisons trop d'intégrations en cascade.

L'algorithme 6.1 contient en pseudo-code l'algorithme d'intégration (la figure 6.1 contient une description rapide du pseudo-code que nous utilisons). Il s'applique à l'ensemble du programme. Le prédicat *primitive-inline*? retourne vrai si, et seulement si, la fonction a été définie par une construction `define-inline`.

[2] Dans les faits, le seuil de taille à partir duquel Bigloo refuse d'intégrer des fonctions n'est pas une constante puisqu'il dépend du degré d'optimisation.

```
D: L'ensemble des définitions du programme

MAXL: Une constante entière

∀d∈D
    d↓exp ← inline-exp( d↓exp, 0, ∅ )

inline-exp( exp, level, stack )=
   ∀e∈exp
       e ← inline-exp( e, level, stack ),
   si exp est un appel de fonction et inline-site?( exp, level, stack )
         alors inline-call( exp, level, stack )
         sinon exp

inline-site?( exp, level, stack )=
   si exp↓fun∈stack
         alors faux
   ou bien si primitive-inline?( exp↓fun )
         alors vrai
   ou bien si level > MAXL
         alors faux
   ou bien si exp↓fun n'est invoquée qu'une seule fois dans tout le progamme
         alors vrai
   sinon small-size?( exp↓fun )
```

Algorithme 6.1: L'expansion en ligne globale d'un programme

La fonction *inline-site*? regroupe les critères que nous avons précédemment énoncés. L'algorithme utilisé mélange donc plusieurs types de critères, notamment la décision d'intégrer une fonction sur *un* site peut être prise alors qu'elle est refusée pour un autre. Ceci signifie que notre algorithme ne prend pas la décision d'intégrer des fonctions mais des sites d'appel. Avant de voir le mécanisme d'intégration, détaillons le prédicat *small-size*?

Les petites fonctions

La notion de « petite fonction » est subjective ! Il ne nous faut pas perdre de vue l'objectif principal : contrôler l'intégration afin que la taille du code ne grossisse que dans des proportions raisonnables. Un critère expéditif consiste à dire qu'une fonction est petite si sa taille est inférieure à une certaine constante k. Ce critère n'est pas satisfaisant. En effet, imaginons une fonction f ayant k arguments se contentant de les additionner. La taille du corps de f serait donc $k+1$: elle ne serait pas intégrée. Pourtant la taille d'une expression contenant une invocation de f est elle-même au moins de taille $k+1$. Ceci signifie que remplacer un appel à f par son corps, n'entraîne aucune augmentation de la taille du programme.

Notre critère pour décider qu'une fonction est petite dépend donc de son nombre d'arguments. L'algorithme 6.2 contient la définition complète du prédicat *small-size*?.

```
K: Une constante entière

O: Une valeur entière représentant le degré d'optimisation

small-size?( fun )=
   soit arity=fun↓arity,
   soit body=fun↓body,
      size-of( body ) <= K * (1 + arity + O)
```

Algorithme 6.2: Une fonction est-elle petite?

Nous souhaitions que des fonctions aussi simples que la négation (`not`) soient intégrées. Pour qu'au niveau 1 d'optimisation le compilateur y parvienne, nous avons fixé la constante $\mathcal{K}$ à 3 (1 pour le nœud `if` plus 1 pour chacune des deux branches).

6.3 Comment faire l'intégration?

La section précédente a montré *quand* un appel de fonction est expansé en ligne. Cette section va montrer *comment* il faut le faire.

Les publications sur l'intégration ne s'intéressent qu'au problème de la décision d'intégration. Les méthodes pour réaliser les intégrations ne sont jamais abordées. En particulier, le problème de l'intégration de fonctions récursives est généralement tout juste survolé. Le compilateur expanse éventuellement le premier appel (celui qui permet de rentrer dans la fonction). Les appels internes ne sont pas expansés. L'article de H. Baker [Bak92a] est l'exception car il présente une technique pour réaliser l'intégration de fonctions récursives. Sa solution consiste principalement à « dérouler » un certain nombre d'appel récursifs. La méthode que nous avons adoptée dans Bigloo est tout-à-fait différente.

Bigloo ne prend plus la décision d'expanser un appel à une fonction f si une précédente invocation de f est déjà en cours d'intégration. Mais Bigloo se distingue des autres compilateurs, car il distingue deux cas d'intégration : l'intégration de fonctions non-récursives et celle des fonctions récursives. La fonction *inline-call* de l'algorithme 6.1 distingue donc deux cas.

```
inline-call( exp, level, stack )=
   soit fun=exp↓fun,
      si recursive?( fun )
           alors do-inline-rec-call( exp, level, stack )
           sinon do-inline-call( exp, level, stack )
```

Le prédicat *recursive*? est vrai si, et seulement si, son argument est une fonction auto-récursive (nous ne nous intéressons ici qu'aux fonctions qui contiennent explicitement des appels auto-récursifs dans leurs corps).

6.3.1 L'intégration de fonctions non récursives

Si la fonction que le compilateur expanse en ligne n'est pas auto-récursive, l'intégration est donc une simple β-réduction. L'algorithme 6.3 contient la définition de la fonction *do-inline-call*. Le corps de la fonction expansée remplace l'appel et il est lui-même soumis au processus d'intégration. C'est lors de cet appel récursif que les variables *stack* et *level* sont modifiées. L'algorithme 6.3 présente l'effet de ce type d'intégration (en ne tenant pas compte des α-conversions).

```
do-inline-call( exp, level, stack )=
   soit fun=exp↓fun,
   soit actuals=exp↓actuals,
   soit formals=fun↓formals,
   soit newf={a|a=inline-exp( a, 1 + level, stack )},
   soit body=body-of( fun ),
      inline-exp( β-reduce( α-convert( body ), formals, newf ), 1 + level, {fun}⋃stack )
```

Algorithme 6.3: L'expansion en ligne par production de "let"

Voici donc la modification qu'effectue sur l'arbre de syntaxe abstraite l'intégration des fonctions non

récursives.

$$\textit{do-inline-call}\ \left[\!\!\left[\begin{array}{l}\texttt{(define (}\textit{fun}\ f_0\ \ldots\ f_n\texttt{)}\\ \quad \textit{Exp}\texttt{)}\\ \\ \ddots\\ \quad \boxed{(\textit{fun}\ a_0\ \ldots\ a_n)}\\ \quad\ \ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{(define (}\textit{fun}\ f_0\ \ldots\ f_n\texttt{)}\\ \quad \textit{Exp}\texttt{)}\\ \\ \ddots\\ \quad \boxed{\begin{array}{l}\texttt{(let ((}f_0\ a_0\texttt{)}\ \ldots\ \texttt{(}f_n\ a_n\texttt{)))}\\ \quad \textit{Exp}\texttt{)}\end{array}}\\ \quad\ \ddots\end{array}\right]\!\!\right]$$

Les variables temporaires introduites par la construction `let` seront susceptibles d'être supprimées par les optimisations ultérieures de Bigloo (comme la propagation des constantes). Mais puisqu'il s'agit d'autres optimisations, nous laissons ces variables apparaître ici.

6.3.2 L'intégration de fonctions simplement récursives

L'intégration d'une fonction récursive exige une attention particulière. La solution qu'adoptent tous les systèmes consiste à n'expanser qu'un nombre fixé d'appels (ceci correspond exactement à un dépliage lorsqu'on déroule des boucles (*loop unrolling*)). La solution que nous proposons est plus efficace, elle consiste à ne pas remplacer l'appel par le corps de la fonction mais à créer une fonction locale ayant pour corps celui de la fonction appelée. L'appel expansé est juste transformé en un appel à la fonction locale.

Nous n'allons pas donner le code de la fonction *do-inline-rec-call* dans notre langage algorithmique car cette fonction construisant totalement un nouvel arbre, le code ne serait pas très lisible. Nous allons cependant montrer les effets de cette fonction. Comme pour l'intégration des fonctions non récursives, nous n'indiquons pas ici les α-conversions en pratique indispensables[3].

$$\textit{do-inline-rec-call}\ \left[\!\!\left[\begin{array}{l}\texttt{(define (}\textit{fun}\ f_0\ \ldots\ f_n\texttt{)}\\ \quad \textit{Exp}\texttt{)}\\ \\ \ddots\\ \quad \boxed{(\textit{fun}\ a_0\ \ldots\ a_n)}\\ \quad\ \ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{(define (}\textit{fun}\ f_0\ \ldots\ f_n\texttt{)}\\ \quad \textit{Exp}\texttt{)}\\ \\ \ddots\\ \quad \boxed{\begin{array}{l}\texttt{(letrec ((}\textit{fun}\ \texttt{(lambda (}f_0\ \ldots\ f_n\texttt{)}\ \textit{Exp}\texttt{)))}\\ \quad (\textit{fun}\ \ a_0\ \ldots\ a_n))\end{array}}\\ \quad\ \ddots\end{array}\right]\!\!\right]$$

Le compilateur ne fait pas seulement cette très simple transformation, il fait en plus une analyse pour déterminer si certains paramètres de la fonction sont invariants. Le compilateur effectue une propagation de constante pour ceux qui le sont. L'intérêt de cette transformation apparaît sur cet exemple :

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

(define (succ x) (+ x 1))

(define (map-succ l)
   (map succ l))
```

[3] L'α-conversion doit être totale pour interdire les partages de branches d'arbres de syntaxe.

Étudions l'intégration de la fonction récursive `map` dans la fonction `map-succ`. La transformation *do-inline-rec-call* simple (sans l'optimisation de la détection des invariants) conduit au code :

```
(define (map-succ l)
   (letrec ((map (lambda (f l)
                   (if (null? l)
                       '()
                       (cons (f (car l)) (map f (cdr l))))))
            (map succ l)))
```

Le paramètre `f` étant invariant, Bigloo le supprime de `map` :

```
(define (map-succ l)
   (letrec ((map (lambda (l)
                   (if (null? l)
                       '()
                       (cons (succ (car l)) (map (cdr l))))))
            (map l)))
```

Enfin, le code obtenu est re-soumis au processus d'intégration donc l'appel à la fonction `succ` est expansé. Le code final obtenu est donc :

```
(define (map-succ l)
   (letrec ((map (lambda (l)
                   (if (null? l)
                       '()
                       (cons (+ 1 (car l)) (map (cdr l)))))))
            (map l)))
```

L'amélioration de l'intégration de la fonction récursive `map` est double : *(i)* l'appel à une fonction locale est mieux traité que l'appel à une fonction globale (Bigloo parvient souvent pour les fonctions locales à compiler les appels terminaux sans consommation de pile), *(ii)* la propagation de constantes permise par l'intégration peut conduire à du code beaucoup plus efficace. Dans l'exemple que nous avons donné, le code *avant* intégration était d'ordre supérieur (la fonction `succ` était utilisée comme objet de première classe) alors que le code *après* intégration n'est plus que du premier ordre.

6.4 L'ordre des expansions en ligne

L'ordre dans lequel les intégrations sont effectuées est important. Deux ordres différents peuvent conduire à deux résultats différents. Prenons la terminologie et les exemples de [HC89]. Désignons par $(A \rightarrow B)$ l'intégration de A dans B et $[XYZ]$ une suite ordonnée d'événements, commençant par X, effectuant Y et terminant par Z. La séquence $[(A \rightarrow B)(B \rightarrow C)]$ est différente de $[(B \rightarrow C)(A \rightarrow B)]$ car, $[(A \rightarrow B)(B \rightarrow C)] \equiv [(B' = A + B)(C' = B' + C)]$ alors que $[(B \rightarrow C)(A \rightarrow B)] \equiv [(C' = B + C)(B' = A + B)]$. Des séquences différentes peuvent également conduire à des résultats équivalents, par exemple $[(C \rightarrow A)(C \rightarrow B)]$ est équivalent à $[(C \rightarrow B)(C \rightarrow A)]$.

Malgré les différences de résultats, nous n'avons pas essayé de trouver une heuristique pour choisir un « bon ordre », nous expansons les fonctions dans l'ordre où se présentent les appels de fonction. Ceci n'a pas une importance capitale car lorsqu'une fonction est expansée, son corps est à nouveau soumis à expansion.

6.5 Les statistiques de l'intégration

Nous allons étudier sur plusieurs programmes de taille et de style divers le processus d'expansion en ligne, puis le nombre de fonctions expansées, le nombre d'appels expansés et l'incidence sur le code produit (la taille des fichiers objet et les temps d'exécutions). Les programmes sont tous présentés dans le chapitre 1. Nous faisons trois compilations: la première n'expanse que les fonctions introduites par la construction

`define-inline`; la seconde (**-O**) expanse aussi les petites fonctions et enfin en mode **-O2** le compilateur peut décider d'intégrer des fonctions plus grosses.

La colonne β indique le nombre d'appels de fonction qui ont été expansés. Sont indiquées ensuite la taille du fichier C produit, et la taille du fichier objet. Enfin, la dernière colonne donne le temps d'exécution du programme.

Voici les mesures sur machines Sparc :

Conform		β	taille .c	taille .o	exec
	–	291	136 k	22 k	42.1 s
	-O	427	128 k	20 k	34.3 s
	-O2	430	144 k	21 k	33.3 s

Queens		β	taille .c	taille .o	exec
	–	47	16 k	4 k	13.2 s
	-O	50	16 k	4 k	13.0 s
	-O2	52	16 k	4 k	13.0 s

Bague		β	taille .c	taille .o	exec
	–	33	13 k	4 k	30 s
	-O	42	12 k	4 k	24.7 s
	-O2	42	12 k	4 k	24.7 s

Peval		β	taille .c	taille .o	exec
	–	762	160 k	25 k	17.6 s
	-O	847	160 k	26 k	16.5 s
	-O2	853	168 k	26 k	16.2 s

Et sur machines Mips :

Conform		β	taille .c	taille .o	exec
	–	291	93 k	39 k	44.3 s
	-O	427	73 k	32 k	33.3 s
	-O2	430	85 k	32 k	33.0 s

Queens		β	taille .c	taille .o	exec
	–	47	14 k	8 k	9.5 s
	-O	50	13 k	8 k	8.8 s
	-O2	52	14 k	8 k	8.8 s

Bague		β	taille .c	taille .o	exec
	–	33	11 k	7 k	46.6 s
	-O	42	10 k	6 k	28.7 s
	-O2	42	10 k	6 k	28.7 s

Peval		β	taille .c	taille .o	exec
	–	762	95 k	40 k	14.9 s
	-O	847	92 k	39 k	14.0 s
	-O2	853	112 k	39 k	13.8 s

Ces mesures appelllent plusieurs remarques :

- Même sans optimisation, le nombre de fonctions expansées est important. Il faut se souvenir que les fonctions de la bibliothèque d'exécution comme `cons`, `car`, `null?`, ... utilisent le mécanisme d'intégration.
- Les fichiers C et les fichiers objet grossissent peu. C'est à peine perceptible quelque soit le mode de compilation et quelque soit le nombre d'appels expansés.
- L'amélioration des performances est sensible, sans être spectaculaire. Les temps avec optimisation (**-O2**) sont meilleurs d'environ 20 % sur Sparc et 27 % sur Mips.
- Il est surprenant de constater que l'intégration n'améliore presque pas les performances du programme `queens`. C'est d'autant plus curieux qu'il ressemble un peu au programme `bague`. Ce programme `queens` possède une petite fonction (`succ`) qui est invoquée un grand nombre de fois. Avec l'intégration les appels à cette fonction disparaissent mais pourtant l'exécution n'est pas plus rapide. Ce phénomène s'explique parfaitement lorsqu'on instrumente une exécution[4]. En fait, ce programme teste principalement l'allocation des listes. La fonction `succ` n'intervient que pour 0.2 % du temps d'exécution alors l'intégrer ou pas est sans importance. Ce sont les autres optimisations de Bigloo qui améliorent un peu les performances.

6.6 Conclusion

L'intégration des fonctions est une méthode qui permet de diminuer, ou même de supprimer dans certains cas, le coût de l'appel de fonction. En plus, grâce à elle, les autres optimisations du compilateur (élimination de sous-expressions communes, propagation de constantes) s'appliquent plus souvent. Cette optimisation trouve tout son sens dans le cadre d'un compilateur de langage applicatif car les appels de fonctions sont très nombreux. Ainsi, sur quelques programmes représentatifs, nous avons mesuré des gains de performances d'environ 20 %.

L'intégration fonctionnelle ne peut se faire que parcimonieusement car elle a une contrepartie : elle fait grossir la taille du code. Nous avons donné dans cette section un algorithme qui prend la décision d'intégration

[4] Pour instrumenter nos programmes, nous avons employé l'utilitaire `pixie` sur machine Mips. Son intérêt est que les programmes qu'il instrumente peuvent être compilés en mode optimisé par les compilateurs C. L'options `-p` n'est pas utilisée. L'instrumentation porte donc sur un programme très proche de celui réellement exécuté.

d'une fonction sur un site d'invocation. L'algorithme prend en compte plusieurs types d'informations comme la taille des fonctions appelées, la profondeur des appels déjà expansés, etc. Nos mesures montrent que l'heuristique que nous avons développée est satisfaisante, car bien que le nombre d'appels que le compilateur expanse soit important, la taille du code n'augmente que très peu.

Chapitre 7

L'analyse de flot de contrôle

Depuis plusieurs années, les analyses de flot de contrôle (l'élaboration du graphe d'appel en présence de fonctions de première classe) sont étudiées dans la littérature sur la compilation des langages fonctionnels. Plusieurs modèles théoriques ont été élaborés et plusieurs algorithmes ont été proposés [Shi88, Roz92, Aye92]. Puisque ces algorithmes sont complexes, l'effort a été concentré sur leur conception. Des questions cruciales, bien que pragmatiques, restent à ce jour sans réponse. À quoi ces algorithmes peuvent-ils réellement servir en pratique ? Qu'améliorent-ils vraiment ? Le travail présenté dans ce chapitre propose deux importantes optimisations qui prouvent l'intérêt des analyses de flot de contrôle :

La réduction des vérifications dynamiques de type : Partant d'une analyse de flot de contrôle célèbre, nous avons obtenu un algorithme d'inférence qui calcule des approximations de type en plus des approximations des valeurs fonctionnelles. Ce nouvel algorithme élimine plus de 65 % des tests de type dynamiques.

La réduction des allocations de fermetures : L'analyse de flot de contrôle calcule des approximations des opérateurs fonctionnels. Ces approximations constituent le fondement de notre schéma d'allocation de fermetures. Cette analyse de fermetures optimisante possède une base théorique sûre et englobe par ses résultats les analyses utilisées jusqu'à ce jour. Notre analyse de fermeture optimise en moyenne plus de 80 % des fermetures.

Si la première optimisation ne concerne que les compilateurs de langages typés dynamiquement comme Scheme ; la seconde, en revanche, est d'un intérêt plus général : elle concerne tous les compilateurs de λ-langages, incluant ceux typés statiquement (comme ML). Les optimisations que nous décrivons sont complètement implantées dans Bigloo. Nous pouvons donc présenter des mesures de performances précises sur des programmes réels. Nos chiffres montrent que nos optimisations diminuent de plus de 65 % les temps d'exécution.

Ce chapitre est organisé de la façon suivante : la première section présente notre analyse de flot de contrôle. La deuxième contient l'étude de l'inférence de types. La troisième section présente l'analyse de fermeture et la dernière partie présente les mesures de performances et démontre l'utilité de nos optimisations.

Ce travail a fait l'objet des publications [Ser94b, Ser95] et du rapport de recherche [Ser93].

7.1 L'analyse de flot de contrôle

Le flot de contrôle (ou encore *contrôle de flux*) peut, dans les langages fonctionnels modernes où les fonctions sont des valeurs de première classe, être de nature très dynamique. Néanmoins, des analyses peuvent en isoler des parties statiques. L'analyse que nous avons implantée dans Bigloo est proche de la « 0cfa » (0th-order Control Flow Analysis) décrite par O. Shivers dans sa thèse de doctorat [Shi91a]. Cette analyse calcule statiquement, pour chaque site d'application d'un programme l'ensemble des fonctions qui peuvent être invoquées lors de toutes les exécutions possibles. Ces approximations peuvent parfois être trop grossières (par exemple, elle peuvent contenir trop d'éléments pour être significatives) mais elles sont sûres.

Puisque le langage que nous compilons est Scheme dans sa totalité, nous avons dû adapter l'analyse de Shivers. De plus, nous calculons également des approximations pour d'autres structures de données que les fermetures. Shivers a donné un formalisme rigoureux pour décrire une classe d'analyse de flot de contrôle. Nous avons concentré notre travail sur l'utilisation, en situation réelle, de l'une d'entres elles.

7.1.1 L'algorithme

Nous présentons ici l'algorithme d'analyse de flot de contrôle (que nous noterons ACFA). Le langage accepté par notre algorithme est Sqil94 (voir la section 3.1). Nous nous démarquons donc nettement de la majorité des travaux antérieurs qui utilisent CPS. Dans l'énoncé de l'algorithme ACFA, nous utilisons deux ensembles. $FunId$ désigne l'ensemble des identificateurs de fonctions et $VarId$ l'ensemble des identificateurs de variables.

Remarque : La forme `call-with-current-continuation` *n'est pas présente dans notre langage intermédiaire car ce n'est pas une forme spéciale mais juste une fonction de la bibliothèque. Cette fonction ne nécessite pas de traitement particulier lors de l'analyse de flot de contrôle.*

Remarque : Une fonction peut seulement être introduite par des variables (constructions `labels`*), ainsi, dans la suite de ce texte, la* portée lexicale d'une fonction *devra être comprise comme étant la portée lexicale de la variable servant à introduire la fonction.*

Le principe de fonctionnement de ACFA

L'algorithme d'analyse de flot de contrôle peut-être vu comme un interprète qui n'aurait pas pour but d'évaluer un programme mais les valeurs possibles de certaines variables. ACFA est un algorithme d'interprétation abstraite [CC77] qui calcule des approximations des opérateurs fonctionnels. Il utilise un processus d'itérations. Chaque itération consiste en un parcours du graphe d'appel du programme. Chaque itération de rang n apporte des approximations des opérateurs fonctionnels qui sont utilisées par l'itération de rang $n+1$. L'algorithme se termine lorsqu'une itération n'apporte aucune nouvelle approximation. Autrement dit, ACFA réalise une itération de point fixe sur les approximations des opérateurs fonctionnels.

Étudions le comportement de l'algorithme ACFA sur le programme :

```
1:  (define (hux a) (funcall a a))
2:
3:  (define (foo b) (funcall b b))
4:
5:  (define (bar)
6:     (labels ((f1 (x) x)
7:              (f2 (y) y)
8:              (f3 (z) z))
9:        (foo (function f1))
10:       (hux (foo (function f2)))
11:       (foo (function f3))))
12:
13: (bar)
```

La première itération se base sur le graphe d'appel statique (celui où les fonctions invoquées sont des constantes). Ce graphe est donné dans la figure 7.1. Initialement seules les valeurs des fonctions constantes sont connues par l'algorithme. Toutes les autres variables ont uniquement $\perp$ comme valeur possible.

Le parcours de l'arbre de la figure 7.1 permet de calculer les premières approximations. Déroulons le comportement de l'algorithme : l'appel de la fonction `foo` (ligne *9*) ajoute l'identificateur `f1` à l'ensemble des approximations de `b`. L'appel de `b` (ligne *3*) ajoute à `x` l'approximation `f1`. L'appel à `foo` de la ligne *10* ajoute aux approximations de `b` la fonction `f2` mais le corps de la fonction `foo` n'est pas à nouveau examiné (une fonction ne peut être examinée qu'une seule fois au cours d'une itération pour empêcher l'algorithme de tomber dans des boucles infinies) ainsi, le résultat possible de `foo` reste l'ensemble { $\perp$, `f1` }. L'appel à `hux` de la ligne *10* ajoute donc `f1` aux approximations de `a`. Ainsi après la première itération, les approximations obtenues sont :

id.	a	b	x	y	z
approx.	{ $\perp$, f1 }	{ $\perp$, f1, f2, f3 }	{ $\perp$, f1 }	{ $\perp$ }	{ $\perp$ }

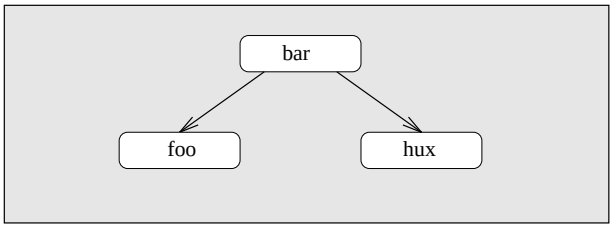


FIG. 7.1 - Le graphe d'appel statique

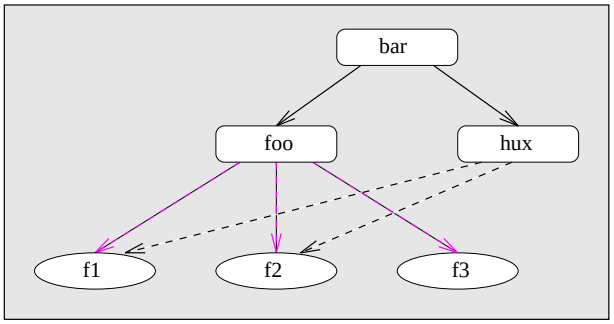


FIG. 7.2 - Le graphe d'appel après la première itération

Lors de la deuxième itération et de l'appel à `b` (ligne *3*) les fonctions `f1`, `f2` et `f3` peuvent être invoquées puisque l'ensemble des approximations de `b` contient ces trois fonctions. Ainsi, `f1`, `f2` et `f3` sont ajoutées aux approximations des paramètres formels de ces trois fonctions. L'appel à `hux` (ligne *10*) ajoute aux approximations de `a` les fonctions `f2` et `f3`. Ainsi, après cette itération, les approximations sont :

id.	a	b	x	y	z
approx.	{ ⊥, f1, f2, f3 }	{ ⊥, f1, f2, f3 }	{ ⊥, f1, f2, f3 }	{ ⊥, f1, f2, f3 }	{ ⊥, f1, f2, f3 }

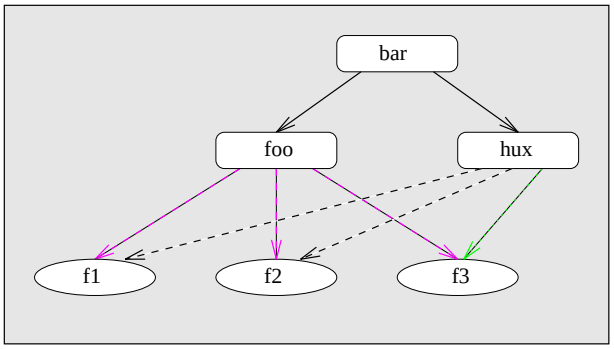


FIG. 7.3 - Le graphe d'appel après la deuxième itération

La troisième itération n'apporte plus aucune information, le point fixe est atteint, l'algorithme s'arrête. Tant que le point fixe n'est pas atteint, les résultats de ACFA ne sont pas exploitables car incomplets. Dans notre exemple une optimisation qui supposerait que le paramètre formel `a` ne peut être lié qu'à la fonction `f1` serait très probablement incorrecte !

L'algorithme ACFA

Pour calculer les approximations, ACFA utilise des informations sur les identificateurs apparaissant dans le programme qui sont décrits par quatre propriétés logiques :

- un prédicat de classe de variable $\mathcal{FUNCTION}$ qui indique si une variable est liée à une fonction.

- Trois prédicats de localité $\mathcal{FOREIGN}$ et $\mathcal{ESCAPE}$, pour les fonctions et $\mathcal{LOCAL}$, pour les autres variables.

Ainsi donc, pour toutes les variables d'un programme :

$\mathcal{LOCAL}(v)$	$\Leftrightarrow$	v est une variable locale.
$\mathcal{FOREIGN}(v)$	$\Leftrightarrow$	v est une fonction externe (définie dans un langage externe comme par exemple le langage d'implantation de la bibliothèque d'exécution).
$\mathcal{ESCAPE}(v)$	$\Leftrightarrow$	v est une fonction qui s'enfuit (fonction définie dans un autre module ou fonction exportée).

Et pour toutes les approximations calculées par ACFA :

$\mathcal{FUNCTION}(x)$	$\Leftrightarrow$	x est un identificateur de fonction.

Remarque : Seules les fonctions globales peuvent être importées ou exportées donc seulement les variables globales peuvent satisfaire le prédicat $\mathcal{ESCAPE}$.

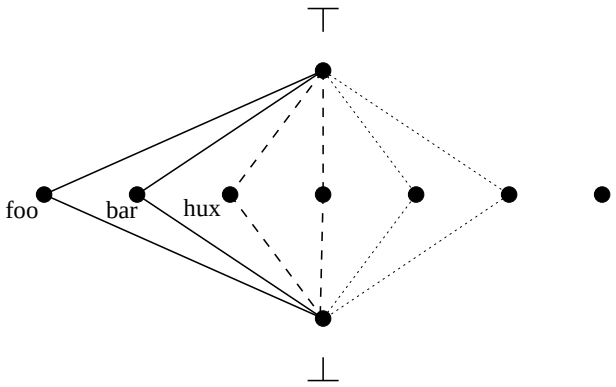


FIG. 7.4 - Le treillis des fermetures

L'algorithme ACFA utilise le treillis des fermetures qui est représenté dans la figure 7.4. C'est-à-dire que chaque approximation est une partie de l'ensemble :

$$R = \{\top, \bot\} \bigcup FunId$$

L'élément maximal incluant toutes les fonctions est noté $\top$. L'absence d'approximation est notée $\bot$. Toute approximation contenant l'élément $\top$ est indéfinie. Les annotations sont portées sur les fonctions et les variables du programme. Elle sont obtenues en utilisant la fonction $\mathcal{A}$. Initialement, toute les approximations ont $\{\bot\}$ comme valeur. La seule opération définie sur les approximations est la fonction d'ajout d'une nouvelle approximation. La définition de cette fonction nommée `add-approx!` est :

$$\forall x \in R, v \in VarId \cup FunId,$$
$$\texttt{si } y = \mathcal{A}(v) \texttt{ alors add-approx!}(v, x) \Rightarrow \mathcal{A}(v) = \texttt{si } x = \bot \texttt{ alors } y \texttt{ sinon } \{x\} \bigcup y$$

ACFA est exprimé dans le langage algorithmique déjà largement utilisé dans cette thèse (voir la figure 6.1). Nous avons découpé l'algorithme en deux parties. La première partie (figure 7.1) contient le cœur de l'algorithme. C'est à dire que la fonction *acfa* fait une discrimination sur l'arbre de syntaxe qui lui est passé en argument.

Voici quelques explications sur cette partie de l'algorithme.

- La fonction *set-top!* annule une approximation en propageant la valeur $\top$. Cette fonction est par exemple utilisée lorsque ACFA perd la trace d'une variable. Étudions le cas de la fonction identité `(define (id`

```
acfa( atree )=
   selon atree
        [[ A ]]:
             {⊤}
        [[ V ]]:
             si LOCAL( var )
                  alors A( var )
                  sinon {⊤}
        [[ (quote datum) ]]:
             {⊤}
        [[ (set! var val) ]]:
             si LOCAL( var )
                  alors ∀x∈acfa( val )
                            add-approx!( var, x )
                  sinon set-top!( acfa( val ) )
        [[ (labels ((fun_1 (arg_1 ...) body_1) ...) body) ]]:
             acfa( body )
        [[ (if si alors sinon) ]]:
             acfa( si ),
             acfa-exp( alors )⋃acfa-exp( sinon )
        [[ (begin atree_1 ... atree_n) ]]:
             acfa( atree_1 ),
             ...,
             acfa( atree_n )
        [[ (function f) ]]:
             {f}
        [[ (funcall f a_1 ... a_n) ]]:
             acfa-unknow-app( atree, acfa( f ), acfa( a_1 ), ..., acfa( a_n ) )
        [[ (failure) ]]:
             {⊥}
        [[ (f a_1 ... a_n) ]]:
             acfa-known-app( atree, f, acfa( a_1 ), ..., acfa( a_n ) )
   fin

set-top!( approx )=
   ∀a∈approx
       set-one-top!( a )

set-one-top!( a )=
   si FUNCTION( a ) et FOREIGN( a ) ou ESCAPE( a )
        alors ∀i∈[1,n]
                  set-top!( a↓_{formals_i} )
   sinon add-approx!( a, ⊤ )
```

Algorithme 7.1: L'algorithme d'analyse de flot de contrôle

`x) x)`. Si cette fonction est exportée, les valeurs du paramètre formel `x` peuvent être utilisées à l'extérieur du programme compilé. Toutes les fonctions qui appartiennent à $\mathcal{A}$(`x`) sont donc comme exportées et doivent donc être marquées de la valeur d'approximation $\top$. La fonction *set-top*! se charge alors de propager la valeur $\top$ depuis le paramètre formel jusqu'aux approximations.

- La forme `failure` retourne l'approximation $\{\bot\}$. Ceci est un point très important. Imaginons le programme suivant :

```
1:  (define (foo f)
2:     (if (procedure? f)
3:         f
4:         (failure)))
5:
6:  (define (bar x)
7:     (labels ((bar (x) x))
8:        (let ((f (foo bar)))
9:           (f x))))
```

  On peut penser que le test `procedure?` a été posé par la passe d'introduction des tests de type. Pour que les approximations de la variable `f` (ligne *8*) ne contiennent que la fonction `bar`, il faut que la forme `if` de la fonction `foo` ne retourne que les valeurs possibles de sa branche « alors ». Pour parvenir à ce résultat, il suffit que la forme `failure` ne retourne aucune approximation (et surtout pas $\top$).

- L'algorithme ACFA est présenté dans une forme simplifiée car ici les programmes sont supposés corrects et totalement α-convertis.

acfa-unknow-app(*atree*, *approx*, $approx_1$, ..., $approx_n$)=
  $\bigcup_{f \in approx}$ *acfa-try-app*(*atree*, *f*, $approx_1$, ..., $approx_n$)

acfa-try-app(*atree*, *f*, $approx_1$, ..., $approx_n$)=
  si $\mathcal{FUNCTION}$(*f*)
    alors *acfa-known-app*(*atree*, *f*, $approx_1$, ..., $approx_n$)
  ou bien si $f = \top$
    alors $\forall i \in [1,n]$
      set-top!($approx_i$),
      $\{\top\}$
  sinon *acfa-error*()

acfa-known-app(*atree*, *var*, $approx_1$, ..., $approx_n$)=
  si $\mathcal{ESCAPE}$(*var*)
    alors *set-top*!(*acfa*($var\downarrow_{body}$)),
      $\forall i \in [1,n]$
        set-top!($approx_i$),
      $\{\top\}$
  ou bien si $\mathcal{FOREIGN}$(*var*)
    alors *acfa-foreign-app*(*atree*, *var*, $approx_1$, ..., $approx_n$)
  sinon *acfa-function-body*(*var*, $approx_1$, ..., $approx_n$)

acfa-function-body(*var*, $approx_1$, ..., $approx_n$)=
  $\forall i \in [1,n]$
    $\forall x \in approx_i$
      `add-approx!`($var\downarrow_{formals_i}$, *x*),
  acfa($var\downarrow_{body}$)

Algorithme 7.2: Les appels de fonctions dans l'analyse de flot de contrôle

La deuxième partie de l'algorithme est donnée dans 7.2. Elle décrit le traitement effectué par ACFA sur les appels fonctionnels. Tel qu'il est présenté, ACFA peut tomber dans des boucles infinies ! Cela peut se

produire lors d'examen de fonctions auto-récursives (dans la fonction *acfa-known-app*). Ces boucles sont évitées de façon immédiate en interdisant plusieurs examens du corps d'une même fonction au cours d'une même itération. Ceci est réalisé au moyen d'estampilles.

7.1.2 Terminaison de ACFA

Puisque notre fonction d'approximation est monotone (on ne fait qu'ajouter des approximations) et que notre treillis ne contient qu'un nombre fini d'éléments, le théorème de point fixe de Tarski [Tar55] prouve la terminaison de notre analyse.

7.2 L'inférence de types

Scheme est typé dynamiquement. Pour éviter les comportements imprédictible les types doivent être vérifiés lors des exécutions. Par exemple, avant d'accéder au `car` d'un objet, il est indispensable de s'assurer d'abord que cet objet est une paire. Puisqu'ils sont nombreux, ces tests ont deux conséquences importantes : ils augmentent la taille des fichiers objets compilés et ils ralentissent les exécutions. Nous montrons dans cette section qu'en améliorant l'algorithme ACFA, nous parvenons à réduire le nombre de tests dynamiques tout en conservant la correction des exécutions.

Avant de décrire l'inférence de types, nous devons rappeler la méthode choisie pour introduire les tests dynamiques dans les programmes Scheme. Les seules opérations qui exigent des vérifications sont :

- L'accès aux structures de données.
- Les opérations arithmétiques.
- Les appels de fonctions calculés[1].

Ces trois classes d'opérations sont, pour des raisons d'efficacité ou de nécessité, implantées dans le langage de la bibliothèque d'exécution. Ce langage est comme nous l'avons montré dans le chapitre 4, connu du compilateur même s'il ne joue aucun rôle particulier. Pendant la compilation, il est considéré comme un langage externe avec lequel le fichier source est connecté. De ce point de vue, les seules opérations qui exigent des vérifications sont implantées dans le langage de la bibliothèque d'exécution. En conséquence, seulement les fonctions externes peuvent introduire des tests de type. Les fonctions externes peuvent retourner des objets Scheme comme des chaînes de caractères, des caractères, etc. Comme les fonctions d'allocations sont également écrites dans le langage de la bibliothèque d'exécution, on sait dès la compilation où les objets Scheme sont introduits. Ces informations sont propagées par la version améliorée de l'algorithme ACFA et il est donc souvent possible d'inférer le type d'une variable.

7.2.1 L'algorithme ACFA$^+$

Puisque ce sont seulement les appels externes qui peuvent introduire de nouveaux objets Scheme, le processus de propagation commence dans la fonction *acfa-foreign-app* jusqu'à présent volontairement laissée sans définition. Une version simplifiée de cette fonction est donnée dans l'algorithme 7.3.

```
acfa-foreign-app( atree, f, approx_1, ..., approx_n )=
   ∀i∈[1,n]
       set-top!( approx_i ),
   type-of-foreign-result( f )
```

Algorithme 7.3: acfa-foreign-app

Ainsi chaque approximation calculée par l'algorithme ACFA$^+$, appartient à l'ensemble des parties de l'ensemble :

[1] Les corrections des appels non calculés étant facilement testées lors de la compilation.

$$R' = R \bigcup \{\texttt{integer}, \texttt{character}, \texttt{pair}, \ldots\}$$

La fonction *acfa-foreign-app* présentée est simplifiée car elle propage l'approximation $\top$ sur toutes les approximations des paramètres effectifs. En pratique, le compilateur dispose d'informations qui lui permettent de ne pas propager cette valeur pour les appels aux fonctions externes de la bibliothèque (par exemple, `+`, `+fx`, `car`, `cons`, etc.). De plus, certaines fonctions externes ont un statut spécial qui oblige le compilateur à les traiter de façon particulière. C'est par exemple le cas des fonctions `push-trace` et `pop-trace`. Ces fonctions permettent de mémoriser une trace d'exécution utilisée pour la mise au point. Quand le compilateur est invoqué en mode de débogage, il insère un appel à `push-trace` en tête et un appel à `pop-trace` en fin de chaque fonction. Ainsi, une fonction tracée a l'aspect :

$$\texttt{(lambda (x) (push-trace } id\texttt{) (pop-trace } id{\downarrow_{body}}\texttt{))}$$

Nous ne sommes pas parvenus à écrire de façon satisfaisante les fonctions `push-trace` et `pop-trace` en Scheme. Nous avons dû les écrire dans le langage de la bibliothèque d'exécution. Ainsi, si l'algorithme ACFA ne réalise pas un traitement particulier lorsqu'il les rencontre, toute l'analyse de flot de contrôle va devenir inutile car toutes les valeurs auront $\top$ dans leurs approximations. Cet exemple met en évidence le besoin d'une discrimination *ad hoc* dans la fonction *acfa-foreign-app*. L'algorithme 7.4 présente une esquisse de cette fonction.

```
acfa-foreign-app( atree, f, approx_1, ..., approx_n )=
   si cint->bint?( f )
        alors {integer}
   ou bien si cstring->bstring?( f )
        alors {string}
   ou bien si cbool->bbool?( f )
        alors {boolean}
   ou bien si push-lambda-trace?( f )
        alors {⊥}
   ou bien si pop-lambda-trace?( f )
        alors approx_1
       ⋮
   sinon ∀i∈[1,n]
              set-top!( approx_i ),
         type-of-foreign-result( f )
```

Algorithme 7.4: acfa-foreign-app$^+$

7.2.2 L'inférence de types

Une fois que les approximations ont été collectées par l'algorithme ACFA$^+$, les applications concernant les vérificateurs de type (`integer?`, `procedure?`, ...) dont les résultats peuvent être déduits sont supprimées. Ces prédicats, comme les constructeurs sont écrits dans le langage de la bibliothèque d'exécution. En conséquence ils n'apparaissent que dans les appels à des fonctions externes.

Examinons sur un exemple le comportement de l'algorithme ACFA$^+$:

```
(define (foo f x)
   (if (integer? x)
       (integer? (f x))
       (if (real? x)
           (integer? (f (inexact->exact x))))))

(define (bar)
   (define (square y)
      ;; *fx désigne la multiplication entière
      (*fx y y))
   (foo square 4))

(bar)
```

Ce programme Scheme, après macro expansion et insertion des tests de type est :

```
 1:  (define (foo f x)
 2:     (if (integer? x)
 3:         (let ((obj (funcall (let ((f (if (procedure? f) f (failure))))
 4:                               (if (= (procedure-arity f) 1) f (failure)))
 5:                             x)))
 6:            (integer? obj))
 7:         (if (or (integer? x) (flonum? x))
 8:             (let ((obj (funcall (let ((f (if (procedure? f) f (failure))))
 9:                                   (if (= (procedure-arity f) 1) f (failure)))
10:                                 (if (flonum? x)
11:                                     (flonum->fixnum (if (flomun? x) x (failure)))
12:                                     x))))
13:                (integer? obj))
14:             #f)))
15:
16:  (define (bar)
17:     (labels ((square (y) (if (integer? y) (*fx y y) (failure))))
18:        (foo (function square) 4)))
19:
20:  (bar)
```

grâce à l'algorithme ACFA$^+$, il peut être réduit en :

```
(define (foo f x)
   (let ((obj (funcall f x)))
      #t))

(define (bar)
   (labels ((square (y) (*fx y y)))
      (foo (function square) 4)))

(bar)
```

Pour parvenir à ce résultat, ACFA$^+$ a eu besoin de deux itérations. Voici une pseudo-trace d'exécution de l'algorithme :

1. ⟦ `(bar)` ⟧ (ligne *20*) ⇒ inspection du corps de `bar`.
2. `add-approx!( square,`*square*`)` où $square \equiv \lambda y.(*fx\ y\ y)$
3. ⟦ `(foo (function square) 4)` ⟧ (ligne *18*) ⇒ `add-approx!( f,`*square*`)`, `add-approx!( x,`*integer*`)`
4. ⟦ `(let ((obj (funcall ...))) ...` ⟧ (ligne *3*) ⇒ `add-approx!( y,`*integer*`)`, inspection du corps de `square`.
5. ⟦ `(*fx y y)` ⟧
6. `add-approx!( obj,`*integer*`)`

Pour résumer voici les toutes les approximations calculées par ACFA$^+$ sur notre exemple :

id.	square	f	x	y	obj
approx.	{ ⊥, *square* }	{ ⊥, *square* }	{ ⊥, *integer* }	{ ⊥, *integer* }	{ ⊥, *integer* }

Étudions un autre exemple où l'algorithme d'inférence de types ACFA$^+$ permet d'améliorer les performances du code produit par Bigloo. Examinons la définition de la fonction de la bibliothèque d'exécution `read-char`. Si `read-char/port` est une fonction réalisant une lecture physique sur un flux d'entrée la définition de `read-char` peut être :

```
(define (read-char . port)
   (let ((port (if (null? port)
                   (current-input-port)
                   (car port))))
      (if (input-port port)
          (read-char/port port)
          (failure))))
```

Maintenant imaginons une petite fonction `user` utilisant `read-char` :

```
(define (user)
   (read-char))
```

Pour des raisons d'efficacité la fonction `read-char` est intégrée dans la fonction `user`. Ainsi, le code de `user` devient :

```
(define (user)
   (let ((port '()))
      (let ((port (if (null? port)
                      (current-input-port)
                      (car port))))
         (if (input-port port)
             (read-char/port port)
             (failure)))))
```

En utilisant les annotations de ACFA$^+$ pour éliminer les tests inutiles, le code de la fonction `user` sera réécrit en :

```
(define (user)
   (let ((port '()))
      (let ((port (current-input-port)))
         (read-char/port port))))
```

L'optimisation classique de propagation de constantes (*constant folding*) supprimera les liaisons inutiles. Ainsi le code final sera :

```
(define (user)
   (read-char/port (current-input-port)))
```

Les deux exemples que nous avons donnés illustrent l'utilité de notre inférence de types ACFA$^+$. Nous présenterons ultérieurement des mesures permettant de cerner le pourcentage de tests de type que notre optimisation élimine.

7.3 L'algorithme ACFA^{++}

Une des lacunes des algorithmes ACFA et ACFA$^+$ est la perte de trace de tous les objets placés dans des structures de données car ces algorithmes ne manipulent que des types (comme `pair`) et non pas des approximations des structures. Examinons le comportement de ACFA$^+$ sur le programme suivant :

```
(define (foo p)
   (funcall (car p) (cdr p)))

(labels ((inc (x) (if (integer? x)
                      (+fx x 1)
                      (error))))
   (foo (cons (function inc) 5)))
```

La seule approximation obtenue est que `p` est une paire mais l'algorithme a perdu la trace de la fonction `inc`, l'appel calculé de la fonction `foo` n'est donc pas optimisé. Il est facile de modifier l'algorithme ACFA$^+$ pour qu'il suive les objets stockés dans des structures de données.

7.3.1 Le nouvel algorithme

La seule modification que l'on doit apporter à ACFA se situe dans la fameuse fonction *acfa-foreign-app* car toutes les allocations et les accès aux structures sont implantés dans le langage de la bibliothèque d'exécution et n'apparaissent donc dans le programme que sous forme d'appels à des fonctions externes. Nous avions déjà dû modifier cette fonction (voir algorithme 7.3) pour réaliser l'inférence de type présentée dans la section 7.2. La modification que nous apportons à présent consiste à détecter les appels aux créations de structures de données pour que la fonction *acfa-foreign-app* ne se contente plus de retourner le type de l'objet mais une représentation de cet objet. L'algorithme 7.5 présente le nouveau corps de cette fonction pour que les objets soient suivis dans les paires.

acfa-foreign-app(*atree*, *f*, $approx_1$, ..., $approx_n$)=
  si `cint->bint?`(*f*)
    alors {`integer`}
  ou bien si `cstring->bstring?`(*f*)
    alors {`string`}
  ou bien si `cbool->bbool?`(*f*)
    alors {`boolean`}
  ou bien si `push-lambda-trace?`(*f*)
    alors $\{\bot\}$
  ou bien si `pop-lambda-trace?`(*f*)
    alors $approx_1$
  ou bien si `is-cons?`(*f*)
    alors {*atree*}
  ou bien si `is-car?`(*f*)
    alors *acfa-car*($approx_1$)
  ou bien si `is-cdr?`(*f*)
    alors *acfa-cdr*($approx_1$)
  ⋮
  sinon $\forall i \in [1,n]$
      set-top!($approx_i$),
    `type-of-foreign-result`(*f*)

acfa-car(*approx*)=
  si $\exists x \in approx$, $(x = \top) \vee (x$ n'est pas l'approximation d'une paire)
    alors $\{\top\}$
    sinon $\bigcup_{a \in approx} a\downarrow_1$

acfa-cdr(*approx*)=
  si $\exists x \in approx$, $(x = \top) \vee (x$ n'est pas l'approximation d'une paire)
    alors $\{\top\}$
    sinon $\bigcup_{a \in approx} a\downarrow_2$

Algorithme 7.5: acfa-foreign-app^{++}

Nous nommons ACFA^{++} la version de l'algorithme de flot de contrôle qui utilise la fonction *acfa-foreign-app* telle qu'elle est définie dans 7.5. La complexité dans le pire cas de ACFA^{++} est la même que celle de ACFA$^+$ mais sa complexité en moyenne est plus élevée. En effet, les approximations que calcule ACFA^{++} sont plus fines donc cet algorithme parvient moins vite à l'approximation $\{\top\}$. Il lui faut donc plus d'itérations pour atteindre son point fixe.

7.4 L'analyse de fermetures

Comme cela a déjà été mentionné, Scheme est un langage typé dynamiquement. Ainsi, les appels de fonctions calculés (les appels où les fonctions invoquées ne sont pas des constantes connues pendant la compilation) doivent être isolés. Il est obligatoire de tester que les objets appliqués sont réellement des fonctions et que l'arité est compatible avec le nombre d'arguments fournis. De plus, Scheme a deux classes de fonctions, les fonctions d'arité fixe et les fonctions d'arité variable ; comme nous l'avons montré dans la section 2.3 les fonctions possèdent deux points d'entrée. Le premier sert aux fonctions d'arité fixe et le deuxième aux fonctions d'arité variable. Cette représentation impose une taille minimum d'au moins quatre mots par fonction (une étiquette, une arité et deux points d'entrée). Nous allons montrer dans cette section que grâce à nos algorithmes de flot de contrôle (ACFA, ACFA$^+$ et ACFA^{++}) nous pouvons réduire la taille des fermetures.

Nos optimisations dépendent de l'usage fait des fermetures. Pour les caractériser, nous allons introduire le concept de *famille de procédures*. Intuitivement, l'ensemble $FunId$ des fonctions d'un programme est divisé en parties telles que tous les éléments d'une partie sont appliqués sur les mêmes sites d'invocations calculés.

7.4.1 Les familles de procédures

En premier lieu donnons quelques définitions. Nous notons $SITE$ l'ensemble des sites d'appels d'un programme. Pour chaque élément de $SITE$ l'algorithme d'analyse de flot de contrôle a calculé une approximation des fonctions qui peuvent être invoquées. C'est-à-dire que : $\forall s = [\![(\texttt{funcall} \ldots)]\!]$ appartenant à $SITE$, l'analyse de flot de contrôle à calculée $\mathcal{A}(f)$.

La fonction $\mathcal{USE}$ caractérise les utilisations des fonctions. $\mathcal{A}$ indique, pour chaque site d'appels, l'ensemble des fonctions qui peuvent être invoquées. Par opposition, $\mathcal{USE}$ indique, pour chaque définition, l'ensemble des sites où la fonction peut être invoquée. La définition de $\mathcal{USE}$ est la suivante :

$$\boxed{\begin{array}{c}\forall f \in FunId,\ \mathcal{USE}(f) \\ \Leftrightarrow \\ \{s \in SITE \,|\, s = [\![(\texttt{funcall}\ g \ldots)]\!] \wedge f \in \mathcal{A}(g)\}\end{array}}$$

Nous introduisons ici les trois propriétés $\mathcal{T}$, $\mathcal{X}$ et $\mathcal{S}$ qui sont utilisées dans le reste de ce chapitre (ainsi que dans le chapitre 8).

Le prédicat $\mathcal{T}$: Une fonction f satisfait le prédicat $\mathcal{T}$ si pour tous ses sites d'appels $[\![(\texttt{funcall}\ g \ldots)]\!]$, tous les éléments appartenant aux approximations de l'ensemble g sont des fonctions satisfaisant également le prédicat $\mathcal{T}$. Ce prédicat permet de grouper ensemble des fonctions suivant leur utilisation. Sa définition formelle est :

$$\boxed{\begin{array}{c}\forall f \in FunId,\ \mathcal{T}(f) \\ \Leftrightarrow \\ \overline{\mathcal{ESC}(f)} \wedge (\forall s = [\![(\texttt{funcall}\ g \ldots)]\!] \in \mathcal{USE}(f),\ \forall a \in \mathcal{A}(g),\ a \neq f,\ \mathcal{FUN}(a) \wedge \mathcal{T}(a))\end{array}}$$

La propriété $\mathcal{X}$: Une fonction f satisfait le prédicat $\mathcal{X}$ si pour tous les sites où elle peut être invoquée, f est la seule fonction pouvant être invoquée. La définition de $\mathcal{X}$ est :

$$\boxed{\begin{array}{c}\forall f \in FunId,\ \mathcal{X}(f) \\ \Leftrightarrow \\ \overline{\mathcal{ESC}(f)} \wedge (\forall s = [\![(\texttt{funcall}\ g \ldots)]\!] \in \mathcal{USE}(f),\ \mathcal{A}(g) = \{f\})\end{array}}$$

La propriété $\mathcal{S}$: Une fonction f satisfait le prédicat $\mathcal{S}$ s'il n'existe aucun site (`funcall`) où f peut être invoquée. La définition cette propriété est :

$$\boxed{\begin{array}{c}\forall f \in FunId,\ \mathcal{S}(f) \\ \Leftrightarrow \\ \overline{\mathcal{ESC}(f)} \wedge (\mathcal{USE}(f) = \emptyset)\end{array}}$$

Remarque : $\forall f \in FunId,\ \mathcal{S}(f) \Rightarrow \mathcal{X}(f),\ \mathcal{X}(f) \Rightarrow \mathcal{T}(f)$

Définition 1 *Soit s un site d'appel $[\![$(`funcall` $f \ldots$)$]\!]$. S'il existe une fonction g dans l'ensemble $\mathcal{A}(f)$ qui satisfait le prédicat $\mathcal{T}$ alors $\mathcal{A}(f)$ est appelée* une famille de procédure.

Proposition 1 *Pour toute fonction f de l'ensemble $FunId$, si f satisfait le prédicat $\mathcal{S}$, alors cette fonction ne requiert aucune allocation (pas de structure pour représenter la fonction et pas d'environnement alloué).*

Si une fonction f satisfait le prédicat $\mathcal{S}$, alors, toutes les invocations de f sont directes (par opposition à des invocations calculées utilisant la construction `funcall`) et ne peuvent survenir que dans la portée lexicale de la définition de f. Ainsi, chaque site d'invocation de f peut être compilé en un branchement direct (si f possède des variables libres, elles peuvent être ajoutées à la liste de ses paramètres formels). □

Proposition 2 *Les fonctions qui ne sont jamais passées en argument ou retournées comme résultat vérifient le prédicat $\mathcal{S}$.*

Cette proposition est évidente car ces fonctions sont toujours invoquées de façon directe ; elles n'apparaissent dans aucun `funcall`. □

Remarque : Cette proposition est très importante car la majorité des fonctions sont toujours invoquées de manière directe. Les résultats de notre analyse de flot de contrôle peuvent donc être très fréquemment utilisés.

Proposition 3 *Pour toute fonction f dans $FunId$, si f satisfait le prédicat $\mathcal{X}$ alors f ne nécessite pas de structure pour représenter la fermeture*[2]*.*

En effet, si une fonction f satisfait le prédicat $\mathcal{X}$ alors :

$$\forall s = [\![(\texttt{funcall}\ g \ldots)]\!] \in \mathcal{USE}(f), \mathcal{A}(g) = \{f\}$$

Cela signifie que seule f peut être invoquée. Les appels calculés peuvent être remplacés (en faisant, si besoin, une passe de *lambda lifting* [Joh85]) par des appels directs qui ne nécessitent pas d'allocation. □

Proposition 4 *Pour toute fonction f dans $FunId$, si f satisfait le prédicat $\mathcal{T}$ alors f peut être allouée de façon « légère ». C'est-à-dire que cette fonction n'a ni besoin d'étiquette, ni besoin de champ contenant l'arité et un seul point d'entrée est utile.*

Cela est vrai car si une fonction f satisfait le prédicat $\mathcal{T}$, alors :

$$\forall s = [\![(\texttt{funcall}\ g \ldots)]\!] \in \mathcal{USE}(f), \forall a \in \mathcal{A}(g), a \in \text{FunId}$$

Cette famille est connue pendant la compilation et donc, la correction des types peut être vérifiée statiquement. □

7.4.2 Les améliorations apportées à la compilation

Nous montrons dans cette section comment Bigloo utilise les prédicats $\mathcal{T}$, $\mathcal{X}$ and $\mathcal{S}$ pour améliorer la compilation des fonctions et des appels fonctionnels. Quatre constructions sont améliorées : `function`, `funcall`, `labels` et les définitions globales. Voici pour chacune d'elles les améliorations réalisées :

- (`labels` (($f \ldots$)$\ldots$)$\ldots$) ou (`define` ($f \ldots$)$\ldots$) :

  Si f satisfait le prédicat $\mathcal{S}$ alors aucune fermeture n'est construite pour f. Les appels à cette fonction seront tous compilés en des branchements directs.

- (`function` f) :

  - Cas simple : si f satisfait le prédicat $\mathcal{X}$ et si tous les éléments de $\mathcal{USE}(f)$ sont dans la portée lexicale de f, la forme est supprimée.

[2] S'il n'est pas utile d'allouer une structure pour représenter la fermeture, en revanche, il peut être indispensable d'allouer un environnement.

- Si f satisfait le prédicat $\mathcal{X}$ et si f a zéro ou une variable libre, alors, aucune allocation n'est requise. La forme est remplacée par la variable libre. Si f a plusieurs variables libres, alors la forme est remplacée par la liste de ces variables.
- Si f satisfait le prédicat $\mathcal{T}$ alors une structure « légère » est allouée pour représenter la fermeture.

- (`funcall` f ...) :
  - Soit ⟦ (`funcall` f ...) ⟧, un site d'appel. S'il existe une unique fonction g dans $\mathcal{A}(f)$ satisfaisant $\mathcal{X}$, alors, un appel direct est compilé.
  - Soit ⟦ (`funcall` f ...) ⟧, un site d'appel. Si tous les éléments de $\mathcal{A}(f)$ vérifient la propriété $\mathcal{T}$, une application « légère » (sans test de type ni test d'arité) est compilée.

- (`funcall` f ...) ou (f ...) : Si le résultat d'une application est connue et s'il a été prouvé que la fonction invoquée ne réalisait aucun effet de bord (information fournie par la passe **Effect**, voir la section 9.3), l'appel de fonction est remplacé par le résultat de cet appel.

7.4.3 Applications & exemples

Nous étudions plusieurs exemples caractéristiques des améliorations précédemment décrites. Le but de cette section est de mieux transmettre l'intuition des optimisations réalisées.

$\mathcal{S}$: les fonctions de premier ordre

Le premier exemple ne contient aucune fonction utilisée comme valeur ou retournée comme résultat :

```
1:  (define (fibo x)
2:     (labels ((fib (x) (if (< x 2)
3:                           1
4:                           (+ (fib (- x 1)) (fib (- x 2)))))))
5:        (if (number? x)
6:            (fib x)
7:            (error "fib" x))))
8:
9:  (fibo 20)
```

La fonction `fib` satisfait donc le prédicat $\mathcal{S}$. L'exécution ne réalise donc aucune allocation, `fib` est donc juste représenté par une fonction C. La compilation, en C, de `fibo` est donc :

```
1:  obj_t static obj_t fibo( obj_t x )
2:  {
3:     return fib( x );
4:  }
5:
6:  static obj_t fib( obj_t x )
7:  {
8:     if( LT_I( x, BINT( 2 ) ) )
9:        return BINT( 1 );
10:    else
11:    {
12:       obj_t aux_1, aux_2;
13:
14:       aux_1 = fib( SUB_I_PTAG( x, PSUB_TAG( 2 ) ) );
15:       aux_2 = fib( SUB_I_PTAG( x, PSUB_TAG( 1 ) ) );
16:       return ADD_I( aux_1, aux_2 );
17:    }
18: }
```

Aucune allocation n'est effectuée. Les appels à `fib` aux lignes *4* et *6* du code source sont compilés en des appels directs C (lignes *3*, *14* et *15*). Par ailleurs, le test `number?` a été supprimé car la seule valeur fournie à `fibo` est `20`.

$\mathcal{X}$: l'opérateur de point fixe

L'exemple que nous utilisons ici est extrait de l'article [Roz92] de G.J. Rozas. Il a été réécrit en remplaçant les formes `lambda` qui n'existent pas en Sqil94 par des formes `labels`. Ainsi, toutes les fonctions de notre programme sont nommées.

```
1:  (let ((fac (labels ((y (f)
2:                         (labels ((g (x)
3:                                    (funcall
4:                                     f
5:                                     (labels ((lam-2 () (funcall x x)))
6:                                        (function lam-2))))))))
7:               (y (labels ((lam-0 (fg)
8:                                   (labels ((lam-1 (n)
9:                                                   (if (= n 0)
10:                                                      1
11:                                                      (* (funcall (funcall fg) (- n 1))
12:                                                         n))))
13:                                    (function lam-1))))
14:                     (function lam-0))))))
15:     (funcall fac 10))
```

Dans ce programme, toutes les fonctions vérifient le prédicat $\mathcal{X}$. C'est-à-dire que tous les appels sont directs. Cela est vrai en particulier pour la fonction `lam-1` dont le code après analyse est :

```
(define (lam-1 lam-1-env n)
   (if (= n 0) 1 (* (lam-1 lam-1-env (- n 1)) n)))
```

Ainsi, l'expression `(funcall (funcall fg) (- n 1))` (lignes *12* et *13*) du programme source a été remplacée par un appel récursif direct.

$\mathcal{X}$: Currification

Nous étudions dans cette section un exemple simple de programme écrit dans un style ML. Dans ce langage, toutes les fonctions sont unaires. Une façon naturelle de représenter les fonctions d'arité supérieure est d'utiliser la currification. Notre exemple illustre l'optimisation réalisée par Bigloo quand une fonction vérifie le prédicat $\mathcal{X}$:

```
(define (plus x)
   (labels ((add-c (y) (+ x y)))
      (function add-c)))

(define (foo x y) (funcall (plus x) y))

(foo 1 2)
```

Dans ce programme, la fonction `add-c` vérifie le prédicat $\mathcal{X}$ puisqu'elle est seulement invoquée dans l'expression `(funcall (plus x) y)`. L'allocation de la fonction est donc remplacée par la variable libre de `add-c`, c'est-à-dire `x`. Ainsi, après optimisation le programme est transformé en :

```
(define (plus x) x)

(define (add-c env y) (+ env y))

(define (foo x y) (add-c (plus x) y))

(foo 1 2)
```

Aucune allocation n'est effectuée.

$\mathcal{X}$: l'analyse ACFA^{++}

Reprenons l'exemple de la section 7.3 :

```
1:  (define (foo p)
2:     (funcall (car p) (cdr p)))
3:
4:  (labels ((inc (x) (if (integer? x)
5:                        (+fx x 1)
6:                        (error))))
7:     (foo (cons (function inc) 5)))
```

L'algorithme ACFA^{++} suit les fonctions, même lorsqu'elles sont placées dans des structures de données. Dans l'exemple, l'approximation calculée pour `p` sera un ensemble contenant une paire dont la première projection est la fonction `inc` et la deuxième projection est le type `integer`. Ainsi la fonction `inc` satisfait le prédicat $\mathcal{X}$. Le code est transformé en :

```
(foo (cons 'dummy 5))

(define (inc env x)
   (+fx x 1))

(define (foo p)
   (inc (car p) (cdr p)))
```

L'appel calculé de la ligne *2* a été remplacé au profit d'un appel direct. Le test de type portant sur le prédicat `integer?` a été supprimé.

$\mathcal{T}$: une sémantique dénotationnelle

Nous allons illustrer l'emploi du prédicat $\mathcal{T}$ en étudiant la compilation d'une sémantique dénotationnelle. Nous donnons un fragment d'une sémantique de Scheme écrite en Scheme dans la figure 7.1.

```
1:  ;; eval: form * env * cont * store -> val
2:  (define (eval e env cont store) ((meaning e) env cont store))
3:
4:  ;; meaning: form -> env * cont * store -> val
5:  (define (meaning e)
6:    (if (atom? e)
7:        (if (symbol? e) (meaning-reference e) (meaning-quotation e))
8:        (case (car e)
9:          ((quote) (meaning-quotation (cadr e)))
10:         ((if) (meaning-alternative (cadr e) (caddr e) (cadddr e)))
11:         ((lambda) (meaning-abstraction (cadr e) (cddr e)))
12:         ((begin) (meaning-sequence (cdr e)))
13:         ...
14:
15: (define (meaning-alternative e1 e2 e3)
16:   (lambda (r k s)
17:      ((meaning e1) r (lambda (v s)
18:                        (if v ((meaning e2) r k s) ((meaning e3) r k s))) s)))
19:
20: (define (meaning-quotation v)
21:   (lambda (r k s) (k v s)))
22: ...
```

Programme 7.1: sémantique dénotationnelle

Chacune des fonctions `meaning-`... (`meaning-reference`, `meaning-quotation`, `meaning-alternative`, etc.) retourne une fonction à trois arguments (`env`, `cont` et `store`). Désignons par M l'ensemble contenant ces fermetures. Chacune d'entre elles peut être invoquée sur tous les sites « `((meaning ...) ...)` » (lignes *02* et *19*). Nous avons donc :

$$\forall f \in M, \mathcal{T}(f)$$

compil.	Le temps exprimé en seconde pour produire le code C.
compil.+cc	Le temps global exprimé en seconde pour produire un exécutable (temps d'édition de liens inclus).
size	Nombre de lignes du programme Sqil94 produit par le compilateur.
.o size	Taille du fichier objet obtenu après la compilation C (taille exprimée en KiloOctet).
failure	Nombre de fois que la forme `failure` est présente dans le fichier produit pour signaler une erreur potentielle dûe à un problème de type ou d'arité.
type error	Nombre de fois qu'une forme `failure` peut survenir à cause d'une erreur de type.
proc	Nombre de fermetures allouées. Dans la colonne **O2** ce chiffre correspond au nombre de fermeture qui n'ont pu bénéficier d'aucune optimisation.
$\mathcal{T}$	Nombre de fermetures qui sont simplifiées car elles vérifient la propriété $\mathcal{T}$.
$\mathcal{X}$	Nombre de fermetures supprimées car les fonctions vérifient la propriété $\mathcal{X}$.
$\mathcal{S}$	Nombre de fermetures supprimées car les fonctions vérifient la propriété $\mathcal{S}$.
run	Temps d'exécution exprimé en seconde.
run unsafe	Temps d'exécution exprimé en seconde pour le même programme mais compilé dans un mode où aucun test dynamique (de type ou d'arité) n'a été émis (option `-unsafe` de Bigloo).

FIG. 7.5 - Les mesures de l'analyse de fermeture

Ainsi toutes ces fonctions forment une famille. Il est possible d'obtenir un interprète de code-octet en réalisant l'optimisation suivante : puisque l'algorithme ACFA a prouvé que toutes les fonctions de M vérifient le prédicat $\mathcal{T}$, il est possible, pendant la compilation, de changer la façon de représenter ces fonctions. Par exemple, on pourrait choisir d'attribuer à chacune d'elles un numéro unique et de les représenter par des couples $< num \times env >$. Tous les appels « `((meaning ...) ...)` » seraient alors transformés en sauts indexés sur la projection de ces couples. Les numéros attribués aux fonctions peuvent être considérés comme des codes-octet et les sites d'appels « `((meaning ...) ...)` » peuvent être vus comme des interprètes de codes-octet. Dans Bigloo, nous avons adopté une solution légèrement plus performante. Plutôt que d'allouer des couples $< num \times env >$, nous allouons pour chaque fonction de M un couple de la forme $< entry\ point \times env >$. Les sauts indexés peuvent être évités et le code produit reste donc plus petit. La section 7.5 contient des chiffres qui mesurent l'efficacité de cette optimisation.

7.5 Les mesures de performances

Cette section présente une étude comparative entre les tailles des codes produits, les temps de compilation et d'exécution de Bigloo lorsque l'analyse de flot de contrôle est activée et lorsqu'elle ne l'est pas. Nous avons étudié plusieurs programmes écrits dans des styles différents. L'échantillon qu'ils constituent est représentatif des styles de programmation habituellement rencontrés en Scheme.

Pour chaque test, trois compilations ont été réalisées : *(i)* la première sans optimisation *(ii)* la deuxième avec des optimisations classiques (comme par exemple la CSE) **(O)** *(iii)* la dernière contient les optimisations classiques (comme dans le mode **(O)**) et en plus, elle contient les optimisations que nous avons décrites dans ce chapitre **(O2)**. Toutes les compilations ont été interrompues avant la production de code C. Nous avons alors mesuré la taille des fichiers compilés (encore exprimés en Sqil94). Pour tous les modes de compilation, nous avons calculé le nombre de tests de type et le nombre d'allocations de fermetures. Nous avons également mesuré les temps de compilation globaux, c'est-à-dire les temps de compilation incluant la compilation du compilateur C (gcc[Sta89] toujours utilisé avec l'option `-O`). Pour chaque mesure, δ est le ratio calculé par la formule $1 - O2/O$. Les relevés de temps d'exécutions sont conformes à la description des mesures de performance du chapitre 1 et ont été effectués sur une machine équipée de processeurs Sparc.

La figure 7.5 contient un résumé des significations des chiffres que nous présentons. Les six programmes que nous utilisons pour mesurer l'efficacité des optimisations de ce chapitre sont : **Conform**, **Pp**, **Earley**, **Dens**, **Peval** et **Queens**. Tous ont déjà été décrits dans le chapitre 1.

Conform		**O**	**O2**	δ
compil.	6.4	6.4	60.5	-845 %
compil.+cc	72.6	66.1	100.5	-52 %
size (lines)	9143	8190	3816	53 %
.o size (Kbytes)	144	136	52	61 %
failure	718	626	203	68 %
type error	459	355	132	63 %
proc	94	94	5	95 %
$\mathcal{T}$	_	_	0	_
$\mathcal{X}$	_	_	16	_
$\mathcal{S}$	_	_	73	_
run	73.5	61.9	24.7	60 %
run unsafe	52.2	43.4	15.8	63 %

Conform utilise beaucoup de fonctions mais peu nombreuses sont celles qui doivent être allouées. L'optimisation des fermetures a un impact particulièrement visible sur ce programme. Cela explique pourquoi le gain obtenu en utilisant ACFA est plus important lorsque la compilation n'introduit pas de tests dynamiques (95 % en mode *unsafe* contre 60 % en mode *safe*).

Pp		**O**	**O2**	δ
compil.	6.1	5.2	35.7	-586 %
compil.+cc	56.5	48.6	59.8	-23 %
size (lines)	6555	5771	3018	48 %
.o size (Kbytes)	87	78	30	62 %
failure	471	404	97	76 %
type error	318	251	81	68 %
proc	52	52	14	73 %
$\mathcal{T}$	_	_	5	_
$\mathcal{X}$	_	_	3	_
$\mathcal{S}$	_	_	30	_
run	12.6	11.4	8.4	26 %
run unsafe	12.6	9.9	6.3	36 %

Pp est, des programmes testés, le moins fonctionnel. Les résultats plus faibles ne sont donc pas surprenants. Le nombre de fermetures optimisées est le plus faible (73 %). Cela explique pourquoi l'accélération est aussi la plus faible de tous les bancs d'essai (26 et 36 %).

Earley		**O**	**O2**	δ
compil.	7.4	6.8	73.9	-986 %
compil.+cc	73.3	56.5	100.5	-77 %
size (lines)	10679	7965	3637	54 %
.o size (Kbytes)	120	90	33	63 %
failure	859	547	214	61 %
type error	626	314	118	62 %
proc	75	75	4	95 %
$\mathcal{T}$	_	_	0	_
$\mathcal{X}$	_	_	0	_
$\mathcal{S}$	_	_	71	_
run	44.4	41.6	10.2	75 %
run unsafe	33.7	33.2	7.0	79 %

Pour ce programme (**Earley**), l'analyse de flot de contrôle semble nécessiter beaucoup de temps. Le temps de compilation est augmenté de 986 % sans la compilation C et de 77 % avec la compilation C. Cela est dû aux tests de type dynamiques. Voici présenté dans ce tableau les temps de compilation en mode *unsafe* :

Earley	**O**	**O2**	δ
compil.	5.7 s	11.9 s	-108 %
compil.+cc	21.5 s	21.1 s	2 %

Earley utilise des vecteurs et chaque accès exige un test de borne. Ce sont ces très nombreux tests qui rendent l'arbre de syntaxe de grosse taille en augmentant le nombre de site d'appels. Cela ralentit l'algorithme ACFA.

Dens		**O**	**O2**	δ
compil.	4.2	4	13.1	-227 %
compil.+cc	29.8	29.4	23.4	20 %
size (lines)	3444	2982	1597	46 %
.o size (Kbytes)	49	45	18	60 %
failure	248	210	25	88 %
type error	156	118	25	79 %
proc	54	54	8	85 %
$\mathcal{T}$	_	_	36	_
$\mathcal{X}$	_	_	4	_
$\mathcal{S}$	_	_	8	_
run	38	39	15.2	61 %
run unsafe	32.5	32.6	15.4	53 %

Dens est une sémantique dénotationnelle, l'optimisation décrite dans la section 7.4.3 s'applique. Cela explique le grand nombre de fonctions satisfaisant le prédicat $\mathcal{T}$. C'est pourquoi les performances sont grandement améliorées lorsque Bigloo utilise l'analyse de flot de contrôle.

Peval		**O**	**O2**	δ
compil.	7.6	6.4	35	-446 %
compil.+cc	86.9	62.4	72.7	-16 %
size (lines)	9458	7550	4789	37 %
.o size (Kbytes)	144	120	54	55 %
failure	724	535	218	59 %
type error	566	377	213	43 %
proc	63	63	15	76 %
$\mathcal{T}$	_	_	8	_
$\mathcal{X}$	_	_	0	_
$\mathcal{S}$	_	_	40	_
run	17.7	17.4	10.7	38 %
run unsafe	13.4	13.4	8.1	39 %

L'inférence de type semble échouer sur **Peval**. Cela est en partie dû à ses nombreux effets de bord. Le seul gain de l'analyse de flot de contrôle est obtenu par l'optimisation des fermetures. Cela explique pourquoi

l'amélioration des temps d'exécutions est presque la même que le mode de compilation soit sûre ou non.

Queens		**O**	**O2**	δ
compil.	2.1	2.1	2.9	-38 %
compil.+cc	10.7	8.7	6.6	24 %
size (lines)	737	586	189	69 %
.o size (Kbytes)	13	12	4	67 %
failure	66	50	7	86 %
type error	47	31	7	77 %
proc	12	12	0	100 %
$\mathcal{T}$	_	_	0	_
$\mathcal{X}$	_	_	3	_
$\mathcal{S}$	_	_	9	_
run	31.3	29.5	12.0	59 %
run unsafe	24.8	24.5	10.7	56 %

Queens est un programme originellement écrit en ML. Les currifications automatiques qui introduisent des fermetures sont éliminées par l'analyse de flot de contrôle.

Nous devons faire plusieurs commentaires sur ces mesures :

- Afin de situer les performances de Bigloo par rapport aux autres compilateurs Scheme, nous avons fait les mêmes mesures pour le compilateur Scheme-to-C de J.F. Bartlett. Nous avons utilisé la version du 15 Mars 1993 avec l'option de compilation : « `-On -Ob -Og -Ot -cc gcc -O` ». Cette option signifie que les compilations sont faites en mode *unsafe* et que l'arithmétique est entière.

scc	**Queens**	**Dens**	**Pp**	**Conform**	**Peval**	**Earley**
compil.+cc	13.8 s	29.9 s	49 s	55 s	80.8 s	111.5 s
run unsafe	22.3 s	30.6 s	18.3 s	35.4 s	13.8 s	9.7 s

- L'algorithme ACFA semble s'exécuter lentement et la complexité du pire cas est d'ailleurs $\mathcal{O}(n^3)$ où n désigne le nombre de fonctions et de site d'appels. Bien que les premières passes de la compilation soient ralenties, le temps de compilation reste raisonnable. Paradoxalement, sur certains programmes (comme (**queens** et **semantics**), l'analyse de flot de contrôle permet même d'avoir une compilation globale plus courte. Cela s'explique par le fait qu'après l'algorithme ACFA le code C produit est de meilleure qualité. Comme le compilateur C est très lent à compiler, le temps perdu dans les passes de Bigloo est rattrapé par un temps de compilation C plus court. Ce paradoxe ne s'observe que sur les programmes où ACFA a permis d'améliorer fortement le code C produit. Sur les autres programmes, le temps passé à réaliser l'analyse de flot de contrôle n'est jamais compensé. Pour des programmes réels, l'algorithme ACFA est parfaitement utilisable. Par exemple, l'auto-amorçage de Bigloo (plus de 30,000 lignes de Scheme) prend 45 minutes sans ACFA et 55 avec.

  De plus, comme c'est indiqué dans nos mesures, le nombre d'itérations pour atteindre le point fixe d'ACFA est toujours très petit. Pour nos exemples, il a toujours été atteint en cinq itérations au plus. Nous utilisons ACFA pour des programmes « réels » et nous n'en avons jamais rencontré qui nécessitent des temps de compilation polynomiaux.

- Les nombres de vérifications de type et de créations de fermetures que nous donnons sont statiques. Bien que cela ne soit pas les mesures idéales, ces mesures sont tout de même révélatrices.

- Les temps d'exécution en mode *unsafe* permettent de distinguer quelle est la part des optimisations des fermetures dans l'amélioration globale obtenue en utilisant les approximations de ACFA. En effet, dans ce mode Bigloo n'émet aucun test de type ou de bornes dynamique qui ralentissent les exécutions. Les mesures montrent que les programmes sont environ 60 % plus rapides avec l'optimisation des fermetures.

- Puisque le mode *unsafe* ne contient aucun test de type, les gains obtenus correspondent aux gains qu'obtiendrait un compilateur ML en utilisant nos optimisations.

En résumé, les mesures de performances font apparaître que l'analyse de flot de contrôle permet de réduire les temps d'exécutions de 60 % pour une compilation dont le temps est augmenté en moyenne de seulement 20 %.

7.6 Comparaison avec d'autres travaux

Trois types de travaux peuvent être comparés au nôtre. Les premiers concernent les analyses de flot de contrôle.

- O. Shivers a publié de nombreux articles sur l'analyse de flot de contrôle dans les langages fonctionnels modernes [Shi88, Shi91b]. Son but a été d'étudier les analyses et leurs applications. La première partie de sa thèse de doctorat est consacrée à l'étude sémantique de l'analyse et son aspect théorique alors que la seconde partie est consacrée à des exemples d'applications. Il a défini une analyse générale dont la *0cfa* n'est qu'un cas particulier. Il a également décrit des analyses plus fines comme la « *1cfa* », qu'il a prototypé dans Orbit [KKR+86], un compilateur Scheme. Shivers a brièvement donné des temps d'exécution de la *1cfa* mais il n'a pas réalisé de mesures précises sur les coûts et les bénéfices qu'on peut attendre de cette analyse en situations réelles. Il n'est donc pas possible de tirer des conclusions sur la pertinence de celle-ci en pratique.

  Shivers a montré plusieurs applications de ses analyses de flot de contrôle. Notre inférence de type est proche de son « *type recovery* », mais les autres optimisations qu'il mentionne (optimisations classiques d'analyse de flot de données) sont déjà réalisées par notre compilateur *sans* l'analyse de flot de contrôle. La principale raison est qu'Orbit utilise CPS comme langage intermédiaire, le controle est donc très dynamique. Dans ce cas, l'analyse de flot de contrôle est très importante pour isoler les parties statiques. Nous n'avons pas ces problèmes car nous utilisons Sqil94 comme langage intermédiaire. En conséquence, nous ne nous servons de l'analyse de flot de contrôle que pour réaliser des optimisations telles que l'allocation optimisée des fermetures.

  L'analyse dite de *1cfa* calcule des approximations plus fines : plutôt que de ne seulement savoir quelles sont les fonctions invoquées sur chaque site d'appel, la *1cfa* permet de savoir comment ces fonctions parviennent aux sites d'appel. Nous ne pensons pas que cette information supplémentaire permette d'améliorer la compilation. Qu'apporte-t-elle de plus à un compilateur? Nous n'avons pas de réponses à ces questions. C'est pourquoi nous avons choisi d'utiliser la *0cfa* plutôt que la *1cfa* dans Bigloo.

- G.J. Rozas présente dans l'article [Roz92], comment, en utilisant une technique proche de la *0cfa*, le compilateur Liar [Roz84] parvient à compiler indifféremment des programmes utilisant la construction `letrec` ou l'opérateur de point fixe Y. Son article met l'accent sur le fonctionnement de son analyse plus que sur l'exploitation de ses résultats. Rozas soulève des problèmes que nous n'avons pas mentionnés ici, comme l'élimination d'environnements inutiles. Malheureusement, bien que son analyse soit implantée dans son compilateur, aucune mesure de performances ne permet d'avoir une idée de l'impact de ses optimisations.

Plusieurs analyses de fermetures doivent être comparées à celle que nous avons présentée dans ce chapitre. Les thèses de D. Kranz [Kra88] et de N. Séniak [Sén91] sont, en grande partie, consacrées aux analyses de fermetures. Tous deux réalisent sensiblement la même analyse. Ils divisent les fonctions en deux catégories : celles qui nécessitent un environnement alloué et celles qui n'en nécessitent pas[3]. Les algorithmes présentés dans ces thèses pour décider d'allouer ou pas les fermetures, correspondent exactement au calcul du prédicat $\mathcal{S}$. Cela signifie, que dans leurs analyses, si une fonction ne satisfait pas $\mathcal{S}$, elle est allouée. Dans notre cas, une fonction ne sera allouée que si, en plus de ne pas vérifier $\mathcal{S}$, elle ne vérifie pas non plus $\mathcal{X}$ (si elle vérifie $\mathcal{T}$, elle ne sera que partiellement allouée). Puisque la propriété que calculent Kranz et Séniak est strictement comprise dans nos propriétés, les résultats que nous obtenons avec l'algorithme ACFA sont toujours au moins aussi bons que les leurs[4].

[3] Kranz est même un peu plus précis car les fermetures qui nécessitent un environnement sont elles-mêmes divisées en deux catégories, celles dont l'environnement est alloué dans le tas et celles dont l'environnement est alloué dans la pile. Bigloo réalise également des allocations en pile mais cette optimisation ne concernant pas que les fermetures, nous ne la présentons pas ici.

[4] En mode de mise au point, Bigloo se contente de réaliser la même analyse que Kranz et Séniak, il ne calcule pas les propriétés $\mathcal{X}$ et $\mathcal{T}$.

Plusieurs travaux ont été menés sur la réduction du nombre de test de type dans les langages typés dynamiquement :

- L'inférence de types que nous avons présentée est très différente de l'algorithme W de Damas et Milner [DM82]. Notre but est de calculer des approximations des types plutôt que de réaliser une inférence globale. En particulier une variable peut, avec notre analyse, être de plusieurs types (par exemple, `integer` *ou* `real`). L'algorithme W type les fonctions *sans* savoir où elles sont appliquées alors que notre algorithme ne type les fonctions *que* parce qu'il sait où elles sont invoquées (et avec quels arguments). En général, notre analyse n'est pas aussi précise que l'algorithme W mais parfois les résultats que nous obtenons sont impossibles avec W. Par exemple, notre inférence de types peut typer une arithmétique générique avec surcharge (notre algorithme différencie les nombres flottants et les nombres entiers).

  Une autre comparaison intéressante est la complexité des deux algorithmes. Tous deux ont une « grosse » complexité (exponentielle pour W et $\mathcal{O}(n^3)$ pour ACFA) et pourtant, tous deux, semblent en général se comporter de façon linéaire !

- F. Henglein présente une optimisation qui consiste à supprimer des tests de type inutiles [Hen92]. Il n'utilise pas d'analyse de contrôle et il semble éliminer plus de tests que notre analyse le permet. Henglein affirme que son analyse supprime entre 60 et 95 % des tests. Son analyse est entièrement consacrée à l'analyse de type et pas du tout à l'amélioration de la compilation des fermetures.

- Cartwright, Wright et Fagan dans les articles [CF91, WC94] et également Thatte dans l'article [Tha90] présentent des systèmes d'inférences de types partiels. Les programmes qui ne sont pas typés statiquement ne sont pas rejetés par l'analyse mais des tests dynamiques y sont introduits. Le but de ces articles n'est pas de montrer comment on peut réduire les tests de type mais d'explorer de nouvelles façons de calculer des types avec des algorithmes partiels.

7.7 Conclusion

En utilisant et en améliorant une analyse déjà connue (la *0cfa*) nous avons conçu deux nouvelles optimisations. La première partie de ce chapitre présente l'algorithme ACFA (et ses variantes) que nous utilisons dans Bigloo. La seconde partie présente les optimisations que nous réalisons et les améliorations que nous avons obtenues dans notre compilateur.

La première optimisation concerne l'allocation des fermetures. Pour plusieurs programmes réels, nous avons mesuré que notre nouvelle analyse de fermetures permet de supprimer 87 % des allocations. En utilisant notre analyse pour réaliser une inférence de type, nous parvenons à supprimer 65 % des tests dynamiques. Le cumul de ces deux optimisations permet d'avoir une diminution du temps des exécutions de 53 % (plus d'un facteur 2) lorsque le compilateur insère des tests de type et une diminution de 59 % des temps d'exécutions lorsque le compilateur n'émet pas de test de type. Par ailleurs, grâce à l'algorithme ACFA, la taille des fichiers objets est réduite de 61 % (plus de deux fois plus petit)[5]. Tous les langages fonctionnels modernes tels ML et Scheme sont dans le champ d'application de nos nouvelles optimisations.

[5]Les performances des programmes dépendent énormément de leur capacité à tenir dans la mémoire cache. Réduire la taille des fichiers objets c'est potentiellement mieux se comporter vis à vis du cache. À terme, avec l'évolution des machines, cette diminution de la taille du code peut devenir aussi importante que la diminution actuelle du temps d'exécution.

Chapitre 8

La globalisation

La projection naturelle de Scheme vers C n'est possible que pour l'ensemble des constructions Scheme qui possèdent un équivalent en C. Ce n'est pas le cas des fonctions Scheme utilisées comme valeur de première classe. Nous allons étudier dans ce chapitre leur compilation vers C. Le processus s'apparente à l'analyse de fermetures ou au *lambda-lifting* des compilateurs traditionnels [Ste78, Joh85]. L'algorithme utilisé dans Bigloo est une amélioration de celui que N. Séniak a proposé dans sa thèse [Sén91].

Nous avons présenté dans le chapitre 7 une analyse de flot de contrôle qui établit les classes d'allocation des fonctions. Elle porte des annotations sur l'arbre de syntaxe abstraite spécifiant si une fonction doit être compilée en une fermeture ou en un simple pointeur de code. Une fermeture est une structure de données dont un des champs est un pointeur de code. Puisque nous compilons vers un C dit « orthodoxe » ce pointeur de code doit être une fonction C (car même pour les fermetures nous voulons utiliser la pile C). Il faut donc compiler une fonction locale Scheme en une fonction C. Pour cela, nous transformons la fonction locale Scheme en une fonction globale Scheme (d'où le nom de *globalisation*) qui est alors compilée normalement vers C.

8.1 Quand globaliser une fonction locale

Une fonction locale doit être globalisée dans deux cas :

- Si l'analyse de flot de contrôle a échoué dans sa tentative de suppression de cette valeur fonctionnelle. Pour reprendre la terminologie du chapitre 7, une fonction doit être globalisée quand elle ne satisfait pas le prédicat $\mathcal{S}$[1].
- Si elle est invoquée par plusieurs fonctions locales elles-mêmes globalisées. En effet si une fonction globalisée g contient des appels à une fonction locale libre f, cette fonction doit l'être également si elle ne peut pas être déplacée dans le corps de g. La condition pour laquelle une fonction locale f peut être déplacée dans une fonction globalisée g est une condition de visibilité. Déplacée dans g, f doit toujours garder visibles les fonctions qu'elle invoque et doit toujours être visible depuis les fonctions l'invoquant.

Examinons l'exemple suivant :

```
(define ...
   (labels ((foo (x) (bar x))
            (bar (x) x))
      foo))
```

La globalisation de `foo` n'entraîne pas la globalisation de `bar` car `bar` peut être déplacée (nous employons également le terme *incorporée* pour désigner qu'une fonction locale a été déplacée) dans `foo`. Le résultat de la globalisation est alors :

[1] Nous négligeons dans ce chapitre le cas où une fonction satisfait le prédicat $\mathcal{X}$ et où l'analyse de flot de contrôle parvient à montrer qu'elle n'est utilisée que dans la portée lexicale de sa définition. Dans un tel cas, la fonction n'a pas besoin d'être globalisée.

```
(define (foo x)
   (labels ((bar (x) x))
      (bar x)))
```

C'est-à-dire que la définition de `bar` est *incorporée* dans la définition de `foo`. En revanche, si `bar` est invoquée dans le corps de la construction `labels`, comme elle doit à la fois être visible depuis le `labels` et depuis la fonction `foo`, elle doit être, elle aussi globalisée. Le code résultant sera donc :

```
(define (foo x)
   (bar x))

(define (bar x)
   x)
```

La propriété $\mathcal{G}(f)$ indique qu'une fonction locale f doit être globalisée. Pour la définir formellement nous devons introduire en préliminaire trois propriétés logiques :

$$\mathcal{E}(f) \Leftrightarrow \overline{\mathcal{S}(f)}$$

$$\mathcal{A}(g, f) \equiv f \text{ est une fonction locale libre dans } g, \text{ appelée dans } g$$

$$\mathcal{L}(g, f) \equiv \text{la définition de } f \text{ peut être déplacée dans celle de } g$$

La propriété $\mathcal{E}$ exprime le fait qu'une fonction est utilisée comme objet de première classe. Elle utilise les résultats de l'analyse de flot de contrôle en utilisant le prédicat $\mathcal{S}$ (voir chapitre 7). Le prédicat $\mathcal{G}$ est alors défini comme suit :

$$\boxed{\begin{array}{c} \forall f \;\; \mathcal{G}(f) \\ \Leftrightarrow \\ \mathcal{E}(f) \vee (\exists g \mid \mathcal{A}(g, f) \wedge \overline{\mathcal{L}(g, f)} \wedge \mathcal{G}(g)) \end{array}}$$

Pour donner la définition formelle du prédicat $\mathcal{L}$, il nous faut d'abord donner les définitions de deux nouvelles propriétés :

$$\mathcal{G}_0(f) \Leftrightarrow \mathcal{E}(f) \vee (\exists g, \mathcal{E}(g) \wedge \mathcal{A}(g, f))$$

$$\mathcal{G}_1(f) \Leftrightarrow \mathcal{G}_0(f) \wedge \overline{\mathcal{E}(f)}$$

La propriété $\mathcal{G}_0(f)$ définit un sur-ensemble de $\mathcal{E}$ (c'est-à-dire que $\mathcal{G}_0(f) \Rightarrow \mathcal{G}(f)$). Intuitivement ce prédicat pourrait servir de critère de globalisation dans un système qui ne réaliserait pas d'intégration (système où $\mathcal{L}(g, f)$ serait toujours faux).
Désignons par A_f^+ l'ensemble : $\{h \mid \mathcal{A}(h, f)\}$ (l'ensemble des fonctions qui appellent f) et par A_f^- l'ensemble : $\{h \mid \mathcal{A}(f, h)\}$ (l'ensemble des fonctions qui sont appelées par f). La définition de $\mathcal{L}$ est alors :

$$\boxed{\begin{array}{c} \forall f, g \;\; \mathcal{L}(f, g) \\ \Leftrightarrow \\ \overline{\mathcal{G}_0(f)} \vee (\mathcal{E}(g) \wedge ((\mathcal{G}_0(f) \wedge (f = g)) \vee (\mathcal{G}_1(f) \wedge (\forall h \in A_f^+ \;\; \mathcal{L}(h, g)) \wedge (\forall h \in A_f^- \;\; \mathcal{E}(h) \vee \mathcal{L}(h, g))))) \end{array}}$$

Voici quelques explications informelles de la propriété $\mathcal{L}$.

- Une fonction locale qui ne vérifie pas la propriété $\mathcal{G}_0$ n'est pas globalisée. Elle n'a donc pas besoin d'être intégrée, il lui suffit de rester en place.
- Une fonction f qui vérifie la propriété $\mathcal{E}$ doit être globalisée. Elle ne peut donc être intégrée que dans elle même !
- Si une fonction vérifie $\mathcal{G}_1$ elle ne peut être intégrée que si elle reste visible depuis toutes les fonctions qui l'invoquent et si elle garde une visibilité sur toutes les fonctions qu'elle invoque.

Le calcul de $\mathcal{G}$ peut se faire en 4 étapes :

1. Calculer l'ensemble $E = \{f \mid \mathcal{E}(f)\}$. Ce calcul peut être fait de façon linéaire en parcourant l'arbre de syntaxe abstraite.

2. Calculer l'ensemble $G_0 = \{f \mid \mathcal{G}_0(f)\}$. Un algorithme linéaire pour calculer cet ensemble consiste à parcourir le graphe représenté par $\mathcal{A}$ à partir des fonctions de E, en marquant les fonctions atteintes. Chaque fonction est visitée au plus une fois.

3. Calculer l'ensemble $G_1 = \{f \mid \mathcal{G}_1(f)\}$ à partir de E et G_0.

4. Calculer la propriété $\mathcal{G}$. Il faut pour cela parcourir le graphe représenté par $\mathcal{A}$ à partir des fonctions de E, en vérifiant pour chaque fonction si elle satisfait le prédicat $\mathcal{L}$ (pour vérifier que deux fonctions f et g satisfont le prédicat $\mathcal{L}$, il suffit de parcourir le graphe d'appels, en marquant que f est intégrée dans g; chaque fonction ne peut être intégrée que dans une seule autre fonction).

8.2 Les variables locales capturées

Une fonction globalisée ou intégrée peut contenir des références à des variables locales libres qui doivent lui rester accessible une fois qu'elle a été déplacée. Pour y parvenir les variables libres sont passées en argument aux fonctions les capturant.

Désignons par $\mathcal{K}(f, v)$ le fait qu'une variable v est capturée par une fonction f. Un cas évident de capture est celui des variables libres de f, mais ce n'est pas le seul. Si f appelle des fonctions globalisées ou intégrées elle doit capturer les variables libres de ces fonctions afin d'être capable de les fournir en arguments supplémentaires. Ainsi dans l'exemple suivant:

```
(define (foo a b)
   (labels ((bar () (+ a (hux)))
            (hux () b))
      ...))
```

Si `bar` est globalisée, elle capture `a` pour ses propres besoins et `b` pour la fonction `hux`. Notons $\mathcal{V}(f, v)$ le fait que v est une variable locale libre dans f. $\mathcal{K}$ est solution de l'équation:

$$\forall f, v \;\; \mathcal{K}(f, v) \Leftrightarrow (\mathcal{G}(f) \vee \exists h \mid \mathcal{G}(h) \wedge \mathcal{L}(h, f)) \wedge (\mathcal{V}(f, v) \vee (\exists g \mid \mathcal{A}(f, g) \wedge \mathcal{K}(g, v)))$$

Comme l'indique N. Séniak [Sén91], un algorithme possible pour calculer $\mathcal{K}$ consiste à réaliser la fermeture transitive de la relation $\mathcal{A}$, puis à collecter les variables capturées par chaque fonction, ce qui est faisable en $\mathcal{O}(n^3)$, où n est le nombre de fonctions. Comme $\mathcal{A}$ ne met en relation que des fonctions définies localement au sein de la même fonction globale, le calcul de $\mathcal{K}$ peut être fait individuellement, fonction globale par fonction globale, et n reste petit.

8.3 La représentation de l'environnement

Lorsqu'une fonction globalisée est toujours appelée de façon directe, on peut lui passer en arguments supplémentaires ces variables capturées car elles sont toujours connues statiquement au site d'appel. En revanche, les variables capturées par des fonctions invoquées de façon calculée doivent être regroupées dans des structures de données qui sont passées comme argument aux fonctions invoquées.

Il existe diverses façons de représenter les environnements lexicaux. Nous avons choisi une représentation plate plutôt qu'une représentation utilisant des blocs chaînés (ou la variante utilisant une table d'indirections (*display*)). Ce choix présente l'inconvénient de pas réaliser de partage physique de blocs car l'extension d'un environnement est réalisée en le recopiant. Mais il présente à nos yeux plusieurs avantages importants. Il est très simple à mettre en œuvre, il est bien adapté aux programmes qui n'abusent pas de fonctions d'ordre supérieur (ce sont principalement ces programmes que Bigloo tente de très bien compiler pour concurrencer les compilateurs de langages itératifs) et surtout, la représentation plate diminue les fuites de mémoire. Arrêtons nous un instant sur ce problème en comparant une représentation chaînée et une représentation plate sur un l'exemple extrait de l'article [SA94]:

```
(define (foo v w)
   (let ((gee (lambda ()
                (let* ((u (car v))
                       (hux (lambda ()
                              (let ((bar (lambda () (+ u w))))
                                 (cons bar u)))))
                  (hux)))))
      (gee)))

(define (loop n res)
   (if (< n 1)
       res
       (let ((s ((foo (make-list n) 0))))
          (loop (- n 1) (cons s res)))))
```

Avec une représentation plate (figure 8.1, partie gauche), chaque évaluation de l'expression `(foo (make-list n) 0)` construit une fermeture `bar` qui contient seulement les entiers `u` et `w`. Le résultat final contient n copies de la fermeture `bar`, ainsi ce programme a une complexité en espace de $\mathcal{O}(n)$. Avec des environnements chaînés (figure 8.1, partie droite), chaque fermeture `bar` contient un pointeur sur la fermeture `hux` qui contient un pointeur sur la liste `w` de taille n. Il en résulte que l'occupation mémoire de ce programme avec une représentation chaînée est $\mathcal{O}(n^2)$.

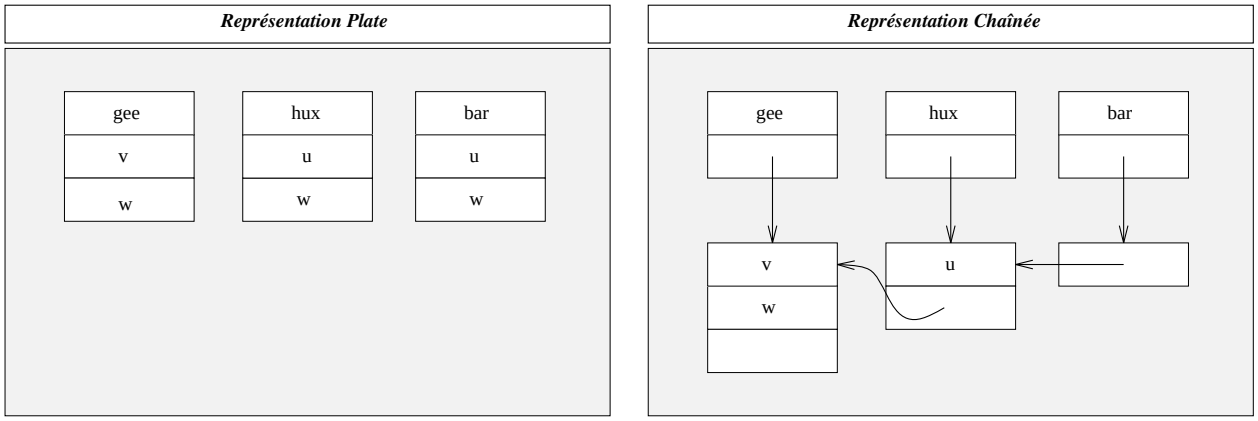


FIG. 8.1 - Les environnement plats (à gauche) et chaînés (à droite)

Le non partage des blocs implique qu'une variable peut être dupliquée dans plusieurs environnements. Pour être conforme à la sémantique de Scheme, les variables capturées doivent donc être placées dans des cellules d'indirection afin que les modifications des variables soient prises en compte par toutes les fonctions qui les capturent. On note $\mathcal{I}(v)$ lorsqu'une variable est placée dans une variable d'indirection. Si on note $\mathcal{W}$ le fait qu'une variable est affectée (est le deuxième argument d'une forme `set!`) alors $\mathcal{I}$ est définie par :

$$\boxed{\forall v \;\; \mathcal{I}(v) \Leftrightarrow \mathcal{W}(v) \wedge (\exists f \mid \mathcal{G}(f) \wedge \mathcal{K}(f, v))}$$

8.4 Représentation des valeurs fonctionnelles

Lorsqu'une fonction f ne satisfait pas le prédicat $\mathcal{S}$ elle est donc globalisée. Deux cas sont alors possibles :

1. La fonction ne satisfait pas non plus le prédicat $\mathcal{X}$ (voir chapitre 7), sa valeur est alors représentée dans le code produit par une structure contenant un pointeur de code ainsi que les variables capturées. La version globalisée de f est une fonction qui prend en premier argument son environnement puis ses paramètres formels explicites. Dans la section 2.3 nous avons présenté une structure servant à représenter

les valeurs fonctionnelles : la structure `procedure_t`. En fait cette structure n'est utilisée que pour les fonctions ne satisfaisant pas le prédicat $\mathcal{T}$. Comme nous l'avons montré dans le chapitre 7, les fonctions qui satisfont ce prédicat possèdent une propriété qui permet d'allouer une structure plus petite : la structure `procedure_light_t` :

```
struct procedure_light {      /*  Les fermetures légères          */
   union object *(*entry)();
   void          *env;
} procedure_light_t;
```

2. La fonction f ne satisfait pas le prédicat $\mathcal{X}$. Bien que dans le code source, f soit utilisée comme objet de première classe, l'analyse de flot de contrôle est parvenue à déterminer tous les sites calculés où f peut être invoquée et à prouver que sur tous ces sites, seule f pouvait être invoquée. Il n'y a donc pas de structure allouée pour cette fonction car toutes les invocations sont compilées comme des invocations directes. En revanche, il peut être indispensable d'allouer un environnement. C'est le cas quand la fonction a plus d'une variable capturée. Cet environnement est un bloc mémoire dont la taille est le nombre de variables capturées.

8.5 Les exemples de globalisations

Pour permettre une comparaison aisée, nous prenons les exemples de Séniak [Sén91]. Le premier est une version de la fonction factorielle.

```
(define (fact n)
   (letrec ((loop (lambda (r m)
                    (if (> m n)
                        r
                        (loop (* m r) (+ m 1))))))
      (loop 1 1)))
```

Puisque la fonction `loop` satisfait le prédicat $\mathcal{S}$, elle n'est pas globalisée. Le code reste donc inchangé.

Le deuxième exemple utilise une addition currifiée.

```
(define (addc n)
   (letrec ((f (lambda (m) (+ n m))))
      f))

(define (two)
   ((addc 1) 1))
```

La fonction `f` satisfait le prédicat $\mathcal{X}$ et elle n'est pas invoquée dans la portée lexicale de sa définition ; elle est donc globalisée. Elle ne contient qu'une variable capturée : `n`. Il n'y a donc pas d'allocation d'environnement. Le code est donc transformé en :

```
(define (addc n)
   n)

(define (f n m)
   (+ n m))

(define (two)
   (f (addc 1) 1))
```

Étudions un exemple où une fonction ne satisfait aucun des prédicats du chapitre 7.

```
(define (foo a b c)
   (letrec ((bar (y) (hux a))
            (hux (x) (cons x b)))
      (hux (cons bar c))))
```

La fonction `bar` est placée dans une structure de données qui est transmise au monde extérieur. L'analyse de flot de contrôle ne parvient pas à suivre cette fonction, elle doit donc être globalisée. Comme la fonction `hux`

est invoquée depuis `bar` *et* depuis `foo`, elle ne peut pas être intégrée dans `bar`. Le code sera donc transformé en :

```
(define (foo a b c)
   (let ((bar (make-procedure bar 2 1)))
      (procedure-env-set! bar 0 a)
      (procedure-env-set! bar 1 b)
      (hux (cons bar c))))

(define (hux b x)
   (cons b x))

(define (bar env y)
   (let ((a (procedure-env-ref env 0))
         (b (procedure-env-ref env 1)))
      (hux b a)))
```

La procédure `bar` a deux variables capturées (car $\mathcal{K}$(`bar`, `a`) et $\mathcal{K}$(`bar`, `b`)) (d'où le paramètre effectif `2` dans l'appel à `make-procedure`) et elle a un paramètre formel (d'où le paramètre effectif `1` dans l'appel à `make-procedure`).

Enfin, étudions le cas de fonctions satisfaisant le prédicat $\mathcal{T}$:

```
(define (execute f a)
   (f a))

(define (compile exp)
   (letrec ((do-num (lambda (x) exp))
            (do-else (lambda (x) x)))
      (if (number? exp)
          do-num
          do-else)))

(define (eval exp)
   (execute (compile exp)))
```

Les deux fonctions `do-num` et `do-else` vérifient le prédicat $\mathcal{T}$. Elle sont donc globalisées et allouées dans des structures `procedure_light_t`. Le code résultat est :

```
(define (execute f a)
   (f a))

(define (compile exp)
   (letrec ((do-num (make-procedure-light do-num 1 1))
            (do-else (make-procedure-light do-else 1 1)))
      (if (number? exp)
          do-num
          do-else)))

(define (do-num env x)
   (procedure-env-ref env 0))

(define (do-else env x)
   x)

(define (eval exp)
   (execute (compile exp)))
```

Ce code ressemble à ce qu'aurait produit la compilation si les fonctions ne satisfaisaient pas le prédicat $\mathcal{T}$ mais il faut se rappeler que la taille d'une structure `procedure_t` est d'au moins 5 mots alors qu'une structure `procedure_light_t` peut tenir en 2 !

8.6 Comparaison avec d'autres travaux

L'analyse de fermeture est la partie cruciale des compilateurs de langages fonctionnels. C'est en premier lieu sur cette partie du compilateur que tout se gagne ou ... que tout se perd ! La littérature est assez fournie dans ce domaine :

- Nos travaux s'inscrivent dans la lignée directe de ceux de N. Séniak, en apportant plusieurs améliorations majeures :
  - Notre critère pour compiler une procédure en fermeture est plus fin que celui de Séniak. Il ne suffit pas qu'une fonction soit utilisée comme objet de première classe, il faut aussi que l'analyse de flot de contrôle ne soit pas parvenue à montrer certaines propriétés. Nous avons déjà fait cette comparaison dans le chapitre 7.
  - Contrairement à N. Séniak, nous distinguons plusieurs catégories de fermetures. Ceci nous permet de faire de l'allocation *à géométrie variable*. Plus les analyses permettent d'isoler des propriétés fines des fonctions, plus les structures allouées sont petites.
  - Enfin, notre critère pour globaliser une fonction est plus fin que celui de Séniak grâce à l'introduction de l'incorporation. Nous avons proposé un moyen de calculer la propriété de globalisation en tenant compte de l'incorporation.
- N. Sundaresan présente dans l'article [Sun90] une compilation des fonctions locales Pascal vers C. La technique proposée est quasiment la même que celle de Séniak. La terminologie change un peu (l'auteur nomme la technique utilisant les environnements alloués de *structurization* et celle utilisant les paramètres supplémentaires de *extended parameter list*) mais le principe est le même. Les calculs de variables capturées sont les mêmes. Et enfin, l'auteur utilise des environnements chaînés plutôt que des représentations plates.
- Dans l'article [Ses89], P. Sestoft présente une analyse dont l'objectif est de remplacer les paramètres formels des fonctions par des variables globales dans le but de réduire la consommation de pile ou de tas. Il nomme ce processus la *globalisation* (à ne pas confondre avec notre emploi du terme globalisation qui pour nous s'applique à des fonctions). Nous devrions pouvoir tirer partie de cette analyse pour ne pas systématiquement allouer les variables capturées dans le tas.
- P. Steckler présente dans [Ste94] une transformation qu'il nomme en anglais *selective and lightweight closure conversion*. Cette transformation est très proche de celle de Séniak car elle consiste à faire la distinction entre les deux sortes de fonctions globalisées : celles qui accèdent à leurs variables capturées via un environnement et celles qui se voient affublées de paramètres supplémentaires.
- L'analyse de fermetures est la partie la plus importante de la thèse de D. Kranz [Kra88] et de l'article [KKR+86]. Son analyse comporte plusieurs phases :
  1. Une analyse d'échappement qui consiste à déterminer quand une fonction est utilisée comme objet de première classe. Cette analyse revient seulement à calculer la propriété $\mathcal{T}$ que nous avons présentée dans le chapitre 7. L'analyse d'échappement de Kranz est donc beaucoup plus grossière que la nôtre.
  2. Une analyse de représentation qui détermine comment les fonctions doivent être implantées :
     - La fonction n'a pas d'environnement. Ses paramètres effectifs peuvent être placés :
       - dans la pile. Ceci correspond au mode d'appel de C. Les fonctions de Kranz sans environnement et avec paramètres dans la pile sont donc compilées comme nos fonctions globales et nos fonctions globalisées dont tous les sites d'appels sont connus.
       - dans des registres. Comme le montre Kranz et comme l'intuition peut le laisser penser, ce type de fonctions est le plus efficace.
     - La fonction a un environnement qui peut être alloué soit dans le tas soit dans la pile. Les environnements sont représentés par ce que l'auteur nomme des *lazy displays*. Il s'agit en fait d'une représentation qui combine les environnements chaînés et les environnements utilisant des *displays*.

8.7 L'autre globalisation

Nous n'avons présenté dans ce chapitre que la globalisation des fonctions locales lorsqu'elles sont utilisées comme objet de première classe. Il existe pourtant un deuxième cas où les fonctions locales doivent être globalisées (nous parlerons de *globalisation pour C*). Puisque nous tentons d'appliquer systématiquement une projection naturelle entre Scheme et C, nous sommes confrontés à un problème épineux pour compiler les fonctions qui restent locales après l'analyse de fermeture. En effet si ces fonctions ne sont invoquées que de façon terminale alors elles peuvent être projetées vers des boucles C. En revanche, si elles sont invoquées de façon non terminale, elles doivent être globalisées pour pouvoir utiliser la pile de C. L'algorithme et le schéma que nous avons adoptés pour cette deuxième globalisation sont exactement ceux de N. Séniak [Sén90] parce qu'ils réalisent la projection naturelle des fonctions Scheme vers des boucles ou des fonctions C et surtout parce que l'optimalité de son algorithme est prouvée[2]. Nous n'allons pas présenter cet algorithme mais nous allons présenter deux exemples de globalisation pour C. Le premier concerne des fonctions mutuellement récursives :

```
(define (mul4 n)
   (letrec ((even (lambda (m)
                    (if (= m 0)
                        #t
                        (odd (- m 1)))))
            (odd  (lambda (m)
                     (if (= m 0)
                         #f
                         (even (- m 1))))))
      (if (even n)
          (even (/ n 2))
          #f)))
```

La fonction **even** est appelée trois fois dans **mul4**, une fois de façon terminale et une fois de façon non terminale. Elle doit être globalisée. En revanche la fonction **odd** n'est invoquée que sur des sites terminaux, elle peut donc être intégrée dans **even**. La code produit pour la fonction **mul4** en C ressemblera donc à:

```
bool mul4( int n )
{
   if( even( n ) )
      return even( n / 2 );
   else
      return FALSE;
}

bool even( int m )
{
even:
   if( m == 0 )
      return TRUE;
   else
   {
      m = m - 1;
      goto odd;
   }
odd:
   if( m == 0 )
      return FALSE;
   else
   {
      m = m - 1;
      goto even;
   }
}
```

[2] L'algorithme est optimal dans le sens de la diminution du nombre de fonctions globalisées.

Les fonctions locales utilisées pour implanter des boucles ne sont invoquées que sur des sites terminaux : elles ne sont donc jamais globalisées. C'est pourquoi le code :

```
(define (foo v)
   (let ((l (vector-length v)))
      (letrec ((loop (lambda (r acc)
                       (if (= r l)
                           acc
                           (loop (+ 1 r)
                                 (+ (vector-ref v r) acc))))))
         (loop 0 0))))
```

sera traduit en :

```
int foo( vect_t v )
{
   int l, r, acc;

   l = vector_length( v );
   acc = 0;
   r = 0;

loop:
   if( r == l )
      return acc;
   else
   {
      acc = v[ r ] + acc;
      r = r + 1;
      goto loop;
   }
}
```

8.8 Conclusion

Nous avons présenté les méthodes utilisées pour compiler les fonctions locales Scheme. Si une fonction locale Scheme ne peut pas être projetée vers une boucle C (soit parce qu'elle est utilisée comme objet de première classe soit parce qu'elle est invoquée de façon non terminale), elle est projetée vers une fonction globale C. Nous appelons cette opération la *globalisation*. La globalisation des fonctions utilisées comme valeurs de première classe est réalisée dans une passe différente de la globalisation des fonctions pour C. L'avantage de cette séparation est de repousser dans les dernières passes du compilateur les contraintes du langage cible (ici la contrainte est que C ne possède pas de fonctions locales).

Chapitre 9

Les analyses de flot de données

Ce chapitre contient les descriptions de plusieurs optimisations « classiques » d'analyse de flot de données (*data flow analysis*). Si ces optimisations sont bien connues pour les compilateurs de langages impératifs comme Fortran, Pascal ou C [ASU86, FL88] elles ont été moins étudiées pour les langages d'ordre supérieur. Nous décrivons ici l'élimination de sous-expressions communes (CSE) et la β-réduction effectuée pendant la compilation. Ces deux optimisations ont plusieurs points communs :

- Elles ont pour but de réduire le nombre d'expressions calculées.
- Elles utilisent des techniques proches (par exemple, elles utilisent toutes les deux abondamment les noms intermédiaires qui sont donnés à certaines expressions).
- Elles utilisent toutes les deux les informations collectées par deux passes précédentes : `Hoist` et `Effect`

Après avoir incorporé ces optimisations dans Bigloo et avoir mesuré leurs impacts sur des programmes tests, nous sommes arrivés aux mêmes conclusions que C. Chambers[1] : ces optimisations n'ont pas une incidence majeure sur les performances des programmes exécutables. En revanche, elles diminuent en proportion assez importante la taille des fichiers C produits, permettant ainsi aux compilateurs C de compiler plus vite. Dans la mesure où, comme nous l'avons déjà mentionné, la lenteur des compilateurs C est un des problèmes (si ce n'est *le* problème) de notre schéma de compilation (compiler vers C plutôt que compiler vers un langage d'assemblage), les optimisations décrites dans ce chapitre ont une importance non négligeable dans Bigloo.

Puisque les analyses que nous présentons ici ont plus pour but de diminuer la taille des fichiers C que d'améliorer les performances des exécutables, nous avons estimé judicieux de placer dans ce chapitre une section présentant la compilation des constantes de Bigloo (la passe **Cnst**). Comme nous le montrerons dans la section 9.7 une mauvaise compilation des constantes peut s'avérer problématique en allongeant de façon inacceptable les temps de compilation.

9.1 L'élimination des sous-expressions communes (`Cse`)

L'élimination de sous-expressions communes (CSE) est une optimisation qui consiste à supprimer des calculs redondants, c'est-à-dire des calculs effectués plusieurs fois pendant une même exécution. C'est une optimisation classique puisqu'elle est réalisée par de très nombreux compilateurs optimisant (gcc, Sml/NJ, SELF, ...).

Généralement cette optimisation s'applique quand le programme est exprimé dans un langage proche d'un langage d'assemblage. Dans Bigloo nous l'utilisons dans un contexte original puisque nous l'utilisons au niveau du langage intermédiaire Sqil qui est de bien plus haut niveau qu'un langage d'assemblage (voir section 3.1). Ainsi nous appliquons la CSE sur des expressions complexes et non pas sur des instructions machine. L'intérêt est que toute expression (donc tout appel de fonction) Sqil peut être supprimée par cette

[1] La thèse de C. Chambers [Cha92c] contient des études de performance qui isolent chaque optimisation. Sont ainsi présentées les mesures des gains de chacune d'entres elles. Les chiffres donnés font état d'une diminution moyenne de moins de 15 % du temps d'exécution pour la CSE.

optimisation. La CSE s'applique donc sur des opérations aussi diverses que des accès mémoire, des opérations arithmétiques, des tests de type ou, toute expression composée présente dans le programme source.

9.1.1 Le principe de la CSE

Le langage intermédiaire nomme les expressions (si le langage intermédiaire est proche d'un langage d'assemblage cela se fera en plaçant le résultat d'une expression dans un registre, si le langage intermédiaire est CPS, l'expression sera le paramètre effectif d'une continuation enfin si le langage est de plus haut niveau comme Sqil, les expressions sont liées à des variables auxiliaires au moyen de la construction `let`). Généralement la CSE utilise ces correspondances *nom/expression*. Si un nom est déjà lié à une expression et que dans la portée lexicale de ce nom le compilateur retrouve une évaluation de l'expression, il pourra la remplacer par un accès à la variable liée. Cette technique est classique (elle est par exemple dans [ASU86] qui est une référence dans la bibliographie portant sur la compilation) et nous l'utilisons dans Bigloo. En plus, nous faisons une CSE particuliere pour les prédicats. Lors de l'évaluation d'une expression conditionnelle (`if` *si alors sinon*) le compilateur sait que dans l'évaluation de l'expression *alors*, l'expression *si* vaut `vrai` et que dans l'expression *sinon*, elle vaut `faux`. Il n'est donc pas utile d'avoir lié le prédicat à une variable pour le réduire si on le retrouve dans les sous-branches d'une autre expression conditionnelle.

Voici donc schématisés les deux types de transformations que réalise la CSE. La première utilise les variables auxilaires.

$$\mathcal{CSE}_{bindings}\left[\!\left[\begin{array}{l}\texttt{(let (... (}var\ exp\texttt{) ...)}\\ \quad\ddots\\ \quad\boxed{exp}\\ \quad\quad\ddots\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\texttt{(let (... (}var\ exp\texttt{) ...)}\\ \quad\ddots\\ \quad\boxed{var}\\ \quad\quad\ddots\texttt{)}\end{array}\right]\!\right]$$

Alors que la deuxième porte sur les prédicats.

$$\mathcal{CSE}_{cond}\left[\!\left[\begin{array}{l}\texttt{(if }si\text{-}exp\\ \quad\ddots\\ \quad\boxed{\texttt{(if }si\text{-}exp\ alors\text{-}exp\ sinon\text{-}exp\texttt{)}}\\ \quad\quad\ddots\\ \quad\ddots\\ \quad\boxed{\texttt{(if }si\text{-}exp\ alors\text{-}exp\ sinon\text{-}exp\texttt{)}}\\ \quad\quad\ddots\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\texttt{(if }si\text{-}exp\\ \quad\ddots\\ \quad\boxed{alors\text{-}exp}\\ \quad\quad\ddots\\ \quad\ddots\\ \quad\boxed{sinon\text{-}exp}\\ \quad\quad\ddots\texttt{)}\end{array}\right]\!\right]$$

9.1.2 La CSE et les effets de bords

Les effets de bords peuvent interdire les deux transformations précédentes. Par exemple pour les deux programmes suivants :

```
(let ((var (foo 3)))          |  (let ((var (print 3)))
   (set! var 6)               |     ...
   (foo 3))                   |     (print 3))
```

L'expression `(foo 3)` ne peut pas être réduite car la variable à laquelle elle était liée est affectée. L'expression `(print 3)` ne peut pas l'être non plus, car elle réalise un effet de bord.

```
cse( atree, p+, p-, already )=
   selon atree
        [[ A ]]:
            atree
        [[ (quote ...) ]]:
            atree
        [[ (begin atree_1 ... atree_n) ]]:
            soit atree_1'=cse( atree_1, p+, p-, already ),
               ⋮
            soit atree_n'=cse( atree_n, p+, p-, already ),
               [[ (begin atree_1' ... atree_n') ]]
        [[ (set! var val) ]]:
            [[ (set! var cse( val, p+, p-, already )) ]]
        [[ (let ((var_1 val_1) ...) body) ]]:
            soit val_1'=cse( val_1, p+, p-, already ),
            soit val_1''=
                  si side-effect?( val_1' ) et La variable var_1 n'est pas affectée et val_1'∈already
                       alors la variable ou l'expression val_1' est liée
                       sinon val_1',
            soit already^1=
                  si side-effect?( val_1' ) et val_1'∉already
                       alors {< var_1 × val_1'' >}⋃already
                       sinon already,
               ⋮
            soit body'=cse( body, p+, p-, already^n ),
               [[ (let ((var_1 val_1') ...) body') ]]
        [[ (labels ((fun_1 (arg_1 ...) body_1) ...) body) ]]:
            soit body_1'=cse( body_1, p+, p-, already ),
               ⋮
            soit body'=cse( body, p+, p-, already ),
               [[ (labels ((fun_1 (arg_1 ...) body_1') ...) body') ]]
        [[ (if si alors sinon) ]]:
            soit si'=cse( si, p+, p-, already ),
               si side-effect?( si' )
                    alors [[ (if si' cse( alors, p+, p-, already ) cse( sinon, p+, p-, already )) ]]
                    sinon si si'∈p+
                                alors cse( alors, p+, p-, already )
                          ou bien si si'∈p-
                                alors cse( sinon, p+, p-, already )
                          sinon soit alors'=cse( alors, {si'}⋃p+, p-, already ),
                                 soit sinon'=cse( sinon, p+, p-, already ),
                                    [[ (if si' alors' sinon') ]]
              ⋮
        [[ (fun arg_1 ...) ]]:
            soit fun'=cse( fun, p+, p-, already ),
            soit arg_1'=cse( arg_1, p+, p-, already ),
               ⋮
            soit atree'=[[ (fun' arg_1' ...) ]],
               si side-effect?( atree' ) et atree'∈already
                    alors la variable ou l'expression atree' est liée
                    sinon atree'
   fin
```

Algorithme 9.1: L'algorithme d'élimination des sous-expressions communes

9.1.3 L'algorithme de CSE

Nous avons conçu un algorithme qui réalise l'élimination des sous-expressions communes (figure 9.1).

- La fonction *cse* prend en argument : *(i)* Un arbre de syntaxe abstraite. *(ii)* Une liste de prédicats retournant la valeur `vrai`. *(iii)* Une liste de prédicats retournant la valeur `faux`. *(iv)* Une liste d'associations *variable/expression*. Elle retourne un nouvel arbre de syntaxe abstraite où des expressions communes ont été retirées.
- La fonction *cse* est invoquée lors d'une compilation sur toutes les définitions globales avec pour arguments : *(i)* L'arbre de syntaxe représentant le corps de la fonction. *(ii)* *(iii)* *(iv)* Trois listes vides.
- La fonction *cse* effectue un parcours de l'arbre de syntaxe qui lui est fourni en argument.
- Deux catégories d'expressions peuvent être factorisées. Les prédicats et les expressions liées à des variables des constructions `let`. C'est donc lors des constructions `if` et `let` que les ensembles p^+, p^- et *already* sont modifiés.
- À cause des restrictions énoncées dans la section 9.1.2 avant d'augmenter un des trois ensembles, il faut s'assurer que l'expression ajoutée ne contient aucun effet de bord. Le prédicat *side-effect*? présenté dans la section 9.3 permet de savoir si une expression ne réalise aucun effet de bord[2].
- La suppression des expressions liées à des variables dans les `let` peut conduire à des expressions liant entres elles deux variables. La suppression de ces liaisons inutiles tombera dans le champ d'application de l'optimisation de β-réduction.
- L'algorithme parcourt l'arbre de syntaxe abstraite de façon linéaire. Supposons n la taille de cet arbre. Pour chaque application examinée à la $i^{ème}$ itération de l'algorithme, l'ensemble *already* est au maximum de taille i (nous négligeons le cas des expressions conditionnelles car il est exactement analogue en complexité à celui des appels fonctionnels). La complexité de l'algorithme 9.1 est donc proportionnelle à $\sum_{i=0}^{n} 1 + i$. C'est-à-dire que la complexité dans le pire cas est $\mathcal{O}(\frac{n^2+3n}{2})$. En pratique, à cause des restrictions dues aux effets de bords, l'ensemble *already* reste très petit. La complexité moyenne semble donc très proche de $\mathcal{O}(n)$.

Note d'implantation : Le prédicat side-effect? retourne la valeur `vrai` *si une expression réalise un effet de bord (par exemple si l'expression invoque une fonction qui réalise un effet de bord). Pour éviter les nombreuses utilisations de ce prédicat (qui peuvent augmenter le temps de la compilation), il est judicieux de remarquer que si la fonction courante ne réalise pas globalement d'effet de bord alors aucune des expressions qui la compose n'en fait.*

9.2 La β-réduction (`Beta`)

L'optimisation dite de β-réduction a pour but de supprimer les liaisons inutiles. Il s'agit comme pour la CSE d'une optimisation couramment réalisée par les compilateurs optimisant. Les techniques mises en œuvre pour cette optimisation sont proches de celles utilisées par la CSE. Le principe de cette optimisation est très simple. Elle se contente d'opérer les transformations suivantes :

$$\beta_{binding}\ \left[\!\!\left[\begin{array}{l} \texttt{(let (... (}var_x\ var_y\texttt{) ...)} \\ \quad \ddots \\ \qquad \boxed{var_x} \\ \qquad\quad \ddots\texttt{)} \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(let (...)} \\ \quad \ddots \\ \qquad \boxed{var_y} \\ \qquad\quad \ddots\texttt{)} \end{array} \right]\!\!\right]$$

[2] En pratique ce n'est qu'une approximation sûre du prédicat *side-effect*? qui est utilisée dans Bigloo.

Si une variable n'est utilisée qu'une seule fois dans sa portée lexicale et que sa valeur de liaison ne fait pas d'effet de bord, elle peut être supprimée :

$$\beta_{1occure}\ \left[\!\!\left[\begin{array}{l} \texttt{(let (...}\ \textit{(var exp)}\ \texttt{...)} \\ \quad \ddots \\ \qquad \boxed{var} \\ \qquad\quad \ddots\texttt{)} \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(let (...)} \\ \quad \ddots \\ \qquad \boxed{exp} \\ \qquad\quad \ddots\texttt{)} \end{array} \right]\!\!\right]$$

Enfin, nous avons amélioré l'optimisation pour qu'elle puisse réaliser la transformation suivante :

$$\beta_{hoisting}\ \left[\!\!\left[\begin{array}{l} \texttt{(let (...}\ \textit{(var exp)}\ \texttt{...)} \\ \quad \ddots \\ \qquad \boxed{\begin{array}{l} \textit{(f (g var))} \\ \vdots \\ \textit{(h (g var))} \end{array}} \\ \qquad\quad \ddots\texttt{)} \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(let (...}\ \textit{(var (g exp))}\ \texttt{...)} \\ \quad \ddots \\ \qquad \boxed{\begin{array}{l} \textit{(f var)} \\ \vdots \\ \textit{(h var)} \end{array}} \\ \qquad\quad \ddots\texttt{)} \end{array} \right]\!\!\right]$$

L'intérêt de cette transformation ($\beta_{hoisting}$) n'étant pas immédiat, voici un exemple d'application. Imaginons le programme suivant :

```
(let ((var (foo ...)))
   (if var ... ...)
   ...
   (if var ... ...))
```

La compilation des expressions conditionnelles conduit au code suivant :

```
(let ((var (foo ...)))
   (if (bbool->cbool var) ... ...)
   ...
   (if (bbool->cbool var) ... ...))
```

la passe d'intégration (**Inline**) place les résultats d'applications dans des variables intermédiaires. Or, dans notre exemple, c'est la passe de typage (**Typage**) qui a incorporé l'appel à la fonction `bbool->cbool` dans l'arbre de syntaxe abstraite. Comme la passe de typage survient après la passe d'intégration, le résultat de cet appel de fonction n'est pas mis dans une variable auxiliaire. Les réductions vues précédemment dans ce chapitre ne factoriseront donc pas les deux appels à `bbool->cbool` mais grâce à la transformation $\beta_{hoisting}$, ce code sera réécrit en :

```
(let ((var (bbool->cbool (foo ...))))
   (if var ... ...)
   ...
   (if var ... ...))
```

9.2.1 La β-réduction et les effets de bords

Comme pour la CSE les effets de bords peuvent interdire les applications des transformations que nous avons données précédemment. Il ne suffit plus seulement de savoir si une variable est affectée, si elle est liée à une expression qui fait des effets de bords. Il faut également savoir si une variable est liée à une valeur qui peut éventuellement être transformée. C'est par exemple le cas dans le programme suivant :

```
(let ((l (cons 1 2)))
   (let ((x (car l)))
      (set-car! l 3)
      x))
```

Bien que la variable `x` ne soit utilisée qu'une seule fois, la construction `let` ne peut pas être supprimée. L'algorithme de β-réduction que nous avons conçu pour Bigloo utilise donc la fonction *side-effect*? déjà présente dans l'algorithme 9.1 mais il utilise également le prédicat *mutable*? qui retourne la valeur `vrai` si une valeur peut être transformée et la valeur `faux` dans le cas contraire.

9.2.2 L'algorithme de β-réduction

L'algorithme effectuant la β-réduction présenté dans la figure 9.2.

- La fonction *β-reduce* prend en arguments : *(i)* Un arbre de syntaxe abstraite. *(ii)* Un ensemble de couples *variable/expression*. Elle retourne un nouvel arbre où des liaisons construites par la forme `let` ont été supprimées.
- La fonction *β-reduce* est invoquée lors d'une compilation sur toutes les définitions globales avec pour arguments : *(i)* l'arbre de syntaxe représentant le corps de la fonction. *(ii)* l'ensemble vide.
- La fonction *β-reduce* effectue un parcours de l'arbre de syntaxe abstraite quelle reçoit en argument.
- Une judicieuse implantation peut supprimer le besoin d'implanter *replace* par une liste. Ainsi une bonne implantation de l'algorithme 9.2 est linéaire en fonction de la taille de l'arbre de syntaxe abstraite.

Note d'implantation : On peut faire la même remarque que pour l'implantation de l'algorithme 9.1, si une fonction ne fait globalement d'effet de bord, il n'est pas utile d'utiliser le prédicat side-effect? ni le prédicat mutable? pour ces sous-expressions.

9.3 La détection des effets de bords (`Effect`)

Les deux algorithmes 9.1 et 9.2 utilisent les prédicats *side-effect*? et *mutable*?. Ces deux prédicats reçoivent en argument un arbre de syntaxe abstraite. Le premier retourne la valeur `vrai` si l'expression correspondant à l'arbre peut éventuellement réaliser un effet de bord, et `faux` dans le cas contraire. Le second retourne la valeur `vrai` si le résultat de l'évaluation de l'expression correspondant à l'arbre de syntaxe abstraite est une valeur modifiable (une liste, un vecteur, une chaîne de caractères, ...); et `faux` dans le cas contraire. Nous allons étudier dans cette section comment sont calculées ces deux propriétés.

9.3.1 *side-effect*?

Une passe entière (`Effect`) est consacrée au calcul du prédicat *side-effect*?. Elle ajoute des annotations dans l'arbre de syntaxe abstraite. La fonction *side-effect*? se contente de les consulter. La passe (`Effect`) consiste en une itération de point fixe sur l'ensemble des fonctions faisant des effets de bords. Un parcours de l'arbre établit $\mathcal{W}$, l'ensemble des fonctions qui utilisent la construction `set!` ou qui invoquent une fonction externe (les fonctions de la bibliothèque d'exécution possèdent des annotations particulières précisant quelles sont celles qui ne font pas d'effet de bord). L'algorithme d'itération est donné en 9.3.

9.3.2 *mutable*?

Le prédicat *mutable*? est réalisé sur le même modèle que *side-effect*?. Toute fonction dont le type du résultat ne peut pas être totalement déterminé est supposée retourner un résultat modifiable. Une itération de point fixe permet de calculer totalement la propriété. Les fonctions de la bibliothèque portent des annotations permettant au compilateur de savoir si leurs résultats sont modifiables.

9.4 La réorganisation de l'arbre de syntaxe abstraite (`Hoist`)

Les deux optimisations précédemment décrites (la CSE et la β-réduction) s'appuient sur les liaisons des expressions dans des blocs lexicaux. Ainsi, une expression *exp* n'est factorisée que si elle apparaît dans la

```
β-reduce( atree, replace )=
   selon atree
        [[ V ]]:
            si atree est une variable globale ou atree∉replace
                 alors atree
            sinon l'expression qui remplace atree dans replace
        [[ A ]]:
            atree
        [[ (quote ...) ]]:
            atree
        [[ (begin atree_1 ...) ]]:
            [[ (begin β-reduce( atree_1, replace ) ...) ]]
        [[ (set! var val) ]]:
            [[ (set! var β-reduce( val, replace )) ]]
        [[ (let ((var_1 val_1) ...) body) ]]:
            soit replace^1=replace,
            soit val_1'=β-reduce( val_1, replace ),
            soit val_1''=si var_1 est affectée ou side-effect?( val_1' ) ou mutable?( val_1' )
                 alors val_1'
            ou bien si var_1 est toujours utilisée comme argument d'une fonction f
                 alors [[ (f var_1) ]]
            ou bien si var_1 n'est utilisée qu'une seule fois
                 alors replace^1 ← {< var_1 × val_1' >}⋃replace^1,
                       la liaison doit être supprimée, on note ceci par la valeur distinguée ◇
            ou bien si val_1' est une variable locale non affectée
                 alors replace^1 ← {< var_1 × val_1' >}⋃replace^1,
                       ◇
            sinon val_1',
               ⋮
            soit body'=β-reduce( body, replace^n ),
               [[ (let (si val_1'' ≠ ◇
                       alors (var_1 val_1'') ...) body') ]]
        [[ (labels ((fun_1 (arg_1 ...) body_1) ...) body) ]]:
            soit body_1'=β-reduce( body_1, replace ),
               ⋮
            soit body'=β-reduce( body, replace ),
               [[ (labels ((fun_1 (arg_1 ...) body_1') ...) body') ]]
        [[ (if si alors sinon) ]]:
            soit si'=β-reduce( si, replace ),
            soit alors'=β-reduce( alors, replace ),
            soit sinon'=β-reduce( sinon, replace ),
               [[ (si si' alors' sinon') ]]
             ⋮
        [[ (fun arg_1 ... arg_n) ]]:
            si n = 1 et
               side-effect?( atree ) et
               mutable?( atree ) et
               arg_1 n'est jamais affectée et est toujours argument de fun
                 alors arg_1
                 sinon soit arg_1'=β-reduce( arg_1, replace ),
                             ⋮
                       soit arg_n'=β-reduce( arg_n, replace ),
                       soit fun=β-reduce( fun, replace ),
                          [[ (fun arg_1 ... arg_n') ]]
   fin
```

Algorithme 9.2: L'algorithme de réduction

$\mathcal{W}$: L'ensemble des fonctions faisant directement des effets de bords (fonctions utilisant `set!`, `set-car!`, ...)

soit w=$\mathcal{W}$,
  tant que $w \neq \emptyset$ **faire**
    soit f=le premier élément de w,
      $w \leftarrow$ tous les éléments sauf le premier de w,
      si f n'est pas marquée comme faisant des effets de bords
        alors marquer f comme faisant des effets de bords,
          $w \leftarrow \{$toutes les fonctions appelée par $f\} \bigcup w$
  fait

Algorithme 9.3: La détection des effets de bords

portée lexicale d'une variable déjà liée à *exp* ou si elle est en position de test dans une branche d'une autre conditionnelle où elle servait également de test. Étudions alors le comportement de ces optimisations sur l'expression suivante : `(foo (car x) (cdr x))`. Après la passe `Typage` le programme est transformé en :

```
(foo (if (pair? x) (car x) (error ...)
     (if (pair? x) (cdr x) (error ...))))
```

Bien que le test `(pair? x)` apparaisse deux fois, il ne pourra pas être factorisé. Pour améliorer les deux optimisations nous avons ajouté une passe à Bigloo : la passe `Hoist`. Cette passe réorganise l'arbre de syntaxe abstraite. Par exemple, sur le code précédent, après application de cette passe l'arbre de syntaxe sera :

```
(if (pair? x)
    (if (pair? x)
        (foo (car x) (cdr x))
        (error ...))
    (error ...))
```

Sur ce code-ci, la CSE et la β-réduction peuvent s'appliquer.

Cette passe est facultative puisqu'elle a seulement pour but d'améliorer le comportement d'optimisations.

9.4.1 Les réécritures

La passe `Hoist` est une passe de réécriture de l'arbre qui consiste à remonter le plus près possible de la racine les expressions qui correspondent à des erreurs d'exécution (par exemple les expressions qui réalisent des vérifications de type). Pour y parvenir, elle recherche des motifs dans l'arbre qu'elle transforme. Nous donnons ici les trois plus importantes transformations effectuées (nous désignons par $\mathcal{F}$ une expression représentant une invocation de la fonction d'erreur). Ces trois transformations repèrent le motif `(if` *si alors* $\mathcal{F}$`)`.

$$\mathcal{HOIST}_{if}\ [\![(\texttt{if}\ \boxed{(\texttt{if}\ si\ alors\ \mathcal{F})}\ alors'\ sinon')]\!] = [\![(\texttt{if}\ si\ (\texttt{if}\ alors\ alors'\ sinon')\ \mathcal{F})]\!]$$

$$\mathcal{HOIST}_{app}\ [\![(\texttt{f}\ \ldots\ \boxed{(\texttt{if}\ si\ alors\ \mathcal{F})}\ \ldots)]\!] = [\![(\texttt{if}\ si\ (\texttt{f}\ \ldots\ alors\ \ldots)\ \mathcal{F})]\!]$$

$$\mathcal{HOIST}_{let}\ \left[\!\left[\begin{array}{l}\texttt{(let (... (}var\ \boxed{\texttt{(if }si\ alors\ \mathcal{F}\texttt{)}}\texttt{) ...)}\\ \quad exp\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\texttt{(if }si\\ \quad \texttt{(let (... (}var\ alors\texttt{) ...)}\\ \quad\quad exp\texttt{)}\\ \quad \mathcal{F}\texttt{)}\end{array}\right]\!\right]$$

Ces trois transformations utilisent la caractéristique bloquante des expressions $\mathcal{F}$. Ainsi il n'est pas utile de dupliquer des fragments d'arbres. Par exemple la transformation $\mathcal{HOIST}_{if}$ appliquée dans un cadre général devrait être :

$$\mathcal{HOIST}_{if_{general}}\ \left[\!\left[\begin{array}{l}\texttt{(if }\boxed{\texttt{(if }si\ alors\ sinon\texttt{)}}\\ \quad alors'\\ \quad sinon'\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\texttt{(if }si\\ \quad\quad \texttt{(if }alors\\ \quad\quad\quad alors'\\ \quad\quad\quad sinon'\texttt{)}\\ \quad\quad \texttt{(if }sinon\\ \quad\quad\quad alors'\\ \quad\quad\quad sinon'\texttt{))}\end{array}\right]\!\right]$$

Les expressions $alors'$ et $sinon'$ doivent être dupliquées. Les trois transformations $\mathcal{HOIST}_{if}$, $\mathcal{HOIST}_{app}$ et $\mathcal{HOIST}_{let}$ ne s'opèrent qu'en présence d'invocations à la fonction d'erreur. Elle contribueront à rendre plus efficace la CSE et la β-réduction quand la compilation placera des tests de type. Une quatrième transformation joue un rôle très important car elle s'applique quelque soit le mode de compilation :

$$\mathcal{HOIST}_{applet}\ \left[\!\left[\begin{array}{l}\texttt{(}f\ \ldots\\ \quad\quad \boxed{\texttt{(let ((}var\ val\texttt{)) }exp\texttt{)}}\\ \quad \ldots\texttt{)}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\texttt{(let ((}var\ val\texttt{))}\\ \quad \texttt{(}f\ \ldots\ exp\ \ldots\texttt{))}\end{array}\right]\!\right]$$

Nous ne donnons pas l'algorithme que nous utilisons pour faire ces transformations en raison de la simplicité de la tâche réalisée. Précisons seulement qu'une implantation efficace utilisant des effets de bords dans l'arbre conduit à une passe linéaire : une descente dans l'arbre suffit pour appliquer toutes les transformations.

9.5 Exemples d'applications

Nous montrons quelques exemples d'applications des optimisations de CSE et de β-réduction.

9.5.1 Les expressions booléennes

Nous avons mentionné dans la section 4.3 une difficulté lors de la compilation des expressions booléennes. L'exemple que nous avions donné est la compilation d'un `ou` binaire. Voici ce programme avant et après la passe `Typage`.

```
(if (let ((ortest a))                    (if (let ((ortest a))
      (if ortest                                (if (bbool->cbool ortest)
          ortest                                    (bbool->cbool ortest)
          b))                                       (bbool->cbool b)))
    x                                        x
    y)                                       y)
```

Grâce à la passe de CSE, ce code devient :

```
(let ((ortest a))
   (if (if (bbool->cbool ortest)
           #t
           (bbool->cbool b))
       x
       y))
```

Lors d'une passe ultérieure, le test portant sur `#t` sera supprimée. On obtiendra donc une compilation efficace de l'expression.

9.5.2 Les tests de type

Nous illustrons ici l'efficacité des deux optimisations de ce chapitre dans la diminution des tests de type. Pour cela nous examinons la compilation de la fonction de la bibliothèque d'exécution Scheme `remq` :

```
(define (remq x y)
   (cond
      ((null? y) y)
      ((eq? x (car y))
       (remq x (cdr y)))
      (else
       (cons (car y) (remq x (cdr y))))))
```

après typage, cette définition devient :

```
(define (remq x y)
   (if (null? y)
       y
       (if (let ((obj25 (car (if (pair? y)
                                 y
                                 (error ...)))))
              (eq? x obj25))
           (remq x (cdr (if (pair? y)
                            y
                            (error ...))))
           (let ((obj21 (remq x (cdr (if (pair? y)
                                         y
                                         (error ...))))))
              (obj10 (car (if (pair? y)
                              y
                              (error ...)))))
              (cons obj10 obj21)))))
```

Grâce à la passe `Hoist`, la définition est transformée en :

```
(define (remq x y)
   (if (null? y)
       y
       (if (pair? y)
           (let ((obj25 (car y)))
              (if (eq? x obj25)
                  (if (pair? y)
                      (remq x (cdr y))
                      (error ...))
                  (if (pair? y)
                      (if (pair? y)
                          (let ((obj21 (remq x (cdr y)))
                                (obj10 (car y)))
                             (cons obj10 obj21))
                          (error ...))
                      (error ...))))
           (error ...))))
```

Et enfin, l'élimination des sous-expressions communes et la β-réduction conduisent à l'obtention du code suivant :

```
(define (remq x y)
   (if (null? y)
       y
       (if (pair? y)
           (let ((obj25 (car y)))
              (if (eq? x obj25)
                  (remq x (cdr y))
                  (cons obj25 (remq x (cdr y)))))
           (error ...))))
```

Ce code est optimal en nombre de tests de type et en accès (pour deux arguments `x` et `y` inconnus) puisque au cours d'une même itération, le prédicat `pair?` et les fonctions d'accès aux listes (`car` et `cdr`) ne sont employées qu'une seule fois.

9.6 L'impact des optimisations

Nous allons étudier pour plusieurs programmes l'impact des deux optimisations présentées dans ce chapitre. Pour cela, nous allons mesurer la taille des fichiers C produits (**.c** en nombre de lignes), la taille des fichiers objet obtenus après la compilation C en mode -O (**.o** en K). Les temps de compilation sans la compilation C (**compil.**) et avec (**compil.+cc**) sont exprimés en secondes. Ces compilations sont faites en mode où les tests de type sont émis et Bigloo n'utilise pas d'autre optimisation que la CSE et la β-réduction. La colonne **run** indique les temps d'exécution. La ligne δ est le rapport entre les mesures obtenues sans optimisation et celles avec optimisations. Tous les temps présentés ici ont été relevés sur machine Mips.

queens	**taille .c**	**taille .o**	**compil.**	**compil.+cc**	**run**
sans optimisation	840	8	1.5	10.1	14.1
CSE + β	745	8	1.5	8.4	13.9
δ	11 %	0 %	0 %	17 %	1 %

Queens est un petit programme. Les gains ne sont pas très spectaculaires. Néanmoins, l'objectif de diminution des temps de compilations est en partie atteint, puisque grâce à nos optimisations, le compilateur C est parvenu à compiler 17 % plus vite le fichier produit par Bigloo. Il n'y a pas de gain de performance du code produit.

pseudoknot	**taille .c**	**taille .o**	**compil.**	**compil.+cc**	**run**
sans optimisation	7667	200	14.6	171.6	37.4
CSE + β	5528	160	13.2	108.9	31.3
δ	28 %	20 %	10 %	37 %	16 %

Sur **pseudoknot**, l'effet de nos optimisations se fait beaucoup plus sentir puisque la compilation (jusqu'à la production du code objet) est plus courte de 37 %. Il faut noter un phénomène d'apparence surprenante : bien que la compilation Bigloo contienne plus de passes, la production de code C est plus rapide de 10 % en mode optimisant. Ceci s'explique parce que les dernières passes travaillent sur un arbre de syntaxe abstraite de taille plus faible.

beval	**taille .c**	**taille .o**	**compil.**	**compil.+cc**	**run**
sans optimisation	4770	65	4.2	61.4	33.7
CSE + β	3667	44	3.7	42.5	30.7
δ	23 %	32 %	12 %	31 %	9 %

Ce programme est significatif des résultats de la CSE et de la β-réduction. Le gain en performance est d'environ 10 % alors que le gain sur le temps global de compilation est de 30 %.

peval	taille .c	taille .o	compil.	compil.+cc	run
sans optimisation	6401	73	4.8	63.4	13.8
CSE + β	5394	54	4.4	50.2	14.8
δ	16 %	26 %	8 %	21 %	-7 %

Les mesures du programme **peval** imposent quelques explications complémentaires. Bien que les fichiers objets soit plus petits (car on calcule moins d'expressions), le temps d'exécution du programme compilé en mode optimisé est plus long de 7 % que celui du programme compilé sans optimisation. Ce comportement défie le sens commun ! Nous avons tenté de trouver l'explication de ce phénomène en instrumentant les exécutions. Notre incompréhension s'en est trouvée accrue car une exécution de la version sans optimisation effectue 306 066 504 cycles alors que la version avec optimisation n'en effectue que 283 309 938. Soit une différence de 7 % en moins pour la version optimisée. Nous sommes totalement incapables d'expliquer pourquoi ce nombre de cycles inférieur de 7 % conduit à une exécution supérieure de 7 %.

En moyenne, le cumul de la CSE et de la β-réduction apporte donc 20 % de réduction sur la taille des fichiers C et objets produits, 27 % de diminution de temps de compilation globale (jusqu'à obtention du fichier objet) et 5 % en temps d'exécution.

9.7 La compilation des constantes (`Cnst`)

Une réflexion superficielle laisse penser que la compilation des constantes est une tâche facile du compilateur. Puisque cette compilation n'a que peu d'incidence sur les performances des exécutables pourquoi ne pas choisir la solution immédiate ? C'est-à-dire pourquoi ne pas produire puis compiler le code qui construira les constantes à l'initialisation du programme ? Cette solution est parfaitement correcte mais elle présente un inconvénient qui peut devenir un problème aigu : le code compilé pour construire les constantes est souvent énorme. Étudions la compilation d'une liste de taille n. Il y a principalement deux types de code que l'on peut produire pour créer cette liste :

- Imbriquer n appels à la fonction de construction des paires (`cons`).
- Utiliser deux variables auxiliaires, une pointant sur la tête de la liste et une autre sur la queue, la création de la liste est alors une séquence de mutation de la queue de la liste.

Imaginons maintenant que la liste soit grande (n pouvant être de l'ordre de 100, 1000, 10000, ...). Ce cas est parfaitement envisageable si la liste est produite par un autre programme (par exemple un générateur d'analyseurs syntaxiques dont les tables seraient implantées en utilisant des listes). Le résultat de la compilation de la constante est donc une suite (éventuellement imbriquée) de n appels de fonctions. La taille de ce code est donc proportionnelle à n, grandissant avec lui. Si n est grand, l'arbre de syntaxe devient immense. La compilation peut donc devenir extrêmement longue. L'article [HFA+94] étudie le comportement de différents compilateurs sur un programme (Pseudoknot de M. Feeley) manipulant des nombres flottants. Ce programme comprend plusieurs tableaux de nombres qui contiennent chacun environ 150 éléments. Bien que 150 soit un nombre petit (à l'échelle informatique) on peut déjà mesurer sur ce programme l'incidence d'une mauvaise compilation des constantes. Comme le montre l'article, certains compilateurs mettent plusieurs heures à compiler Pseudoknot ! Ce n'est pas acceptable. Bigloo, pour la compilation de Pseudoknot a un comportement qui peut sembler paradoxal. Il existe une version C de Pseudoknot. Elle utilise, comme la version Scheme, des tableaux de nombres flottants. Le temps de compilation du programme Scheme par Bigloo est plus de deux fois inférieur à celui de la compilation du programme C par gcc. Ce phénomène étrange trouve son explication uniquement dans la compilation des constantes : celle de Bigloo est beaucoup plus performante que celle de gcc.

9.7.1 Compiler les constantes sans production de code

La compilation canonique qui produit du code qui construit les constantes lors de l'initialisation n'est pas satisfaisante. Comme nous compilons vers C, il existe à notre avis deux méthodes pour la remplacer :

- Produire des déclarations C statiques. Cette solution est présentée dans [Que94]. En plus d'être esthétique elle a plusieurs propriétés intéressantes :
  - Toutes les constantes Scheme seront compilées en des constantes C. Un accès à une constante Scheme sera compilé en un accès à une constante C. Ceci s'inscrit parfaitement dans l'optique d'une projection naturelle comme celle qui est développée dans les sections 2.6.2 et 10.3.
  - Comme presque toujours avec la projection naturelle, les performances du code produit seront très bonnes (meilleures que pour les autres techniques que nous mentionnons) car le compilateur C connaîtra dès la compilation les adresses où se situeront les constantes. Ainsi les accès aux champs des structures ne nécessiteront pas d'indirections.
  - Les temps de compilations seront plus courts que la méthode produisant le code chargé d'allouer les constantes à l'initialisation car l'arbre de syntaxe abstraite du programme ne sera pas encombré de code énorme.

  Malheureusement, cette technique présente un inconvénient majeur : elle ne permet qu'une compilation très difficile des symboles. Les symboles doivent satisfaire le prédicat `eq?`. Donc deux symboles identiques utilisés dans deux modules différents doivent être placés à la même adresse. Ce n'est pas possible si un symbole est compilé en une structure C statique. Cet inconvénient nous a incité à utiliser un autre schéma de compilation.
- Écrire les constantes dans une chaîne de caractères (ou éventuellement dans un fichier annexe) qui est lue à l'initialisation. Les initialisations des constantes sont alors toutes réalisées par le lecteur standard. Les constantes sont allouées dans le tas de Bigloo et non pas dans celui de C comme avec la solution précédente.

C'est cette dernière solution que Bigloo utilise.

9.7.2 La compilation des constantes de Bigloo

Commençons par illustrer la compilation des constantes par un exemple :

```
(print '(1 2 3 4))
(print 'toto)
(print "toto\"toto")
(print 3.435)
(print '#(1 2 (2 3)))
```

Comme nous l'avons déjà mentionné, toutes les constantes sont regroupées dans une seule chaîne qui est ici :

```
"#(1 2 (2 3)) #\"toto\\\"toto\" (1 2 3 4) FOO".
```

Pendant la phase d'initialisation, les constantes sont placées dans un vecteur : le tableau des constantes. Chaque occurrence d'une constante est alors compilée en C par un accès à ce tableau. Nous avons pris soin d'allouer le tableau des constantes dans la zone statique C et non pas dans le tas de Bigloo car ainsi, cette solution a les mêmes performances que celle qui place les constantes dans des variables globales. En revanche cette solution est moins efficace que celle qui projette directement les constantes Scheme sur des constantes C car un accès à un champ des constantes coûte une indirection.

9.7.3 L'auto-amorçage

Pour initialiser un module, Bigloo lit donc les constantes dans une chaîne de caractères. La lecture peut être effectuée par le lecteur standard car la chaîne de caractère est une chaîne Bigloo. Se pose alors le problème suivant : comment le module définissant le lecteur peut être initialisé alors que pour ce faire, il a besoin du

lecteur (donc d'être déjà initialisé)? Ce problème n'a qu'une solution : le lecteur ne doit pas être initialisé de la même façon. Les constantes doivent être compilées autrement lorsqu'on compile ce module. En fait c'est toute la bibliothèque d'exécution de Bigloo qui est compilée dans un mode où l'initialisation des constantes n'utilise pas le lecteur. Les chaînes de caractères sont placées dans la zone statique, les autres constantes sont compilées par production de code. Pour la bibliothèque le choix de produire du code est raisonnable car comme nous maîtrisons son code, nous évitons d'y placer des grosses constantes.

9.8 Conclusion

Les deux optimisations (CSE et β-réduction) sont classiques et habituellement réalisées par les compilateurs optimisants. En particulier, les compilateurs C les appliquent. Toutefois, les inclure dans Bigloo présente des intérêts. Comme nous l'avons dit, les temps de compilation globaux, incluant les optimisations sont plus courts (de 27 % sur nos exemples) et les temps d'exécution sont diminués (de 5 % sur nos exemples). Ceci signifie que les optimisations réalisées par Bigloo sur son arbre de syntaxe abstraite ne sont pas réalisées par le compilateur qui a en charge la compilation des résultats de compilation Scheme. Ceci s'explique simplement : pour réaliser ses optimisations Bigloo utilise des informations de pureté sur les fonctions de la bibliothèque (par exemple que la fonction d'addition générique `+` ne fait pas d'effet de bord et retourne une valeur qui ne peut pas être modifiée). Ces informations donnent un cadre d'application plus large aux optimisations précitées. Naturellement le compilateur C ne dispose pas de ces informations. Il n'est donc pas capable de réduire les expressions que Bigloo parvient à éliminer en mode optimisant.

Par ailleurs, puisque les optimisations de ce chapitre ont plus d'effet sur les temps de compilation que sur les performances des programmes produits, nous avons exposé ici notre compilation des constantes. Nous utilisons une technique qui permet des temps de compilation beaucoup plus courts que les compilateurs qui produisent le code créant les constantes lors de la phase d'initialisation du programme exécutable.

Chapitre 10

La compilation de ML : Camloo

Ce chapitre contient la description de l'extension de Bigloo qui lui permet de compiler des programmes ML. Pour cela nous avons détourné le compilateur Caml-light et développé une petite passerelle entre l'arbre de syntaxe abstraite de ce compilateur et Scheme. L'objectif principal de ce travail est de valider les techniques de compilation utilisées dans Bigloo. Après un développement très court (moins de 4 mois et environ 3000 lignes de code) Bigloo est devenu capable de compiler du Scheme et du Caml. Comme nous allons le montrer dans ce chapitre, il est même devenu un *très bon* compilateur ML, comparable aux meilleurs compilateurs totalement dédiés à ce langage. Ce chapitre est structuré de la façon suivante: *(i)* En premier lieu nous allons décrire la technologie employée pour compiler le langage Caml. *(ii)* Nous donnerons ensuite l'architecture complète de la compilation de ML. Nous montrerons comment sont réparties les différentes tâches de la compilation. *(iii)* Nous détaillerons ensuite la passerelle entre l'arbre de syntaxe abstraite du compilateur Caml-light et Scheme. Nous introduirons le concept de *projection naturelle*. *(iv)* Enfin, nous comparerons notre compilation de ML avec d'autres compilateurs.

Le travail présenté dans ce chapitre à été réalise en collaboration avec Pierre Weis, chargé de recherches à l'Inria-Rocquencourt. Il a donné lieu à la publication [SW94].

10.1 La technologie mise en œuvre pour le compilateur Caml

Pour atteindre la compatibilité totale que nous nous sommes fixée comme objectif secondaire, nous avons opté pour un schéma simple : re-diriger un compilateur Caml déjà existant sur Bigloo. Notre nouveau compilateur Caml-light partagera l'ensemble de la partie haute du compilateur de codes-octet Caml-light. Ceci nous assure évidemment une compatibilité totale au niveau du code source. Nous allons montrer que charger Bigloo de la partie basse de la compilation de Caml-light permet d'obtenir un compilateur Caml performant. La figure 10.1 expose schématiquement l'architecture du compilateur Caml.

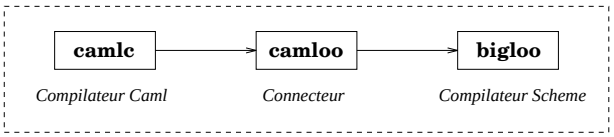


FIG. 10.1 - L'architecture globale du compilateur Caml

En résumé, nous avons obtenu notre nouveau compilateur en ajoutant à Bigloo, au début de la compilation une passe issue du compilateur Caml-light. Nous avons simplement connecté les deux compilateurs. La sortie du premier servant d'entrée au second dans l'optique des *pipe* d'Unix. Si nous avons été obligés d'adapter légèrement le compilateur Caml, en revanche Bigloo n'a nécessité quasiment aucun changement. Le principal effort de ce travail a porté sur l'implantation du connecteur reliant les deux compilateurs.

Ainsi donc, le compilateur Caml-light est stoppé juste avant qu'il n'émette les codes-octet. L'émission est remplacée par une sauvegarde (sous forme d'un texte) de son arbre de syntaxe abstraite.

Le connecteur se contente de projeter cet arbre de syntaxe abstraite sur les constructions du langage Scheme (nous nommons ce traducteur Camloo).

L'implantation totale du nouveau compilateur a requis 3000 lignes de code source (400 de Caml et 2600 lignes de Scheme).

10.2 L'architecture globale

Dans cette section, nous montrons comment compiler des modules Caml-light, c'est-à-dire comment compiler des *implémentations* et des *interfaces*.

10.2.1 La compilation des implémentations

Pour la compilation d'une implémentation Caml (un fichier `.ml`), le compilateur Caml-light effectue son travail habituel. Il fait l'analyse lexicale et syntaxique, il effectue la passe d'inférence de types. Il se charge de la résolution des noms, de la compilation du filtrage et enfin, il produit son arbre de syntaxe abstraite. Cet arbre (que nous appellerons λ-arbre) est écrit dans un fichier portant le suffixe `.lam`.

Le connecteur Camloo se charge alors de traduire le fichier `.lam` en un module Scheme compilable par Bigloo.

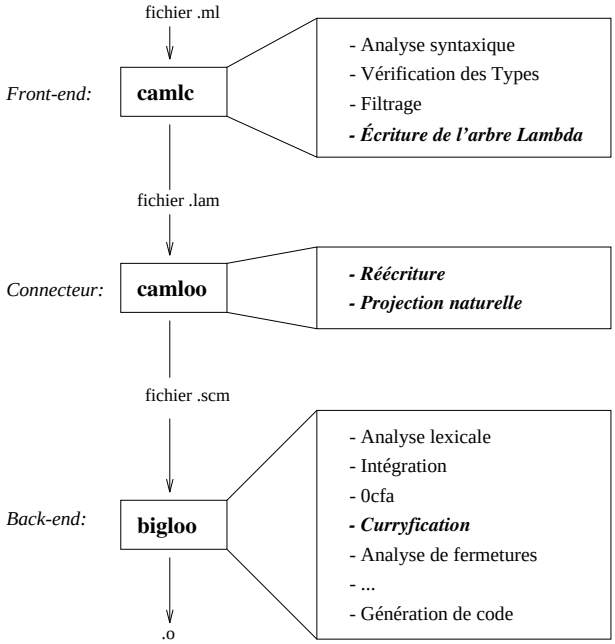


FIG. 10.2 - La compilation des implémentations Caml

Dans la figure 10.2, apparaissent en gras les passes ajoutées pour obtenir le nouveau compilateur Caml.

10.2.2 La compilation des interfaces

Pour la compilation des fichiers d'interface (fichiers `.mli`), le compilateur Caml-light produit traditionnellement un fichier `.zi` mais en plus, nous lui faisons générer un fichier `.sci` qui contient des clauses d'importations pour le langage de module de Bigloo.

10.2.3 Compiler indifféremment Scheme ou Caml-light

Bigloo a un mécanisme intégré qui permet d'ajouter des passes sur la partie haute de la compilation [Ser94a]. Comme l'invocation de ces pré-compilateurs peut être associée à des suffixes, Caml-light et Camloo

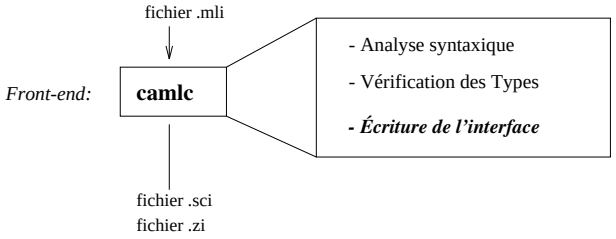


FIG. 10.3 - La compilation des interfaces Caml

sont invoqués quand Bigloo compile des fichiers `.ml` ou `.mli`. Vue depuis Bigloo, la passe de compilation Caml est seulement une passe supplémentaire avant la compilation du code Scheme (dans l'esprit d'une passe de macro expansion). Ainsi, pour compiler un module Caml-light, il suffit d'invoquer normalement Bigloo en lui donnant comme source le fichier Caml. C'est pourquoi le nouveau compilateur se nomme également Bigloo et Bigloo est ainsi devenu un compilateur Scheme *et* Caml-light.

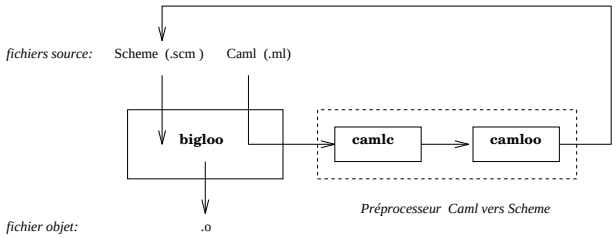


FIG. 10.4 - Bigloo: un compilateur Scheme *et* Caml-light

10.3 Le connecteur Camloo

Comme nous l'avons déjà mentionné, les deux compilateurs (Caml-light et Bigloo) sont reliés au moyen de fichiers texte. La dernière passe de Caml-light est modifiée pour que ce compilateur produise le fichier contenant la représentation du λ-arbre. Ce fichier n'est pas directement compilable par Bigloo: un petit programme doit traduire cette représentation en un module Scheme. Cette traduction consiste en une projection des constructions et valeurs ML sur leurs équivalents Scheme. La correction des projections est facile à obtenir car les deux langages sont assez proches. En fait la difficulté est d'effectuer une projection efficace. Heureusement, pour ce travail, l'efficacité vient de la simplicité: nous n'avons qu'à projeter les constructions ML en leur *équivalents naturels* Scheme, en évitant les encodages superflus. Si nous n'adoptions pas cette démarche, l'efficacité serait compromise même si la sémantique de ML était préservée, car le compilateur de bas niveau ne pourrait pas optimiser ce programme encodé. Par exemple, il est important que les fonctions ML soient projetées sur les fonctions n-aires de Scheme et non pas sur des fonctions systématiquement unaires. On bénéficie ainsi des optimisations de Bigloo sur les appels contenant plusieurs arguments.

10.3.1 Un exemple de projection de ML sur Scheme

Étudions en premier lieu un exemple simple: le module `len.ml` implantant la fonction `list_length`. Ce module est :

```
let rec list_length = function
    [] -> 0
  | x :: l -> 1 + list_length l;;
```

Le λ-arbre obtenu grâce au compilateur Caml-light est le fichier `len.lam` suivant :

```
(lprim
 (pset_global (qualifiedident "len" "list_length"))
 ((lfunction
   (lswitch 2 (lvar 0)
    (((constrconstant
       (qualifiedident "builtin" "[]") 0 2)
       (lshared (lconst (scatom 0)) -1))
     ((constrregular
       (qualifiedident "builtin" "::") 1 2)
       (lshared
        (lprim paddint
          ((lconst (scatom 1))
          (lapply (lprim (pget_global (qualifiedident "len"  "list_length")) ())
                  ((lprim (pfield 1) ((lvar 0)))))))
       -1)))))))
```

Cette expression est difficilement lisible. Nous nous contenterons de dire qu'elle définit la variable globale `list_length` du module `len`, que le filtrage a été expansé (`lswitch`), que les noms des constructeurs ont été explicités (comme dans l'expression `"builtin" "::"`) et que les primitives sont qualifiées (`paddint` au lieu de `+`). Comme le lecteur peut le remarquer, Caml-light utilise la notation de De Bruijn [DB72]. Les noms des variables locales ont donc disparu au profit d'un index.
Pour ce λ-arbre, camloo produit le fichier Scheme `len.scm` qui ressemble fortement à ce qu'un programmeur Lisp aurait pu écrire :

```
(define list_length@len
   (lambda (x1)
      (if (null? x1)
          0
          (+fx 1 (list_length@len (cdr x1))))))
```

Cet exemple illustre la projection directe des fonctions, listes et entiers ML sur leur correspondants Scheme. En plus, il faut remarquer que le fichier Scheme n'utilise pas l'arithmétique générale mais l'arithmétique entière (traduction évidente de la primitive Caml-light `paddint`).

10.3.2 La projection générale

Nous décrivons dans cette section le schéma général de la projection ML vers Scheme. Nous verrons ultérieurement quelques exceptions utilisées pour projeter encore plus directement des structures ML (par exemple les listes) et des constructions syntaxiques (par exemple les fonctions currifiées).

La projection des valeurs

- Les valeurs fonctionnelles : il n'y a pas de difficulté à projeter les fonctions Caml sur les procédures Scheme car les deux langages sont basés sur le λ-calcul avec des fonctionnalités d'ordre supérieur.

- Les objets basiques Caml comme les entiers, les chaînes de caractères, les caractères, les tableaux et les nombres flottants, sont projetés sur leurs équivalents Scheme. Toutefois, un petit problème apparaît pour les caractères car le compilateur Caml-light les représente comme des entiers. Quelques primitives ont donc été changées pour tenir compte de la différence entre entiers et caractères dans Bigloo. Que les objets basiques Caml soit projetés sur leurs équivalents Scheme présente un avantage immense : toutes les fonctions Scheme qui manipulent ces valeurs, pourront être utilisées dans les programmes CamlLa première exploitation de cette possibilité se trouve dans l'implantation de la bibliothèque d'exécution de Caml où toutes les fonctions qui existaient déjà pour la bibliothèque de Scheme n'ont pas eu à être réécrites pour ML (fonctions arithmétiques, fonctions calculant des longueurs `vector-length`, manipulant des chaînes de caractères `substring`, ...).

- Les constructeurs constants ML sont projetés sur les constructeurs constants de Bigloo (par exemple, `true` et `false` sont projetés sur les booléens Scheme `#t` et `#f`.

- Les valeurs obtenues au moyen de constructeurs non constants sont projetées sur les vecteurs Scheme. Par exemple, l'application d'un constructeur `C` aux arguments `x, y, z` est traduite en un vecteur Bigloo `#(x y z)`. Le constructeur `C` est inscrit dans l'étiquette attachée aux vecteurs (8 bits des vecteurs y sont consacrés).

- Les exceptions : les valeurs obtenues par des constructeurs extensibles sont également projetées sur des vecteurs Scheme.

La projection des exceptions nécessite quelques explications supplémentaires. Puisque les exceptions sont génératives, chacune doit posséder sa propre étiquette unique. Celle-ci est attribuée par Caml-light lors de l'édition de liens. Heureusement, Scheme possède des symboles qui sont une association entre des noms et des objets uniques. Ainsi, les exceptions ML donnent lieu à la création de symboles Lisp qui jouent le rôle d'identificateur d'exception. Le nom des symboles est composé du nom de l'exception d'un compteur (comptant le nombre de redéfinitions de l'exception) et du nom du module définissant l'exception.

Ainsi, l'exception `Out_of_memory` du module de la bibliothèque `exc` est représentée par le symbole `'Out_of_memory1@exc`. Pour les exceptions non constantes, le symbole est placé dans le premier champ du vecteur représentant l'exception. Ainsi le résultat de la projection de `Failure "hd"` est le vecteur `#(Failure1@exc "hd")`.

La projection de la construction `try`...`raise`

Les autres constructions ML ne posent pas de problèmes particuliers (principalement des boucles et des conditionnelles); la dernière difficulté est posée par les gestionnaires (*handler*) d'exception. Nous aurions pu adopter une projection triviale pour `try`...`with`... and `raise` utilisant la fonction de la bibliothèque Scheme `call/cc`. Il aurait alors suffi de définir une macro `try-with` (la variable `raise` étant liée au pilote courant)[1] :

```
(define raise (lambda (x) (error "uncaught exception" x)))

(define-macro (try-with computation match-handler)
   `((call/cc
      (lambda (return)
         (let ((previous-raise raise))
            (set! raise (lambda (exc)
                          (set! raise previous-raise)
                          (return (lambda () (,match-handler exc)))))
            (let ((result ,computation))
               (set! raise previous-raise)
               (lambda () result)))))))
```

Malheureusement, comme nous l'avons déjà expliqué dans la section 2.4, cette fonction est très difficile à implanter dans un schéma d'exécution utilisant une pile. Produisant du code C utilisant la pile C, Bigloo n'échappe pas à la règle ! Comme le code Caml utilise abondamment le mécanisme des exceptions, nous avons besoin d'une implantation plus efficace. En C, nous utilisons les facilités offertes par `setjmp` et `longjmp` (nous savons depuis [Rob89] qu'il est facile d'ajouter à C, sans modification du compilateur, un mécanisme d'exceptions très proche de celui de ML en utilisant `setjmp`, `longjmp`), mais là aussi, ces fonctions ne sont pas destinées à être utilisées aussi souvent que dans le code C produit par une compilation Caml. Il en résulte de médiocres performances pour des programmes utilisant intensivement `try` et `raise`. C'est pourquoi pour certains portages, notre compilateur utilise directement un petit code écrit en langage d'assemblage (sur Sparc il fait 19 lignes).

Le filtrage

Camloo ne se charge pas de traduire le filtrage issu de Caml. Le filtrage est expansé par le compilateur Caml-light en des imbrications de constructions conditionnelles. Camloo se contente de projeter ces constructions sur leurs équivalents Scheme.

[1] Cette implantation est dûe à Christian Queinnec.

Nous avons renoncé à utiliser le filtrage de Bigloo pour garantir la compatibilité. L'intérêt est aussi que nous bénéficions gratuitement des extensions de Caml-light comme les *streams*.

10.3.3 Les modules

Les modules de Caml-light sont projetés par Camloo sur les modules de Bigloo. Les modules de Bigloo sont auto-contenus : l'interface et l'implémentation sont dans le même fichier. Quand notre version du compilateur Caml-light compile une interface, il produit deux fichiers : un traditionnel `.zi` et un module Bigloo sans implémentation (fichier `.sci`) qui contient uniquement d'éventuelles définitions de primitives trouvées dans le fichier d'interface. Quand Caml-light compile un fichier d'implémentation, Camloo produit un module Bigloo contenant le corps de l'implémentation et une en-tête de module Bigloo contenant les prototypes des toutes les liaisons exportées.

Cette projection des modules de Caml-light sur ceux de Bigloo oblige à ne compiler que des modules ayant une implémentation. Ceci est une restriction par rapport au langage de module de Caml-light, mais chaque programme *exécutable* est compilable par Bigloo. Ceci ne semble donc pas être un grave problème.

10.3.4 Les optimisations de la projection

La projection des fonctions currifiées

Le schéma général de projection des fonctions Caml décrit précédemment est correct mais inefficace car les fonctions currifiées du langage ML donneront lieu à des créations de multiples invocations de fonctions Scheme avec créations de fermetures intermédiaires.

En effet, ML ne possède pas de fonction n-aires. Ceci est généralement contourné en utilisant des fonctions currifiées ou éventuellement des fonctions qui prennent des tuples en argument. Puisque l'encodage classique des fonctions n-aires Caml est d'utiliser des fonctions currifiées, Camloo doit les projeter sur les fonctions Scheme n-aires. Il faudrait également que les fonctions utilisant des tuples soient projetées, elles aussi, sur des fonctions n-aires. Ceci reste à faire.

Comme cela est dit au chapitre 7, la *0cfa* qu'utilise Bigloo optimise déjà les currifications. Cette analyse n'est toutefois pas suffisamment puissante car elle est locale aux modules. Il nous a donc fallu concevoir une optimisation globale qui permette de projeter les fonctions ayant n arguments currifiés sur les fonctions de Scheme ayant n arguments, spécialement pour les fonctions globales.

Cette nouvelle optimisation requiert la collaboration de Camloo *et* de Bigloo. Camloo produit pour chaque fonction globale (exportée ou non) deux points d'entrée : le premier est destiné aux applications partielles (les applications qui construisent des fermetures), le second est appelé directement quand la fonction est invoquée avec tous ses arguments currifiés (application totale).

Donnons un exemple :

```
let rec map f = function
    [] -> []
  | x :: l -> f x :: map f l;;
map succ [1; 2];;
```

La procédure `map` donne lieu à la création de deux points d'entrée, `map` et `2-map`, `2-map` est invoquée directement sans création de fermeture :

```
(define map (lambda (x1) (lambda (x2) (2-map x1 x2))))

(define 2-map (lambda (x1 x2)
                (if (null? x2)
                    '()
                    (let ((obj2 (2-map x1 (cdr x2)))
                          (obj1 (x1 (car x2))))
                       (cons obj1 obj2)))))

(2-map succ (let ((obj2 (cons 2 '()))) (cons 1 obj2)))
```

En plus, cette projection des fonctions currifiées ML sur les fonctions n-aires n'est pas restreinte aux fonctions locales à un module : notre compilateur optimise totalement les fonctions currifiées exportées. Quand un

module exporte une fonction currifiée, Camloo génère une clause d'exportation pour les deux points d'entrée (`map` et `2-map`) dans l'exemple précédent). Ainsi, quand un module importe une fonction currifiée, Bigloo importe les deux points d'entrée, il peut donc retrouver le point d'entrée direct quand c'est nécessaire.

La projection des listes

Nous avons accordé une attention particulière à la projection des listes ML en Scheme. La principale raison est la compatibilité entre le code Scheme et le code Caml. De plus, les paires sont manipulées efficacement car elles ont une étiquette spéciale qui permet de les implanter en n'utilisant que deux mots.

Cette optimisation est très simple à réaliser : pour allouer les listes Caml comme des listes Scheme, il suffit de détecter les applications des constructeurs `::@builtin` et `[]@builtin`. Les accès aux listes n'ont lieu que dans la partie *expression* d'un filtrage dont le filtre a pu prouver une manipulation de listes. Nous pouvons donc remplacer les fonctions d'accès génériques aux structures par les fonctions `car` et `cdr` (voir l'exemple de `list_length`).

La projection des références

La projection naïve des références qui utilise un schéma classique de la projection des constructeurs non constants (et emploie donc des vecteurs Scheme) est inefficace : `ref 0` devient `#(0)`. L'inconvénient est que chaque consultation de la référence occasionne une indirection dans un vecteur et chaque écriture occasionne une sauvegarde dans le vecteur. Ceci est incontournable quand les références sont utilisées comme valeur de premier ordre (passées à des fonctions ou mises dans des structures de données) mais en pratique, cela arrive rarement. Ainsi la projection qui utilise les variables impératives Scheme s'applique dans le cas général. Elle utilise les variables et l'affectation (`set!`) Scheme. Avec cette projection il n'y a plus d'allocation mémoire, la référence peut même rester dans un registre.

Examinons un exemple qui est une boucle utilisant une référence comme compteur de contrôle.

```
let x = ref 10 in
  while !x > 0 do
    print_int !x;
    x := !x + 1
  done;;
```

La projection de ML vers Scheme transforme la construction `while` en une fonction récursive et la variable ML `x` est traduite en une variable Scheme.

```
(let ((x1 10))
  (letrec ((loop (lambda ()
                   (if (>fx x1 0)
                       (begin
                          (print_int@io x1)
                          (set! x1 (+fx x1 1))
                          (loop))
                       '()))))
     (loop)))
```

La variable `x` est ensuite traduite en une variable C :

```
{
   obj x1;

   x1 = 10;

loop:

   if( GT( x1, 0 ) )
   {
      print_int__io( x1 );
      x1 = ADD( x1, 1 );
      goto loop;
   }
   else
      BNIL;
}
```

Cette optimisation est réalisée par une analyse purement syntaxique du λ-arbre : les variables liées à des références sont projetées sur des variables Scheme sauf si elles sont utilisées comme valeur de premier ordre (la variable apparaît sans déréférenciation : `x` au lieu de `!x`).

Cette transformation de programme est correcte et facile à réaliser dans Camloo, car les références Caml-light qui ne sont pas utilisées comme objet de premier ordre ont la même sémantique que les variables Scheme. En particulier, Bigloo peut placer dans des zones du tas des variables qui sont partagées par plusieurs fermetures.

10.4 Les modifications apportées aux deux compilateurs

10.4.1 Les modifications de Caml-light

Les modifications que nous avons apportées au compilateur Caml-light sont mineures. Exceptée la passe de sauvegarde du λ-tree (qui n'est qu'un imprimeur) les rares modifications qui ont été apportées, l'ont été pour que le code produit soit plus performant. Par exemple, le compilateur Caml-light « oublie » l'arité et les noms des constructeurs après la compilation du filtrage, car c'est inutile pour le reste de sa compilation. Il nous a fallu les conserver pour aider Bigloo à produire du meilleur code.

10.4.2 Les modifications de Bigloo

Les vérifications de type dynamiques

Du point de vue de l'efficacité, un problème important subsiste: Scheme est un langage dont la vérification des types se fait lors des exécutions. Ce n'est dans les faits pas si important car plusieurs analyses (la *0cfa* et la CSE) en suppriment une grande majorité (environ 75 %). Néanmoins, nous voulons tirer partie du typage statique de ML pour ne plus faire aucun test lors des exécutions. Ceci n'a pas été très compliqué car comme presque tous les compilateurs Lisp, Bigloo possède un drapeau de compilation qui permet de supprimer l'émission des tests de type. Ainsi, quand nous compilons du code Caml, nous pouvons utiliser de façon sûre l'option `unsafe` de Bigloo.

La bibliothèque d'exécution

La plupart des exigences de ML pour la bibliothèque d'exécution telle que la gestion automatique de la mémoire (*GC*) ou l'allocation dynamique des structures (vecteurs, fermetures, chaînes, ...) sont satisfaites par la bibliothèque Scheme. Bigloo les prête donc gratuitement à ML puisque les valeurs ML sont projetées sur les valeurs Scheme.

Les besoins spécifiques de Caml-light ont été implantés en utilisant l'interface externe de Bigloo (voir le chapitre 4).

Pour résumer, excepté l'optimisation des fonctions currifiées, aucune autre modification n'a été faite à Bigloo pour parvenir à compiler efficacement le langage Caml. La version couramment distribuée de Bigloo compile indifféremment du Scheme *et* du Caml.

10.5 Comparaison avec d'autres travaux

D'autres compilateurs ont déjà été obtenus en ré-utilisant des systèmes existants. Nous comparons Bigloo avec trois autres systèmes disponibles : Caml V3.1, sml2c et Camlot.

Caml V3.1 Ce compilateur Caml [W*al.*91] est basé sur la bibliothèque d'exécution du système Le-Lisp [CDD+86]. Pour obtenir le compilateur, les réalisateurs ont dû écrire entièrement les parties hautes et le générateur de code. Grâce à la machine virtuelle LLM3, la portabilité à été obtenue sans effort. Si le code Caml compilé avait utilisé le compilateur Complice de Le-Lisp, cette approche aurait été très proche de la nôtre (excepté bien entendu que Bigloo produit du langage C et non pas du langage d'assemblage).

sml2c Ce compilateur est basé sur une réécriture du générateur de code du compilateur Sml/NJ [TAL91, AM87]. La représentation CPS des programmes est compilée vers une machine cible dont les instructions sont implantées en C. D'autre part, les compilateurs partagent la même bibliothèque d'exécution et utilise les mêmes optimisations de haut niveau.

À cause de la méthodologie employée par sml2c (génération de code C implantant les instructions d'une machine virtuelle) le code C produit est très différent de celui qu'un utilisateur écrirait « à la main ». Ce code C est donc difficilement optimisé par le compilateur C. À l'opposé, Bigloo fait de gros efforts pour produire du code C ressemblant à du code écrit à la main. L'idée est la même que pour la projection naturelle de ML vers Scheme et a les mêmes bonnes propriétés : la projection naturelle de Scheme vers C produit du code C que le compilateur peut très bien optimiser[2].

Le principal intérêt de l'approche de sml2c est de minimiser le code d'assemblage indispensable (en moyenne, chaque bibliothèque d'exécution requiert environ 500 lignes de langage d'assemblage). Toutefois, nous avons le même avantage de portabilité avec Bigloo (19 lignes de langage d'assemblage pour le portage sur Sparc et 0 sur les autres machines).

camlot L'approche de Régis Cridlig est proche de la nôtre [Cri92]. Il a modifié le compilateur Caml-light et a écrit un nouveau compilateur inférieur (*back-end*) et une nouvelle bibliothèque d'exécution. Le code C produit par Camlot est lisible : comme Bigloo, Camlot réalise une transformation naturelle de ML vers C.

Les trois compilateurs que nous venons de décrire partagent la même idée, celle de la réutilisation. Certains réutilisent les parties basses d'un compilateur existant, d'autres les parties hautes. En général, l'effort fourni par ces compilateurs correspond « en gros » à l'implantation d'un « demi » compilateur complet.

10.6 Les mesures de performance de la compilation ML

Cette section contient des mesures de performances de Bigloo compilant du ML et celles d'autres compilateurs ML. La façon dont avons obtenu les chiffres est conforme aux principes que nous avons énoncés dans le chapitre 1. Nous avons toujours mesuré les performances de programmes autonomes (pour Sml/NJ nous donc utilisé `exportFn`).

Les fichiers C produits par Bigloo, Camlot et sml2c ont tous été compilés par le compilateur gcc utilisé avec l'option `-O2`. La figure 10.5 décrit les différentes versions utilisées et les modes de compilation. Les compilateurs Bigloo, Camlot et Caml-light possèdent un mode de compilation où les tests de bornes des indices des tableaux sont omis ; nous l'avons utilisé.

La figure 10.6 décrit brièvement les programmes ML que nous avons employés. Les tailles sont données en nombre de lignes. Ces programmes ne sont pas présentés dans le chapitre 1 car ce chapitre ne présente que les programmes Scheme qui sont utilisés tout au long de ce document.

[2] Permettant également un emploi raisonnable des débogueurs symboliques.

	Compilateur				
Banc d'essai	**Camlc**	**Camlot**	**Bigloo**	**Sml/NJ**	**Sml2c**
sort	260.0 s	15.8 s	*9.0* s	27.9 s	47.7 s
life	44.4 s	4.5 s	3.2 s	*2.8* s	5.4 s
takc	30.8 s	*1.4* s	*1.4* s	11.6 s	33.4 s
taku	36.2 s	6.7 s	7.1 s	*4.2* s	8.6 s
boyer	12.6 s	8.9 s	*5.9* s	12.0 s	8.6 s
soli	28.6 s	1.3 s	*1.2* s	4.4 s	6.8 s
kb	73.6 s	80.6 s	46.7 s	*22.1* s	37.9 s
queens	95.0 s	_	*13.3* s	31.6 s	45.3 s
ffib	39.2 s	*2.2* s	*2.2* s	5.2 s	9.4 s
fft	163.0 s	28.4 s	*22.8* s	226.0 s	_

Tableau 10.1: Les temps d'exécution sur machines Sparc

	Compilateur				
Banc d'essai	**Camlc**	**Camlot**	**Bigloo**	**Sml/NJ**	**Sml2c**
sort	414.8 s	18.0 s	*10.2* s	25.0 s	51.6 s
life	71.1 s	4.2 s	3.7 s	*2.3* s	4.9 s
takc	49.6 s	*2.5* s	*2.5* s	11.3 s	28.8 s
taku	56.7 s	5.9 s	6.3 s	*3.5* s	7.4 s
boyer	17.9 s	*2.4* s	2.7 s	3.6 s	7.1 s
soli	41.8 s	1.5 s	*1.4* s	3.5 s	7.1 s
kb	105.3 s	32.3 s	39.9 s	*10.6* s	30.4 s
queens	143.5 s	_	*8.3* s	13.0 s	42.1 s
ffib	63.8 s	*2.8* s	*2.8* s	4.4 s	8.3 s
fft	187.1 s	28.3 s	*22.8* s	52.4 s	105.8 s

Tableau 10.2: Les temps d'exécution sur machines Mips

Les temps d'exécution des figures 10.1 et 10.2 montrent que Bigloo compile très efficacement les appels de fonctions, en particulier quand les fonctions sont currifiées (voir **takc** et **ffib**). D'autre part, l'optimisation des fonctions n-aires « décurrifiées » est réellement nécessaire (voir **taku**). Bigloo et Camlot sont plus lents que les autres compilateurs sur le programme **kb**. C'est probablement dû à leur *GC* qui n'utilise pas de générations (technique très efficace pour ce test).

Bigloo a de bonnes performances pour le programme **sort** ; c'est en partie dû à sa bonne compilation des références. Bigloo compile les boucles efficacement, qu'elles soient impératives (**sort** et **soli**) ou plus fonctionnelles (**queens** et **life**). Les exceptions sont efficacement implantées dans Bigloo (c'est aussi le cas de Camlot) puisque le programme **boyer** qui les utilise intensivement est bien compilé par ce compilateur.

Le compilateur Sml/NJ se comporte mieux sur machines Mips. Ceci est en accord avec l'article de A. Diwan, D. Tarditi & E. Moss [DTM94]. En revanche, il a des performances plutôt pauvres sur machines Sparc (moins bonnes performances que sml2c sur le programme **boyer**).

Les temps pour les compilateurs qui produisent du C se ressemblent sur machines Sparc et Mips. Bigloo et Camlot ont des performances assez proches, sensiblement meilleures que celles de sml2c.

La figure 10.7 donne pour chaque compilateur, la taille moyenne des fichiers objets obtenus.

Comme le lecteur peut le remarquer, alors que Bigloo implante deux jeux de primitives (un pour Scheme et un pour ML) les exécutables qu'il produit sont de tailles raisonnables. Caml-light produit des programmes extrêmement compacts mais ce ne sont pas des exécutables au sens d'Unix. Les exécutables du compilateur Sml/NJ sont gros sur les machines à base de processeurs Mips : la version expérimentale 1.03f n'est pas compilable avec l'option `-noshare` sur cette architecture.

compilateur	version	compilation
camlc	0.6	`-O fast`
camlot	0.6	`-O fast -f`
bigloo	v1.7-0.2	`-unsafe -O3`
sml/nj	1.03f	`-noshare` sur Sparc
sml2c		`-ffunction-integration -cf -O2` `-gcc -funsafe-arithmetic`

FIG. 10.5 - Les drapeaux de compilation

programs	taille	description	fonctionnalité testée
sort	79	L'algorithme du "quicksort".	Tableaux, boucles et références.
life	148	Le jeu de la vie.	Manipulation de listes et de chaînes.
takc	11	La fonction de Takeuchi.	Appels currifiés de fonctions.
taku	12	La fonction de Takeuchi.	Appels de fonctions decurrifiés.
boyer	1765	Démonstrateur de théorèmes.	Exceptions et traitements symboliques.
soli	110	Résolution du jeu du solitaire.	Tableaux et boucles.
kb	537	Complétion naïve de Knuth-Bendix.	Exceptions et traitements symboliques.
queens	71	Résolution du problème des reines.	Manipulation des listes.
ffib	13	Variante de la fonction de Fibonacci.	Utilisation de fermetures.
fft	176	Une transformée de Fourier.	Arithmétique flottante.

FIG. 10.6 - La description des programmes de tests

10.7 Conclusion et perspectives

L'objectif de se travail était de montrer que les techniques de compilation employées par Bigloo ne sont pas exclusivement applicables à la compilation de programmes Scheme. Nous avons montré qu'elles s'appliquent parfaitement à la compilation de ML. En effet, en très peu de temps (moins de 4 mois), nous avons étendu Bigloo de telle sorte qu'il puisse compiler le langage Caml-light. Notre compilateur est parfaitement conforme à la description de ce langage [WL93] puisqu'en plus de se compiler lui-même, il a compilé avec succès le système Coq d'aide à la démonstration automatique [DFHH93]. Le nouveau Bigloo est donc un compilateur Scheme *et* ML.

En plus de valider nos techniques de compilation (en les inscrivant dans un contexte plus large), ce travail ouvre de nouvelles perspectives. Il est possible avec Bigloo de totalement mélanger les codes Scheme et les codes ML. Un exécutable peut être composé d'une partie écrite en Scheme et d'une partie écrite en Caml. À notre sens, ces deux langages s'en trouvent rapprochés. En plus, comme nous l'avons déjà montré Bigloo s'interface largement avec C, il est envisageable d'écrire des programmes mélangeant ces trois langages.

Nous allons donner ici un exemple de programme mélangeant les trois langages. Trois fichiers seront compilés séparément (deux avec Bigloo et le dernier avec un compilateur C). L'édition de liens les utilisera

compilateur	sur Sparc	sur Mips
camlc	7 k	19 k
camlot	128 k	129 k
bigloo	288 k	232 k
sml/nj	335 k	3867 k
sml2c	387 k	447 k

FIG. 10.7 - Taille des exécutables

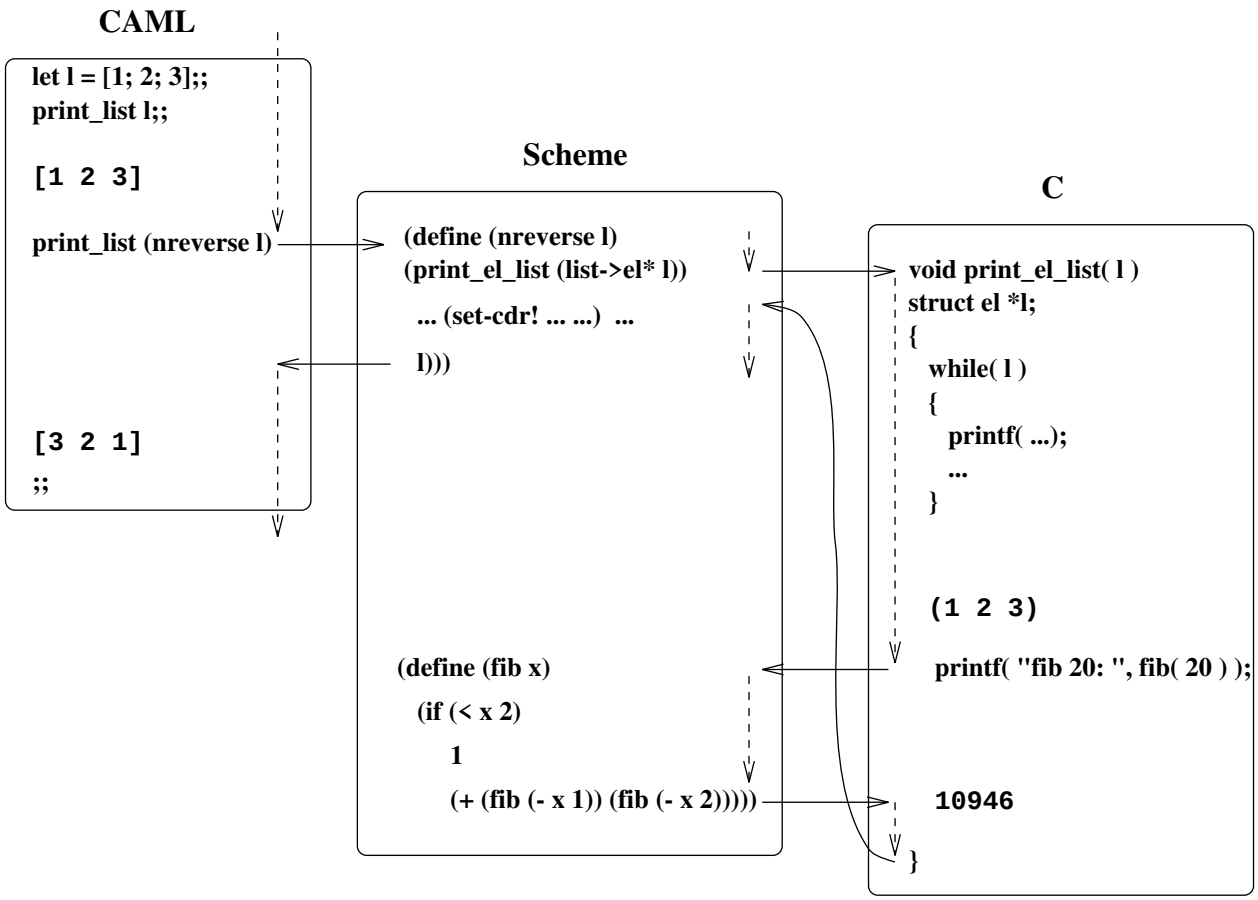


FIG. 10.8 - Une exécution mixant Scheme, ML et C

pour produire un exécutable unique.

- Le programme ML utilise Scheme (il invoque la fonction `nreverse` faisant un renversement physique de listes).
- Scheme utilise ML (la liste allouée par ML).
- Scheme utilise C (les structures `struct el` et la fonction `print_el_list`).
- C utilise Scheme (en invoquant la fonction `fib`).

Nous ne donnons pas le code de ces trois programmes, en revanche, nous montrons dans la figure 10.8 un schéma décrivant une exécution de ce programme composite.

La figure 10.9 résume de façon humoristique les possibilités offertes par Bigloo en matière de croisement de langages.

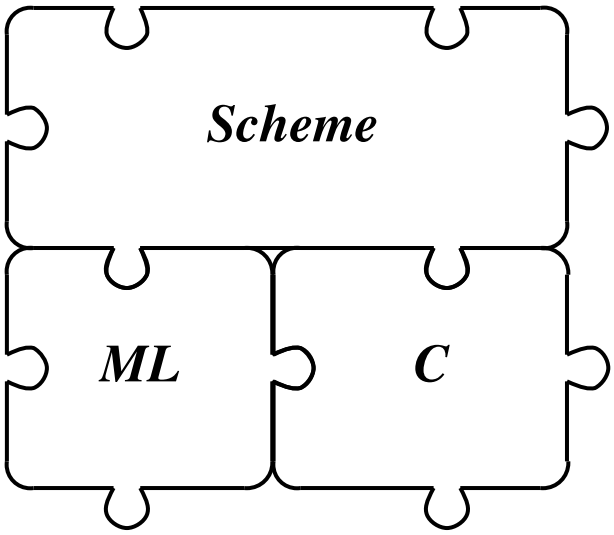


FIG. 10.9 - Le puzzle

Chapitre 11

Les mesures de performances

Voici venu le temps des mesures de performances. Nous allons comparer 6 compilateurs de dialectes de Lisp. En plus de Bigloo, ces compilateurs sont: Scheme-to-C, Orbit, Complice, Akcl et Cmucl.

- Scheme-to-C [Bar89b] a beaucoup de points communs avec Bigloo:
  - Son langage cible est C.
  - Il utilise un schéma de compilation l'amenant à produire du C orthodoxe.
  - Il est conçu pour produire des petites applications autonomes utilisant un gestionnaire mémoire conservatif.
  - Il utilise un mécanisme d'intégration proche de celui de Bigloo pour compiler les fonctions locales.

  Puisque ce compilateur est antérieur, il nous a servi de point de comparaison pendant toute la conception et le développement de Bigloo[1].

- Orbit est un compilateur optimisant [KKR+86] qui produit du code natif. Il a longtemps servi de référence en matière de compilation de Scheme. Nous avons rencontré de nombreux problèmes en utilisant le portage sur Mips. Nous ne présentons donc pas toujours de temps pour Orbit sur les machines équipées de ce processeur.

- Complice est le compilateur du système Le-Lisp [CDD+86]. Il produit du code LLM3 qui est expansé vers la machine cible.

- Austin Kyoto Common Lisp (Akcl) est une version améliorée de Kyoto Common Lisp. Il s'agit d'un compilateur Common Lisp qui produit du C. Akcl utilise un *GC* conservatif comme Bigloo mais il produit du C hétérodoxe.

- Carnegie Mellon University Common Lisp (Cmucl) est un compilateur de Common Lisp produisant du code natif. Il n'existe pas de portage de ce compilateur sur machines équipées de processeur Mips. Nous ne l'avons donc testé que sur Sparc. Ce compilateur jouit actuellement d'une bonne réputation [Mac92b] car il est l'un des compilateurs Common Lisp qui produit le code le plus performant.

Nous aurions souhaité comparer plus de compilateurs Scheme mais malheureusement, peu fonctionnent sur Sparc et Mips. Gambit, le compilateur de M. Feeley ne tourne que sur machine équipée de processeurs Motorola 68k, Liar, le compilateur de G. Rozas fonctionne sur Motorola 68k, sur Mips mais pas sur Sparc. Nous aurions aimé tester le compilateur Chez Scheme, mais n'étant pas dans le domaine public nous n'avons pu en obtenir une version. Nous nous sommes adressés à ses auteurs pour leur demander les temps d'exécution de Chez Scheme sur les bancs d'essai de Gabriel, mais ils n'ont pas souhaité nous les fournir.

[1] Nous profitons de ce moment pour adresser nos sincères remerciements à J. Bartlett pour son compilateur.

11.1 Les bancs d'essai de Gabriel

Bien qu'ayant émis dans le chapitre 1 des réticences à utiliser les bancs d'essai de Gabriel [Gab85] nous allons commencer par étudier les comportements de différents compilateurs sur ces programmes. La raison pour laquelle nous nous sommes résigné à les utiliser est que ces tests constituent une référence pour les mesures d'exécution des programmes Lisp. Ils représentent une sorte de passage obligé. Ces programmes existent dans presque tous les dialectes et permettent donc de comparer des systèmes différents. Les performances de Bigloo sur ces programmes vont nous permettre de le situer dans le panorama général des compilateurs Lisp. La figure 11.1 contient une description sommaire des compilateurs que nous utilisons pour cette première série de test.

compilateur	version	dialecte	compilation
Bigloo	1.7	Scheme	`-O4 -unsafe`
S2c	15mar93jfb	Scheme	`-O -On -Ob -Ot -Og`
Orbit	T3.1	Scheme	
Complice	15.24	Le-Lisp	`(defvar #:complice:parano-flag ())`
Akcl	1.605	Common Lisp	`(speed 3) (safety 0) (space 0)`
Cmucl	Python 1.0 (17e)	Common Lisp	`(speed 3) (safety 0) (space 0)`

FIG. 11.1 - Les versions des compilateurs

Nous avons utilisé trois compilateurs Scheme, deux compilateurs Common Lisp et le compilateur le plus performant du système Le-Lisp v15. Pour tous les programmes et tous les compilateurs nous avons utilisé l'arithmétique des petits entiers. En Scheme et Le-Lisp, nous avons employé des fonctions spécialisées. En Common Lisp, nous avons inclus dans les codes sources des annotations de type. Il est d'ailleurs possible que les compilateurs Akcl et Cmucl ne soient pas toujours parvenus à tirer parti des informations qui leur ont été fournies. Cela explique peut-être leurs surprenantes mauvaises performances pour certains tests.

Pour tous les compilateurs, nous avons utilisé des modes où les tests de type et tests de bornes ne sont pas émis. Les performances que nous donnons correspondent donc à des exécutions non sûres. Nous tenons d'ailleurs à apporter quelques explications concernant le système T. Dans une communication personnelle, les auteurs nous ont expliqué qu'il n'y avait plus d'option de compilation permettant de passer en mode non sûr car lorsqu'elle existait, cette option ne contribuait pas à rendre les performances meilleures. En étudiant le code produit, nous avons constaté qu'il n'y a effectivement pas besoin d'option car le code produit est *toujours* non sûr ! Étudions le cas de la compilation de la fonction :

```
(define (foo x)
   (vector-set! x 1 (fx+ 27 (vector-ref x 1))))
```

Le code en langage d'assemblage produit par Orbit est :

```
96:     00000068
100:    001C0241                        Procedure ^FOO_5 (lambda (K_3 X_4) ...)
104:    E4046002    ld 2(a1),a2         ($CONTENTS-LOCATION 1 ^C_11 $VECTOR-ELT X_4 (QUOTE 0))
108:    A484A004    add $4,a2,a2        ($FIXNUM-ADD 1 ^C_14 (QUOTE 1) V_10)
112:    E4246002    st a2,2(a1)         ($SET-LOCATION 1 ^B_21 $VECTOR-ELT VAL_13 X_4 (QUOTE 0))
116:    A2800012    add a2,zero,a1      Return from procedure (K_3 0 VAL_13)
120:    81C3E000    jmpl 0(link),zero
124:    9A803FFE    add $-2,zero,scratch
```

On voit qu'il n'y a ni test de type ni test de bornes émis dans ce programme. D'ailleurs l'exécution produit :

```
==> (load "foo.so")
;Loading  foo.so into SCHEME-ENV
FOO
==> (foo '#())
** Error: reference to non-existent memory
```

Étrange message ! En traçant les appels systèmes nous avons observé:

```
read (0, (foo '#())
"(foo '#())\n", 512) = 11
- SIGSEGV (11)
sigsetmask (0) = 0x402
write (1, "\n", 1) =
1
write (1, "** Error: ", 10) = ** Error: 10
write (1, "reference to non-existent memory", 32) = reference to non-existent memory32
```

Cette trace montre qu'Orbit se contente de rattraper le signal `11` en guise de gestion d'erreur. Ceci confirme qu'Orbit n'émet pas de tests de type ni de tests de bornes. Cela explique également pourquoi, sous l'interprète, invoquer la fonction `string->number` sur un argument qui n'est pas une chaîne produit le message d'erreur : `** Error: reference to non-existent memory` ....

	Compilateur					
Banc d'essai	**Bigloo**	**S2c**	**Orbit**	**Complice**	**Akcl**	**Cmucl**
Dderiv	*0.57*	0.80		1.23	1.40	1.80
Deriv	*0.44*	0.57	0.82	1.00	1.23	0.59
Destru	*0.15*	0.27	0.30	0.38	0.37	0.32
Div2-iter	*0.19*	0.27	0.30	0.50	0.58	0.48
Div2-rec	0.97	1.10	0.56	*0.48*	1.51	0.66
Puzzle	*0.54*	0.55	0.59	1.46	3.00	9.56
Tak	*0.02*	0.03	0.06	0.08	0.38	0.05
Fread	*0.02*	0.96	0.10	0.04	0.03	0.06
Boyer	3.22	4.70	5.64	*2.53*	5.82	4.40

Tableau 11.1: Temps d'exécution des bancs d'essai de Gabriel sur Sparc

Analysons les chiffres relevés dans le tableau 11.1[2] :

- La première observation qui doit être faite à la lecture des performances est que l'amélioration des machines conjuguée à l'amélioration des compilateurs conduit à des temps d'exécutions très faibles. Les deux exemples les plus significatifs sont **Tak** et **Fread**. S'ils ne sont exécutés qu'une seule fois, invariablement les mesures de performances donnent 0 seconde de temps système et 0 seconde de temps `cpu` lorsqu'ils sont compilés par Bigloo. Nous avons donc été obligés de répéter un grand nombre de fois les tests pour avoir des résultats significatifs. Par exemple, **Tak** et **Fread** sont répétés 500 fois, **Destru** et **Div2-iter** 60 fois, etc. Cette pratique n'est pas sans poser les problèmes que nous avons déjà soulevés dans le chapitre 1. En particulier, nous incluons les temps de *GC*. Ceci explique pourquoi les chiffres que nous donnons ici peuvent être différents de ceux déjà publiés.

- La deuxième remarque évidente est que les deux compilateurs Common Lisp semblent étrangement peiner sur certains programmes (**Puzzle** pour Cmucl et **Tak** pour Akcl). Nous ne sommes pas capables d'expliquer ce phénomène. Les programmes sources que nous avons utilisés nous ont été communiqués par l'équipe de maintenance de Akcl. En prenant leur code nous nous sommes mis à l'abri de relevés de performances incorrects dûs à l'absence d'annotations dans les programmes (annotations `(declare ...)` ou `(the ...)`).

- Il nous faut donner quelques explications sur le comportement des différents compilateurs sur le programme **Div2-rec**. Ce test constitue sur Sparc un point singulier puisque les rapports entre compilateurs s'inversent ! Bigloo et Scheme-to-C qui ont généralement de bonnes performances sont presque les plus lents sur ce test. Ceci trouve son explication dans l'utilisation des fenêtres de registres des machines Sparc. **Div2-rec** est un programme test qui réalise une descente récursive non terminale dans un arbre. Étudions le comportement d'un programme presque similaire :

[2] Les temps présentés sont, comme cela est précisé dans le chapitre 1, formés de l'addition du temps cpu et du temps système. Ils sont exprimés en secondes. Le symbole $\frown$ indique que la compilation du test a échoué ou que l'exécution du programme compilé a été incorrecte.

```
(define (foo n acc)
   (if (=fx n 1)
       acc
       (+fx n (foo (-fx n 1) acc))))

(define (main argv)
   (let ((n (string->number (cadr argv)))
         (m 500000))
      (let loop ((m m)
                 (acc 0))
         (if (=fx m 0)
             (print acc)
             (loop (-fx m 1) (foo n acc))))))
```

Ce programme invoque 500000 fois la fonction `foo`. Cette fonction empile `n` appels récursifs. Nous avons étudié les performances des codes produits par deux compilateurs (Bigloo et Orbit) lorsqu'on fait varier `n`. La figure 11.2 contient deux courbes où l'axe des abscisses correspond aux valeurs de `n` alors que l'axe des ordonnées correspond aux temps d'exécutions.

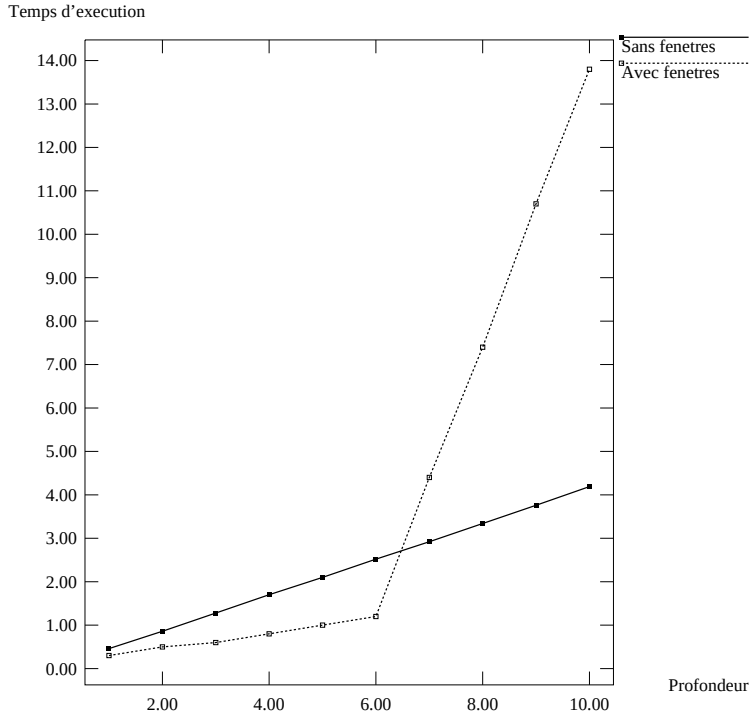


FIG. 11.2 - Les fenêtres de registres sur Sparc

Ces deux courbes sont parfaitement significatives. Le compilateur C qui compile le fichier produit par Bigloo génère un exécutable qui utilise les fenêtres de registres alors que le compilateur Orbit qui produit directement du code natif ne s'en sert pas. La stratégie utilisant les fenêtres de registres est gagnante tant que la profondeur globale des appels imbriqués est inférieure au nombre de fenêtres. La figure 11.2 permet d'établir que la machine Sparc utilisée possède 7 fenêtres (`1` (pour l'appel à `main`) + `6` appels imbriqués à `foo`). La figure établit implacablement que si la stratégie utilisant les fenêtres de registres est gagnante, si les récursions ne sont pas profondes, elle est terriblement perdante quand elles ne le sont pas (dès la profondeur de 11, la version utilisant les fenêtres est 4 fois plus lente que la version qui ne les utilise pas) ! Le style *standard* des programmes C n'utilise que parcimonieusement les récursions. Ce n'est pas le cas, au contraire, des langages fonctionnels. L'incidence des fenêtres de registres des Sparc

est donc importante, on peut même se demander si ce n'est pas la raison majeure de l'inefficacité des compilateurs les utilisant[3].

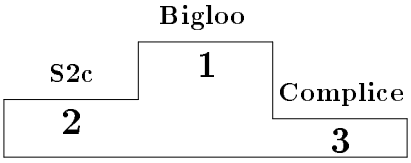
Afin d'être plus complet, nous avons également mesuré les performances des compilateurs fonctionnant sur machine Mips (ds5000/200). Pour Le-Lisp nous avons utilisé les temps fournis dans la distribution standard. Ce ne sont donc des temps `cpu` ; le temps `système` n'est pas compté pour ce compilateur sur cette architecture. En revanche, pour les autres compilateurs, nous continuons à donner les sommes `cpu + système`.

	Compilateur			
Banc d'essai	**Bigloo**	**S2c**	**Orbit**	**Complice**
Dderiv	*0.56*	0.69		0.96
Deriv	*0.49*	0.60	0.76	0.85
Destru	*0.16*	0.33	0.22	0.30
Div2-iter	*0.16*	0.26	0.28	0.29
Div2-rec	*0.22*	0.31	0.35	0.28
Puzzle	*0.63*	0.73	*0.63*	1.44
Tak	*0.05*	0.06	0.06	0.08
Fread	*0.02*	0.10	0.09	
Boyer	2.99	4.50	5.22	*1.52*

Tableau 11.2: Temps d'exécution des bancs d'essai de Gabriel sur Mips

La hiérarchie des compilateurs est la même que sur Sparc. Nous devons préciser à la décharge de Complice qu'aucune de ces deux machines ne lui est favorable. Il est probable que des mesures sur machines à base de MC 68k l'aurait placé en meilleure position. Ceci nous semble être révélateur des problèmes de portabilité. Il est difficile de faire un portage de Le-Lisp. Il est donc prévisible et normal qu'il y ait de *bons* portages et de *moins bons* portages. Les temps de Complice sur Sparc et Mips nous servent autant à montrer que l'objectif de la portabilité est crucial que la qualité intrinsèque de Bigloo.

Pour résumer les mesures de performance sur les bancs d'essai de Gabriel nous avons établi un classement des compilateurs en attribuant des points en fonction des classements de chaque test. Il en résulte :



Les programmes de Gabriel permettent de penser que Bigloo se compare très favorablement aux autres compilateurs sur les programmes de premier ordre. Nous allons le comparer aux autres compilateurs Scheme sur des programmes utilisant l'ordre supérieur.

11.2 Les bancs d'essai utilisant des fonctions d'ordre supérieur

Nous avons mesuré les performances des programmes produits par les trois compilateurs Scheme (Bigloo, Scheme-to-C et Orbit) sur trois tests utilisant abondamment des fonctions d'ordre supérieur : **Dens**, **Leval** et **Church**. Les temps sont donnés dans le tableau 11.3.

	Compilateur (Sparc)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Dens	15.4	29.8	*13.7*
Leval	*10.9*	16.0	11.2
Church	*14.2*	36.2	30.5

	Compilateur (Mips)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Dens	*9.1*	33.1	⌒
Leval	*4.9*	12.1	12.4
Church	*16.3*	38.6	36.1

Tableau 11.3: Les fonctions d'ordre supérieur

[3] Cela donne raison aux choix des nouveaux compilateurs comme Twobit [CH94, Han92] qui renoncent à l'emploi des fenêtres.

Nous ne sommes pas parvenu à exécuter **Dens** avec Orbit sur les machines équipées de processeur Mips. Ceci est regrettable car comme ce compilateur produit un code rapide sur Sparc, une mesure sur Mips aurait permis de mieux cerner sa valeur. Nous aurions aimé savoir si le même phénomène (Orbit est bon sur Sparc mais en revanche, il est très nettement distancé sur Mips) que sur le test **Leval** se reproduisait.

Le tableau 11.4 contient les rapports normalisés. Les chiffres présentés correspondent aux calculs :

$$\frac{\text{(temps sur Sparc + temps sur Mips) de l'autre compilateur}}{\text{(temps sur Sparc + temps sur Mips) de Bigloo}}$$

	Compilateur		
Rapport moyen	**Bigloo**	**S2c**	**Orbit**
Dens	1.00	2.57	⌒
Leval	1.00	1.78	1.49
Church	1.00	2.45	2.17

Tableau 11.4: Rapports normalisés entre compilateur

Bigloo a de bonnes performances sur les programmes utilisant des fonctions d'ordre supérieur. L'explication se situe à la fois dans l'analyse de flot de contrôle (chapitre 7) et dans la compilation des fermetures (chapitre 8) où nos efforts portent leurs fruits. Sur le programme **Church** l'analyse de flot de contrôle est parvenue à transformer certains des appels calculés en appels directs. Sur la sémantique dénotationnelle (**Dens**) c'est l'optimisation décrite dans la section 7.4.3 qui s'applique. Les compilateurs Orbit et Scheme-to-C ne réalisent pas les mêmes optimisations sur les programmes très fonctionnels, ce qui explique pourquoi leurs performances sont moindres.

11.3 La compilation du contrôle

Nous utilisons pour mesurer la qualité de la compilation du contrôle de premier ordre, les deux tests **Peval** et **Conform**. Bien que **Conform** soit écrit en Scheme standard, nous ne sommes pas parvenus à le faire fonctionner avec le système T. Les mesures sont regroupées dans le tableau 11.5.

	Compilateur (Sparc)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Peval	*14.9*	24.9	33.0
Conform	*13.5*	34.6	⌒

	Compilateur (Mips)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Peval	*17.6*	30.0	35.0
Conform	*14.0*	35.7	⌒

Tableau 11.5: La compilation du contrôle

Le tableau 11.6 contient les rapports normalisés des exécutions sur Sparc et Mips.

	Compilateur		
Rapport moyen	**Bigloo**	**S2c**	**Orbit**
Peval	1.00	1.67	2.09
Conform	1.00	2.56	⌒

Tableau 11.6: Rapports normalisés entre compilateur

Bigloo semble à l'aise sur ces deux programmes puisque quelque soit l'architecture, c'est toujours lui qui permet d'obtenir les meilleures performances. Ils appartiennent à la classe des programmes sur laquelle nous avons concentré une partie de nos efforts : programmes utilisant peu l'ordre supérieur et n'allouant pas énormément de données (en instrumentant les exécutions, nous avons constaté que pour ces deux programmes, les temps de calcul consacrés à la gestion mémoire (allocation + *GC*) sont inférieur à 10 %).

Les bonnes performances de Bigloo s'expliquent principalement par le fait que son analyse de flot de contrôle couplée à son analyse de fermeture lui permet de produire des programmes exécutables qui n'allouent aucune mémoire dans le tas pour le contrôle.

11.4 Les entrées/sorties

Le seul test d'*entrée/sortie* dont nous disposons est le *pretty-printer* de Marc Feeley. Malheureusement, nous ne sommes pas parvenus à le compiler avec Orbit. Ce compilateur est donc absent de ces tests.

	Compilateur (Sparc)	
Processeur	**Bigloo**	**S2c**
Pp	*10.8*	35.3

	Compilateur (Mips)	
Processeur	**Bigloo**	**S2c**
Pp	*7.5*	33.2

Tableau 11.7: Les entrées sorties

	Compilateur	
Rapport moyen	**Bigloo**	**S2c**
Pp	1.00	3.74

Tableau 11.8: Rapports normalisés entre compilateur

Bigloo a de biens meilleures performances que Scheme-to-C. Cela s'explique principalement par la meilleure qualité de son lecteur car la compilation du contrôle ne pose pas de problème majeur dans ce test. Scheme-to-C et Bigloo produisent d'ailleurs à-peu-près le même code. Les travaux que nous avions menés sur l'analyse lexicale en Scheme [Ser92] permettent à Bigloo de lire très vite les données Lisp. Ce phénomène pouvait déjà s'observer sur le test **Fread** (voir section 11.1).

11.5 Les vecteurs

	Compilateur (Sparc)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Beval	9.6	23	*8.8*
Bague	*23.3*	37.6	68.8
Sievev	*15.6*	70.7	49.7

	Compilateur (Mips)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Beval	*9.0*	23.8	11.4
Bague	*24.7*	54.9	75.5
Sievev	*17.2*	63.8	22.0

Tableau 11.9: Les manipulations de vecteurs

	Compilateur		
Rapport moyen	**Bigloo**	**S2c**	**Orbit**
Beval	1.00	2.52	1.08
Bague	1.00	1.93	3.01
Sievev	1.00	4.10	2.19

Tableau 11.10: Rapports normalisés entre compilateur

Scheme-to-C semble être en difficulté sur tous les programmes présents dans cette série de test. Sa plus mauvaise performance est sur le programme **sievev**. La différence avec Bigloo s'explique principalement par la compilation de la fonction :

```
(define (mod x y)
   (- x (* (/ x y) y)))
```

Il semble que Scheme-to-C ne soit pas capable de n'utiliser que son arithmétique entière pour compiler cette fonction (alors que nous avons invoqué le compilateur avec l'option `-On` qui a pour but de le forcer à considérer que tous les nombres sont des petits entiers). En instrumentant une exécution, on peut constater que le code produit alloue des nombres flottants ! De plus certains appels aux fonctions arithmétiques ne sont pas intégrés. Le cumul de ces deux erreurs explique ses mauvais résultats.

Scheme-to-C ne produit pas non plus de bons exécutables pour le programme **Beval**. L'explication est d'un autre ordre. Ce compilateur expanse les constructions `case` en cascade de `if` or **Beval** est un interprète

de code-octet. Tout le temps de l'exécution est passé dans le moteur de l'interprète qui est formé d'un `case` à 50 branches. Comme ces branches sont étiquetées par des petits entiers, Bigloo le compile comme un `switch` C. Cette compilation a été présentée dans le chapitre 5.

11.6 Les listes

Pour mesurer l'efficacité de la gestion des listes, nous avons utilisé deux tests: **Queens** et **Earley**. En plus de tester les listes, ces deux programmes nécessitent une bonne compilation du contrôle. **Earley** contient de nombreuses fonctions locales et **Queens** comprend quelques applications de fonctions d'ordre supérieur.

	Compilateur (Sparc)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Queens	12.1	28.3	*9.9*
Earley	*14.7*	15.2	33.7

	Compilateur (Mips)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Queens	*7.7*	16.3	⌢
Earley	*14.4*	16.5	⌢

Tableau 11.11: Les listes

Le tableau 11.6 contient les rapports normalisés des exécutions sur Sparc et Mips.

	Compilateur		
Rapport moyen	**Bigloo**	**S2c**	**Orbit**
Queens	1.00	2.25	⌢
Earley	1.00	1.09	⌢

Tableau 11.12: Rapports normalisés entre compilateur

Encore une fois, il est regrettable que nous n'ayons pu faire fonctionner les programmes compilés par Orbit sur Mips. Néanmoins, nous pouvons penser que si ce compilateur est le plus rapide sur le programme **Queens** sur Sparc il en aurait été autrement sur Mips car nous pensons que c'est l'emploi des fenêtres de registres (comme sur le test de Gabriel **div2-rec**) qui explique ses meilleures performances. Pour nous en convaincre, nous avons ré-écrit ce programme en transformant les récursions de manière à les rendre terminales. Les temps de cette nouvelle version de **Queens** sont donnés sur Sparc dans le tableau 11.13. Comme cela était prévisible, Bigloo et Scheme-to-C qui utilisent les fenêtres de registres voient leur performances améliorées alors que Orbit qui ne les utilise pas a des performances stationnaires.

	Compilateur (Sparc)		
Banc d'essai	**Bigloo**	**S2c**	**Orbit**
Queens-tail	*8.5*	23.9	10.1

Tableau 11.13: La version récursive terminale de **Queens**

Pour expliquer l'écart important qui sépare Bigloo et Scheme-to-C, nous devons nous référer au tableau 11.14 qui indique les pourcentages des temps globaux d'exécutions consacrés à la gestion mémoire (allocation + *GC*). L'analyse de flot de contrôle de Bigloo a transformé l'appel à la fonction currifiée en un appel direct. Scheme-to-C ne fait pas cette analyse et il alloue donc des fermetures. Comme cette fonction currifiée est fréquemment utilisée dans le test, le volume d'allocation de Scheme-to-C est plus important que celui de Bigloo. En revanche, sur le programme **Earley**, les temps de gestion mémoire sont les mêmes pour les deux compilateurs. Comme ces temps représentent une part importante de l'exécution (45 % pour les deux compilateurs) et que leur gestionnaire mémoire est de technologie assez proche (racines ambiguës et conservatif) il en résulte que les deux compilateurs ont des performances assez semblables.

11.7 Conclusion

Tous les programmes de tests qui nous ont servi à relever nos mesures, mis bout à bout, forment plus de 5000 lignes de Scheme. D'après leur nombre et leurs différents auteurs, ces lignes constituent un échantillon

	Compilateur	
Banc d'essai	**Bigloo**	**S2c**
Queens	35 %	45 %
Earley	45 %	45 %

Tableau 11.14: Le temps consacré à la gestion mémoire lors des exécutions

représentatif des styles de programmation rencontrés en Scheme. Au fil des mesures, nous avons pu constater que Bigloo se comparait toujours très favorablement aux autres compilateurs Lisp. Les bonnes performances que nous obtenons nous permettent de valider notre approche. Bigloo partait avec deux handicaps liés à son langage cible: *(i)* produire du C plutôt que du langage machine nous empêchait d'utiliser des techniques rentables pour les langages fonctionnels comme le passage d'arguments dans des registres *(ii)* , produire du C orthodoxe nous permet de bénéficier des nombreuses optimisations des compilateurs C, mais compromet l'efficacité du *GC* en empêchant l'emploi des techniques modernes comme les générations.

Les précédents travaux qui ont été menés laissaient penser que produire du code natif permettait à un compilateur d'être environ deux fois plus rapide que s'il produisait du C (voir par exemple les travaux [Sén91, FMRW94]). Notre approche qui a consisté à multiplier les analyses de haut niveau nous a permis de hisser Bigloo au niveau des meilleurs compilateurs produisant du code natif. Bien sûr, des compilateurs produisant du code natif pourraient parfaitement réaliser nos analyses et optimisations et il est alors tout à fait probable que le facteur 2 réapparaîtrait.

Chapitre 12

Conclusion

Le propos de cette thèse est de présenter des techniques permettant la réalisation de compilateurs de langages fonctionnels portables *et* performants. Nous avons distingué deux catégories de portabilités : la portabilité d'exécution et la portabilité de compilation.

La portabilité d'exécution

La portabilité d'exécution a été obtenue en choisissant pour langage cible un langage de haut niveau jouant le rôle de langage d'assemblage portable. Le langage C étant le candidat le plus sérieux à ce poste, c'est lui que nous avons choisi. Cette orientation permet à Bigloo de fonctionner sur la quasi totalité des machines Unix. De plus, puisque C est à peu près bien compilé sur toutes les machines, notre compilateur a des performances homogènes sur toutes les architectures. Ceci nous semble constituer un critère de portabilité aussi important que le simple fait de tourner sur plusieurs machines.

La portabilité de compilation

Les analyses et optimisations présentées dans cette thèse ont un spectre d'application très large. Pour le prouver nous avons montré dans le chapitre 10 qu'en plus de Lisp, nos techniques s'appliquent parfaitement à la compilation de Caml. Les nombreuses mesures de performances montrent que Bigloo est l'un des meilleurs compilateurs ML, validant ainsi nos optimisations.

L'efficacité

L'efficacité du code produit a été notre objectif principal. Nous ne voulions pas que la portabilité nous conduise à des performances moyennes. La nature de notre langage cible nous a interdit l'emploi de techniques particulièrement bien adaptées à la compilation des langages fonctionnels. Nous avons compensé cela par deux moyens :

- Nous avons orienté notre compilateur pour que le code produit puisse bénéficier des nombreuses optimisations réalisées par les compilateurs C (réduction en force (*strength-reduction*), qualité du code produit (*peephole optimizations*, *branch tensioning*, ...), déplacement de code (*code motion*), etc.). Pour cela, nous avons compilé chaque construction Scheme possédant un équivalent en C en une projection, que nous avons nommée *projection naturelle*, vers cet équivalent. Le code C ainsi obtenu a des allures de C écrit à la main (nous nommons ce code du C *orthodoxe*). Ainsi, nos programmes C générés tombent naturellement dans le champ d'application des optimisations des compilateurs C.

- Puisque nous avons perdu la possibilité de faire des optimisations sur les parties basses (*back end*) de la compilation (contrôler l'allocation de registres nous est évidemment impossible), nous avons multiplié les optimisations de haut niveau. Nous avons présenté tout un arsenal d'analyses et d'optimisations s'appliquant aux langages fonctionnels (intégration fonctionnelle, β-réduction à la compilation, élimination des sous-expressions communes, analyses de flot de contrôle d'ordre supérieur, etc.).

Dans le chapitre 11 nous avons montré que sur un vaste échantillon de programmes, notre compilateur est l'un des plus rapides. Les deux machines modernes (à base de processeurs Sparc et Mips) sur lesquelles nous avons fait nos mesures, montrent également qu'il a des performances homogènes, ce qui n'est pas forcément le cas de nos concurrents qui produisent du code natif. Cela confirme que le choix de C comme cible permet l'alliance entre portabilité et efficacité.

Extensibilité

L'extensibilité est une des caractéristiques de Bigloo qui le distingue des précédents compilateurs. Elle se divise en deux catégories :

L'extensibilité supérieure: Notre compilateur à été conçu de façon à ce qu'il soit possible de lui ajouter dynamiquement des phases de pré-compilation. Nous avons présenté dans le chapitre 10 l'extension qui lui permet de compiler des programmes Caml-light. D'autres extensions existent ; citons, par exemple, la compilation des programmes MEROON ou encore le solveur de contraintes de Jean-Marie Geffroy et Daniel Diaz.

L'extensibilité inférieure: Au moyen d'une interface externe complète, le code produit par Bigloo peut être mélangé à du code compilé écrit dans d'autres langages de programmation. Le chapitre 4 contient la présentation de l'interface avec le langage C.

Langages fonctionnels, langages impératifs

Au delà des objectifs techniques de cette thèse, notre ambition est de contribuer à rendre les langages fonctionnels plus compétitifs vis-à-vis des langages impératifs. Qu'en est-il maintenant ? Aucune des mesures présentées dans le chapitre 11 ne comparait des programmes C et des programmes Lisp. La raison en est que nous cherchions à prouver dans ce chapitre que nos techniques de compilation étaient bien adaptées au monde des langages fonctionnels. Le principe de nos mesures a donc été de prendre tel quel des programmes Lisp, de les compiler avec plusieurs compilateurs dont Bigloo et de comparer les performances des codes produits. Dans cette optique il n'est pas possible de comparer des programmes C et des programmes Lisp car justement ces deux langages sont très différents : on n'écrit pas un programme C comme on écrit un programme Lisp. De plus il n'est pas pertinent de comparer les performances d'un programme C écrit dans un style Lisp. La question importante est : *un programme Lisp (ou ML) peut-il concurrencer en performances un programme C, écrit par un habitué de ce langage ?* Nous allons apporter trois éléments de réponse :

- L'article [HFA+94] décrit une expérience où la résolution d'un problème faisant intervenir du calcul numérique flottant a été programmée dans plusieurs langages. Une version C (écrite dans un style C avec des passages d'adresses, des allocations en pile, ...) est comparée à plusieurs versions fonctionnelles. La vitesse du code produit par Bigloo est entre trois et quatre fois plus lente que celle du code produit par le compilateur C. Il existe à cela deux explications.
  - Les nombres flottants de Bigloo ont la même taille que les `double` de C. Ces nombres sont alloués dans des structures pour permettre le typage dynamique et l'implantation des prédicats standards. Bien que grâce à une nouvelle optimisation, la plupart des allocations des nombres flottants du programme soient faites en pile, il n'en reste pas moins que le cumul des nombres alloués dans le tas et des structures de données manipulées dans le programme constitue une lourde charge pour le gestionnaire mémoire. Ainsi, en instrumentant les exécutions, nous avons constaté que 44 % du temps global est consacré à la gestion mémoire. Le programme C est écrit de telle façon que toutes les allocations sont faites en pile. C'est-à-dire que la version C incorpore, de fait, un algorithme de gestion explicite de mémoire *ad hoc*. D'une façon générale on ne s'étonnera pas qu'une gestion spécifique soit plus efficace qu'une gestion générale. Bien que Bigloo, comparativement à d'autres systèmes fonctionnels, alloue peu de mémoire (grâce à sa compilation efficace du contrôle), la gestion mémoire implicite reste pénalisante.
  - En imaginant que l'on puisse faire disparaître le coût de cette gestion mémoire, les performances de la version compilée par Bigloo n'en resteraient pas moins inférieures à celles du programme C

(environ 50 % plus lentes). Cela s'explique probablement par le fait que l'arithmétique flottante de Bigloo est moins efficace que celle de C car toutes les opérations nécessitent des indirections dans des structures. Des techniques comme celles de X. Leroy [Ler92] permettent de faire partiellement disparaître ce surcoût.

L'expérience **Pseudoknot** nous semble être représentative du cas extrême défavorisant les langages fonctionnels. Généralement ces langages ne sont pas conçus pour les calculs flottants et donc les efforts des concepteurs de compilateurs ne portent pas sur ce type de programmes. C'est pourquoi la gestion des flottants est souvent un peu lourde et surtout très coûteuse en mémoire. Néanmoins cette expérience montre nettement que nos efforts pour nous rapprocher des performances des langages impératifs sans *GC* doivent, à l'avenir, porter sur l'allocation. Les langages fonctionnels modernes comme Scheme ou ML ne permettent pas au programmeur de spécifier que telle ou telle allocation doit être faite en pile (cela serait contraire à la philosophie même de ces langages où toute la gestion mémoire est cachée), il faut donc que les compilateurs prennent eux-mêmes cette initiative. D'une manière générale nous pensons que les optimisations futures importantes auront pour objectif la diminution du volume des allocations dans le tas. Nous avons entrepris la conception et la formalisation d'un algorithme prenant la décision de déplacer des allocations du tas vers la pile. Pour l'instant nous n'en sommes qu'à l'état de prototype mais les résultats obtenus sont d'ores et déjà encourageants. L'état de nos recherches ne nous a toutefois pas permis de présenter l'algorithme dans ce document. Terminer cet algorithme est le principal objectif de nos travaux futurs.

- L'expérience du programme **Pseudoknot** compare deux implantations du même algorithme. La seconde expérience que nous montrons maintenant, concerne deux algorithmes différents utilisés pour résoudre un même type de problème : l'analyse lexicale. Les différences d'implantation s'expliquent simplement par les orientations différentes des langages. La version Scheme code les automates déterministes par des fonctions alors que la version C les code par des tableaux. L'article [Ser92] contient la description du système (**Rgc**) d'analyse lexicale efficace pour Scheme inclus maintenant dans Bigloo. Cet article mesure et compare les performances des systèmes **Lex**, **Flex** et **Rgc**. Il apparaît que la lecture et l'analyse lexicale réalisées par Bigloo sont environ 260 % fois plus rapide que celles de C et **Lex** et 20 % fois plus rapide que celles de C et **Flex**. Cet exemple nous sert à montrer que sur des programmes réels, Scheme n'est pas obligatoirement plus lent que C. L'analyse lexicale est un exemple assez complet puisqu'elle met en œuvre des mécanismes de lecture, de manipulation d'entiers et de caractères, mécanismes que l'on pourrait, a priori, penser parfaitement convenir au langage C.

- Le dernier exemple d'application réelle existant en C et Scheme que nous allons comparer est celui des interprètes Lisp ! En effet, existent (dans le domaine public) plusieurs interprètes Scheme écrit en C (les plus fréquemment utilisés sont **Scm**, **Elk** ou encore **Siod**). Chacun de ces interprètes possède une implantation différente des autres. Il est intéressant de comparer les performances de ces interprètes avec celles de l'interprète de Bigloo car ce dernier est totalement écrit en Scheme. Il encore plus intéressant de constater que l'interprète de Bigloo est un des plus rapides (le plus rapide sur certaines architectures comme les machines équipées de processeurs Mips). L'ambition de ce paragraphe n'est pas de vanter les mérites de notre interprète, mais de montrer que programmer en Scheme n'est pas moins efficace que programmer en C. Nous devons cependant reconnaître que cet exemple « grandeur nature » est un peu biaisé. Nous avons expliqué précédemment que la principale source d'inefficacité de Lisp vient de son gestionnaire mémoire. L'exemple des analyseurs lexicaux appuie cet argument puisque les automates de **Rgc** ne consomment pas de mémoire et sont largement aussi rapides que ceux implantés en C. Le cas des interprètes C est ambigu car même si ces interprètes sont écrits en C, ils sont obligés d'avoir un *GC* pour parvenir à interpréter correctement les programmes. Il n'est donc pas étonnant qu'un interprète C ne soit pas plus rapide qu'un interprète écrit en Scheme et compilé par un compilateur efficace.

En conclusion, nous pouvons donc affirmer qu'il est possible d'écrire des applications performantes en Lisp ou ML car Bigloo est un compilateur qui produit un code performant. Pour certains types d'applications, les programmes qu'il compile sont largement aussi rapides que des programmes C équivalents, compilés par les compilateurs optimisants standard. Nous espérons que l'alliance de performances raisonnables et l'ouverture de Scheme et ML au monde extérieur (grâce à des interfaces externes sophistiquées) contribueront à rendre ces langages crédibles dans des contextes autres qu'universitaires.

Bibliographie

[ACT92] Apple Computer (Eastern Research) et Technology. – *Dylan, An object-oriented dynamic language.* – Apple Computer, Inc., avril 1992.

[AM87] Appel (A. W.) et MacQueen (D. B.). – A Standard ML Compiler. *In : Functional Programming Languages and Computer Architecture*, éd. par Kahn (G.). pp. 301–324. – Springer-Verlag 274, 1987.

[App87] Appel (A.). – Garbage collection can be faster than stack allocation. *Information Processing Letters*, vol. 25, n° 4, juin 1987, pp. 275–279.

[App92] Appel (A.). – *Compiling with continuations.* – Cambridge University Press, 1992.

[AS94] Appel (A.) et Shao (Z.). – *An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures.* – Rapport technique n° CS-TR-450-94, Princeton University, 1994.

[ASU86] Aho (A.), Sethi (R.) et Ullman (J.). – *Compilers: Principles, Techniques and Tools.* – Addison-Wesley, 1986.

[Att94] Attardi (G.). – *The Embeddable Common Lisp.* – Rapport technique n° unpublished, Corso Italia 40, I-56125 Pisa, Italy, Dipartimento di Informatica, Università di Pisa, 1994.

[Aye92] Ayers (A.). – Efficient Closure Analysis with Reachability. *In : Bigre 81-82, WSA '92 Workshop on Static Analysis.* – septembre 1992.

[Bak92a] Baker (H.). – Inlining Semantics for Subroutines which are recursive. *ACM Sigplan Notices*, vol. 27, n° 12, décembre 1992, pp. 39–46.

[Bak92b] Baker (H.). – *The Buried Binding and Dead Binding Problems of Lisp 1.5: Sources of Incomparability in Garbage Collector Measurements.* – Rapport technique n° 5(2):11–19, Lisp Pointers, 1992.

[Bak94] Baker (H.). – *CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A <1>.* – Rapport technique n° TR-Nbl-94-01, 1994.

[Bar88] Bartlett (J.F.). – *Compacting Garbage Collection with Ambiguous Roots.* – Research Report n° 88/2, Palo Alto, Cal., Digital Western Research Laboratory, février 1988.

[Bar89a] Bartlett (J.F.). – *Mostly-Copying Garbage Collection Picks Up Generations and C++.* – Technical Note n° TN-12, Palo Alto, Cal., Digital Western Research Laboratory, octobre 1989.

[Bar89b] Bartlett (J.F.). – *Scheme->C a Portable Scheme-to-C Compiler.* – Research Report 89 n° 1, Palo Alto, California, DEC Western Research Laboratory, janvier 1989.

[Boe91] Boehm (H.J.). – Space efficient conservative garbage collection. *In : ACM Conference on Programming Language Design and Implementation, SIGPLAN Notices 28, 6*, pp. 197–206. – 1991.

[BW88] Boehm (H.J.) et Weiser (M.). – Garbage collection in an uncooperative environment. *Software — Practice and Experience*, vol. 18, n° 9, septembre 1988, pp. 807–820.

[CC77] Cousot (P.) et Cousot (R.). – Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In : Symposium on Principles of Programming Languages*, pp. 238–252. – Los Angeles, CA, janvier 1977.

[CDD+86] Chailloux (J.), Devin (M.), Dupont (F.), Hullot (J.M.), Serpette (B.) et Vuillemin (J.). – *Le_Lisp version 15.2. Le manuel de référence.* – Rapport technique n° L-003, France, INRIA-Rocquencourt, 1986.

[CF91] Cartwright (R.) et Fagan (M.). – Soft Typing. *In : Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 278–292. – Toronto, Ontario, Canada, juin 1991.

[CH94] Clinger (W.D.) et Hansen (L.T.). – Lambda, the Ultimate Label or A Simple Optimizing Compiler for Scheme. *In : Conference Record of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 128–139. – Orlando, Florida, USA, 1994.

[Cha85] Chailloux (J.). – *La machine LLM3.* – Rapport technique n° 55, France, INRIA, juin 1985.

[Cha91] Chailloux (E.). – *Compilation des langages fonctionnels: CeML un traducteur ML vers C.* – Paris (France), Thèse de doctorat d'université, Université Paris 7, novembre 1991.

[Cha92a] Chailloux (E.). – A Conservative Garbage Collector with Ambiguous Roots for Static Typechecking Languages. *In: IWMM 92*, éd. par Bekkers (Y.) et Cohen (J.), pp. 218–247. – St. Malo, France, septembre 1992.

[Cha92b] Chailloux (E.). – An efficient way to compile ML to C. *In: Workshop on ML and its applications.* – San Francisco, California, USA, 1992.

[Cha92c] Chambers (C.). – *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* – Departement of Computer Science, Technical report stan-cs-92-1240, Stanford University, mars 1992.

[Cri92] Cridlig (R.). – An optimizing ML to C compiler. *In: Workshop on ML and its applications.* – San Francisco, California, USA, 1992.

[DB72] De Bruijn (N.G.). – λ-calculus notation with nameless dummies, a tool for automatic formual manipulation. *Indag. Math.*, vol. 34, 1972.

[DC94] Dean (F.) et Chambers (C.). – Towards Better Inlining Decisions Using Inlining Trials. *In: Conference Record of the ACM Conference on Lisp and Functional Programming*, pp. 273–282. – Orlando, Florida, US, juin 1994.

[Del91] Delacour (V.). – *Gestion mémoire automatique pour langages de programmation de haut niveau.* – Paris (France), Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), juin 1991.

[DFH86] Dybvig (R.K.), Friedman (D.P.) et Haynes (C.T.). – Expansion-passing style: Beyond conventional macros. *In: Conference Record of the 1986 ACM Conference on Lisp and Functional Programming*, pp. 143–150. – 1986.

[DFHH93] Dowek (G.), Felty (A.), Herbelin (H.) et Huet (G.). – *The COQ proof assistant user's guide.* – Rapport technique n° 154, France, Inria-Rocquencourt, 1993.

[DLKR92] Dolenc (A.), Lemmke (A.), Keppel (D.) et Reilly (G.V.). – *Notes on Writing Portable Programs in C.* – Rapport technique, Helsinki University of Thechnology and CS&E, University of Washington and Dept. of Computer Science, Brown University, July 1992.

[DM82] Damas (L.) et Milner (R.). – Principle Type Inference for Functional Programs (extended abstract). *In: 9th ACM Symposium on Principle of Programming Langugages*, pp. 207–212. – 1982.

[DPS94] Davis (H.), Parquier (P.) et Séniak (N.). – Sweet Harmony: the Talk/C++ Connection. *In: Conference Record of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 121–127. – Orlando, Florida, USA, 1994.

[DTM94] Diwan (A.), Tarditi (D.) et Moss (E.). – Memory Subsystem Performance of Programs Using Copying Garbage Collection. *In: Symposium on Principles of Programming Languages*, pp. 1–14. – 1994.

[Dur85] Durand (R.). – *Conception des programmes applicatifs: méthodologie et transformations.* – Paris (France), Thèse de doctorat d'état, Université Paris VI, juin 1985.

[FL88] Fischer (C.N.) et LeBlanc (R.J. Jr.). – *Crafting A Compiler.* – The Benjamin/Cummings Publishing Company, 1988.

[FMRW94] Feeley (M.), Miller (J.), Rozas (G.) et Wilson (J.). – *Compiling Higher-Order Languages into Fully Tail-Recursive Portable* C. – Rapport technique n° Unpublished, 1994.

[Gab85] Gabriel (R.). – *Performance and Evaluation of Lisp Systems.* – Cambridge, Massachusetts, MIT Press, 1985.

[Gud93] Gudeman (D.). – *Representing Type Information in Dynamically Typed Languages.* – Rapport technique, Departement of Computer Science, Gould-Simpson Building, The University of Arizona, Tucson, AZ 85721, University of Arizona, avril 1993.

[H*al.*91] Hanson (C.) et *al.* – *MIT Scheme reference manual.* – Rapport technique n° 1281, Artificial Intelligence Laboratory, Cambridge, MA, Massachusetts Institute of Technology, janvier 1991.

[Han92] Hansen (L.T.). – *The impact of programming style on the performance of Scheme programs.* – M.s. thesis, University of Oregon, août 1992.

[HBH93] Hammond (K.), Burn (G.L.) et Howe (D.B.). – Siking your cache. *In: Proceedings of the 1993 Glasgow Workshop on Functional Programming*, éd. par O'Donnel (J.T.) et Hammond (K.), pp. 58–68. – Ayr, Scotland, juillet 1993.

[HC89] Hwu (W.) et Chang (P.). – Inline function expansion for compiling C programs. *In : Conference on Programming Language Design and Implementation*. ACM. – Portland, Oregon, juin 1989.

[Hen92] Henglein (F.). – Global Tagging Optimization by Type Inference. *In : ACM Conference on Lisp and Functional Programming*. – 1992.

[HFA+94] Hartel (P. H.), Feeley (M.), Alt (M.), Augustsson (L.), Baumann (P.), Beemster (M.), Chailloux (E.), Flood (C. H.), Grieskamp (W.), van Groningen (J. H. G.), Hammond (K.), Hausman (B.), Ivory (M. Y.), Lee (P.), Leroy (X.), Loosemore (S.), Röjemo (N.), Serrano (M.), Talpin (J.-P.), Thackray (J.), Weis (P.) et Wentworth (P.). – Pseudoknot: a Float-Intensive Benchmark for Functional Compilers (DRAFT). *In : Implementation of Functional Languages*, éd. par Glauert (J. R. W.). pp. 13.1–13.34. – School of Information Systems, Univ. of East Anglia, Norwich NR4 7TJ, UK, septembre 1994.

[Hof93] Hoffmann (U.). – *Using C as Target Code for Translating Highlevel Programming Languages*. – Rapport technique n° APPLY/CAU/IV.4/2, Christian-Albrechts-University of Kiel, 1993.

[HS91] Harbisson (S.P.) et Steele (G.L.). – *C a Reference Manual*. – Englewood Cliffs, NJ 07632, Prentice Hall Software Series, 1991, third edition édition.

[IEE91] IEEE Std 1178-1990. – *IEEE Standard for the Scheme Programming Language*. – New York, NY, Institute of Electrical and Electronic Engineers, Inc., 1991.

[ISO90] ISO/IEC. – *9899 Programming Language - C*. – Rapport technique n° DIS 9899, ISO, July 1990.

[Joh85] Johnson (T.). – *Lambda Lifting :* Transforming Programs to Recursive Equations. *In : Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 190–203. – 1985.

[KH92] Kane (G.) et Heinrich (J.). – *MIPS RISC Architecture*. – Prentice-Hall, 1992.

[KKR+86] Kranz (D.), Kesley (R.), Rees (J.), Hudak (P.), Philbin (J.) et Adams (N.). – ORBIT: An optimizing compiler for Scheme. *In : Symposium on Compiler Construction*. ACM, pp. 219–233. – Palo Alto, California, juin 1986.

[Kra88] Kranz (D.A.). – *ORBIT: An Optimizing Compiler For Scheme*. – Thèse de PhD, Yale university, février 1988.

[Ler90] Leroy (X.). – *The ZINC experiment: an economical implementation of the ML language*. – Rapport technique n° RT-117, France, Inria-Rocquencourt, février 1990.

[Ler92] Leroy (X.). – Unboxed objects and polymorphic typing. *In : Symposium on Principles of Programming Languages*, pp. 177–188. – Albequerque, New Mexico, janvier 1992.

[LH83] Lieberman (H.) et Hewitt (C.). – A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, vol. 26, n° 6, juin 1983, pp. 419–429.

[Mac92a] MacLachlan (R.A.). – *CMU Common Lisp User's Manual*. – Rapport technique, Pittsburgh, PA 15213, USA, Carnegie Mellon University, mai 1992.

[Mac92b] MacLachlan (R.A.). – The Python Compiler for CMU Common Lisp. *In : Conference Record of the ACM Conference on Lisp and Functional Programming*, pp. 235–246. – San Francisco, California, USA, juin 1992.

[MB93] Mateu-Brule (L.). – *Stratégies avancées de gestion de blocs de contrôle*. – Paris (France), Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), février 1993.

[MG69] Morris et Goscinny (R.). – *Les rivaux de Painful Gulch*. – Paris, France, Dupuis, 1969.

[Pat92] Patel (D.). – *Using load-foreign and define-foreign under UNIX*. – Documenation of the release 3.1, 1992.

[Ple91] Pleban (U.F.). – Compilation Issues in the Screme Implementation for the 88000. *In : Topics in Advanced Language Implementation*, éd. par Lee (P.). pp. 157–188. – The MIT Press, 1991.

[QG92] Queinnec (C.) et Geffroy (J-M.). – Partial evaluation applied to symbolic pattern matching with intelligent backtrack. *In : Workshop on Static Analysis*, éd. par Billaud (M), Castéran (P), Corsini (MM), Musumbu (K) et Rauzy (A). pp. 109–117. – Bordeaux (France), septembre 1992.

[Que90] Queinnec (C.). – *Le filtrage : une application de (et pour) Lisp*. – Paris (France), InterÉditions, 1990.

[Que93] Queinnec (C.). – Designing MEROON v3. *In : Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, éd. par Rathke (Christian), Kopp (Jürgen), Hohl (Hubertus) et Bretthauer (Harry). – Sankt Augustin (Germany), septembre 1993.

[Que94] Queinnec (C.). – *Les langages Lisp.* – InterEditions, 1994.

[RA82] Rees (J.A.) et Adams (N.I.). – T: a Dialect of Lisp or, LAMBDA: the Ultimate Software tool. *In : Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114–122. – 1982.

[RM92] Rose (J. R.) et Muller (H.). – Integrating the Scheme and C languages. *In : Conference Record of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 247–259. – 1992.

[Rob89] Roberts (E.S.). – *Implementing Exceptions in C.* – Rapport technique n° 26154, 130 Lytton Avenue. Palo Alto, California 94301. USA, DEC SRC, 1989.

[Roz84] Rozas (G.J.). – *Liar, an Algol like compiler for Scheme.* – S.b. thesis, Cambridge, Mass., Massachusetts Institute of Technology, janvier 1984.

[Roz92] Rozas (G.J.). – Taming the Y operator. *In : Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming.* ACM, pp. 226–234. – juin 1992.

[SA94] Shao (Z.) et Appel (A.). – Space-Efficient Closure Representations. *In : Conference Record of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 150–161. – Orlando, Florida, USA, 1994.

[Sch77] Scheifler (R.W.). – An Analysis of Inline Substitution for a Structured Programming Language. *CACM*, vol. 20, n° 9, septembre 1977, pp. 647–654.

[Sén90] Séniak (N.). – *Efficient Compilation of Local Functions using C as a Back-end.* – Rapport technique, Palaiseau, France, LIX, École Polytechnique, 1990.

[Sén91] Séniak (N.). – *Théorie et pratique de Sqil: un langage intermédiaire pour la compilation des langages fonctionnels.* – Thèse de PhD, Université Pierre et Marie Curie (Paris VI), novembre 1991.

[Ser92] Serrano (M.). – Rgc: un générateur d'analyseurs lexicaux efficaces en scheme. *In : Avancées applicatives, Actes des journées JFLA*, éd. par Queinnec (C.), pp. 235–252. – février 1992.

[Ser93] Serrano (M.). – De l'utilisation des analyses de flot de contrôle dans la compilation des langages fonctionnels. *In : Actes des journées du GDR de Programmation*, éd. par Lescanne (P.). – 1993.

[Ser94a] Serrano (M.). – *Bigloo user's manual.* – RT n° 0169, France, INRIA-Rocquencourt, décembre 1994.

[Ser94b] Serrano (M.). – Using Higher Order Control Flow Analysis When Compiling Functional Languages. *In : 6th International Symposium on Programming Language Implementation and Logique Programming*, éd. par Hermenegildo (M.) et Penjam (J.), pp. 447–449. – Madrid, Spain, septembre 1994.

[Ser95] Serrano (M.). – Control Flow Analysis: a Functional Languages Compilation Paradigm. *In : Symposium on Applied Computing (SAC '95), Special Track on Programming Languages*, p. To appear. – Nashville, Tennessee, USA, février 1995.

[Ses89] Sestoft (P.). – Replacing function parameters by global variables. *In : Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture.* pp. 39–53. – London. UK., septembre 1989.

[Sex88] Sexton (H.). – *Foreign Functions and Common Lisp.* – Rapport technique n° 1(5):11–23, Lisp Pointers, 1988.

[SH87] Steenkiste (P.A.) et Hennessy (J.). – Tags and Type Checking in LISP: Hardware and Software Approaches. *In : Architectural support for programming languages and operating systems*, pp. 50–59. – Palo Alto. CA US, 1987.

[Shi88] Shivers (O.). – Control flow analysis in scheme. *In : Conference on Programming Language Design and Implementation.* – Atlanta, Georgia, juin 1988.

[Shi91a] Shivers (O.). – *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* – Pittsburgh, PA 15213, CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, mai 1991.

[Shi91b] Shivers (O.). – The Semantics of Scheme Control-Flow Analysis. *In : Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* – Yale University, New Haven, Conn., juin 1991.

[SJ84] Saint-James (E.). – Recursion is more efficient than iteration. *In : ACM Symposium on Lisp and Functionnal Programming.* – Austin Texas, USA, 1984.

[SJ90] Saint-James (E.). – Transformations de programmes à l'exécution: puissance et efficience. *In : Actes des Journées francophones des Langages applicatifs, Bigre 69*, pp. 110–117. – 1990.

[SJ93] Saint-James (E.). – *La programmation applicative.* – Paris, France, Hermès, 1993.

[SK93] Schreiner (W.) et Kepler (J.). – *Compiling a Functionnal Language to Efficient*

SACLIB C. – Rapport technique n° FWF S5302-PHY, Research Institue for Symbolic Computation, 1993.

[Sta89] Stallman (R.M.). – *Using and Porting GNU CC.* – Rapport technique, Free Software Foundation, Inc., avril 1989.

[Ste78] Steele (G.L.). – *Rabbit: a Compiler for Scheme.* – MIT AI Memo n° 474, Cambridge, Mass., Massachusetts Institute of Technology, mai 1978.

[Ste84] Steele (G.L.). – *COMMON LISP (The language).* – Burlington MA (USA), Digital Press (DEC), 1984, 2nd édition.

[Ste91] Steenkiste (P.A.). – The Implementation of Tags and Run-Time Type Checking. *In : Topics in Advanced Language Implementation*, éd. par Lee (P.). pp. 3–24. – The MIT Press, 1991.

[Ste94] Steckler (P.A.). – *Correct Higher-Order Program Transformation.* – Doctor of philosphy, Faculty of the Graduate School of the College of Computer Science of Northeastern University, juillet 1994.

[Sto77] Stoy (J.E.). – *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* – Cambridge, Massachusetts, M.I.T Press, 1977.

[Sun87] Sun Microsystems, Inc., Mountain View, California. – *The SPARC Architecture Manual*, août 1987.

[Sun90] Sundaresan (N.). – Translation of Nested Pascal Routines to C. *SIGPLAN Notices*, vol. 25, n° 5, 1990.

[SW94] Serrano (M.) et Weis (P.). – $1 + 1 = 1$: an optimizing caml compil. *In : ACM SIGPLAN Workshop on ML and its Applications.* pp. 101–111. – Orlando (Florida, USA), 1994.

[TAL91] Tarditi (D.), Acharya (A.) et Lee (P.). – *No assembly require: Compiling Standard ML to C.* – Rapport technique n° CMU-CS-90-187, Pittsburg, Pennsylvania, School of Computer Science, Carnegie Mellon University, mars 1991.

[Tar55] Tarski (A.). – A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, vol. 5, 1955, pp. 285–309.

[Tha90] Thatte (S.). – Quasi-static Typing. *In : Proceedings of the ACM Symposium of Principles of Programming Languages*, pp. 367–381. – 1990.

[W*al.*91] Weis (P.) et *al.* – *The CAML Reference manual.* – Rapport technique n° 121, INRIA-Rocquencourt, 1991.

[WC94] Wright (A.) et Cartwright (R.). – A Practical Soft Type System for Scheme. *In : Conference Record of the ACM Conference on Lisp and Functional Programming*, pp. 250–262. – Orlando, Florida, USA, juin 1994.

[WL93] Weis (P.) et Leroy (X.). – *Le langage CAML.* – InterEditions, Paris, 1993.

[WLM92] Wilson (P.R.), Lam (M.S.) et Moher (T.G.). – Caching Considerations for Generational Garbage Collection. *In : Conference on Lisp and Functional Programming*, pp. 32–42. – 1992.

[YH88] Yuasa (T.) et Hagiya (M.). – *Kyoto Common Lisp Report.* – Rapport technique, Kyoto University, Research Institute for Mathematical Sciences, 1988.

[Zor90] Zorn (B.). – Comparing Mark-and-sweep and Stop-and-copy Garbage Collection. *In : ACM SIGPLAN '90 Conference on Lisp and Functional Programming*, pp. 87–97. – 1990.

Annexe A

La page de manuel de Bigloo

Version 1.7

NAME

bigloo v1.7 – a Scheme compiler

SYNOPSIS

bigloo [-o *output*] [-q] [-m *module-name*] [-s] [-v] [-i] [-E] [-C] [-[cA]] [-g[*level*]] [-cg] [-unsafe[*atrsv*]] [-O[*level*]] [-farithmetic] [-fshared-data] [-w] [-copt *args*] [-afile *file-name*] [-access *module-name file-name*] [-l*library*] [-call/cc] [-mklib] [-mkheap] [-heap *file-name*] [<-/+>rm] [-char <7/8>bit] [-nil] [-cc *compiler*] [-version] [-query] [-help] *input*

DESCRIPTION

Bigloo is a Scheme compiler. Scheme is defined in IEEE standard for the Scheme Programming Language but Bigloo does not entirely conform to it. The compiler produces C files which are then compiled by any Ansi C compiler to produce *.o* or *executable* files. **Bigloo** is a module compiler which means that it is allowed to compile several files and to link them together to produce an unique executable.

OPTIONS

-m *module-name*	set name of the current module.
-q	Do not load an init file.
-s	compile silently. Print only warning and error messages.
-v	be verbose, print some compilation information.
-i	interprete rather than compile the *input*. If no input file is specified then enter the read-eval-print loop.
-E	Stop the compilation after the macro expansion.
-C	Stop the compilation before the C compilation.
-[cA]	Don't link the file, only produce a *.o* file.
-g[*level*]	Put debug informations in the object code.
1	Only trace global functions.

2 Trace global and local functions.

3 Trace global and local functions and enable assertions.

-cg Produce some debug information for *.o* file.

-unsafe[*atrsv*] The compiler produces 'runtime safe test' to ensure the safety of the executable. It's possible to suppress several or all this runtime test. -unsafe suppress all of it. Here are the meaning of the partial flags:

a Suppress test on arity. This means that before an application to an unknown function the compiler won't produce a test to be sure that the function posses a correct arity in regard to the number of arguments which are provided to it.

t Suppress test on type. Don't check the type of an object before using it. For example don't test if an object is a pair before applying the 'car' function.

r Don't check the range before accessing a vector.

s Don't check that an object is a well tagged structure before accessing to a field. This option is a specialisation of the -t flag.

v Don't check version soundness.

-O[*level*] Optimize the object code. -O with the *level* omitted is equivalent to -O1. *level* is one of:

1 which means, do several optimisations like, inlining, common subexpression elimination.

2 which means, do all the optimisations performed in level 1 and do a control flow analysis.

3 which means, do all the optimisations performed in level 2 and performed a more clever control flow analysis.

4 which means, do all the optimisations performed in level 3 and performed a heap to stack transformation.

-farithmetic Suppress genericity of arithmetic operators.

-fshared-data Attempt to share constant data.

-w Suppress warning messages.

-copt *args* Invoke the C compiler with args. (args has to be a quoted string).

-cc *compiler* Use *compiler* to compile C files.

-afile *file-name* Set the 'access-file' file to be named *file-name*.

-access *module-name file-name* Add an access between *module-name* and *file-name*.

-l*library* Link with object library *library*.

-call/cc Enable the use of the 'call/cc' library function. Without this option program that uses 'call/cc' will be badly compiled. The programs that do not contain side-effect will perhaps look correct but they are not. If you want 'call/cc' use this option.

-mklib Use the compiler to compile the bigloo's librarie. This needs a compilation special mode.

-mkheap	Don't compile, produce a saved image for the compiler.
-heap *file-name*	Use *file-name* as heap file.
<-/+>rm	Remove or don't remove temporary C files
-char *<7/8>bit*	Compile in heigth bit clean mode or not.
-nil	Evaluate '() as #f in conditionnal.
-version	Print the current version of Bigloo.
-query	Print the current configuration.
-help	Give a brief description of the options.

CONFIGURATION FILE

Each **Bigloo**'s user can use a special configuration file. This file must be named '~/.bigloorc'. This file allows to modify the behaviours of the compiler. See the documentation for details.

ENVIRONMENT VARIABLES

BIGLOOHEAP	Set the environment size (in megabytes). The default value is 4.
BIGLOOLIB	The path to find the Bigloo's librarie.
BIGLOOINCLUDE	The path to find the Bigloo's include.

FILES

/.../lib/bigloo/1.7/bigloo.heap – saved heap image for the compiler.
/.../lib/bigloo/1.7/bigloo.h – definitions which are #include'd in the C code.
/.../lib/bigloo/1.7/libbigloo.a – library.
/.../lib/bigloo/1.7/libbigloo_u.a – library.
~/.bigloorc – user 'runtime-command' configuration file.
/.../.bigloorc – idem.

SEE ALSO

scc(1), K2(1), gcc(1), camloo(1)

AUTHOR

Manuel SERRANO (Inria-Rocquencourt) Manuel.Serrano@inria.fr

Annexe B

La page de manuel de Camloo

Version 0.2

NAME

camloo v0.2 – the Bigloo front end for Caml

SYNOPSIS

camloo [-help] [-only-ml] [-after-ml] [*camlc-option*] *input*

DESCRIPTION

Camloo is an extension package to compile Caml files using the Bigloo compiler. The resulting compiler can be considered as the optimizing option of the regular Caml Light compiler (camlc). Bigloo (with Camloo) has some minors differences with the Caml Light system:

. Camloo **integers** size have the C's integer size minus 2 bits.

. Camloo can't compile Caml Light modules which have no implementation.

. Camloo does not support 'let rec' definitions for values (eg. records).

OPTIONS

Camloo have not to be called directly. It is automatically invoked by Bigloo when it encounters a source file name with the *'.ml'* suffix. Alternatively the '*-extend caml*' option forces Bigloo to invoke Camloo. This is also the only way to pass arguments to Camloo: when Bigloo finds the *-extend caml* option in the command line, it invokes Camloo with the rest of the command line. Camloo arguments have the following meaning:

-help Prints all the Camloo's options.

-only-ml Produces a Scheme file but does not compile it.

-after-ml Compiles a Scheme file gained from a Caml source file.

camlc-option All the Camlc options (without the -c option)

EXAMPLES

Here are some examples of Camloo invocation:

Prints the Camloo's help message: $ bigloo -extend caml -help

Compiles the 'foo.ml' source file $ bigloo foo.ml or $ Bigloo foo.ml

Compiles the 'bar.mli' source file: $ bigloo bar.mli

BIGLOO'S OPTIONS

Bigloo options keep the same meaning with Camloo. For example, to compile a file without linking it, use the *'-A'* option:

example: $ bigloo -A foo.ml

SEE ALSO

bigloo(1), camlc(1)

AUTHOR

Manuel SERRANO, Pierre WEIS (Inria-Rocquencourt) {Manuel.Serrano,Pierre.Weis}@inria.fr

Annexe C

Le fichier bigloo.h

bigloo.h

```
/*=====================================================================*/
/*    serrano/these/src/bigloo.h ...                                   */
/*    -------------------------------------------------------------   */
/*    Author      :  Manuel Serrano                                    */
/*    Creation    :  Wed Sep  1 17:15:00 1993                          */
/*    Last change :  Wed Nov  2 08:22:34 1994 (serrano)                */
/*    -------------------------------------------------------------   */
/*    Les choses de 'Bigloo'                                           */
/*=====================================================================*/
#ifndef BIGLOO_H
#define BIGLOO_H

#if defined( NULL )
#   undef NULL
#endif

/*---------------------------------------------------------------------*/
/*    Les includes indispensables                                      */
/*---------------------------------------------------------------------*/
#include <stdio.h>
#include <setjmp.h>
#include <errno.h>
#if( defined( sun ) && defined( sparc ) )
#   include <stdlib.h>
#endif
#include <math.h>
#if defined( sony_news )
#   include <news/machparam.h>
#endif
#include <limits.h>

/*---------------------------------------------------------------------*/
/*    Les sites ou sont ranges les libraries et les includes           */
/*---------------------------------------------------------------------*/
#define LIBRARY_DIR() "/home/cornas/icsla/serrano/prgm/project/bigloo/lib/1.7"
#define INCLUDE_DIR() "/home/cornas/icsla/serrano/prgm/project/bigloo/lib/1.7"
#define CC() "gcc"
/*---------------------------------------------------------------------*/
/*    !!! WARNING !!! WARNING !!! WARNING !!! WARNING !!! WARNING !!!  */
/*    Attention, le trois macros ci-dessus doivent imperativement etre */
/*    ligne 35 et 37 (pour la distribution)                            */
/*---------------------------------------------------------------------*/

/*---------------------------------------------------------------------*/
/*    Le type booleen (je ne veux plus fixer que c'est unsigned char)  */
/*---------------------------------------------------------------------*/
#define bool_t int

/*---------------------------------------------------------------------*/
/*    Quelques macros 'system dependent'                               */
/*---------------------------------------------------------------------*/
#define DOWN 55
#define UP   66

/*--- Les casts pour 'longjmp'/'setjmp' -------------------------------*/
#define JMP_BUF void
#define JMP_VAL int

/*--- setjmp/_setjmp en fonction des signaux --------------------------*/
#define SIGNALS_WANTED

#if( defined( SIGNALS_WANTED ) )
#   define _setjmp setjmp
#   define _longjmp longjmp
#endif

/*--- La configuration machine dependente -----------------------------*/
#if defined( SPARC ) || defined( sparc )
#   if( !defined( SPARC ) )
#      define SPARC
#   endif
#   include <sys/signal.h>
#   define STACK_GROWS DOWN
#   define PTR_ALIGNMENT 2
    /* Le nombre de fenetre de registre sur sparc */
#   define BIG_ENDIAN
#   if defined( __svr4__ )
#      include <sys/regset.h>
#      define SETJMP setjmp
#      define LONGJMP longjmp
#   else
#      define SETJMP _setjmp
#      define LONGJMP _longjmp
#   endif
#   define NB_WINDOW_REGISTER SPARC_MAXREGWINDOW
#else
#   if( defined( PYR ) || defined( pyr ) \
        || (defined( sony_news ) && defined( r3000 )) )
#      define STACK_GROWS DOWN
#      define PTR_ALIGNMENT 2
#      define BIG_ENDIAN
#      define SETJMP _setjmp
#      define LONGJMP _longjmp
#   else
#      if( defined( i386 ) )
#         define STACK_GROWS DOWN
#         define PTR_ALIGNMENT 2
#         define LITTLE_ENDIAN
#         undef SIGBUS
#         define SIGBUS SIGUSR1
#         define SETJMP _setjmp
#         define LONGJMP _longjmp
#      else
#         if( defined( __pa_risc ) ||  \
              defined(_PA_RISC1_0) ||  \
              defined(_PA_RISC1_1) )
#            define STACK_GROWS UP
#            define PTR_ALIGNMENT 2
#            define BIG_ENDIAN
#            define SIGBUS _SIGBUS
#            define SETJMP _setjmp
#            define LONGJMP _longjmp
#         else
#            if( defined( sun ) && defined( mc68000 ) )
#               define STACK_GROWS DOWN
#               define PTR_ALIGNMENT 2
#               define BIG_ENDIAN
#               define SETJMP _setjmp
#               define LONGJMP _longjmp
#            else
                /* thank's to Drew Whitehouse [Drew.Whitehouse@anu.edu.au] */
#               if( defined( sgi) || defined( ultrix ) && defined( mips ) )
#                  define STACK_GROWS DOWN
#                  define PTR_ALIGNMENT 2
#                  if defined( ultrix ) && defined( mips )
#                     define LITTLE_ENDIAN
#                  else
#                     define BIG_ENDIAN
#                  endif
#                  define SETJMP _setjmp
#                  define LONGJMP _longjmp
#               else
#                  if( defined( _IBMR2 ) )
#                     define STACK_GROWS DOWN
#                     define PTR_ALIGNMENT 2
#                     define SETJMP setjmp
```

```
#                   define LONGJMP longjmp
#                else
#                   if( defined( NeXT ) && defined( mc68000 ) )
#                      define STACK_GROWS DOWN
#                      define PTR_ALIGNMENT 2
#                      define BIG_ENDIAN
#                      define SETJMP setjmp
#                      define LONGJMP longjmp
#                   else
                       --> error "I need to know the way the c-stack grows,
                                  see 'public/grows.c'"
#                   endif
#                endif
#             endif
#          endif
#       endif
#    endif
#  endif
#endif

#if( !defined( NB_WINDOW_REGISTER ) )
#  define NB_WINDOW_REGISTER 0
#endif

/*---------------------------------------------------------------------*/
/*    Quelques messages d'erreur personnel.                            */
/*---------------------------------------------------------------------*/
#define EHEAP       500  /* Pas assez de memoire pour allouer le tas   */
#define EMEMORY     501  /* Plus assez de place dans le tas            */
#define ETARGS      502  /* on passe trop d'args a apply               */

/*---------------------------------------------------------------------*/
/*    Les macros du GC ...                                             */
/*---------------------------------------------------------------------*/
#define NO_GC           1
#define BOEHM_GC        2

#define GC BOEHM_GC

/*---------------------------------------------------------------------*/
/*    Il y a plusieurs formes d'objets:                                */
/*    Les objets allouees:                                             */
/*           +--------+--------+--------+--------+                    */
/*           |....signed fixed point value.....??|                    */
/*           +--------+--------+--------+--------+                    */
/*                                                                     */
/*    Les objets immediats 30 bits:                                    */
/*           +--------+--------+--------+--------+                    */
/*           |....signed fixed point value.....??|                    */
/*           +--------+--------+--------+--------+                    */
/*                                                                     */
/*    Les objets immediats 6 bits:                                     */
/*           +--------+--------+--------+--------+                    */
/*           |........................|xxxxxx??|                    */
/*           +--------+--------+--------+--------+                    */
/*                                                                     */
/*    Les objets immediats 8 bits:                                     */
/*           +--------+--------+--------+--------+                    */
/*           |.................|xxxxxxxx|......??|                    */
/*           +--------+--------+--------+--------+                    */
/*                                                                     */
/*---------------------------------------------------------------------*/

/*---------------------------------------------------------------------*/
/*    Ou sont les 'tags' et quel 'mask' cela represente.               */
/*---------------------------------------------------------------------*/
#define TAG_SHIFT      PTR_ALIGNMENT
#define TAG_MASK       ((1 << PTR_ALIGNMENT) - 1)

/*---------------------------------------------------------------------*/
/*    Les 'tags' des pointeurs ...                                     */
/*---------------------------------------------------------------------*/
#if( GC == BOEHM_GC )
#  define TAG_INT       1     /*  Les integer sont tagues  ....01     */
#  define TAG_CNST      2     /*  Les cnsts sont taguees   ....10     */
#  define TAG_STRUCT    0     /*  Les pointer sont tagues  ....00     */
#  define TAG_PAIR      3     /*  Les pairs sont taguees   ....11     */
#else
#  if( GC == NO_GC )
#     define TAG_INT    0     /*  Les integer sont tagues  ....00     */
#     define TAG_CNST   2     /*  Les cnsts sont taguees   ....10     */
#     define TAG_STRUCT 1     /*  Les pointer sont tagues  ....01     */
#     define TAG_PAIR   3     /*  Les pairs sont taguees   ....11     */
#  else
       --> error "Unknown garbage collector type"
#  endif
#endif

/*---------------------------------------------------------------------*/
/*    La taille de la table hashage                                    */
/*---------------------------------------------------------------------*/
#define HASH_TABLE_SIZE_SHIFT 12
#define HASH_TABLE_SIZE       (1 << HASH_TABLE_SIZE_SHIFT)

/*---------------------------------------------------------------------*/
/*    Les 'header' des structures ...                                  */
/*---------------------------------------------------------------------*/
#define HEADER_PAIR                 ((header_t)BINT( 0 ))
#define HEADER_STRING               ((header_t)BINT( 1 ))
#define HEADER_VECTOR               ((header_t)BINT( 2 ))
#define HEADER_PROCEDURE            ((header_t)BINT( 3 ))
#define HEADER_TRUE                 ((header_t)BINT( 4 ))
#define HEADER_FALSE                ((header_t)BINT( 5 ))
#define HEADER_UNSPEC               ((header_t)BINT( 6 ))
#define HEADER_NIL                  ((header_t)BINT( 7 ))
#define HEADER_SYMBOL               ((header_t)BINT( 8 ))
#define HEADER_STACK                ((header_t)BINT( 9 ))
#define HEADER_INPUT_PORT           ((header_t)BINT( 10 ))
#define HEADER_OUTPUT_PORT          ((header_t)BINT( 11 ))
#define HEADER_RGRAMMAR             ((header_t)BINT( 12 ))
#define HEADER_CELL                 ((header_t)BINT( 13 ))
#define HEADER_EOF                  ((header_t)BINT( 14 ))
#define HEADER_STRUCT               ((header_t)BINT( 15 ))
#define HEADER_REAL                 ((header_t)BINT( 16 ))
#define HEADER_EOA                  ((header_t)BINT( 17 ))
#define HEADER_FOREIGN              ((header_t)BINT( 18 ))
#define HEADER_OUTPUT_STRING_PORT   ((header_t)BINT( 19 ))
#define HEADER_BINARY_PORT          ((header_t)BINT( 20 ))

/*---------------------------------------------------------------------*/
/*    Les differents objects 'Sqic' :                                  */
/*---------------------------------------------------------------------*/
typedef long  int_t;
typedef int_t header_t;

typedef union object {
   int_t              integer;   /*  Les entiers                      */

   header_t           header;    /*  Un champs un peu fictif mais     */
                                 /*  il est utile pour pouvoir acceder */
                                 /*  au header des objets sans savoir  */
                                 /*  quel est leur type. Tous les      */
                                 /*  headers sont en tete des struct   */
                                 /*  on peut donc le header global     */
                                 /*  plutot que les header locaux      */

   struct pair {                 /*  Les pairs.                       */
#if( !(defined( TAG_PAIR ) ) )
      header_t        header;    /*  Le header est facultatif, il     */
#endif
      union object   *car;       /*  depend du GC qu'on utilise.      */
      union object   *cdr;       /*  Dans tous les cas, il y a biensur */
   } pair_t;                     /*  un 'car' et un 'cdr' :-)         */

#if( defined( ALLOCATE_CONSTANT ) )
   struct boolean {              /*  Les booleens qui sont au nombre  */
      header_t        header;    /*  de 2: true et false              */
   } boolean_t;
#endif

   struct string {               /*  Les chaines de char, juste une   */
      header_t        header;    /*  longueur, la chaine C suit.      */
      union object   *length;
   } string_t;

   struct vector {               /*  Les vecteurs, un header et une   */
      header_t        header;    /*  taille (ATTENTION: sur 22 bits,  */
      union object   *length;    /*  voir la macro vector-length).    */
   } vector_t;

   struct procedure {            /*  Les fermetures                   */
      header_t        header;
      union object *(*entry)();
      union object *(*va_entry)();
      long            arity;
      void            *env;      /*  Ce champs est utilise pour etre  */
   } procedure_t;                /*  sur que l'alignement est correct. */

   struct procedure_light {      /*  Les fermetures legeres           */
      union object *(*entry)();
      void            *env;
   } procedure_light_t;

   struct procedure_extra_light {/*  Les fermetures extra-legeres:    */
      void           *env;
   } procedure_extra_light_t;

   struct symbol {               /*  Les symboles, un nom et une      */
      header_t        header;    /*  valeur                           */
      char           *name;
      union object   *cval;
   } symbol_t;

   struct output_port {          /* Les output_port                   */
      header_t        header;    /* grosso-modo cette structure       */
      FILE           *file;      /* comporte juste un file et son nom */
```

```
323         char          *name;
324      } output_port_t;
325
326      struct output_string_port {   /*  Les output_string_port           */
327         header_t        header;    /* Cette structure comporte:         */
328         char           *buffer;    /* - un buffer                       */
329         long            size;     /* - une taille                      */
330         long            offset;   /* - un offset                       */
331      } output_string_port_t;
332
333      struct input_port {           /*  Les input_port                   */
334         header_t        header;    /*  un input_port est:               */
335         union object   *class;     /*    - une classe                   */
336         char           *name;      /*    - une chaine                   */
337         FILE           *file;      /*    - un file                      */
338         union object   *bufsiz;    /*    - une taille                   */
339         union object   *eof;       /*    - un flag                      */
340         union object   *backward;  /*    - un backward                  */
341         union object   *forward;   /*    - un forward                   */
342         union object   *remember;  /*    - un souvenir                  */
343         union object   *mark;      /*    - un marqueur                  */
344         char           *annexe;    /*    - une annexe (cf grands tokens) */
345         union object   *anxsiz;    /*    - la taille de l'annexe        */
346         char           *buffer;    /*    - un buffer                    */
347      } input_port_t;
348
349      struct binary_port {          /*  Les binary_port                  */
350         header_t        header;    /*  ces ports sont constitues de:    */
351         char           *name;      /*    - un nom de fichier            */
352         FILE           *file;      /*    - un pointeur sur un fichier   */
353         bool_t          io;        /*    - 0 en entree 1 en sortie      */
354      } binary_port_t;
355
356      struct cell {                 /*  Les cellules. Ces objets sont    */
357         header_t        header;    /*  utilisees quand il y a des var   */
358         union object   *obj;       /*  capturees qui sont en plus ecrite */
359      } cell_t;
360
361      struct structure {            /*  Les structures,                  */
362         header_t        header;    /*  sont constituees de :            */
363         union object   *key;       /*                       - une cle   */
364         union object   *length;    /*                       - une long. */
365         union object   *slot;      /*                       - des slots */
366      } struct_t;
367
368      struct real {                 /*  Les nombres flottants            */
369   #if( !defined( TAG_REAL ) )
370         header_t        header;
371   #endif
372         double          real;
373      } real_t;
374
375      struct stack {                /*  Les piles de 'call/cc'           */
376         header_t        header;    /*  sont:                            */
377         union object   *self;      /*        - un ptr sur soit meme     */
378         union object   *size;      /*        - une taille               */
379         void           *stack;     /*        - un espace memoire        */
380      } stack_t;
381
382      struct foreign {              /*  Les types etrangers              */
383         header_t        header;
384         union object   *id;
385         void           *value;
386      } foreign_t;
387   } *obj_t;
388
389   typedef obj_t (*function_t)();
390
391   /*---------------------------------------------------------------------*/
392   /*    Les procedures d'allocations                                     */
393   /*---------------------------------------------------------------------*/
394   #if( GC == DELACOUR_GC )
395   #   define ALLOCATE( size )
396   #   define ALLOCATE_ATOMIC( size )
397   #   define INLINE_ALLOCATE( size )
398   #   define INLINE_ALLOCATE_ATOMIC( size )
399   #   define INIT_ALLOCATION() 1
400   #   define FREE_ALLOCATION() 1
401   #else
402   #   if( GC == BOEHM_GC )
403   #      if( !defined( GC_PRIVATE_H ) )
404             extern obj_t GC_malloc();
405             extern obj_t GC_malloc_atomic();
406   #      endif
407   #      if( defined( GC_DEBUG ) )
408   #         if( !defined( GC_PRIVATE_H ) )
409                extern obj_t GC_debug_malloc();
410                extern obj_t GC_debug_malloc_atomic();
411   #         endif
412   #         define GC_malloc         GC_debug_malloc
413   #         define GC_malloc_atomic  GC_debug_malloc_atomic
414   #      endif
415   #      define ALLOCATE( size ) (obj_t)(GC_malloc( size ))
416   #      define ALLOCATE_ATOMIC( size ) GC_malloc_atomic( size )
417   #      define INLINE_ALLOCATE( size ) (obj_t)(GC_malloc( size ))
418   #      define INLINE_ALLOCATE_ATOMIC( size ) GC_malloc_atomic( size )
419   #      if( (TAG_STRUCT != 0) && (TAG_PAIR != 0) )
420   #         define INIT_ALLOCATION( size )                  \
421                ( GC_init(),                                 \
422                  GC_expand_hp( size ),                      \
423                  GC_register_displacement( TAG_STRUCT ),    \
424                  GC_register_displacement( TAG_PAIR ),      \
425                  1 )
426   #      else
427   #         if( TAG_STRUCT != 0 )
428   #            define INIT_ALLOCATION( size )               \
429                   ( GC_init(),                              \
430                     GC_expand_hp( size ),                   \
431                     GC_register_displacement( TAG_STRUCT ), \
432                     1 )
433   #         else
434   #            if( TAG_PAIR != 0 )
435   #               define INIT_ALLOCATION( size )            \
436                      ( GC_init(),                           \
437                        GC_expand_hp( size ),                \
438                        GC_register_displacement( TAG_PAIR ), \
439                        1 )
440   #            else
441                  define INIT_ALLOCATION( size )            \
442                     ( GC_init(), GC_expand_hp( size ) )
443   #            endif
444   #         endif
445   #      endif
446   #      define FREE_ALLOCATION();
447   #   else
448   #      if( GC == NO_GC )
449             extern obj_t heap_alloc();
450   #         define ALLOCATE( size ) heap_alloc( size )
451   #         define ALLOCATE_ATOMIC( size ) ALLOCATE( size )
452   #         define INLINE_ALLOCATE( size ) ALLOCATE( size )
453   #         define INLINE_ALLOCATE_ATOMIC( size ) ALLOCATE( size )
454   #         define INIT_ALLOCATION( size ) init_heap( size )
455   #         define FREE_ALLOCATION() free_heap()
456   #      else
457             --> error "Unknown garbage collector type"
458   #      endif
459   #   endif
460   #endif
461
462   /*---------------------------------------------------------------------*/
463   /*    Les macros qui servent a taguer/detaguer                         */
464   /*---------------------------------------------------------------------*/
465   #define TAG( val, shift, tag )   ((long)(((long)(val) << shift) | tag))
466   #define UNTAG( val, shift, tag ) ((long)((long)(val) >> shift))
467
468   /*---------------------------------------------------------------------*/
469   /*    Les macros de conversions utilisees par 'Sqic'                   */
470   /*    ------------------------------------------------------------     */
471   /*    Attention, il est normal que pour faire la conversion 'bigloo->c'*/
472   /*    j'utilise une soustraction et non pas un 'and'. En faisant comme */
473   /*    ca, le compilateur C peut bien optimiser les access aux          */
474   /*    differents champs.                                               */
475   /*---------------------------------------------------------------------*/
476   #define BINT( i )          (obj_t)TAG( i, TAG_SHIFT, TAG_INT )
477   #define CINT( i )          (long)UNTAG( i, TAG_SHIFT, TAG_INT )
478
479   #define BREF( r )          ((obj_t)((long)r | TAG_STRUCT))
480   #define CREF( r )          ((obj_t)((long)r - TAG_STRUCT))
481
482   #define BLIGHT( l )        BPAIR( l )
483   #define CLIGHT( l )        CPAIR( l )
484
485   #if( defined( TAG_PAIR ) )
486   #   define BPAIR( p )      ((obj_t)((long)p | TAG_PAIR))
487   #   define CPAIR( p )      ((obj_t)((long)p - TAG_PAIR))
488   #else
489   #   define BPAIR( p )      BREF( p )
490   #   define CPAIR( p )      CREF( p )
491   #endif
492
493   #if( defined( TAG_REAL ) )
494   #   define BREAL( p )      ((obj_t)((long)p | TAG_REAL))
495   #   define CREAL( p )      ((obj_t)((long)p - TAG_REAL))
496   #else
497   #   define BREAL( p )      BREF( p )
498   #   define CREAL( p )      CREF( p )
499   #endif
500
501   #define BFUN( f )          ((obj_t)(f))
502   #define CFUN( f )          ((obj_t (*)())(f))
503
504   #define BCNST( c )         (obj_t)TAG( c, TAG_SHIFT, TAG_CNST )
505   #define CCNST( c )         (long)UNTAG( c, TAG_SHIFT, TAG_CNST )
506
507   #define BCONT( c )         ((obj_t)(c))
508   #define CCONT( c )         (c)
```

```
509
510  #define TRUEP( c )         ((bool_t)(c != BFALSE))
511
512  #define BCHAR( i )         ((obj_t)((long)BCHARH + \
513                                      ((long)((bool_t)(i) << 8))))
514  #define CCHAR( i )         (char)((long)(i)>>8)
515
516  #define CTRUE              ((bool_t)1)
517  #define CFALSE             ((bool_t)0)
518
519  #define FAILURE( p, m, o ) return( the_failure( p, m, o ) )
520
521  #define FUNCTION_ADDRESS( f ) (function_t)(&(f))
522
523  /*---------------------------------------------------------------------*/
524  /*    Le 'CASTING'                                                     */
525  /*---------------------------------------------------------------------*/
526  #define CBOOL_TO_BBOOL( o ) (o ? BTRUE : BFALSE )
527  #define BBOOL_TO_CBOOL( o ) (o != BFALSE)
528
529  #define CSTRING_TO_BSTRING( s ) c_string_to_string( s )
530  #define BSTRING_TO_CSTRING( s ) ((char *)CREF( s ) + STRING_SIZE)
531
532  #define CHAR_TO_INT( c ) BINT( (unsigned char)(CCHAR( c )) )
533  #define INT_TO_CHAR( i ) BCHAR( CINT( i ) )
534
535  #define DOUBLE_TO_REAL( d ) (make_real( d ))
536  #define REAL_TO_DOUBLE( r ) (REAL( r ).real)
537
538  #define CVOID_TO_BVOID( e ) (e, BUNSPEC)
539  #define BVOID_TO_CVOID( e ) ((void)e)
540
541  /*---------------------------------------------------------------------*/
542  /*    Les 'constantes' peuvent etre soit allouees soit constantes.     */
543  /*---------------------------------------------------------------------*/
544  #if defined( ALLOCATE_CONSTANT )
545  #   define BNIL          nil_object
546  #   define BFALSE        false_object
547  #   define BTRUE         true_object
548  #   define BUNSPEC       unspec_object
549  #   define BEOF          end_of_file_object
550  #   define BEOA          end_of_argument_object
551      extern obj_t nil_object, unspec_object, end_of_file_object;
552      extern obj_t true_object, false_object;
553  #else
554  #   define BNIL          ((obj_t)BCNST( 0 ))
555  #   define BFALSE        ((obj_t)BCNST( 1 ))
556  #   define BTRUE         ((obj_t)BCNST( 2 ))
557  #   define BUNSPEC       ((obj_t)BCNST( 3 ))
558  #   define BEOF          ((obj_t)BCNST( 4 ))
559  #   define BCHARH        ((obj_t)BCNST( 5 ))
560  #   define BEOA          ((obj_t)BCNST( 6 ))
561  #endif
562
563  /*---------------------------------------------------------------------*/
564  /*    Les macros GENERALES                                             */
565  /*    -----------------------------------------------------------      */
566  /*    Les macros concernant tous les objects. On trouve ici les        */
567  /*    macros qui ne sont pas propre a une categorie d'object en        */
568  /*    particulier.                                                     */
569  /*---------------------------------------------------------------------*/
570  #define OBJ_SIZE               ((long)(sizeof( ((obj_t)0) ) ))
571  #define HEADER( o )            (CREF( o )->header)
572
573  #if( TAG_STRUCT != 0 )
574  #   define POINTERP( o )       ((((long)o) & TAG_MASK) == TAG_STRUCT)
575  #else
576  #   define POINTERP( o )       (o && ((((long)o) & TAG_MASK) == TAG_STRUCT))
577  #endif
578
579  #if( TAG_CNST != 0 )
580  #   define CNSTP( o )          ((((long)o) & TAG_MASK) == TAG_CNST)
581  #else
582  #   define CNSTP( o )          (o && ((((long)o) & TAG_MASK) == TAG_CNST))
583  #endif
584
585  #if( TAG_SHIFT <= LONG_MAX )
586  #   define BOUND_CHECK( o, v ) ((unsigned long)o < (unsigned long)v)
587  #else
588  #   define BOUND_CHECK( o, v ) (((long)o >= 0) && ((long)o < (long)v))
589  #endif
590
591  /*---------------------------------------------------------------------*/
592  /*    Les deux macros les plus dangereuses de la terre. A n'           */
593  /*    utilisez que si on sait vraiment ce qu'on fait.                  */
594  /*---------------------------------------------------------------------*/
595  #define PEEK( v, i ) (*((obj_t *)(((long)CREF( v )) + (OBJ_SIZE * CINT( i )))))
596  #define POKE( var, i, val ) (PEEK( var, i ) = val, var)
597
598  /*---------------------------------------------------------------------*/
599  /*    Il existe plusieurs procedures d'allocation. Les 'atomic'        */
600  /*    concernent les allocations qui ne contiennent pas de pointeur.   */
601  /*---------------------------------------------------------------------*/
602  #define MAKE_OBJECT( size, head, an_object )                          \
603     (an_object = ALLOCATE( size ),                                     \
604      an_object->header = head, an_object)
605
606  #define MAKE_INLINE_OBJECT( size, head, an_object )                   \
607     (an_object = INLINE_ALLOCATE( size ),                              \
608      an_object->header = head, an_object)
609
610  #define MAKE_ATOMIC_OBJECT( size, head, an_object )                   \
611     (an_object = ALLOCATE_ATOMIC( size ),                              \
612      an_object->header = head, an_object)
613
614  #define EQP( o1, o2 ) ((long)o1 == (long)o2)
615
616  #define BOOLEANP( o ) (((long)o == (long)BTRUE) || ((long)o == (long)BFALSE))
617
618  #define NOT( o ) (!o)
619
620  /*---------------------------------------------------------------------*/
621  /*    La manipulation des SYMBOLS (brrr !)                             */
622  /*---------------------------------------------------------------------*/
623  #define SYMBOLP( o ) (POINTERP( o ) && (HEADER( o ) == HEADER_SYMBOL))
624
625  #define SYMBOL( o )  (CREF( o )->symbol_t)
626
627  #define GET_HASH_NUMBER( s ) (obj_t)(get_hash_number( SYMBOL( s ).name ))
628
629  #define SYMBOL_SIZE (sizeof( ((obj_t)0)->symbol_t ))
630
631  #define SYMBOL_TO_STRING( o ) c_string_to_string( SYMBOL( o ).name )
632
633  #define GET_SYMBOL_PLIST( o ) (SYMBOL( o ).cval)
634
635  #define SET_SYMBOL_PLIST( o, v ) (GET_SYMBOL_PLIST( o ) = v)
636
637  /*---------------------------------------------------------------------*/
638  /*    le tripotage des PAIRs                                           */
639  /*---------------------------------------------------------------------*/
640  #define PAIR_SIZE (sizeof( ((obj_t)0)->pair_t ))
641
642  #define PAIR( o ) (CPAIR( o )->pair_t)
643
644  #if( GC == BOEHM_GC )
645      extern obj_t make_pair();
646
647  #   define MAKE_PAIR( a, d ) make_pair( a, d )
648
649  #   define MAKE_INLINE_PAIR( a, d ) MAKE_PAIR( a, d )
650
651  #else
652  #   if( defined( __GNUC__ ) )
653  #      define MAKE_PAIR( a, d )                                        \
654             ( { obj_t a_pair, an_object;                               \
655                 a_pair = MAKE_INLINE_OBJECT( PAIR_SIZE, HEADER_PAIR,   \
656                                              an_object );              \
657                 a_pair->pair_t.car = a;                                \
658                 a_pair->pair_t.cdr = d;                                \
659                 BPAIR( a_pair ); } )
660  #   else
661  #      define MAKE_PAIR( a, d )                                        \
662           (a_pair = MAKE_INLINE_OBJECT( PAIR_SIZE, HEADER_PAIR,        \
663                                         an_object ),                   \
664            a_pair->pair_t.car = a,                                     \
665            a_pair->pair_t.cdr = d,                                     \
666            BPAIR( a_pair ) )
667  #   endif
668  #   define MAKE_INLINE_PAIR( a, d ) MAKE_PAIR( a, d )
669  #endif
670
671  #define INIT_STACK_PAIR( p, a, d ) ( p.car = (a), p.cdr = (d), p )
672  #define SPAIR_REF( p ) BPAIR( &p )
673
674  #if( !(defined( TAG_PAIR ) ) )
675  #   define PAIRP( c ) (POINTERP( c ) && (HEADER( c ) == HEADER_PAIR))
676  #else
677  #   define PAIRP( c ) ((c && ((((long)c)&TAG_MASK) == TAG_PAIR)))
678  #endif
679
680  #define NULLP( c ) ((long)(c) == (long)BNIL)
681
682  #define CAR( c )   (PAIR( c ).car)
683  #define CDR( c )   (PAIR( c ).cdr)
684
685  #define SET_CAR( c, v ) ((CAR( c ) = v), c)
686  #define SET_CDR( c, v ) ((CDR( c ) = v), c)
687
688  /*---------------------------------------------------------------------*/
689  /*    Les CHARs                                                        */
690  /*---------------------------------------------------------------------*/
691  #define CHARP( o ) (((long)(o) & (long)(BCHARH)) == (long)BCHARH)
692
693  /* Soit pour des FILE soit pour des STRING */
694  #define WRITE_CHAR( o, p )              \
```

```
    ( OUTPUT_STRING_PORTP( p ) ?          \
        strputc( (long)CCHAR( o ), p ) :  \
        ((obj_t)(fputc( (long)CCHAR( o ), OUTPUT_PORT( p ).file )), o) )

/* la meme mais juste pour les FILE C */
#define WRITE_CHAR_F( o, p )       \
   ( (obj_t)(fputc( (long)CCHAR( o ), OUTPUT_PORT( p ).file )), o )

#define CHAR_LT( o1, o2 ) ((long)o1 < (long)o2)

#define CHAR_GT( o1, o2 ) ((long)o1 > (long)o2)

#define CHAR_LE( o1, o2 ) ((long)o1 <= (long)o2)

#define CHAR_GE( o1, o2 ) ((long)o1 >= (long)o2)

extern int toupper(), tolower();

#define CHAR_UPCASE( o )   BCHAR( toupper( CCHAR( o ) ) )
#define CHAR_DOWNCASE( o ) BCHAR( tolower( CCHAR( o ) ) )

/*---------------------------------------------------------------------*/
/*      Les STRINGs de caracteres                                      */
/*---------------------------------------------------------------------*/
#define STRINGP( o ) (POINTERP( o ) && (HEADER( o ) == HEADER_STRING))

#define STRING( o )  (CREF( o )->string_t)

#define STRING_SIZE (sizeof( ((obj_t)0)->string_t))

#define STRING_LENGTH( s )  STRING( s ).length

#define STRING_REF( v, i ) BSTRING_TO_CSTRING( v )[ CINT( i ) ]

#define STRING_SET( s, i, c ) \
   ((STRING_REF( s, i ) = CCHAR( c )), s )

#define STRING_COPY( s )  \
   (c_string_to_string((char *)(CREF( s ) + STRING_SIZE)))

/*---------------------------------------------------------------------*/
/*    Les macros concernant les VECTORs                                */
/*---------------------------------------------------------------------*/
#define VECTOR_SIZE (sizeof( ((obj_t)0)->vector_t))
#define VECTOR_LENGTH_SHIFT 24
#define VECTOR_LENGTH_MASK (~(unsigned long)(0xff << VECTOR_LENGTH_SHIFT))

#define VECTOR( o ) CREF( o )->vector_t

#define VECTORP( o ) (POINTERP( o ) && (HEADER( o ) == HEADER_VECTOR))

#if( (PTR_ALIGNMENT == TAG_SHIFT) )
#   define VECTOR_REF( v, i ) \
        (*((obj_t *)((long)CREF( v ) + (VECTOR_SIZE - TAG_INT) + ((long)i))))
#else
#   define VECTOR_REF( v, i ) \
        (*((obj_t *)(((long)CREF( v )) + VECTOR_SIZE + (OBJ_SIZE * CINT( i )))))
#endif

#define VECTOR_SET( v, i, o ) ((VECTOR_REF( v, i ) = o), v)

/*--- Les vecteurs alloues en pile ---------------------------------*/
#define svector_t( n )                                  \
   struct {                                             \
      header_t header;                                  \
      obj_t    length;                                  \
      obj_t    elements[ n ];                           \
   }

#define INIT_STACK_INITIALIZED_VECTOR( vec, klen, init )          \
   ( INIT_STACK_VECTOR( vec, klen ),                              \
     fill_vector_with_init( &vec, CINT( klen ), init ),           \
     vec )

#define INIT_STACK_VECTOR( vec, klen )                            \
   ( vec.header = HEADER_VECTOR,                                  \
     vec.length = klen,                                           \
     vec )

#define SVECTOR_REF( vec ) BREF( &vec )

/*---------------------------------------------------------------------*/
/*    !!! WARNING !!! WARNING !!! WARNING !!! WARNING !!! WARNING !!!  */
/*    -----------------------------------------------------------    */
/*    Pour pouvoir coder une information supplementaire sur les        */
/*    vecteurs, je limite leur taille a 2^22. Cela signifie que        */
/*    l'info peut-etre codee sur les 8 bits de poids fort.             */
/*---------------------------------------------------------------------*/
#define VECTOR_LENGTH( v ) \
   ((obj_t)((unsigned long)(VECTOR( v ).length) & VECTOR_LENGTH_MASK))

#define VECTOR_TAG_SET( v, tag )              \
    ( VECTOR( v ).length =                    \
        (obj_t)((long)VECTOR_LENGTH( v ) |       \
             (((long)CINT( tag )) << VECTOR_LENGTH_SHIFT)) )

#define VECTOR_TAG( v )                                        \
      BINT( ((obj_t)(((unsigned long)(VECTOR( v ).length) &  \
           ~VECTOR_LENGTH_MASK) >> VECTOR_LENGTH_SHIFT)) )

/*---------------------------------------------------------------------*/
/*    L'ARITHMETIQUE qui comme elle se doit est divisee en deux        */
/*    parties, l'ARITHMETIQUE ENTIERE et l'ARITHMETIQUE FLOTANTE. En   */
/*    plus il y a quelques macros valables pour les deux arithmetiques.*/
/*---------------------------------------------------------------------*/
/*--- l'ARITHMETIQUE ENTIERE --------------------------------------------*/
#define INTEGERP( o ) ((((long)o) & TAG_MASK) == TAG_INT)

#if( !TAG_INT )
#   define ADD_I( a, b ) ((obj_t)((long)( a ) + (long)( b )))
#   define SUB_I( a, b ) ((obj_t)((long)( a ) - (long)( b )))
#   define MUL_I( a, b ) ((obj_t)((CINT( a ) * (long)b)))
#   define DIV_I( a, b ) (obj_t)(BINT( CINT( a )/CINT( b ) ))
#   define ADD_I_PTAG( a, b ) (ADD_I( a, b ))
#   define SUB_I_PTAG( a, b ) (SUB_I( a, b ))
#   define PSUB_TAG( a ) BINT( a )
#   define PADD_TAG( a ) BINT( a )
#else
#   define ADD_I( a, b ) ((obj_t)(((long)( a ) - TAG_INT) + (long)( b )))
#   define SUB_I( a, b ) ((obj_t)(((long)( a ) - (long)( b )) | TAG_INT))
#   define MUL_I( a, b ) (BINT( (CINT( a )    * CINT( b ) )))
#   define DIV_I( a, b ) (BINT( (CINT( a )    / CINT( b ) )))
#   define ADD_I_PTAG( a, b ) ((obj_t)((long)(a) + (long)(b)))
#   define SUB_I_PTAG( a, b ) ((obj_t)((long)(a) - (long)(b)))
#   define PADD_TAG( a ) ((obj_t)((long)BINT( a ) + (long)TAG_INT))
#   define PSUB_TAG( a ) ((obj_t)((long)BINT( a ) - (long)TAG_INT))
#endif

#define EQ_I( x, y ) (((long)x) == ((long)y))
#define LT_I( x, y ) (((long)x) <  ((long)y))
#define LE_I( x, y ) (((long)x) <= ((long)y))
#define GT_I( x, y ) (((long)x) >  ((long)y))
#define GE_I( x, y ) (((long)x) >= ((long)y))

#define NEG_I( x ) (BINT( -CINT( x )))
#define ABS_I( x ) (LT_I( x, BINT(  0 )) ? BINT( -CINT( x ) ) : x)

#define BITOR( x, y )   (obj_t)(((long)x) | ((long)y))
#define BITAND( x, y )  (obj_t)(((long)x) & ((long)y))
#define BITXOR( x, y )  BINT( CINT( x ) ^ CINT( y ) )
#define BITNOT( x )     BINT( ~CINT(x) )
#define BITLSH( x, y )  BINT( CINT(x) << CINT(y) )
#define BITRSH( x, y )  BINT( CINT(x) >> CINT(y) )
#define BITURSH( x, y ) BINT((unsigned long)CINT(x) >> (unsigned long)CINT(y))

#define REMAINDER_I( a, b ) (BINT( (CINT( a ) % CINT( b )) ))
#define QUOTIENT_I( x, y ) DIV_I( x, y )

#define ODDP_I( x )  (CINT( x ) & 0x1)
#define EVENP_I( x ) (!ODDP_I( x ))

/*--- l'ARITHMETIQUE FLOTANTE -------------------------------------------*/
#define REAL_SIZE (sizeof( ((obj_t)0)->real_t ))

#if( !defined( TAG_REAL ) )
#   define REALP( o ) (POINTERP( o ) && (HEADER( o ) == HEADER_REAL))
#endif

#define REAL( o ) CREAL( o )->real_t

#define ADD_R( a, b )  (REAL( a ).real + REAL( b ).real)
#define SUB_R( a, b )  (REAL( a ).real - REAL( b ).real)
#define MUL_R( a, b )  (REAL( a ).real * REAL( b ).real)
#define DIV_R( a, b )  (REAL( a ).real / REAL( b ).real)
#define NEG_R( a )     (-REAL( a ).real)

#define EQ_R( x, y )   (REAL( x ).real) == (REAL( y ).real)
#define LT_R( x, y )   (REAL( x ).real) < (REAL( y ).real)
#define LE_R( x, y )   (REAL( x ).real) <= (REAL( y ).real)
#define GT_R( x, y )   (REAL( x ).real) > (REAL( y ).real)
#define GE_R( x, y )   (REAL( x ).real) >= (REAL( y ).real)

#define ZEROP_R( x )   (REAL( x ).real) == (double)(0.0)
#define POSITIVEP_R(x) (REAL( x ).real) > (double)(0.0)
#define NEGATIVEP_R(x) (REAL( x ).real) < (double)(0.0)

#define ABS_R( x )     (REAL( x ).real < 0.0 ?            \
                          make_real( -REAL( x ).real ) :  \
                          x)
#define ROUND_R( x )   (floor( x + 0.5 ))

#define SREAL_REF( r ) BREAL( &r )

#if( defined( TAG_REAL ) )
#   define INIT_STACK_REAL( r, v ) ( r.real = (double)(v), r )
#else
```

```
#   define INIT_STACK_REAL( r, v ) \
        ( r.header = HEADER_REAL, r.real = (double)(v), r )
#endif

/*--- Les fonctions de converstions arithmetiques ---------------------*/
#define INT_TO_REAL(x) (make_real( (double)(CINT( x )) ))
#define REAL_TO_INT(x) (BINT( (long)(REAL( x ).real) ))

/*----------------------------------------------------------------------*/
/*    La manipulation des PROCEDUREs                                    */
/*----------------------------------------------------------------------*/
#define PROCEDURE_SIZE (sizeof( ((obj_t)0)->procedure_t))

#define PROCEDURE( o ) CREF( o )->procedure_t

#define PROCEDURE_ENTRY( fun ) (obj_t)(PROCEDURE( fun ).entry)
#define PROCEDURE_VA_ENTRY( fun ) (obj_t)(PROCEDURE( fun ).va_entry)

#define PROCEDUREP( fun ) \
   (POINTERP( fun ) && (HEADER( fun ) == HEADER_PROCEDURE))

#define PROCEDURE_ARITY( fun ) (PROCEDURE( fun ).arity)

#define VA_PROCEDUREP( fun ) ( PROCEDURE_ARITY( fun ) < 0 )

#define PROCEDURE_CORRECT_ARITYP( fun, num )      \
   ( (PROCEDURE_ARITY( fun ) == num) ||           \
     (VA_PROCEDUREP( fun ) &&                     \
       ((-num - 1) <= (PROCEDURE_ARITY( fun )))) )

#define PROCEDURE_ENV_REF( p, i ) \
   (*((obj_t *)(((long)&(PROCEDURE( p ).env)) + (OBJ_SIZE * i))))

#define PROCEDURE_ENV_SET( p, i, o ) ((PROCEDURE_ENV_REF( p, i ) = o), p)

/*--- Les procedures allouees en pile ---------------------------------*/
#define sprocedure_t( n )                                   \
   struct {                                                 \
      header_t header;                                      \
      obj_t    (*entry)();                                  \
      obj_t    (*va_entry)();                               \
      long     arity;                                       \
      obj_t    elements[ n ];                               \
   }

#define INIT_STACK_FX_PROCEDURE( proc, a_entry, a_arity, size )   \
   ( proc.header = HEADER_PROCEDURE,                              \
     proc.entry  = a_entry,                                       \
     proc.arity  = a_arity,                                       \
     proc )

#define SPROCEDURE_REF( proc ) BREF( &proc )

/*--- Les procedures 'light' allouees en pile -------------------------*/
#define sprocedure_light_t( n )                             \
   struct {                                                 \
      obj_t    (*entry)();                                  \
      obj_t    elements[ n ];                               \
   }

#define INIT_STACK_LIGHT_PROCEDURE( proc, a_entry, size )         \
   ( proc.entry  = a_entry,                                       \
     proc )

#define SPROCEDURE_LIGHT_REF( proc ) BLIGHT( &proc )

/*----------------------------------------------------------------------*/
/*    La manipulation des PROCEDURE_LIGHTs                              */
/*----------------------------------------------------------------------*/
#define PROCEDURE_LIGHT_SIZE \
   (sizeof( ((obj_t)0)->procedure_light_t ))

#define PROCEDURE_LIGHTP( o ) PAIRP( o )

#define PROCEDURE_LIGHT( o ) CLIGHT( o )->procedure_light_t

#define PROCEDURE_LIGHT_ENTRY( fun ) (obj_t)(PROCEDURE_LIGHT( fun ).entry)

#define PROCEDURE_LIGHT_ENV_REF( p, i ) \
   (*((obj_t *)(((long)&(PROCEDURE_LIGHT( p ).env)) + (OBJ_SIZE * i))))

#define PROCEDURE_LIGHT_ENV_SET( p, i, o ) \
   ((PROCEDURE_LIGHT_ENV_REF( p, i ) = o), p)

/*----------------------------------------------------------------------*/
/*    La manipulation des PROCEDURE_EXTRA_LIGHTs                        */
/*----------------------------------------------------------------------*/
#define PROCEDURE_EXTRA_LIGHT_SIZE \
   (sizeof( ((obj_t)0)->procedure_extra_light_t ))

#define PROCEDURE_EXTRA_LIGHT( o ) (o->procedure_extra_light_t)

#define PROCEDURE_EXTRA_LIGHT_ENTRY( fun ) \
   (obj_t)(PROCEDURE_EXTRA_LIGHT( fun ).entry)

#define PROCEDURE_EXTRA_LIGHT_ENV_REF( p, i ) \
   (*((obj_t *)(((long)&(PROCEDURE_EXTRA_LIGHT( p ).env)) + (OBJ_SIZE * i))))

#define PROCEDURE_EXTRA_LIGHT_ENV_SET( p, i, o ) \
   ((PROCEDURE_EXTRA_LIGHT_ENV_REF( p, i ) = o), p)

/*----------------------------------------------------------------------*/
/*      Les CELLules                                                    */
/*----------------------------------------------------------------------*/
#define CELL_SIZE (sizeof( ((obj_t)0)->cell_t))

#define CELL( o ) CREF( o )->cell_t

#define CELLP( o ) \
   ( POINTERP( o ) && (HEADER( o ) == HEADER_CELL) )

#if( defined( __GNUC__ ) )
#   define MAKE_CELL( v )                                  \
      ( { obj_t a_cell;                                    \
          MAKE_OBJECT( CELL_SIZE, HEADER_CELL, a_cell );   \
          a_cell->cell_t.obj = (obj_t)(v),                 \
          BREF( a_cell ); } )
#else
#   define MAKE_CELL( v )                                  \
      ( MAKE_OBJECT( CELL_SIZE, HEADER_CELL, a_cell ),     \
        a_cell->cell_t.obj = (obj_t)(v),                   \
        BREF( a_cell ) )
#endif

#define CELL_SET( c, v ) ((CELL( c ).obj = v), c)

#define CELL_REF( c )    (CELL( c ).obj)

/*----------------------------------------------------------------------*/
/*    Les macros d'acces aux OUTPUT_PORTs                               */
/*----------------------------------------------------------------------*/
#define OUTPUT_PORT_SIZE (sizeof( ((obj_t)0)->output_port_t ))

#define OUTPUT_PORT( o ) CREF( o )->output_port_t

#define OUTPUT_PORTP( o ) \
   ( POINTERP( o ) && ( (HEADER( o ) == HEADER_OUTPUT_PORT) || \
                        (HEADER( o ) == HEADER_OUTPUT_STRING_PORT) ) )

#define FLUSH_OUTPUT_PORT( o )  \
   ( OUTPUT_STRING_PORTP( o ) ? \
     strport_flush( o ) :       \
     ((fflush( OUTPUT_PORT( o ).file )), o) )

#define BOUTPUT_PORT_TO_CFILE( o )                                  \
   ( OUTPUT_STRING_PORTP( o ) ?                                     \
     FAILURE( c_string_to_string( "output-port-to-file" ),          \
              c_string_to_string( "argument can't be a string port"), \
              o )                                                   \
     : OUTPUT_PORT( o ).file )

#define CFILE_TO_BOUTPUT_PORT( f ) (make_output_port( "<c-port>", f))

/*----------------------------------------------------------------------*/
/*    Les OUTPUT_STRING_PORTs                                           */
/*----------------------------------------------------------------------*/
#define OUTPUT_STRING_PORT_SIZE (sizeof( ((obj_t)0)->output_string_port_t ))

#define OUTPUT_STRING_PORT( o ) CREF( o )->output_string_port_t

#define OUTPUT_STRING_PORTP( o ) \
   ( HEADER( o ) == HEADER_OUTPUT_STRING_PORT )

#define OUTPUT_STRING_PORT_BUFFER_SIZE 1024

#define END_OF_STRING_PORTP( o ) \
   ( OUTPUT_STRING_PORT( o ).offset == OUTPUT_STRING_PORT( o ).size )

/*----------------------------------------------------------------------*/
/*    Les macros d'acces et de tripotages des ports binaires.           */
/*----------------------------------------------------------------------*/
#define BINARY_PORT_SIZE (sizeof( ((obj_t)0)->binary_port_t ))

#define BINARY_PORT( o ) CREF( o )->binary_port_t

#define BINARY_PORTP( o ) \
   ( POINTERP( o ) && (HEADER( o ) == HEADER_BINARY_PORT) )

#define BINARY_PORT_IN   ((bool_t)0)
#define BINARY_PORT_OUT  ((bool_t)1)

#define BINARY_PORT_INP( p ) (BINARY_PORT( o ).io == BINARY_PORT_IN)

/*----------------------------------------------------------------------*/
/*    Les macros d'acces aux INPUT_PORTs                                */
/*----------------------------------------------------------------------*/
```

```
#define CLASS_FILE    BINT( 0 )
#define CLASS_CONSOLE BINT( 1 )
#define CLASS_STRING  BINT( 2 )

#define INPUT_PORT_SIZE (sizeof( ((obj_t)0)->input_port_t ))

#define INPUT_PORT( o ) CREF( o )->input_port_t

#define INPUT_PORTP( o ) \
    (POINTERP( o ) && (HEADER( o ) == HEADER_INPUT_PORT))

#define BUFFER( p ) ((unsigned char *)&(INPUT_PORT( p ).buffer))

#define EOF_OBJECTP( o ) ( o == BEOF )

/*--- Les macros de lecture ----------------------------------------*/
#define INPUT_PORT_READ_CHAR( p )                                   \
   (  INPUT_PORT( p ).forward = ADD_I_PTAG( INPUT_PORT( p ).forward,        \
                                            PSUB_TAG( 1 ) ),                \
     (long)((long)BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward ) - 1 ] ) )

#define READ_CHAR( p )                                              \
   ( INPUT_PORT( p ).backward = INPUT_PORT( p ).forward,                    \
     INPUT_PORT( p ).forward = ADD_I_PTAG( INPUT_PORT( p ).forward,         \
                                           PSUB_TAG( 1 ) ),                 \
     INPUT_PORT_REMEMBER_REF( p ),                                          \
     BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward )-1 ] ?               \
     BCHAR( BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward )-1 ] ) :      \
        !input_port_fill_buffer( p )?                                       \
           INPUT_PORT( p ).forward =                                        \
             SUB_I_PTAG( INPUT_PORT( p ).forward, PSUB_TAG( 1 ) ),          \
           reset_eof( p ),                                                  \
           BEOF :                                                           \
           ( INPUT_PORT( p ).forward =                                      \
               ADD_I_PTAG( INPUT_PORT( p ).forward, PSUB_TAG( 1 ) ),        \
             BCHAR( BUFFER( p )[ (long)CINT( INPUT_PORT( p ).backward ) ] ) ) )

#define PEEK_CHAR( p )                                              \
   ( INPUT_PORT( p ).backward = INPUT_PORT( p ).forward,                    \
     BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward ) ] ?                 \
      BCHAR( BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward )] ) :        \
       (!input_port_fill_buffer( p )) ?                                     \
          BEOF :                                                            \
          (INPUT_PORT( p ).forward =                                        \
             ADD_I_PTAG( INPUT_PORT( p ).forward, PSUB_TAG( 1 ) ),          \
          BCHAR( BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward ) ] ) ))

#define INPUT_PORT_THROW_CHAR( p, n )                               \
   INPUT_PORT( p ).backward = ADD_I( INPUT_PORT( p ).backward, n )

#define INPUT_PORT_REMEMBER_REF( p )                                \
   INPUT_PORT( p ).remember = INPUT_PORT( p ).forward

#define INPUT_PORT_REMEMBER_BACK_REF( p )                           \
   INPUT_PORT( p ).remember = SUB_I_PTAG( INPUT_PORT( p ).forward,          \
                                          PSUB_TAG( 1 ) )

#define INPUT_PORT_EOFP( p )                                        \
   ( (INPUT_PORT( p ).eof &&                                                \
      (BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward ) ] == '\0')) ?     \
     BTRUE : BFALSE )

#define INPUT_PORT_EOLP( p )                                        \
   ( (BUFFER( p )[ (long)CINT( INPUT_PORT( p ).forward ) ] == '\n') ?       \
      BTRUE : BFALSE )

#define INPUT_PORT_BOLP( p )                                        \
   ( (EQ_I( INPUT_PORT( p ).backward, BINT( 0 ) ) ||                        \
      (BUFFER( p )[ (long)CINT( INPUT_PORT( p ).backward ) - 1 ] == '\n')) ?\
     BTRUE : BFALSE )

#define INPUT_PORT_GET_LENGTH( p )                                  \
   ( INPUT_PORT( p ).annexe ? ADD_I( INPUT_PORT( p ).anxsiz,                \
                                  SUB_I( INPUT_PORT( p ).backward,          \
                                         INPUT_PORT( p ).mark) )            \
                            : SUB_I( INPUT_PORT( p ).backward,              \
                                     INPUT_PORT( p ).mark) )

#define INPUT_PORT_STOLE_CHAR( p )                                  \
   ( INPUT_PORT( p ).backward = ADD_I_PTAG( INPUT_PORT( p ).backward,       \
                                            PSUB_TAG( 1 ) ),                \
     INPUT_PORT( p ).forward = INPUT_PORT( p ).remember =                   \
     INPUT_PORT( p ).backward,                                              \
      BUFFER( p )[ (long)CINT( INPUT_PORT( p ).remember ) - 1 ] ?           \
        BCHAR( BUFFER( p )[ (long)CINT( INPUT_PORT( p ).remember )-1 ] ) :  \
        BEOF )

#define INPUT_PORT_AJUST_CURSOR( p )                                \
   ( INPUT_PORT( p ).forward  = INPUT_PORT( p ).remember,                   \
     INPUT_PORT( p ).mark     = INPUT_PORT( p ).backward,                   \
     INPUT_PORT( p ).backward = INPUT_PORT( p ).forward )                   \

#define INPUT_PORT_TOKEN_TOO_LARGEP( p ) (INPUT_PORT( p ).annexe != 0L)


/*---------------------------------------------------------------------*/
/*    Les STRUCtures                                                   */
/*---------------------------------------------------------------------*/
#define STRUCT_SIZE (sizeof( ((obj_t)0)->struct_t))

#define STRUCT( o ) CREF( o )->struct_t

#define STRUCTP( c ) (POINTERP( c ) && (HEADER( c ) == HEADER_STRUCT))

#define STRUCT_KEY( c ) STRUCT( c ).key

#define STRUCT_LENGTH( c ) STRUCT( c ).length

#define STRUCT_SLOT_REF( c, i )                                     \
   (*((obj_t *)(((long)&(STRUCT( c ).slot)) + (OBJ_SIZE * CINT( i ) ))))

#define STRUCT_SLOT_SET( c, i, o ) ((STRUCT_SLOT_REF( c, i ) = o), c)

/*---------------------------------------------------------------------*/
/*    Les 'STACK' (cf. call/cc)                                        */
/*---------------------------------------------------------------------*/
#define STACK_SIZE  (sizeof( ((obj_t)0)->stack_t))

#define STACK( _o_ ) CREF( _o_ )->stack_t

#define STACKP( _s_ ) (POINTERP( _s_ ) && (HEADER( _s_ ) == HEADER_STACK))

#define MAKE_STACK( _size_, aux ) \
   (BREF( MAKE_OBJECT( STACK_SIZE + (long)_size_, HEADER_STACK, aux )))

/*---------------------------------------------------------------------*/
/*    Les 'LOCATION's                                                  */
/*---------------------------------------------------------------------*/
#define LOCATION( x )        BREF( &x )
#define LOCATION_REF( x )    (*((obj_t *)CREF( x )))
#define LOCATION_SET( x, y ) ((LOCATION_REF( x ) = (obj_t)y ), y)

/*---------------------------------------------------------------------*/
/*    Les 'ENV'                                                        */
/*---------------------------------------------------------------------*/
#define ENV_REF( env, offset ) (((obj_t *)env)[ offset ])
#define ENV_SET( env, offset, value ) (ENV_REF( env, offset ) = value, value)

/*---------------------------------------------------------------------*/
/*    Les macros, structures et externes de debug bigloo.              */
/*    -------------------------------------------------------------    */
/*    Le champ 'link' et la variable 'link' sont tous les deux         */
/*    INDISPENSABLES. La variable sert pour les retours multiples, le  */
/*    champ sert pour l'affichage de la pile (voir debug.c).           */
/*---------------------------------------------------------------------*/
struct dframe {
   obj_t          symbol;
   struct dframe *link;
};

#define PUSH_LAMBDA_TRACE( name )                \
   struct dframe  frame;                         \
   struct dframe *link;                          \
                                                 \
   frame.symbol = name;                          \
   frame.link   = top_of_frame;                  \
   link         = top_of_frame;                  \
   top_of_frame = &frame;

#define POP_LAMBDA_TRACE( res )  \
   ( top_of_frame = link, res )

#define GET_LAMBDA_STACK() BREF( top_of_frame )
#define SET_LAMBDA_STACK( t ) \
   ( top_of_frame = (struct dframe *)CREF( t ), BUNSPEC )

extern struct dframe *top_of_frame;

/*---------------------------------------------------------------------*/
/*    Les macros utilisee pour les 'foreign_t'                         */
/*---------------------------------------------------------------------*/
#define FOREIGNP( o ) (POINTERP( o ) && (HEADER( o ) == HEADER_FOREIGN))
#define FOREIGN( o )  CREF( o)->foreign_t
#define FOREIGN_SIZE  (sizeof( ((obj_t)0)->foreign_t))

#define FOREIGN_ID( o )    FOREIGN( o ).id
#define FOREIGN_VALUE( o ) FOREIGN( o ).value

#define FOREIGN_ISP( o, key ) (FOREIGNP( o ) && (EQP( FOREIGN_ID( o ), key)))
#define BFOREIGN_TO_CFOREIGN( o ) (FOREIGN_VALUE( o ))

#define FOREIGN_EQP( o1, o2 ) (FOREIGN_VALUE( o1 ) == FOREIGN_VALUE( o2 ))
#define FOREIGN_NULLP( o1 ) (!FOREIGN_VALUE( o1 ))

#define FOREIGN_STRUCT_REF( o, tname, slot ) \
   (((tname)(FOREIGN_VALUE( o )))->slot)
#define FOREIGN_STRUCT_SET( o, tname, slot, value ) \
```

```
1253     (FOREIGN_STRUCT_REF( o, tname, slot ) = value, BUNSPEC)
1254
1255  #define FOREIGN_ARRAY_REF( tname, array, offset ) \
1256     (((tname)array)[ offset ])
1257
1258  #define FOREIGN_ARRAY_SET( tname, array, offset, value ) \
1259     (FOREIGN_ARRAY_REF( tname, array, offset ) = value, BUNSPEC)
1260
1261  /*------------------------------------------------------------------------*/
1262  /*    Les definitions de certains objets alloues                          */
1263  /*------------------------------------------------------------------------*/
1264  #define DEFINE_STRING( name, str, len )                    \
1265     static struct { header_t header;                        \
1266                     obj_t    length;                        \
1267                     char     string[len+(4-(len % 4))]; }   \
1268        name = { HEADER_STRING, BINT( len ), str }
1269
1270  #if( !defined( TAG_REAL ) )
1271  #   define DEFINE_REAL( name, flonum )                     \
1272        static struct { header_t header;                     \
1273                        double   real; }                     \
1274           name = { HEADER_REAL, flonum }
1275  #endif
1276
1277  /*------------------------------------------------------------------------*/
1278  /*    Le tableau des constantes (pour l'initialisation des modules).      */
1279  /*    ------------------------------------------------------------------  */
1280  /*    Ces deux macros servent a l'initialisation des constantes. C'est    */
1281  /*    un peu astucieux la facon dont c'est fait. Il faut regarder le      */
1282  /*    fichier 'comptime/Cnst/read-alloc.scm' pour comprendre comment      */
1283  /*    ca marche.                                                          */
1284  /*------------------------------------------------------------------------*/
1285  /* Cette premiere macro est juste utilisee dans un hack   */
1286  /* pour la passe Cgen et Cnst. Le pbm est que la          */
1287  /* variable __cnst soit utilisee. C'est le cas puisqu'on  */
1288  /* la passe a cette macro bidon...                        */
1289  #define DECLARE_CNST_TABLE( dummy )                        \
1290     BUNSPEC
1291
1292  #define CNST_TABLE_SET( offset, value )                    \
1293     ( __cnst[ CINT( offset ) ] = value, BUNSPEC )
1294
1295  #define CNST_TABLE_REF( offset )                           \
1296     __cnst[ offset ]
1297
1298  /*------------------------------------------------------------------------*/
1299  /*    Les marks pour l'externement lineaire                               */
1300  /*------------------------------------------------------------------------*/
1301  #define TAG_MARK (unsigned long)((unsigned long)1 << ((8 * OBJ_SIZE) - 1))
1302
1303  #define IS_MARKP( x )                                      \
1304     ( CNSTP( (unsigned long)x ) && (TAG_MARK & (unsigned long) x) )
1305
1306  #define BINT_TO_MARK( x )                                  \
1307     (obj_t)( (unsigned long)( (unsigned long)BCNST( CINT( x ) ) | TAG_MARK ) )
1308
1309  #define MARK_TO_BINT( x )                                  \
1310     (obj_t)( BINT((unsigned long)CCNST( (unsigned long)x & (~TAG_MARK) ) ) )
1311
1312  /*------------------------------------------------------------------------*/
1313  /*    Les macros de l'allocation en pile ...                              */
1314  /*------------------------------------------------------------------------*/
1315  #define STACK_REFERENCE( o ) ((obj_t)&o)
1316
1317  /*------------------------------------------------------------------------*/
1318  /*    Les recuperations externes                                          */
1319  /*------------------------------------------------------------------------*/
1320  #if( !defined( __GNUC__ ) )
1321     extern obj_t an_object, a_cell, a_pair;
1322  #endif
1323
1324  extern obj_t __ContinueValue;
1325  extern obj_t c_constant_string_to_string();
1326  extern bool_t input_port_fill_buffer();
1327
1328  extern obj_t c_string_to_string();
1329  extern obj_t c_string_to_symbol();
1330
1331  extern obj_t apply();
1332  extern obj_t eval_apply();
1333
1334  extern double strtod();
1335  extern obj_t make_real();
1336
1337  extern obj_t strputc();
1338  extern obj_t strputs();
1339  extern obj_t strport_flush();
1340
1341  extern obj_t the_failure();
1342  extern int fflush();
1343
1344  #endif
```

Annexe D

Un exemple complet de compilation

Le fichier source :

source.scm

```
;*---------------------------------------------------------------------*/
;*    serrano/these/src/complet/source.scm ...                         */
;*                                                                     */
;*    Author      :  Manuel Serrano                                    */
;*    Creation    :  Tue May 12 09:19:00 1992                          */
;*    Last change :  Tue Oct 25 17:14:19 1994 (serrano)                */
;*                                                                     */
;*    La resolution des huits reines d'apres L. Augustsson (lml)       */
;*---------------------------------------------------------------------*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module kons
   (main main)
   (static (memq o l)))

(define (memq obj list)
   (let loop ((list list))
      (if (null? list)
          #f
          (if (eq? (car list) obj)
              list
              (loop (cdr list))))))


(define (main argv)
   (memq (car argv) '(1 2 3)))
```

L'arbre de syntaxe abstraite après la passe d'intégration :

source.inlining

```
(define kons-initialisation@kons
  (lambda ()
    (begin
      (check-for-version-soundness!@__system
        'kons
        #"Bigloo (v1.7)"
        #a-33)
      0)))

(define memq@kons
  (lambda (obj@215 list@216)
    (begin
      (labels
        ((loop@217
           (list@218)
           (if (_c-null? list@218)
             #f
             (if (let ((obj1@223 (_c-car list@218)))
                   (_c-eq? obj1@223 obj@215))
               list@218
               (loop@217
                 (let ((c-cdr@226 (_c-cdr list@218))) c-cdr@226))))))
        (loop@217 list@216)))))

(define main@kons
  (lambda (argv@219)
    (begin
      (let ((list@228 '(1 2 3)))
        (let ((obj@227 (_c-car argv@219)))
          (labels
            ((loop@229
               (list@230)
               (if (_c-null? list@230)
                 #f
                 (if (let ((obj1@231 (_c-car list@230)))
                       (_c-eq? obj1@231 obj@227))
                   list@230
                   (loop@229
                     (let ((c-cdr@232 (_c-cdr list@230))) c-cdr@232))))))
            (loop@229 list@228)))))))
```

L'arbre de syntaxe abstraite après l'insertion des tests de type :

source.typage

```
(define kons-initialisation@kons
  (lambda ()
    (begin
      (check-for-version-soundness!@__system
        'kons
        (let ((string-cast@233
               (_c-cstring->bstring #"Bigloo (v1.7)")))
          string-cast@233)
        (_c-cchar->bchar #a-33))
      (_c-cint->bint 0))))

(define memq@kons
  (lambda (obj@215 list@216)
    (begin
      (labels
        ((loop@217
           (list@218)
           (cif (_c-null? list@218)
                (_c-cbool->bbool #f)
                (cif (let ((obj1@223
                            (_c-car
                              (cif (_c-pair? list@218)
                                   list@218
                                   (failure
                                     'memq@kons
                                     (type-error@__error 'pair
                                                         'list@218)
                                     list@218)))))
                       (_c-eq? obj1@223 obj@215))
                     list@218
                     (loop@217
                       (let ((c-cdr@226
                              (_c-cdr
                                (cif (_c-pair? list@218)
                                     list@218
                                     (failure
                                       'memq@kons
                                       (type-error@__error 'pair
                                                           'list@218)
                                       list@218)))))
                         c-cdr@226))))))
        (loop@217 list@216)))))

(define main@kons
  (lambda (argv@219)
    (begin
      (let ((list@228 '(1 2 3)))
        (let ((obj@227
```

```
                (_c-car
                  (cif (_c-pair? argv@219)
                       argv@219
                       (failure
                         'main@kons
                         (type-error@__error 'pair 'argv@219)
                         argv@219)))))
          (labels
            ((loop@229
               (list@230)
               (cif (_c-null? list@230)
                    (_c-cbool->bbool #f)
                    (cif (let ((obj1@231
                                (_c-car
                                  (cif (_c-pair? list@230)
                                       list@230
                                       (failure
                                         'main@kons
                                         (type-error@__error 'pair
                                                             'list@230)
                                         list@230)))))
                           (_c-eq? obj1@231 obj@227))
                         list@230
                         (loop@229
                           (let ((c-cdr@232
                                  (_c-cdr
                                    (cif (_c-pair? list@230)
                                         list@230
                                         (failure
                                           'main@kons
                                           (type-error@__error 'pair
                                                               'list@230)
                                           list@230)))))
                             c-cdr@232))))))
             (loop@229 list@228)))))
```

Le résultat de l'analyse de flot de contrôle :

source.cfa

```
(define main@kons
  (lambda (argv@219)
    (begin
      (let ((list@228 '(1 2 3)))
        (let ((obj@227
                (_c-car
                  (cif (_c-pair? argv@219)
                       argv@219
                       (failure
                         'main@kons
                         (type-error@__error 'pair 'argv@219)
                         argv@219)))))
          (labels
            ((loop@229
               (list@230)
               (cif (_c-null? list@230)
                    (_c-cbool->bbool #f)
                    (cif (let ((obj1@231
                                (_c-car
                                  (cif (_c-pair? list@230)
                                       list@230
                                       (failure
                                         'main@kons
                                         (type-error@__error 'pair
                                                             'list@230)
                                         list@230)))))
                           (_c-eq? obj1@231 obj@227))
                         list@230
                         (loop@229
                           (let ((c-cdr@232
                                  (_c-cdr
                                    (cif (_c-pair? list@230)
                                         list@230
                                         (failure
                                           'main@kons
                                           (type-error@__error 'pair
                                                               'list@230)
                                           list@230)))))
                             c-cdr@232))))))
             (loop@229 list@228)))))

(define kons-initialisation@kons
  (lambda ()
    (begin
      (check-for-version-soundness!@__system
        'kons
        (let ((string-cast@233
                (_c-cstring->bstring #"Bigloo (v1.7)")))
          string-cast@233)
        (_c-cchar->bchar #a-33))
      (_c-cint->bint 0))))
```

L'arbre de syntaxe abstraite après la passe **Hoist** :

source.hoist

```
(define main@kons
  (lambda (argv@219)
    (begin
      (let ((list@228 '(1 2 3)))
        (cif (_c-pair? argv@219)
             (let ((obj@227 (_c-car argv@219)))
               (labels
                 ((loop@229
                    (list@230)
                    (cif (_c-null? list@230)
                         (_c-cbool->bbool #f)
                         (cif (_c-pair? list@230)
                              (let ((obj1@231 (_c-car list@230)))
                                (cif (_c-eq? obj1@231 obj@227)
                                     list@230
                                     (cif (_c-pair? list@230)
                                          (let ((c-cdr@232 (_c-cdr list@230)))
                                            (loop@229 c-cdr@232))
                                          (failure
                                            'main@kons
                                            (type-error@__error 'pair
                                                                'list@230)
                                            list@230))))
                              (failure
                                'main@kons
                                (type-error@__error 'pair 'list@230)
                                list@230)))))
                  (loop@229 list@228)))
             (failure
               'main@kons
               (type-error@__error 'pair 'argv@219)
               argv@219))))))

(define kons-initialisation@kons
  (lambda ()
    (begin
      (let ((string-cast@233
              (_c-cstring->bstring #"Bigloo (v1.7)")))
        (begin
          (check-for-version-soundness!@__system
            'kons
            string-cast@233
            (_c-cchar->bchar #a-33))
          (_c-cint->bint 0))))))
```

L'arbre de syntaxe abstraite après l'élimination des sous-expressions communes :

source.cse

```
(define main@kons
  (lambda (argv@219)
    (begin
      (let ((list@228 '(1 2 3)))
        (cif (_c-pair? argv@219)
             (let ((obj@227 (_c-car argv@219)))
               (labels
                 ((loop@229
                    (list@230)
                    (cif (_c-null? list@230)
                         (_c-cbool->bbool #f)
                         (cif (_c-pair? list@230)
                              (let ((obj1@231 (_c-car list@230)))
                                (cif (_c-eq? obj1@231 obj@227)
                                     list@230
                                     (let ((c-cdr@232 (_c-cdr list@230)))
                                       (loop@229 c-cdr@232))))
                              (failure
                                'main@kons
                                (type-error@__error 'pair 'list@230)
                                list@230)))))
                  (loop@229 list@228)))
             (failure
               'main@kons
               (type-error@__error 'pair 'argv@219)
               argv@219))))))

(define kons-initialisation@kons
  (lambda ()
    (begin
```

```
        (let ((string-cast@233
               (_c-cstring->bstring #"Bigloo (v1.7)")))
          (begin
            (check-for-version-soundness!@__system
              'kons
              string-cast@233
              (_c-cchar->bchar #a-33))
            (_c-cint->bint 0))))))

```

Le résultat de l'analyse de fermeture :

source.glo

```
(define kons-initialisation@kons
  (lambda ()
    (begin
      (set! kons-initialisation_env@kons
        (_make-fx-procedure
          _kons-initialisation@kons
          0
          0))
      (set! main_env@kons
        (_make-fx-procedure _main@kons 1 0))
      (begin
        (let ((string-cast@233
                (_c-cstring->bstring #"Bigloo (v1.7)")))
          (begin
            (check-for-version-soundness!@__system
              'kons
              string-cast@233
              (_c-cchar->bchar #a-33))
            (_c-cint->bint 0)))))))

(define _kons-initialisation@kons
  (lambda (env@235) (kons-initialisation@kons)))

(define main@kons
  (lambda (argv@219)
    (begin
      (let ((list@228 '(1 2 3)))
        (cif (_c-pair? argv@219)
             (let ((obj@227 (_c-car argv@219)))
               (labels
                 ((loop@229
                    (list@230)
                    (begin
                      (cif (_c-null? list@230)
                           (_c-cbool->bbool #f)
                           (cif (_c-pair? list@230)
                                (let ((obj1@231 (_c-car list@230)))
                                  (cif (_c-eq? obj1@231 obj@227)
                                       list@230
                                       (let ((c-cdr@232 (_c-cdr list@230)))
                                         (loop@229 c-cdr@232))))
                                (failure
                                  'main@kons
                                  (type-error@__error 'pair 'list@230)
                                  list@230))))))
                  (loop@229 list@228)))
             (failure
               'main@kons
               (type-error@__error 'pair 'argv@219)
               argv@219))))))

(define _main@kons
  (lambda (env@234 argv@219) (main@kons argv@219)))

```

Le fichier C produit :

source.c

```
/*=====================================================================*/
/*   source.c                                                          */
/*   Bigloo (v1.7)                                                     */
/*   Manuel Serrano (c)       Fri Jul 29 12:18:24 MET DST 1994         */
/*=====================================================================*/
#include <bigloo.h>

static obj_t REQUIRE_INITIALISATION__KONS_211 = BUNSPEC;
extern obj_t make_fx_procedure();
extern obj_t ___ERROR_INITIALISATION___ERROR_92();
static obj_t __cnst[7];
extern obj_t ___SYSTEM_INITIALISATION___SYSTEM_109();
obj_t MAIN_ENV_KONS_147 = BUNSPEC;
extern obj_t CHECK_FOR_VERSION_SOUNDNESS____SYSTEM_45();
extern obj_t ___READER_INITIALISATION___READER_22();
#if defined( __STDC__ )
extern obj_t bigloo_main(obj_t);
#else
extern obj_t bigloo_main();
#endif
extern obj_t OPEN_INPUT_STRING___R4_PORTS_6_10_1_200();
#if defined( __STDC__ )
extern obj_t MAIN_KONS_222(obj_t);
#else
extern obj_t MAIN_KONS_222();
#endif
extern obj_t TYPE_ERROR___ERROR_146();
#if defined( __STDC__ )
extern obj_t __MAIN_KONS_133(obj_t, obj_t);
#else
extern obj_t __MAIN_KONS_133();
#endif
obj_t KONS_INITIALISATION_ENV_KONS_207 = BUNSPEC;
#if defined( __STDC__ )
extern obj_t __KONS_INITIALISATION_KONS_62(obj_t);
#else
extern obj_t __KONS_INITIALISATION_KONS_62();
#endif
extern obj_t READ___READER_198;
extern obj_t KONS_INITIALISATION_KONS_199();
extern obj_t c_constant_string_to_string();


extern void _bigloo_main();
#if defined( __STDC__ )
void main(int argc, char *argv[])
#else
void main(argc, argv)
int argc;
char *argv[];
#endif
{
  _bigloo_main(argc, argv);
}
obj_t
#if defined( __STDC__ )
KONS_INITIALISATION_KONS_199()
#else
KONS_INITIALISATION_KONS_199()
#endif
{

  if (BBOOL_TO_CBOOL(REQUIRE_INITIALISATION__KONS_211))
  {
    REQUIRE_INITIALISATION__KONS_211 = BFALSE;
    ___READER_INITIALISATION___READER_22();
    ___ERROR_INITIALISATION___ERROR_92();
    ___SYSTEM_INITIALISATION___SYSTEM_109();
    DECLARE_CNST_TABLE(__cnst[7]);
    {
      obj_t PORT_236_140;
      {
        obj_t AUX_240_127;
        AUX_240_127 = c_constant_string_to_string("ARGV@219 LIST@230 PAIR
MAIN@KONS (1 2 3 ) KONS #\"Bigloo (v1.7)\" ");
        PORT_236_140 = OPEN_INPUT_STRING___R4_PORTS_6_10_1_200(AUX_240_127);
      }
      {
        obj_t OFFSET_237_240;
        {
          OFFSET_237_240 = BINT(6);
          {
        _LOOP_238_65:

            if (LT_I(OFFSET_237_240, BINT(0)))
              BUNSPEC;
            else
            {
              obj_t AUX_241_17;
              AUX_241_17 = PROCEDURE_ENTRY(READ___READER_198)
                             (READ___READER_198, PORT_236_140, BEOA);
              CNST_TABLE_SET(OFFSET_237_240, AUX_241_17);
              {
                obj_t OFFSET_242_177;
                OFFSET_242_177 = OFFSET_237_240;
                {
                  OFFSET_237_240 = SUB_I(OFFSET_242_177, BINT(1));
                  goto _LOOP_238_65;
                }
              }
            }
          }
        }
      }
    }
    {
      KONS_INITIALISATION_ENV_KONS_207 =
       make_fx_procedure(__KONS_INITIALISATION_KONS_62, 0, 0);
      MAIN_ENV_KONS_147 = make_fx_procedure(__MAIN_KONS_133, 1, 0);
      {
```

```
          obj_t STRING_CAST_233_27;
          STRING_CAST_233_27 = CNST_TABLE_REF(0);
          {
            CHECK_FOR_VERSION_SOUNDNESS___SYSTEM_45(CNST_TABLE_REF(1),
                                    STRING_CAST_233_27,
                                    BCHAR('ß'));
            return BINT(0);
          }
        }
      }
    }
    else
      return BUNSPEC;
  }

  obj_t
  #if defined( __STDC__ )
  __KONS_INITIALISATION_KONS_62(obj_t ENV_235_203)
  #else
  __KONS_INITIALISATION_KONS_62(ENV_235_203)
  obj_t ENV_235_203;
  #endif
  {

    return KONS_INITIALISATION_KONS_199();
  }

  obj_t
  #if defined( __STDC__ )
  MAIN_KONS_222(obj_t ARGV_219_239)
  #else
  MAIN_KONS_222(ARGV_219_239)
  obj_t ARGV_219_239;
  #endif
  {

    {
      obj_t LIST_228_64;
      LIST_228_64 = CNST_TABLE_REF(2);
      if (PAIRP(ARGV_219_239))
      {
        obj_t OBJ_227_0;
        OBJ_227_0 = CAR(ARGV_219_239);
        {
          obj_t LIST_230_222;
          {
            LIST_230_222 = LIST_228_64;
            {
          _LOOP_229_168:

              if (NULLP(LIST_230_222))
                return BFALSE;
              else if (PAIRP(LIST_230_222))
              {
                obj_t OBJ1_231_64;
                OBJ1_231_64 = CAR(LIST_230_222);
                if (EQP(OBJ1_231_64, OBJ_227_0))
                  return LIST_230_222;
                else
                {
                  obj_t C_CDR_232_224;
                  C_CDR_232_224 = CDR(LIST_230_222);
                  {
                    LIST_230_222 = C_CDR_232_224;
                    goto _LOOP_229_168;
                  }
                }
              }
              else
                FAILURE(CNST_TABLE_REF(3),
                 TYPE_ERROR___ERROR_146(CNST_TABLE_REF(4),
                                CNST_TABLE_REF(5)),
                 LIST_230_222);
            }
          }
        }
      }
      else
        FAILURE(CNST_TABLE_REF(3),
            TYPE_ERROR___ERROR_146(CNST_TABLE_REF(4),
                           CNST_TABLE_REF(6)),
            ARGV_219_239);
    }
  }

  obj_t
  #if defined( __STDC__ )
  __MAIN_KONS_133(obj_t ENV_234_76, obj_t ARGV_219_239)
  #else
  __MAIN_KONS_133(ENV_234_76, ARGV_219_239)
  obj_t ENV_234_76, ARGV_219_239;
  #endif
  {

    return MAIN_KONS_222(ARGV_219_239);
  }

  obj_t
  #if defined( __STDC__ )
  bigloo_main(obj_t ARGV_239_87)
  #else
  bigloo_main(ARGV_239_87)
  obj_t ARGV_239_87;
  #endif
  {

    {
      KONS_INITIALISATION_KONS_199();
      return MAIN_KONS_222(ARGV_239_87);
    }
  }
```

Annexe E

Les programmes de tests

fib/bigloo/fib.scm

```
(module fib)

(define (fib x)
   (if (<fx x 2)
       1
       (+fx (fib (-fx x 1)) (fib (-fx x 2)))))

(print (fib 34))
```

fibs/bigloo/fibs.scm

```
(module fibstring
   (main main))


(define (fib x)
   (if (<fx x 2)
       (make-string 1)
       (string-append (fib (-fx x 1)) (fib (-fx x 2)))))


(define (main argv)
   (print (string-length (fib (string->integer (cadr argv))))))
```

tak/bigloo/tak.scm

```
(module tak)

(define tak
   (lambda (x y z)
      (if (not (<fx y x))
          z
          (tak (tak (-fx x 1) y z)
               (tak (-fx y 1) z x)
               (tak (-fx z 1) x y)))))

(print (tak 37 12 6))
```

semantique/bigloo/semantique.scm

```
;;; 30 Dec 89, version 2.6
;;;=============================================
;;; The denotational semantics of pattern-matching
;;;    C. Queinnec   Ecole Polytechnique --- INRIA
;;;          queinnec@poly.polytechnique.fr
;;;=============================================
;;; These programs are not written to be efficient, they
;;; only provide canonical semantics and compilation
;;; for pattern matching. A small functional language is also
;;; given and two new forms of function with pattern
;;; matching. Errors are roughly handled. Standardization
;;; of patterns is weak.

;;;=============================================================
;;; Semantics of pattern matching

;;; Domains
;;; e, ee       : Exp
;;; r, rr       : Env   = Id -> Exp + {unbound-pattern}
;;; a           : Seg   = Id -> Exp * Env * Alt -> Ans
;;; m           : Rep   = Id -> Exp * Env * Alt -> Ans
;;; k           : MCont = Exp * Env * Alt -> Ans
;;; z, zz       : Alt   = Unit -> Ans
;;; n           : Id
;;; f           : Pattern
;;; [[ ]]       : Meaning = Exp * Env * Seg * Rep * MCont * Alt -> Ans


;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module interprete)

;*---------------------------------------------------------------------*/
;*    L'interprete                                                     */
;*---------------------------------------------------------------------*/
(define (wrong m1 m2)
   (print m1)
   (print m2)
   m1)

(define (match f e)
  ((m-g f) e r.init a.init m.init
                    (lambda (e r z) #t)
                    (lambda () #f)))

(define (m-g f)
  (case (car f)
    ((*sexp) (m-sexp-g))
    ((*quote) (m-quote-g (cadr f)))
    ((*or) (m-or-g (cadr f) (caddr f)))
    ((*and) (m-and-g (cadr f) (caddr f)))
    ((*not) (m-not-g (cadr f)))
    ((*setq) (m-setq-g (cadr f) (caddr f)))
    ((*eval) (m-eval-g (cadr f)))
    ((*cons) (m-cons-g (cadr f) (caddr f)))
    ((*ssetq-append) (m-ssetq-append-g (cadr f) (caddr f) (cadddr f)))
    ((*eval-append) (m-eval-append-g (cadr f) (caddr f)))
    ((*end-ssetq) (m-end-ssetq-g (cadr f)))
    ((*times) (m-times-g (cadr f) (caddr f) (cadddr f)))
    ((*end-times) (m-end-times-g (cadr f)))
    (else (wrong "Unrecognized pattern" f)) ) )

(define (m-sexp-g)
  (lambda (e r a m k z) (k e r z)) )

(define (m-quote-g ee)
  (lambda (e r a m k z)
    (if (eq? e ee)
        (k e r z)
        (z) ) ) )

(define (m-or-g f1 f2)
  (lambda (e r a m k z)
    ((m-g f1)
     e r a m (lambda ()
               ((m-g f2)
                e r a m k z ) ) ) ) )

(define (m-and-g f1 f2)
  (lambda (e r a m k z)
    ((m-g f1)
```

```
      e r a m (lambda (ee rr zz)
                ((m-g f2)
                 e rr a m k zz ) )
      z ) ) )

(define (m-not-g f)
  (lambda (e r a m k z)
    ((m-g f)
     e r a m (lambda (ee rr zz) (z))
             (lambda () (k e r z)) ) ) )

(define (m-setq-g n f)
  (lambda (e r a m k z)
    ((m-g f)
     e r a m (lambda (ee rr zz)
               (if (eq? (rr n) unbound-pattern)
                   (k e (extend rr n e) zz)
                   (wrong "Cannot rebind pattern" n) ) )
     z ) ) )

(define (m-eval-g n)
  (lambda (e r a m k z)
    (if (eq? (r n) unbound-pattern)
        (wrong "Unbound pattern" n)
        (if (eq? (r n) e)
            (k e r z)
            (z) ) ) ) )

(define (m-cons-g f1 f2)
  (lambda (e r a m k z)
    (if (pair? e)
        ((m-g f1)
         (car e) r a.init m.init
         (lambda (ee rr zz)
           ((m-g f2)
            (cdr e) rr a m k zz ) )
         z )
        (z) ) ) )

(define (m-ssetq-append-g n f1 f2)
  (lambda (e r a m k z)
    ((m-g f1)
     e r (extend a.init n
                 (lambda (ee rr zz)
                   (if (eq? (rr n) unbound-pattern)
                       ((m-g f2)
                        ee (extend rr n (cut e ee)) a m k zz )
                       (wrong "cannot rebind" n) ) ) )
     m.init (lambda (ee rr zz)
              (wrong "Ssetq not ended" f1) )
     z ) ) )

(define (m-eval-append-g n f)
  (lambda (e r a m k z)
    (if (eq? (r n) unbound-pattern)
        (wrong "Unbound segment" n)
        (check e (r n) (lambda (ee)
                         ((m-g f)
                          ee r a m k z ) )
                       z ) ) ) )

(define (m-end-ssetq-g n)
  (lambda (e r a m k z)
    ((a n) e r z) ) )

;;; corrected version thanks to ML
(define (m-times-g n f1 f2)
  (lambda (e r a m k z)
    (labels ((ltry (e r z)
                   ((m-g f2)
                    e r a m k
                    (lambda ()
                      ((m-g f1)
                       e r a.init
                       (extend m.init n ltry)
                       (lambda (ee rr zz)
                         (wrong "Times not ended" f1) )
                       z ) ) ) ) ))
      (ltry e r z))))

(define (m-end-times-g n)
  (lambda (e r a m k z)
    ((m n) e r z) ) )

(define (a.init n)
  (lambda (e r z)
    (wrong "No current ssetq for" n) ) )

(define (m.init n)
  (lambda (e r z)
    (wrong "No current repetition named" n) ) )

(define (r.init n)
  unbound-pattern )

(define unbound-pattern '**unbound-pattern**)

(define (check e ee fn z)
  (if (and (pair? e)
           (pair? ee)
           (eq? (car e) (car ee)) )
      (check (cdr e) (cdr ee) fn z)
      (if (null? ee) (fn e) (z)) ) )

(define (extend fn pt im)
  (lambda (x) (if (eq? pt x) im (fn x))) )

(define (cut e ee)
  (if (eq? e ee) '()
      (cons (car e) (cut (cdr e) ee)) ) )

;*---------------------------------------------------------------------*/
;*    Le jeux d'essai                                                  */
;*---------------------------------------------------------------------*/
;;; Le symbol a
(define f0 '(*quote a))

;;; La liste (a)
(define f1 '(*cons (*quote a) (*quote ())))

;;; Une liste de 0 ou + a.
(define f2 '(*times x
                    (*cons (*quote a) (*end-times x))
                    (*quote ())))

;;; Une liste (a ... b)
(define f3 '(*times x
                    (*cons (*quote a) (*end-times x))
                    (*cons (*quote b) (*quote ()))))

;;; Une liste  de deux elements egaux
(define f4 '(*cons (*setq y (*sexp))
                   (*cons (*eval y) (*quote ()))))

(let ((l (make-vector 16)))
   (print (let loop ((i 2000))
            (if (=fx i 0)
                l
                (begin
                   (vector-set! l 0 (match f0 'a))
                   (vector-set! l 1 (match f0 'b))
                   (vector-set! l 2 (match f1 '(a)))
                   (vector-set! l 3 (match f1 '(b)))
                   (vector-set! l 4 (match f2 '(a a a a)))
                   (vector-set! l 5 (match f2 '(a b a a)))
                   (vector-set! l 6 (match f2 '(a a a a a a a a b b b a b a)))
                   (vector-set! l 7 (match f2 '(a b a a b b a b a a b a b a)))
                   (vector-set! l 8 (match f3 '(a a a b)))
                   (vector-set! l 9 (match f3 '(a a a a)))
                   (vector-set! l 10 (match f3 '(a a a b a b b b b b b a b)))
                   (vector-set! l 11 (match f3 '(a a a a a a a a a a a b a)))
                   (vector-set! l 12 (match f4 '(e e)))
                   (vector-set! l 13 (match f4 '(e f)))
                   (vector-set! l 14 (match f4 '((e f h e f) (h r g h e))))
                   (vector-set! l 15 (match f4 '((e f g  j l) (e f g  j l))))
                   (loop (-fx i 1)))))))
```

dens/bigloo/dens.scm

```
;*=====================================================================*/
;*    serrano/these/src/dens/bigloo/dens.scm ...                       */
;*    -------------------------------------------------------------    */
;*    Author      :  Manuel Serrano                                    */
;*    Creation    :  Tue Mar 29 14:53:14 1994                          */
;*    Last change :  Wed Jun  1 10:11:34 1994 (serrano)                */
;*    -------------------------------------------------------------    */
;*    Une semantique denotationnelle de Scheme ecrite en Scheme        */
;*    par Christian Queinnec.                                          */
;*=====================================================================*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module den)

;;; 18 April 1990, Version 3.7

;       Un petit Scheme en Scheme avec un style
;             fortement de'notationnel
;       Christian Queinnec
;   \'Ecole Polytechnique \& INRIA-Rocquencourt
;   91128 Palaiseau Cedex --- France

;   Meaning  : Form -> Env * Cont * Store -> Val
```

```
; r   Env    : Var -> Loc
; s   Store  : Loc -> Val
; k   Cont   : Val * Store -> Val
; v   Val    : Fun | Int | Pair
; f   Fun    : (Val...) * Cont * Store -> Val
;     Cons   : Loc * Loc
; e   Form

(define (wrong exp1 exp2)
   (error "bigloo" exp1 exp2))

(define (atom? exp) (not (pair? exp)))

(define (meaning e)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e)
                      (meaning-quotation e) )
      (case (car e)
        ((quote) (meaning-quotation (cadr e)))
        ((lambda) (meaning-abstraction (cadr e) (cddr e)))
        ((if) (meaning-alternative (cadr e) (caddr e) (cadddr e)))
        ((begin) (meaning-sequence (cdr e)))
        ((set!) (meaning-assignment (cadr e) (caddr e)))
        (else (meaning-application (car e) (cdr e))) ) ) )

;;; Iterator

(define (meaning* e*)
  (if (pair? e*)
      (lambda (r k s)
        ((meaning (car e*))
         r
         (lambda (v s1)
           ((meaning* (cdr e*))
            r
            (lambda (v* s2)
              (k (cons v v*) s2) )
            s1 ) )
         s ) )
      (meaning-quotation '()) ) )

(define (meaning-quotation v)
  (lambda (r k s)
    (k v s) ) )

(define (meaning-reference n)
  (lambda (r k s)
    (k (s (r n)) s) ) )

(define (meaning-alternative e1 e2 e3)
  (lambda (r k s)
    ((meaning e1)
     r
     (lambda (v s1)
       (if v
           ((meaning e2) r k s1)
           ((meaning e3) r k s1) ) )
     s ) ) )

(define (meaning-assignment n e)
  (lambda (r k s)
    ((meaning e)
     r
     (lambda (v s1)
       (k v (extend s1 (r n) v)) )
     s ) ) )

(define (meaning-abstraction n* e*)
  (lambda (r k s)
    (k (lambda (v* k1 s1)
         (allocate s1 (length n*)
                      (lambda (a*)
                        ((meaning-sequence e*)
                         (extend* r n* a*)
                         k1
                         (extend* s1 a* v*) ) ) ) )
       s ) ) )

(define (meaning-application e e*)
  (lambda (r k s)
    ((meaning e)
     r
     (lambda (f s1)
       ((meaning* e*)
        r
        (lambda (v* s2)
          (f v* k s2) )
        s1 ) )
     s ) ) )

(define (meaning-sequence e*)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (lambda (r k s)
            ((meaning (car e*))
             r
             (lambda (v s1)
               ((meaning-sequence (cdr e*))
                r k s1 ) )
             s ) )
          (meaning (car e*)) )
      (meaning-quotation '()) ) )

(define (s.init a)
  (wrong "No value for " a) )

(define (r.init n)
  (wrong "No location for " n) )

(define (extend fn pt im)
  (lambda (x) (if (equal? pt x) im (fn x))) )

(define (extend* fn pts ims)
  (if (pair? pts)
      (if (pair? ims)
          (extend (extend* fn (cdr pts) (cdr ims))
                  (car pts) (car ims) )
          (wrong "Too less images" #f) )
      (if (pair? ims)
          (wrong "Too much points" #f)
          fn ) ) )

;;; Locations are represented by growing positive integers.
;;; No attempt is done to garbage collect locations.
(define allocate
  (let ((loc 0))
    (lambda (s n q)
      (let loop ((n n)
                 (a* '()) )
        (if (<= n 0) (q a*)
                     (begin (set! loc (- loc 1))
                            (loop (- n 1)
                                  (cons loc a*) ) ) ) ) ) ) )

(define-macro (definitial name value)
  '(allocate s.init 1
                    (lambda (a*)
                      (set! r.init
                            (extend r.init ',name (car a*)) )
                      (set! s.init
                            (extend s.init (car a*) ,value) )
                      ',name ) ) )

(define-macro (defprimitive name value arity)
  '(definitial
    ,name
     (lambda (v* k s)
       (if (= ,arity (length v*))
           (k (apply ,value v*) s)
           (wrong "Incorrect arity"
                  (list ',name v*) ) ) ) ) )

(definitial cons
  (lambda (v* k s)
    (if (= 2 (length v*))
        (allocate s 2
                  (lambda (a*) (k a* (extend* s a* v*))) )
        (wrong "incorrect arity" (list 'cons v*)) ) ) )

(definitial car
  (lambda (v* k s)
    (if (= 1 (length v*))
        (if (cons? (car v*))
            (k (s (car (car v*))) s)
            (wrong "Not a Cons" #f) )
        (wrong "incorrect arity" (list 'cons v*)) ) ) )

(definitial cdr
  (lambda (v* k s)
    (if (= 1 (length v*))
        (if (cons? (car v*))
            (k (s (cadr (car v*))) s)
            (wrong "Not a Cons" #f) )
        (wrong "incorrect arity" (list 'cons v*)) ) ) )

(defprimitive pair? cons? 1)
(defprimitive eq? eq? 2)

(defprimitive + +fx 2)
(defprimitive - -fx 2)
(defprimitive < <fx 2)
(defprimitive > >fx 2)
(defprimitive = =fx 2)
(defprimitive * *fx 2)
(defprimitive <= <=fx 2)
(defprimitive >= >=fx 2)
```

```
212
213   (definitial call/cc
214     (lambda (v1* k1 s1)
215       (if (= 1 (length v1*))
216           ((car v1*)
217            (list (lambda (v2* k2 s2)
218                    (if (= 1 (length v2*))
219                        (k1 (car v2*) s2)
220                        (wrong "Incorrect arity" (list 'k v2*)) ) ))
221            k1
222            s1 )
223           (wrong "Incorrect arity" (list ',name v1*)) ) ) )
224
225   (definitial t #t)
226   (definitial f #f)
227   (definitial nil '())
228
229   ;;; Some free global locations:
230   (definitial fib 'void)
231
232   (define (cons? e)
233     (and (pair? e)
234          (integer? (car e))
235          (< (car e) 0)
236          (pair? (cdr e))
237          (integer? (cadr e))
238          (< (cadr e) 0)
239          (null? (cddr e)) ) )
240
241   (define (convert e s)
242     (letrec ((convert (lambda (e)
243                         (if (cons? e)
244                             (cons (convert (s (car e)))
245                                   (convert (s (cadr e))) )
246                             e ) )))
247       (convert e) ) )
248
249   (define faux-read
250     (let ((l '((set! fib (lambda (x)
251                            (if (< x 2)
252                                1
253                                (+ (fib (- x 1)) (fib (- x 2))))))
254                (fib 14)
255                '())))
256       (lambda ()
257         (if (null? l)
258             l
259             (let ((res (car l)))
260               (if (not (null? l))
261                   (set! l (cdr l)))
262               res)))))
263
264   (let ((s.current s.init))
265     (let loop ((exp (faux-read)))
266       (if (or (eof-object? exp)
267               (null? exp))
268           'done
269           (begin
270             ((meaning exp)
271              r.init
272              (lambda (v s.final)
273                (set! s.current s.final)
274                (print (convert v s.final)) )
275              s.current)
276              (loop (faux-read))))))
277
```

leval/bigloo/leval.scm

```
1   ;*=====================================================================*/
2   ;*    serrano/these/src/leval/bigloo/leval.scm ...                     */
3   ;*    -------------------------------------------------------------    */
4   ;*    Author      :  Manuel Serrano                                    */
5   ;*    Creation    :  Thu Mar 24 15:15:02 1994                          */
6   ;*    Last change :  Fri Aug 26 12:53:44 1994 (serrano)                */
7   ;*    -------------------------------------------------------------    */
8   ;*    Un petit interprete de byte code                                 */
9   ;*=====================================================================*/
10
11  ;*---------------------------------------------------------------------*/
12  ;*    Le module                                                        */
13  ;*---------------------------------------------------------------------*/
14  (module _ewal
15     (main main)
16     (static (extend-env a b)
17             (update-global! a b c)))
18
19  ;*---------------------------------------------------------------------*/
20  ;*    Les environments ...                                             */
21  ;*---------------------------------------------------------------------*/
22  (define the-global-environment '())
23
24  (define (errow a b c)
25     (print "*** ERROW: " a #\Newline b " -- " c))
26
27  ;*---------------------------------------------------------------------*/
28  ;*    ewal ...                                                         */
29  ;*    sexp x env --> sexp                                              */
30  ;*---------------------------------------------------------------------*/
31  (define (ewal exp . env)
32     (for-each (lambda (x)
33                 ((compile-define (car x) (cdr x)) '()))
34               env)
35     (meaning (compile exp '() #f) '()))
36
37  ;*---------------------------------------------------------------------*/
38  ;*    meaning ...                                                      */
39  ;*---------------------------------------------------------------------*/
40  (define (meaning pre-compiled-expression dynamic-env)
41     (pre-compiled-expression dynamic-env))
42
43  ;*---------------------------------------------------------------------*/
44  ;*    compile ...                                                      */
45  ;*    s-exp x env x { t, f } --> (lambda () ...)                       */
46  ;*    -----------------------------------------------------------      */
47  ;*    La phase d'expansion a genere une syntaxe correcte. On n'a donc  */
48  ;*    plus du tout a la tester maintenant.                             */
49  ;*---------------------------------------------------------------------*/
50  (define (compile exp env tail?)
51     (cond
52        ((not (pair? exp))
53         (let ((atom exp))
54            (cond
55               ((symbol? atom)
56                (compile-ref (variable atom env) tail?))
57               (else
58                (compile-cnst atom tail?)))))
59        ((eq? (car exp) 'module)
60         (lambda (dynamic-env) (unspecified)))
61        ((eq? (car exp) 'quote)
62         (let ((cnst (cadr exp)))
63            (compile-cnst cnst tail?)))
64        ((eq? (car exp) 'if)
65         (let ((si (cadr exp))
66               (alors (caddr exp))
67               (sinon (cadddr exp)))
68            (compile-if (compile si env #f)
69                        (compile alors env tail?)
70                        (compile sinon env tail?))))
71        ((eq? (car exp) 'begin)
72         (let ((rest (cdr exp)))
73            (compile-begin rest env)))
74        ((eq? (car exp) 'define)
75         (let ((var (cadr exp))
76               (val (caddr exp)))
77            (compile-define var (compile val '() #f))))
78        ((eq? (car exp) 'set!)
79         (let ((var (cadr exp))
80               (val (caddr exp)))
81            (compile-set (variable var env) (compile val env #f))))
82        ((eq? (car exp) 'lambda)
83         (let ((formals (cadr exp))
84               (body    (caddr exp)))
85            (compile-lambda formals
86                            (compile body (extend-env formals env) #t)
87                            tail?)))
88        ((not (pair? (car exp)))
89         (let ((fun (car exp))
90               (args (cdr exp)))
91            (let ((actuals (map (lambda (a) (compile a env #f)) args)))
92               (cond
93                  ((symbol? fun)
94                   (let ((proc (variable fun env)))
95                      (cond
96                         ((global? proc)
97                          (compile-global-application proc
98                                                      actuals
99                                                      tail?))
100                        (else
101                         (compile-application (compile-ref proc #f)
102                                              actuals
103                                              tail?)))))
104                 ((procedure? fun)
105                  (compile-compiled-application fun actuals tail?))
106                 (else
107                  (errow "ewal" "Not a procedure" fun))))))
108        (else
109         (let ((fun (car exp))
110               (args (cdr exp)))
111            (let ((actuals (map (lambda (a) (compile a env #f)) args))
112                  (proc    (compile fun env #f)))
113               (compile-application proc actuals tail?))))))
114
115 ;*---------------------------------------------------------------------*/
```

```
;*    variable ...                                                         */
;*---------------------------------------------------------------------*/
(define (variable symbol env)
   (let ((offset (let loop ((env   env)
                            (count 0))
                    (cond
                       ((null? env)
                        #f)
                       ((eq? (car env) symbol)
                        count)
                       (else
                        (loop (cdr env) (+fx count 1)))))))
      (if offset
          offset
          (let ((global (assq symbol the-global-environment)))
             (if (not global)
                 '#(,symbol)
                 global)))))

;*---------------------------------------------------------------------*/
;*    global? ...                                                      */
;*---------------------------------------------------------------------*/
(define (global? variable)
   (pair? variable))

;*---------------------------------------------------------------------*/
;*    dynamic? ...                                                     */
;*---------------------------------------------------------------------*/
(define (dynamic? variable)
   (vector? variable))

;*---------------------------------------------------------------------*/
;*    compile-ref ...                                                  */
;*---------------------------------------------------------------------*/
(define (compile-ref variable tail?)
   (cond
      ((global? variable)
       (lambda (dynamic-env) (cdr variable)))
      ((dynamic? variable)
       (lambda (dynamic-env) (let ((global (assq (vector-ref variable 0)
                                                 the-global-environment)))
                                (if (not global)
                                    (errow "ewal"
                                           "Unbound variable"
                                           (vector-ref variable 0))
                                    (cdr global)))))
      (else
       (case variable
          ((0)
           (lambda (dynamic-env) (car dynamic-env)))
          ((1)
           (lambda (dynamic-env) (cadr dynamic-env)))
          ((2)
           (lambda (dynamic-env) (caddr dynamic-env)))
          ((3)
           (lambda (dynamic-env) (cadddr dynamic-env)))
          (else
           (lambda (dynamic-env)
              (do ((i 0 (+fx i 1))
                   (env dynamic-env (cdr env)))
                  ((=fx i variable) (car env)))))))))

;*---------------------------------------------------------------------*/
;*    compile-set ...                                                  */
;*---------------------------------------------------------------------*/
(define (compile-set variable value)
   (cond
      ((global? variable)
       (lambda (dynamic-env) (update-global! variable value dynamic-env)))
      ((dynamic? variable)
       (lambda (dynamic-env)
          (let ((global (assq (vector-ref variable 0)
                              the-global-environment)))
             (if (not global)
                 (errow "ewal"
                        "Unbound variable"
                        (vector-ref variable 0))
                 (update-global! global value dynamic-env)))))
      (else
       (case variable
          ((0)
           (lambda (dynamic-env) (set-car! dynamic-env
                                           (value dynamic-env))))
          ((1)
           (lambda (dynamic-env) (set-car! (cdr dynamic-env)
                                           (value dynamic-env))))
          ((2)
           (lambda (dynamic-env) (set-car! (cddr dynamic-env)
                                           (value dynamic-env))))
          ((3)
           (lambda (dynamic-env) (set-car! (cdddr dynamic-env)
                                           (value dynamic-env))))
          (else
           (lambda (dynamic-env)
              (do ((i 0 (+fx i 1))
                   (env dynamic-env (cdr env)))
                  ((=fx i variable) (set-car! env
                                              (value dynamic-env)))))))))

;*---------------------------------------------------------------------*/
;*    compile-cnst ...                                                 */
;*---------------------------------------------------------------------*/
(define (compile-cnst cnst tail?)
   (lambda (dynamic-env) cnst))

;*---------------------------------------------------------------------*/
;*    compile-if ...                                                   */
;*---------------------------------------------------------------------*/
(define (compile-if test then else)
   (lambda (dynamic-env) (if (test dynamic-env)
                             (then dynamic-env)
                             (else dynamic-env))))

;*---------------------------------------------------------------------*/
;*    compile-begin ...                                                */
;*---------------------------------------------------------------------*/
(define (compile-begin body env)
   (cond
      ((and (pair? body) (and (null? (cdr body))))
       ;; le cas degenere
       (let ((rest (compile (car body) env #t)))
          (lambda (dynamic-env) (rest dynamic-env))))
      (else
       (let ((body (let loop ((rest body))
                      (cond
                         ((null? rest)
                          (errow "ewal" "Illegal form" body))
                         ((null? (cdr rest))
                          (cons (compile (car rest) env #t) '()))
                         (else
                          (cons (compile (car rest) env #f)
                                (loop (cdr rest))))))))
          (lambda (dynamic-env) (let _loop_ ((body body))
                                   (if (null? (cdr body))
                                       ((car body) dynamic-env)
                                       (begin
                                          ((car body) dynamic-env)
                                          (_loop_ (cdr body)))))))))

;*---------------------------------------------------------------------*/
;*    init-the-global-environment! ...                                 */
;*---------------------------------------------------------------------*/
(define (linit-the-global-environment!)
   (if (pair? the-global-environment)
       'done
       ;; je ne peux pas utiliser de constante car quand cette fonction
       ;; sera appelle, je ne suis pas qu'elles soient initialisee.
       (set! the-global-environment (cons (cons #f #f) '()))))

;*---------------------------------------------------------------------*/
;*    compile-define ...                                               */
;*    ---------------------------------------------------------------  */
;*    On ne rajoute pas en tete car elle contient la definition de     */
;*    'the-module-environment'. On rajoute donc en deuxieme.           */
;*---------------------------------------------------------------------*/
(define (compile-define var val)
   (lambda (dynamic-env)
      (let ((cell (assq var the-global-environment)))
         (if (pair? cell)
             (begin
                (print "*** WARNING:bigloo:ewal\nredefinition of variable -- "
                       var)
                (update-global! cell val dynamic-env))
             (begin
                (set-cdr! the-global-environment
                          (cons (car the-global-environment)
                                (cdr the-global-environment)))
                (set-car! the-global-environment
                          (cons var (val dynamic-env)))
                var)))))

;*---------------------------------------------------------------------*/
;*    define-primitive! ...                                            */
;*    ---------------------------------------------------------------  */
;*    Cette fonction est juste une forme abregee de la precedente, qui */
;*    construit le '(lambda () ...)' absent                            */
;*---------------------------------------------------------------------*/
(define (ldefine-primitive! var val)
   (set-cdr! the-global-environment
             (cons (car the-global-environment)
                   (cdr the-global-environment)))
   (set-car! the-global-environment (cons var val)))

;*---------------------------------------------------------------------*/
;*      update-global! ...                                             */
;*---------------------------------------------------------------------*/
```

```
(define (update-global! variable val dynamic-env)
   (set-cdr! variable (val dynamic-env))
   (car variable))

;*---------------------------------------------------------------------*/
;*    extend-env ...                                                   */
;*---------------------------------------------------------------------*/
(define (extend-env extend old-env)
   (let _loop_ ((extend extend))
      (cond
         ((null? extend)
          old-env)
         ((not (pair? extend))
          (cons extend old-env))
         (else
          (cons (car extend) (_loop_ (cdr extend)))))))

;*---------------------------------------------------------------------*/
;*    pair ...                                                         */
;*---------------------------------------------------------------------*/
(define (pair n l)
   (if (< n 0)
       (let loop ((n n)
                  (l l))
          (cond
             ((= -1 n)
              #t)
             ((not (pair? l))
              #f)
             (else
              (loop (+ 1 n) (cdr l)))))
       (let loop ((n n)
                  (l l))
          (cond
             ((= 0 n)
              (null? l))
             ((not (pair? l))
              #f)
             (else
              (loop (- n 1) (cdr l)))))))

;*---------------------------------------------------------------------*/
;*    compile-lambda ...                                               */
;*---------------------------------------------------------------------*/
(define (compile-lambda formals body tail?)
   (cond
      ((null? formals)
       (lambda (dynamic-env)
          (lambda ()
             (body dynamic-env))))
      ((pair 1 formals)
       (lambda (dynamic-env)
          (lambda (x)
             (body (cons x dynamic-env)))))
      ((pair 2 formals)
       (lambda (dynamic-env)
          (lambda (x y)
             (body (cons x (cons y dynamic-env))))))
      ((pair 3 formals)
       (lambda (dynamic-env)
          (lambda (x y z)
             (body (cons x (cons y (cons z dynamic-env)))))))
      ((pair 4 formals)
       (lambda (dynamic-env)
          (lambda (x y z t)
             (body (cons x (cons y (cons z (cons z dynamic-env))))))))
      ((symbol? formals)
       (lambda (dynamic-env)
          (lambda x
             (body (cons x dynamic-env)))))
      ((pair -1 formals)
       (lambda (dynamic-env)
          (lambda (x . y)
             (body (cons x (cons y dynamic-env))))))
      ((pair -2 formals)
       (lambda (dynamic-env)
          (lambda (x y . z)
             (body (cons x (cons y (cons z dynamic-env)))))))
      ((pair -3 formals)
       (lambda (dynamic-env)
          (lambda (x y z . t)
             (body (cons x (cons y (cons z (cons z dynamic-env))))))))
      (else
       (lambda (dynamic-env)
          (lambda x
             (let ((new-env (let _loop_ ((actuals x)
                                         (formals formals))
                              (cond
                                 ((null? formals)
                                  (if (not (null? actuals))
                                      (errow "ewal"
                                             "Too many arguments provided"
                                             actuals)
                                      dynamic-env))
                                 ((null? actuals)
                                  (errow "ewal"
                                         "Too fee arguments provided"
                                         formals))
                                 ((not (pair? formals))
                                  (cons actuals dynamic-env))
                                 (else
                                  (cons (car actuals)
                                        (_loop_ (cdr actuals)
                                                (cdr formals))))))))
                (body new-env)))))))

;*---------------------------------------------------------------------*/
;*    compile-global-application ...                                   */
;*---------------------------------------------------------------------*/
(define (compile-global-application proc actuals tail?)
   (case (length actuals)
      ((0)
       (lambda (dynamic-env) ((cdr proc))))
      ((1)
       (lambda (dynamic-env) ((cdr proc) ((car actuals) dynamic-env))))
      ((2)
       (lambda (dynamic-env) ((cdr proc) ((car actuals) dynamic-env)
                                         ((cadr actuals) dynamic-env))))
      ((3)
       (lambda (dynamic-env) ((cdr proc) ((car actuals) dynamic-env)
                                         ((cadr actuals) dynamic-env)
                                         ((caddr actuals) dynamic-env))))
      ((4)
       (lambda (dynamic-env) ((cdr proc) ((car actuals) dynamic-env)
                                         ((cadr actuals) dynamic-env)
                                         ((caddr actuals) dynamic-env)
                                         ((cadddr actuals) dynamic-env))))
      (else
       (lambda (dynamic-env)
          (apply (cdr proc) (map (lambda (v) (v dynamic-env)) actuals))))))

;*---------------------------------------------------------------------*/
;*    compile-compiled-application ...                                 */
;*---------------------------------------------------------------------*/
(define (compile-compiled-application proc actuals tail?)
   (case (length actuals)
      ((0)
       (lambda (dynamic-env) (proc)))
      ((1)
       (lambda (dynamic-env) (proc ((car actuals) dynamic-env))))
      ((2)
       (lambda (dynamic-env) (proc ((car actuals) dynamic-env)
                                   ((cadr actuals) dynamic-env))))
      ((3)
       (lambda (dynamic-env) (proc ((car actuals) dynamic-env)
                                   ((cadr actuals) dynamic-env)
                                   ((caddr actuals) dynamic-env))))
      ((4)
       (lambda (dynamic-env) (proc ((car actuals) dynamic-env)
                                   ((cadr actuals) dynamic-env)
                                   ((caddr actuals) dynamic-env)
                                   ((cadddr actuals) dynamic-env))))
      (else
       (lambda (dynamic-env)
          (apply proc (map (lambda (v) (v dynamic-env)) actuals))))))

;*---------------------------------------------------------------------*/
;*    compile-application ...                                          */
;*---------------------------------------------------------------------*/
(define (compile-application proc actuals tail?)
   (case (length actuals)
      ((0)
       (lambda (dynamic-env) ((proc dynamic-env))))
      ((1)
       (lambda (dynamic-env) ((proc dynamic-env)
                              ((car actuals) dynamic-env))))
      ((2)
       (lambda (dynamic-env) ((proc dynamic-env)
                              ((car actuals) dynamic-env)
                              ((cadr actuals) dynamic-env))))
      ((3)
       (lambda (dynamic-env) ((proc dynamic-env)
                              ((car actuals) dynamic-env)
                              ((cadr actuals) dynamic-env)
                              ((caddr actuals) dynamic-env))))
      ((4)
       (lambda (dynamic-env) ((proc dynamic-env)
                              ((car actuals) dynamic-env)
                              ((cadr actuals) dynamic-env)
                              ((caddr actuals) dynamic-env)
                              ((cadddr actuals) dynamic-env))))
      (else
       (lambda (dynamic-env)
          (apply (proc dynamic-env) (map (lambda (v) (v dynamic-env))
                                         actuals))))))
```

```
;*---------------------------------------------------------------------*/
;*    Les inits                                                        */
;*---------------------------------------------------------------------*/
(linit-the-global-environment!)

(ldefine-primitive! '+ +)
(ldefine-primitive! '- -)
(ldefine-primitive! '< <)
(ldefine-primitive! 'eq? eq?)
(ldefine-primitive! 'car car)
(ldefine-primitive! 'cdr cdr)
(ldefine-primitive! 'cons cons)
(ldefine-primitive! 'null? null?)
(ldefine-primitive! 'unspecified unspecified)

;*---------------------------------------------------------------------*/
;*    main ...                                                         */
;*---------------------------------------------------------------------*/
(ewal '(define fib (lambda (x)
                      (if (< x 2)
                          1
                          (+ (fib (- x 1)) (fib (- x 2)))))))

(define (main argv)
   (if (not (null? (cdr argv)))
       (let loop ((i (string->integer (cadr argv)))
                  (r (ewal '(fib 25))))
          (if (=fx i 1)
              (print r)
              (loop (-fx i 1) (ewal '(fib 25)))))))
```

beval/bigloo/beval.scm

```
;*---------------------------------------------------------------------*/
;*    serrano/these/src/beval/bigloo/beval.scm ...                     */
;*                                                                     */
;*    Author      :  Manuel Serrano                                    */
;*    Creation    :  Wed Feb 10 08:35:54 1993                          */
;*    Last change :  Fri Aug 26 11:55:19 1994 (serrano)                */
;*                                                                     */
;*    Un test d'evaluateur                                             */
;*---------------------------------------------------------------------*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module _ewal
   (main main))

;*---------------------------------------------------------------------*/
;*    Les environments ...                                             */
;*---------------------------------------------------------------------*/
(define the-global-environment '())

(linit-the-global-environment!)
(ldefine-primitive! '+ +fx)
(ldefine-primitive! '- -fx)
(ldefine-primitive! '< <fx)
(ldefine-primitive! 'eq? eq?)
(ldefine-primitive! 'car car)
(ldefine-primitive! 'cdr cdr)
(ldefine-primitive! 'cons cons)
(ldefine-primitive! 'null? null?)

(define (errow a b c)
   (print "*** ERROW: " a #\Newline b " -- " c))

;*---------------------------------------------------------------------*/
;*    ewal ...                                                         */
;*    sexp x env --> sexp                                              */
;*---------------------------------------------------------------------*/
(define (ewal exp . env)
   (for-each (lambda (x)
                ((compile-define (car x) (cdr x)) '()))
             env)
   (meaning (compile exp '() #f) '()))

;*---------------------------------------------------------------------*/
;*    meaning ...                                                      */
;*---------------------------------------------------------------------*/
(define (meaning byte-code runtime-env)
   (case (car byte-code)
      ((-2) ;; null
       '())
      ((-3) ;; faux
       #f)
      ((-4) ;; vrai
       #t)
      ((-5) ;; 1
       1)
      ((-1) ;; les constantes
       (cdr byte-code))
      ((0) ;; la premiere variable de l'env
       (car runtime-env))
      ((1) ;; la deuxieme
       (cadr runtime-env))
      ((2) ;; la troisieme
       (caddr runtime-env))
      ((3) ;; la quatrieme
       (cadddr runtime-env))
      ((4) ;; une variable qui est loin dans l'environment
       (do ((i 0 (+fx i 1))
            (env runtime-env (cdr env)))
           ((=fx i (cdr byte-code)) (car env))))
      ((5) ;; une variable globale
       (cdr (cdr byte-code)))
      ((6) ;; les variables globales indefinies
       (let ((global (assq (vector-ref (cdr byte-code) 0)
                           the-global-environment)))
          (if (not global)
              (error "ewal"
                     "Unbound variable"
                     (vector-ref (cdr byte-code) 0))
              (cdr global))))
      ((7) ;; l'affectation de la premiere variable de l'env
       (set-car! runtime-env (meaning (cdr byte-code) runtime-env)))
      ((8) ;; l'affectation de la deuxieme variable de l'env
       (set-car! (cdr runtime-env) (meaning (cdr byte-code) runtime-env)))
      ((9) ;; l'affectation de la troisieme variable de l'env
       (set-car! (cddr runtime-env) (meaning (cdr byte-code) runtime-env)))
      ((10) ;; l'affectation de la quatrieme variable de l'env
       (set-car! (cdddr runtime-env) (meaning (cdr byte-code) runtime-env)))
      ((11) ;; l'affectation de la nieme variable de l'env
       (do ((i 0 (+fx i 1))
            (env runtime-env (cdr env)))
           ((=fx i (cadr byte-code))
            (set-car! env (meaning (cddr byte-code) runtime-env)))))
      ((12) ;; l'affectation des variables globales
       (my-update-global! (cadr byte-code) (cddr byte-code) runtime-env))
      ((13) ;; l'affectation des variables globales non definies
       (let ((global (assq (vector-ref (cadr byte-code) 0)
                           the-global-environment)))
          (if (not global)
              (error "ewal"
                     "Unbound variable"
                     (vector-ref (cadr byte-code) 0))
              (my-update-global! global (cddr byte-code) runtime-env))))
      ((14) ;; if
       (if (meaning (cadr byte-code) runtime-env)
           (meaning (caddr byte-code) runtime-env)
           (meaning (cdddr byte-code) runtime-env)))
      ((15) ;; begin
       (let _loop_ ((body (cdr byte-code)))
          (if (null? (cdr body))
              (meaning (car body) runtime-env)
              (begin
                 (meaning (car body) runtime-env)
                 (_loop_ (cdr body))))))
      ((16) ;; define
       (let ((var (cadr byte-code))
             (val (cddr byte-code)))
          (let ((cell (assq var the-global-environment)))
             (if (pair? cell)
                 (begin
                    (print
                     "*** WARNING:bigloo:ewal\nredefinition of variable -- "
                     var)
                    (my-update-global! cell val runtime-env))
                 (begin
                    (set-cdr! the-global-environment
                              (cons (car the-global-environment)
                                    (cdr the-global-environment)))
                    (set-car! the-global-environment
                              (cons var (meaning val runtime-env)))
                    var)))))
      ((17) ;; lambda-0
       (lambda ()
          (meaning (cdr byte-code) runtime-env)))
      ((18) ;; lambda-1
       (lambda (x)
          (meaning (cdr byte-code) (cons x runtime-env))))
      ((19) ;; lambda-2
       (lambda (x y)
          (meaning (cdr byte-code) (cons x (cons y runtime-env)))))
      ((20) ;; lambda-3
       (lambda (x y z)
          (meaning (cdr byte-code)
                   (cons x (cons y (cons z runtime-env))))))
      ((21) ;; lambda-4
       (lambda (x y z)
          (meaning (cdr byte-code)
                   (cons x (cons y (cons z (cons z runtime-env)))))))
```

```
        ((22) ;; lambda--1
         (lambda x
            (meaning (cdr byte-code)
                     (cons x runtime-env))))
        ((23) ;; lambda--2
         (lambda (x . y)
            (meaning (cdr byte-code)
                     (cons x (cons y runtime-env)))))
        ((24) ;; lambda--3
         (lambda (x y . z)
            (meaning (cdr byte-code)
                     (cons x (cons y (cons z runtime-env))))))
        ((25) ;; lambda--4
         (lambda (x y z  . t)
            (meaning (cdr byte-code)
                     (cons x (cons y (cons z (cons z runtime-env)))))))
        ((26) ;; lambda-n
         (lambda x
            (let ((new-env (let _loop_ ((actuals x)
                                        (formals (cadr byte-code)))
                             (cond
                                ((null? formals)
                                 (if (not (null? actuals))
                                     (error "ewal"
                                            "Too many arguments provided"
                                            actuals)
                                     runtime-env))
                                ((null? actuals)
                                 (error "ewal"
                                        "Too fee arguments provided"
                                        formals))
                                ((not (pair? formals))
                                 (cons actuals runtime-env))
                                (else
                                 (cons (car actuals)
                                       (_loop_ (cdr actuals)
                                               (cdr formals)))))))))
              (meaning (cddr byte-code) new-env))))
        ((27) ;; app-0
         ((cdr (cadr byte-code))))
        ((28) ;; app-1
         ((cdr (cadr byte-code)) (meaning (caddr byte-code) runtime-env)))
        ((29) ;; app-2
         ((cdr (cadr byte-code)) (meaning (caddr byte-code) runtime-env)
                                 (meaning (cadddr byte-code) runtime-env)))
        ((30) ;; app-3
         ((cdr (cadr byte-code)) (meaning (caddr byte-code) runtime-env)
                                 (meaning (cadddr byte-code) runtime-env)
                                 (meaning (cadddr (cdr byte-code)) runtime-env)))
        ((31) ;; app-4
         ((cdr (cadr byte-code)) (meaning (caddr byte-code) runtime-env)
                                 (meaning (cadddr byte-code) runtime-env)
                                 (meaning (cadddr (cdr byte-code)) runtime-env)
                                 (meaning (cadddr (cddr byte-code)) runtime-env)))
        ((32) ;; app-n
         (apply (cdr (cadr byte-code)) (map (lambda (v)
                                              (meaning v runtime-env))
                                            (cddr byte-code))))
        ((33) ;; app-0
         ((cadr byte-code)))
        ((34) ;; app-1
         ((cadr byte-code) (meaning (cddr byte-code) runtime-env)))
        ((35) ;; app-2
         ((cadr byte-code) (meaning (caddr byte-code) runtime-env)
                           (meaning (cadddr byte-code) runtime-env)))
        ((36) ;; app-3
         ((cadr byte-code) (meaning (caddr byte-code) runtime-env)
                           (meaning (cadddr byte-code) runtime-env)
                           (meaning (cadddr (cdr byte-code)) runtime-env)))
        ((37) ;; app-4
         ((cadr byte-code) (meaning (caddr byte-code) runtime-env)
                           (meaning (cadddr byte-code) runtime-env)
                           (meaning (cadddr (cdr byte-code)) runtime-env)
                           (meaning (cadddr (cddr byte-code)) runtime-env)))
        ((38) ;; app-n
         (apply (cadr byte-code) (map (lambda (v)
                                        (meaning v runtime-env))
                                      (cddr byte-code))))
        ((39) ;; app-0
         ((meaning (cadr byte-code) runtime-env)))
        ((40) ;; app-1
         ((meaning (cadr byte-code) runtime-env)
          (meaning (caddr byte-code) runtime-env)))
        ((41) ;; app-2
         ((meaning (cadr byte-code) runtime-env)
          (meaning (caddr byte-code) runtime-env)
          (meaning (cadddr byte-code) runtime-env)))
        ((42) ;; app-3
         ((meaning (cadr byte-code) runtime-env)
          (meaning (caddr byte-code) runtime-env)
          (meaning (cadddr byte-code) runtime-env)
          (meaning (cadddr (cdr byte-code)) runtime-env)))
        ((43) ;; app-4
         ((meaning (cadr byte-code) runtime-env)
          (meaning (caddr byte-code) runtime-env)
          (meaning (cadddr byte-code) runtime-env)
          (meaning (cadddr (cdr byte-code)) runtime-env)
          (meaning (cadddr (cddr byte-code)) runtime-env)))
        ((44) ;; app-n
         (apply (meaning (cadr byte-code) runtime-env)
                (map (lambda (v)
                       (meaning v runtime-env))
                     (cddr byte-code))))))

;*---------------------------------------------------------------------*/
;*    compile ...                                                      */
;*    s-exp x env x { t, f } --> (lambda () ...)                       */
;*    -----------------------------------------------------------      */
;*    La phase d'expansion a genere une syntaxe correcte. On n'a donc  */
;*    plus du tout a la tester maintenant.                             */
;*---------------------------------------------------------------------*/
(define (compile exp env tail?)
   (cond
      ((not (pair? exp))
       (let ((atom exp))
          (cond
             ((symbol? atom)
              (compile-ref (variable atom env) tail?))
             (else
              (compile-cnst atom tail?)))))
      ((eq? (car exp) 'quote)
       (let ((cnst (cadr exp)))
          (compile-cnst cnst tail?)))
      ((eq? (car exp) 'if)
       (let ((si (cadr exp))
             (alors (caddr exp))
             (sinon (cadddr exp)))
          (compile-if (compile si env #f)
                      (compile alors env tail?)
                      (compile sinon env tail?))))
      ((eq? (car exp) 'begin)
       (let ((rest (cdr exp)))
          (compile-begin rest env)))
      ((eq? (car exp) 'define)
       (let ((var (cadr exp))
             (val (caddr exp)))
          (compile-define var (compile val '() #f))))
      ((eq? (car exp) 'set!)
       (let ((var (cadr exp))
             (val (caddr exp)))
          (compile-set (variable var env) (compile val env #f))))
      ((eq? (car exp) 'lambda)
       (let ((formals (cadr exp))
             (body    (caddr exp)))
          (compile-lambda formals
                          (compile body (my-extend-env formals env) #t)
                          tail?)))
      ((not (pair? (car exp)))
       (let ((fun (car exp))
             (args (cdr exp)))
          (let ((actuals (map (lambda (a) (compile a env #f)) args)))
             (cond
                ((symbol? fun)
                 (let ((proc (variable fun env)))
                    (cond
                       ((global? proc)
                        (compile-global-application proc
                                                    actuals
                                                    tail?))
                       (else
                        (compile-application (compile-ref proc #f)
                                             actuals
                                             tail?)))))
                ((procedure? fun)
                 (compile-compiled-application fun actuals tail?))
                (else
                 (errow "ewal" "Not a procedure" fun))))))
      (else
       (let ((fun (car exp))
             (args (cdr exp)))
          (let ((actuals (map (lambda (a) (compile a env #f)) args))
                (proc    (compile fun env #f)))
             (compile-application proc actuals tail?))))))

;*---------------------------------------------------------------------*/
;*    variable ...                                                     */
;*---------------------------------------------------------------------*/
(define (variable symbol env)
   (let ((offset (let loop ((env   env)
                            (count 0))
                    (cond
                       ((null? env)
                        #f)
                       ((eq? (car env) symbol)
                        count)
                       (else
```

```
                        (loop (cdr env) (+fx count 1)))))))
      (if offset
          offset
          (let ((global (assq symbol the-global-environment)))
             (if (not global)
                 '#(,symbol)
                 global)))))

;*---------------------------------------------------------------------*/
;*    global? ...                                                      */
;*---------------------------------------------------------------------*/
(define (global? variable)
   (pair? variable))

;*---------------------------------------------------------------------*/
;*    dynamic? ...                                                     */
;*---------------------------------------------------------------------*/
(define (dynamic? variable)
   (vector? variable))

;*---------------------------------------------------------------------*/
;*    compile-ref ...                                                  */
;*---------------------------------------------------------------------*/
(define (compile-ref variable tail?)
   (cond
      ((global? variable)
       '(5 . ,variable))
      ((dynamic? variable)
       '(6 . ,variable))
      (else
       (case variable
          ((0 1 2 3)
           '(,variable))
          (else
           '(4 . ,variable))))))

;*---------------------------------------------------------------------*/
;*    compile-set ...                                                  */
;*---------------------------------------------------------------------*/
(define (compile-set variable value)
   (cond
      ((global? variable)
       '(12 ,variable . ,value))
      ((dynamic? variable)
       '(13 ,variable . ,value))
      (else
       (case variable
          ((0)
           '(7 . ,value))
          ((1)
           '(8 . ,value))
          ((2)
           '(9 . ,value))
          ((3)
           '(10 . ,value))
          (else
           '(11 ,variable . ,value))))))

;*---------------------------------------------------------------------*/
;*    compile-cnst ...                                                 */
;*---------------------------------------------------------------------*/
(define (compile-cnst cnst tail?)
   (cond
      ((null? cnst)
       '(-2))
      ((and (boolean? cnst) (not cnst))
       '(-3))
      ((boolean? cnst)
       '(-4))
      ((eq? cnst 1)
       '(-5))
      (else
       '(-1 . ,cnst))))

;*---------------------------------------------------------------------*/
;*    compile-if ...                                                   */
;*---------------------------------------------------------------------*/
(define (compile-if test then else)
   '(14 ,test ,then . ,else))

;*---------------------------------------------------------------------*/
;*    compile-begin ...                                                */
;*---------------------------------------------------------------------*/
(define (compile-begin body env)
   (cond
      ((and (pair? body) (and (null? (cdr body))))
       ;; le cas degenere
       (compile (car body) env #t))
      (else
       (let ((body (let loop ((rest body))
                      (cond
                         ((null? rest)
                          (error "ewal" "Illegal form" body))
                         ((null? (cdr rest))
                          (cons (compile (car rest) env #t) '()))
                         (else
                          (cons (compile (car rest) env #f)
                                (loop (cdr rest))))))))
          '(15 ,@body)))))

;*---------------------------------------------------------------------*/
;*    linit-the-global-environment! ...                                */
;*---------------------------------------------------------------------*/
(define (linit-the-global-environment!)
   (if (pair? the-global-environment)
       'done
       ;; je ne peux pas utiliser de constante car quand cette fonction
       ;; sera appellee, je ne suis pas sur que le module soit deja
       ;; initialise.
       (set! the-global-environment (cons (cons #f #f) '()))))

;*---------------------------------------------------------------------*/
;*    compile-define ...                                               */
;*    -------------------------------------------------------------    */
;*    On ne rajoute pas en tete car elle contient la definition de     */
;*    'the-module-environment'. On rajoute donc en deuxieme.           */
;*---------------------------------------------------------------------*/
(define (compile-define var val)
   '(16 ,var . ,val))

;*---------------------------------------------------------------------*/
;*    ldefine-primitive! ...                                           */
;*    -------------------------------------------------------------    */
;*    Cette fonction est juste une forme abregee de la precedente, qui */
;*    construit le '(lambda () ...)' absent                            */
;*---------------------------------------------------------------------*/
(define (ldefine-primitive! var val)
   (set-cdr! the-global-environment
             (cons (car the-global-environment)
                   (cdr the-global-environment)))
   (set-car! the-global-environment (cons var val)))

;*---------------------------------------------------------------------*/
;*      my-update-global! ...                                          */
;*---------------------------------------------------------------------*/
(define (my-update-global! variable val runtime-env)
   (set-cdr! variable (meaning val runtime-env))
   (car variable))

;*---------------------------------------------------------------------*/
;*    my-extend-env ...                                                */
;*---------------------------------------------------------------------*/
(define (my-extend-env extend old-env)
   (let _loop_ ((extend extend))
      (cond
         ((null? extend)
          old-env)
         ((not (pair? extend))
          (cons extend old-env))
         (else
          (cons (car extend) (_loop_ (cdr extend)))))))

;*---------------------------------------------------------------------*/
;*    compile-lambda ...                                               */
;*---------------------------------------------------------------------*/
(define (compile-lambda formals body tail?)
   (cond
      ((null? formals)
       '(17 . ,body))
      ((null? (cdr formals))
       '(18 . ,body))
      ((null? (cddr formals))
       '(19 . ,body))
      ((null? (cdddr formals))
       '(20 . ,body))
      ((null? (cddddr formals))
       '(21 . ,body))
      (else
       '(26 ,formals . ,body))))

;*---------------------------------------------------------------------*/
;*    compile-global-application ...                                   */
;*---------------------------------------------------------------------*/
(define (compile-global-application proc actuals tail?)
   (case (length actuals)
      ((0 1 2 3)
       '(,(+fx (length actuals) 27) ,proc ,@actuals))
      (else
       '(32 ,proc ,@actuals))))

;*---------------------------------------------------------------------*/
;*    compile-compiled-application ...                                 */
;*---------------------------------------------------------------------*/
(define (compile-compiled-application proc actuals tail?)
   (case (length actuals)
      ((0 1 2 3)
```

```
         '(,(+fx (length actuals) 33) ,proc ,@actuals))
        (else
         '(38 ,proc ,@actuals))))

;*---------------------------------------------------------------------*/
;*    compile-application ...                                          */
;*---------------------------------------------------------------------*/
(define (compile-application proc actuals tail?)
   (case (length actuals)
      ((0 1 2 3)
       '(,(+fx (length actuals) 39) ,proc ,@actuals))
      (else
       '(44 ,proc ,@actuals))))

;*---------------------------------------------------------------------*/
;*    main ...                                                         */
;*---------------------------------------------------------------------*/
(ewal '(define fib (lambda (x)
                      (if (< x 2)
                          1
                          (+ (fib (- x 1)) (fib (- x 2)))))))

(define (main argv)
   (if (not (null? (cdr argv)))
       (let loop ((i (string->integer (cadr argv)))
                  (r (ewal '(fib 26))))
          (if (=fx i 1)
              (print r)
              (loop (-fx i 1) (ewal '(fib 26)))))))
```

queens/bigloo/queens.scm

```
;*---------------------------------------------------------------------*/
;*    serrano/these/src/queens/bigloo/queens.scm ...                   */
;*                                                                     */
;*    Author      :  Manuel Serrano                                    */
;*    Creation    :  Tue May 12 09:19:00 1992                          */
;*    Last change :  Wed Oct 12 09:10:59 1994 (serrano)                */
;*                                                                     */
;*    La resolution des huits reines d'apres L. Augustsson (lml)       */
;*---------------------------------------------------------------------*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module reine)

;*---------------------------------------------------------------------*/
;*    succ ...                                                         */
;*---------------------------------------------------------------------*/
(define (succ x)
   (+fx 1 x))

;*---------------------------------------------------------------------*/
;*    mmap ...                                                          */
;*---------------------------------------------------------------------*/
(define (mmap f)
   (labels ((map_rec (l acc)
                     (cond ((null? l)
                            (reverse! acc))
                           (else
                            (map_rec (cdr l) (cons (f (car l)) acc))))))
      (lambda (l)
         (map_rec l '()))))

;*---------------------------------------------------------------------*/
;*    filter ...                                                       */
;*---------------------------------------------------------------------*/
(define (filter p l)
   (let filter ((l l)
                (acc '()))
      (cond
         ((null? l)
          (reverse! acc))
         ((p (car l))
          (filter (cdr l) (cons (car l) acc)))
         (else
          (filter (cdr l) acc)))))

;*---------------------------------------------------------------------*/
;*    count ...                                                        */
;*---------------------------------------------------------------------*/
(define (count from to)
   (let loop ((from from)
              (to   to)
              (acc  '()))
      (if (>fx from to)
          (reverse! acc)
          (loop (succ from)
                to
                (cons from acc)))))

;*---------------------------------------------------------------------*/
;*    concmap ...                                                      */
;*---------------------------------------------------------------------*/
(define (concmap f l)
   (if (null? l)
       '()
       (append (f (car l)) (concmap f (cdr l)))))

;*---------------------------------------------------------------------*/
;*    nsoln ...                                                        */
;*---------------------------------------------------------------------*/
(define (nsoln nq)
   (labels ((safe (d x l)
                  (if (null? l)
                      #t
                      (let ((q (car l)))
                         (and (not (=fx x q))
                              (and (not (=fx x (+fx q d)))
                                   (and (not (=fx x (-fx q d)))
                                        (safe (+fx d 1) x (cdr l)))))))))
            (ok (l)
                (if (null? l)
                    #t
                    (safe 1 (car l) (cdr l)))))
      (let ((pos_l (count 1 nq)))
         (labels ((testcol (b) (filter ok ((mmap (lambda (q) (cons q b)))
                                           pos_l))))
            (labels ((gen (n)
                          (if (=fx n 0)
                              '(())
                              (let ((g (gen (-fx n 1))))
                                 (concmap testcol g)))))
               (length (gen nq)))))))

;*---------------------------------------------------------------------*/
;*    nsola ...                                                        */
;*---------------------------------------------------------------------*/
(define (nsoln_a nq)
   (labels ((ok (l)
                (if (null? l)
                    #t
                    (let* ((x (car l))
                           (l (cdr l)))
                       (labels ((safe (x d l)
                                      (if
                                       (null? l)
                                       #t
                                       (let* ((q (car l))
                                              (l (cdr l)))
                                          (and
                                           (not (=fx x q))
                                           (and (not (=fx x (+fx q d)))
                                                (and (not (=fx x (-fx q d)))
                                                     (safe x (+fx d 1) l)))))))))
                          (safe x 1 l))))))
      (labels ((gen (n)
                    (if (=fx n 0)
                        '(())
                        (concmap (lambda (b)
                                    (filter ok
                                            ((mmap (lambda (q) (cons q b)))
                                             (count 1 nq))))
                                 (gen (-fx n 1))))))
         (length (gen nq)))))

;*---------------------------------------------------------------------*/
;*    les formes top-level                                             */
;*---------------------------------------------------------------------*/
(print (cons (nsoln 10) (nsoln_a 10)))
```

bague/bigloo/bague.scm

```
;*=====================================================================*/
;*    serrano/these/src/bague/bigloo/bague.scm ...                     */
;*    -------------------------------------------------------------    */
;*    Author      :  Pierre Weis                                       */
;*    Creation    :  Fri Apr  1 10:00:21 1994                          */
;*    Last change :  Tue Oct 11 15:16:53 1994 (serrano)                */
;*    -------------------------------------------------------------    */
;*    Resolution recursive du Baguenaudier: bench les appels de        */
;*    fonctions et les acces aux vecteurs                              */
;*    avec 21 pierres le nombre de coups est 1398101                   */
```

```
;*    avec 24 pierres le nombre de coups est 11184810                  */
;*    f (n+1) = 2*f(n) + n mod 2 avec f 1 = 1                          */
;*=====================================================================*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module bague
   (main main))

(define nombre-de-coups 0)
(define nombre-de-pierres 24)

(define une-pierre 1)
(define une-case-vide 0)

(define jeu (make-vector nombre-de-pierres une-pierre))

(define (init-jeu)
   (set! nombre-de-coups 0)
   (let loop ((i (-fx nombre-de-pierres 1)))
      (if (<fx i 0)
          'done
          (begin
             (vector-set! jeu i une-pierre)
             (loop (-fx i 1))))))

(define (la-case n)
   (-fx n 1))

(define (enleve-la-pierre n)
   (if (eq? (vector-ref jeu (la-case n))  une-pierre)
       (vector-set! jeu (la-case n) une-case-vide)
       (error "bague" "cannot remove a stone from an empty slot" n)))

(define (pose-la-pierre n)
   (if (eq? (vector-ref jeu (la-case n)) une-case-vide)
       (vector-set! jeu (la-case n) une-pierre)
       (error "bague" "cannot lay a stone on a non empty slot" n)))

(define (autorise-mouvement n)
   (case n
      ((1) #t)
      ((2) (eq? (vector-ref jeu (la-case 1)) une-pierre))
      (else
       (and (eq? (vector-ref jeu (la-case (-fx n 1))) une-pierre)
            (letrec ((ok (lambda (b i)
                            (if (>fx i (la-case (-fx n 2)))
                                b
                                (ok (and b (eq? (vector-ref jeu i)
                                                une-case-vide))
                                    (+fx i 1))))))
              (ok #t 0))))))

(define (enleve-pierre n)
   (set! nombre-de-coups (+fx nombre-de-coups 1))
   (if (autorise-mouvement n)
       (enleve-la-pierre n)
       (error "bague" "forbidden action" n)))

(define (pose-pierre n)
   (set! nombre-de-coups (+fx nombre-de-coups 1))
   (if (autorise-mouvement n)
       (pose-la-pierre n)
       (error "bague" "forbidden action" n)))

(define (main argv)
   (letrec ((bague (lambda (n)
                      (case n
                         ((1) (enleve-pierre 1))
                         ((2) (enleve-pierre 2)
                              (enleve-pierre 1))
                         (else
                          (bague (-fx n 2))
                          (enleve-pierre n)
                          (repose (-fx n 2))
                          (bague (-fx n 1))))))
            (repose (lambda (n)
                       (case n
                          ((1) (pose-pierre 1))
                          ((2) (pose-pierre 1)
                               (pose-pierre 2))
                          (else
                           (repose (-fx n 1))
                           (bague (-fx n 2))
                           (pose-pierre n)
                           (repose (-fx n 2)))))))
      (init-jeu)
      (bague nombre-de-pierres)
      (print nombre-de-coups)))
```

sievev/bigloo/sievev.scm

```
;*=====================================================================*/
;*    serrano/these/src/sievev/bigloo/sievev.scm ...                   */
;*    -------------------------------------------------------------    */
;*    Author      :  Manuel Serrano                                    */
;*    Creation    :  Fri Apr  1 11:51:09 1994                          */
;*    Last change :  Tue Oct 11 15:31:04 1994 (serrano)                */
;*    -------------------------------------------------------------    */
;*    Le crible d'erathostene en utilisant des vecteurs                */
;*=====================================================================*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module sievev
   (main main))

(define (mod x y)
   (-fx x (*fx (/fx x y) y)))

(define (remove-multiples n v)
   (let loop ((i (-fx (vector-length v) 1)))
      (if (>=fx i n)
          (begin
             (if (=fx (mod (+fx i 2) n) 0)
                 (vector-set! v i #f))
             (loop (-fx i 1))))))

(define (tri-premiers v)
   (letrec ((tri-rec (lambda (i)
                        (if (<fx i (-fx (vector-length v) 1))
                            (begin
                               (if (vector-ref v i)
                                   (remove-multiples (+fx i 2) v))
                               (tri-rec (+fx i 1)))))))
      (tri-rec 0)))

(define (imprime-nombre-de-premiers v)
   (let ((premiers 0))
      (let loop ((i (-fx (vector-length v) 1)))
         (if (>fx i 0)
             (begin
                (if (vector-ref v i)
                    (set! premiers (+fx premiers 1)))
                (loop (-fx i 1)))))
      (print premiers)))

(define (main argv)
   (let ((v (make-vector 10000 #t)))
      (tri-premiers v)
      (imprime-nombre-de-premiers v)))
```

church/bigloo/church.scm

```
;*---------------------------------------------------------------------*/
;*    .../church.scm ...                                               */
;*                                                                     */
;*    Author      :  Manuel Serrano                                    */
;*    Creation    :  Tue May 12 11:26:16 1992                          */
;*    Last change :  Thu Mar  4 13:56:24 1993  (serrano)               */
;*                                                                     */
;*    Les entiers de Church                                            */
;*---------------------------------------------------------------------*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module double
   (main main)
   (include "misc/bigloo.sch"))

;*---------------------------------------------------------------------*/
;*    double ...                                                       */
;*---------------------------------------------------------------------*/
(define double
      (lambda (f)
         (lambda (x) (f (f x)))))

;*---------------------------------------------------------------------*/
;*    quad ...                                                         */
;*---------------------------------------------------------------------*/
(define quad  (double double))

;*---------------------------------------------------------------------*/
;*    oct ...                                                          */
```

```
;*---------------------------------------------------------------------*/
(define oct (quad quad))

;*---------------------------------------------------------------------*/
;*    succ ...                                                         */
;*---------------------------------------------------------------------*/
(define succ (lambda (x) (+fx 1 x)))

;*---------------------------------------------------------------------*/
;*    repeat ...                                                       */
;*---------------------------------------------------------------------*/
(define lrepeat
      (lambda (n)
         (if (>fx n 0)
             (begin
                (lrepeat (-fx n 1))
                (((double oct) succ) 1))
             (((double oct) succ) 1))))

;*---------------------------------------------------------------------*/
;*    Les formes top-level                                             */
;*---------------------------------------------------------------------*/
(define (main argv)
   (repeat (string->integer (cadr argv))
           (lrepeat 10)
           65537))
```

conform/bigloo/conform.scm

```
(module conform
   (main main))

;;; Functional and unstable

(define (vector-copy v)
  (let* ((length (vector-length v))
         (result (make-vector length)))
    (let loop ((n 0))
      (vector-set! result n (vector-ref v n))
      (if (= n length)
          v
          (loop (+ n 1))))))

(define (sort obj pred)
  (define (loop l)
    (if (and (pair? l) (pair? (cdr l)))
        (split l '() '())
        l))

  (define (split l one two)
    (if (pair? l)
        (split (cdr l) two (cons (car l) one))
        (merge (loop one) (loop two))))

  (define (merge one two)
    (cond ((null? one) two)
          ((pred (car two) (car one))
           (cons (car two)
                 (merge (cdr two) one)))
          (else
           (cons (car one)
                 (merge (cdr one) two)))))

  (cond ((or (pair? obj) (null? obj))
         (loop obj))
        ((vector? obj)
         (sort! (vector-copy obj) pred))
        (else
         (error '() "sort: argument should be a list or vector" obj))))

;; This merge sort is stable for partial orders (for predicates like
;; <=, rather than like <).

(define (sort! v pred)
  (define (sort-internal! vec temp low high)
    (if (< low high)
        (let* ((middle (quotient (+ low high) 2))
               (next (+ middle 1)))
          (sort-internal! temp vec low middle)
          (sort-internal! temp vec next high)
          (let loop ((p low) (p1 low) (p2 next))
            (if (not (> p high))
                (cond ((> p1 middle)
                       (vector-set! vec p (vector-ref temp p2))
                       (loop (+ p 1) p1 (+ p2 1)))
                      ((or (> p2 high)
                           (pred (vector-ref temp p1)
                                 (vector-ref temp p2)))
                       (vector-set! vec p (vector-ref temp p1))
                       (loop (+ p 1) (+ p1 1) p2))
                      (else
                       (vector-set! vec p (vector-ref temp p2))
                       (loop (+ p 1) p1 (+ p2 1)))))))))

  (if (not (vector? v))
      (error '() "sort!: argument not a vector" v))

  (sort-internal! v
                  (vector-copy v)
                  0
                  (- (vector-length v) 1))
  v)

;; SET OPERATIONS
; (representation as lists with distinct elements)

(define (adjoin element set)
  (if (memq element set) set (cons element set)))

(define (eliminate element set)
  (cond ((null? set) set)
        ((eq? element (car set)) (cdr set))
        (else (cons (car set) (eliminate element (cdr set))))))

(define (intersect list1 list2)
  (let loop ((l list1))
    (cond ((null? l) '())
          ((memq (car l) list2) (cons (car l) (loop (cdr l))))
          (else (loop (cdr l))))))

(define (union list1 list2)
  (if (null? list1)
      list2
      (union (cdr list1)
             (adjoin (car list1) list2))))

;; GRAPH NODES

; (define-structure
;   (internal-node
;    (print-procedure (unparser/standard-method
;                      'graph-node
;                      (lambda (state node)
;                        (unparse-object state (internal-node-name node))))))
;   name
;   (green-edges '())
;   (red-edges '())
;   blue-edges)

; Above is MIT version; below is portable

(define make-internal-node vector)
(define (internal-node-name node) (vector-ref node 0))
(define (internal-node-green-edges node) (vector-ref node 1))
(define (internal-node-red-edges node) (vector-ref node 2))
(define (internal-node-blue-edges node) (vector-ref node 3))
(define (set-internal-node-name! node name) (vector-set! node 0 name))
(define (set-internal-node-green-edges! node edges) (vector-set! node 1 edges))
(define (set-internal-node-red-edges! node edges) (vector-set! node 2 edges))
(define (set-internal-node-blue-edges! node edges) (vector-set! node 3 edges))

; End of portability stuff

(define (make-node name . blue-edges)   ; User's constructor
  (let ((name (if (symbol? name) (symbol->string name) name))
        (blue-edges (if (null? blue-edges) 'NOT-A-NODE-YET (car blue-edges))))
    (make-internal-node name '() '() blue-edges)))

(define (copy-node node)
  (make-internal-node (name node) '() '() (blue-edges node)))

; Selectors

(define name internal-node-name)
(define (make-edge-getter selector)
  (lambda (node)
    (if (or (none-node? node) (any-node? node))
        (error '() '() "Can't get edges from the ANY or NONE nodes")
        (selector node))))
(define red-edges (make-edge-getter internal-node-red-edges))
(define green-edges (make-edge-getter internal-node-green-edges))
(define blue-edges (make-edge-getter internal-node-blue-edges))

; Mutators

(define (make-edge-setter mutator!)
  (lambda (node value)
    (cond ((any-node? node)
           (error '() '() "Can't set edges from the ANY node"))
          ((none-node? node) 'OK)
          (else (mutator! node value)))))
(define set-red-edges! (make-edge-setter set-internal-node-red-edges!))
(define set-green-edges! (make-edge-setter set-internal-node-green-edges!))
```

```
(define set-blue-edges! (make-edge-setter set-internal-node-blue-edges!))

;; BLUE EDGES

; (define-structure
;   (blue-edge
;    (print-procedure
;     (unparser/standard-method
;      'blue-edge
;      (lambda (state edge)
;        (unparse-object state (blue-edge-operation edge))))))
;   operation arg-node res-node)

; Above is MIT version; below is portable

(define make-blue-edge vector)
(define (blue-edge-operation edge) (vector-ref edge 0))
(define (blue-edge-arg-node edge) (vector-ref edge 1))
(define (blue-edge-res-node edge) (vector-ref edge 2))
(define (set-blue-edge-operation! edge value) (vector-set! edge 0 value))
(define (set-blue-edge-arg-node! edge value) (vector-set! edge 1 value))
(define (set-blue-edge-res-node! edge value) (vector-set! edge 2 value))

; End of portability stuff

; Selectors
(define operation (lambda (x) (blue-edge-operation x)))
(define arg-node  (lambda (x) (blue-edge-arg-node x)))
(define res-node  (lambda (x) (blue-edge-res-node x)))

; Mutators
(define set-arg-node! (lambda (x y) (set-blue-edge-arg-node! x y)))
(define set-res-node! (lambda (x y) (set-blue-edge-res-node! x y)))

; Higher level operations on blue edges

(define (lookup-op op node)
  (let loop ((edges (blue-edges node)))
    (cond ((null? edges) '())
          ((eq? op (operation (car edges))) (car edges))
          (else (loop (cdr edges))))))

(define (has-op? op node)
  (not (null? (lookup-op op node))))

; Add a (new) blue edge to a node

; (define (adjoin-blue-edge! blue-edge node)
;   (let ((current-one (lookup-op (operation blue-edge) node)))
;     (cond ((null? current-one)
;            (set-blue-edges! node
;              (cons blue-edge (blue-edges node))))
;           ((and (eq? (arg-node current-one) (arg-node blue-edge))
;                 (eq? (res-node current-one) (res-node blue-edge)))
;            'OK)
;           (else (error '() "Two non-equivalent blue edges for op"
;                        blue-edge node)))))

;; GRAPHS

; (define-structure
;   (internal-graph
;    (print-procedure
;     (unparser/standard-method 'graph
;      (lambda (state edge)
;        (unparse-object state (map name (internal-graph-nodes edge)))))))
;   nodes already-met already-joined)

; Above is MIT version; below is portable

(define make-internal-graph vector)
(define (internal-graph-nodes graph) (vector-ref graph 0))
(define (internal-graph-already-met graph) (vector-ref graph 1))
(define (internal-graph-already-joined graph) (vector-ref graph 2))
(define (set-internal-graph-nodes! graph nodes) (vector-set! graph 0 nodes))

; End of portability stuff

; Constructor

(define (make-graph . nodes)
  (make-internal-graph nodes (make-empty-table) (make-empty-table)))

; Selectors

(define graph-nodes (lambda (x) (internal-graph-nodes x)))
(define already-met (lambda (x) (internal-graph-already-met x)))
(define already-joined (lambda (x) (internal-graph-already-joined x)))

; Higher level functions on graphs

(define (add-graph-nodes! graph nodes)
  (set-internal-graph-nodes! graph (cons nodes (graph-nodes graph))))


(define (copy-graph g)
  (define (copy-list l) (vector->list (list->vector l)))
  (make-internal-graph
   (copy-list (graph-nodes g))
   (already-met g)
   (already-joined g)))

(define (clean-graph g)
  (define (clean-node node)
    (if (not (or (any-node? node) (none-node? node)))
        (begin
          (set-green-edges! node '())
          (set-red-edges! node '()))))
  (for-each clean-node (graph-nodes g))
  g)

(define (canonicalize-graph graph classes)
  (define (fix node)
    (define (fix-set object selector mutator)
      (mutator object
               (map (lambda (node)
                      (find-canonical-representative node classes))
                    (selector object))))
    (if (not (or (none-node? node) (any-node? node)))
        (begin
          (fix-set node green-edges set-green-edges!)
          (fix-set node red-edges set-red-edges!)
          (for-each
           (lambda (blue-edge)
             (set-arg-node! blue-edge
                            (find-canonical-representative
                             (arg-node blue-edge) classes))
             (set-res-node! blue-edge
                            (find-canonical-representative
                             (res-node blue-edge) classes)))
           (blue-edges node))))
    node)
  (define (fix-table table)
    (define (canonical? node)
      (eq? node (find-canonical-representative node classes)))
    (define (filter-and-fix predicate-fn update-fn list)
      (let loop ((list list))
        (cond ((null? list) '())
              ((predicate-fn (car list))
               (cons (update-fn (car list)) (loop (cdr list))))
              (else (loop (cdr list))))))
    (define (fix-line line)
      (filter-and-fix
       (lambda (entry) (canonical? (car entry)))
       (lambda (entry) (cons (car entry)
                             (find-canonical-representative
                              (cdr entry) classes)))
       line))
    (if (null? table)
        '()
        (cons (car table)
              (filter-and-fix
               (lambda (entry) (canonical? (car entry)))
               (lambda (entry) (cons (car entry) (fix-line (cdr entry))))
               (cdr table)))))
  (make-internal-graph
   (map (lambda (class) (fix (car class))) classes)
   (fix-table (already-met graph))
   (fix-table (already-joined graph))))

;; USEFUL NODES

(define none-node (make-node 'none '#T))
(define (none-node? node) (eq? node none-node))

(define any-node (make-node 'any '()))
(define (any-node? node) (eq? node any-node))

;; COLORED EDGE TESTS

(define (green-edge? from-node to-node)
  (cond ((any-node? from-node) '#F)
        ((none-node? from-node) '#T)
        ((memq to-node (green-edges from-node)) '#T)
        (else '#F)))

(define (red-edge? from-node to-node)
  (cond ((any-node? from-node) '#F)
        ((none-node? from-node) '#T)
        ((memq to-node (red-edges from-node)) '#T)
        (else '#F)))

;; SIGNATURE

; Return signature (i.e. <arg, res>) given an operation and a node

(define sig
```

```
    (let ((none-comma-any (cons none-node any-node)))
      (lambda (op node)                    ; Returns (arg, res)
        (let ((the-edge (lookup-op op node)))
          (if (not (null? the-edge))
              (cons (arg-node the-edge) (res-node the-edge))
              none-comma-any)))))

; Selectors from signature

(define (arg pair) (car pair))
(define (res pair) (cdr pair))

;; CONFORMITY

(define (conforms? t1 t2)
  (define nodes-with-red-edges-out '())
  (define (add-red-edge! from-node to-node)
    (set-red-edges! from-node (adjoin to-node (red-edges from-node)))
    (set! nodes-with-red-edges-out
          (adjoin from-node nodes-with-red-edges-out)))
  (define (greenify-red-edges! from-node)
    (set-green-edges! from-node
                      (append (red-edges from-node) (green-edges from-node)))
    (set-red-edges! from-node '()))
  (define (delete-red-edges! from-node)
    (set-red-edges! from-node '()))
  (define (does-conform t1 t2)
    (cond ((or (none-node? t1) (any-node? t2)) '#T)
          ((or (any-node? t1) (none-node? t2)) '#F)
          ((green-edge? t1 t2) '#T)
          ((red-edge? t1 t2) '#T)
          (else
           (add-red-edge! t1 t2)
           (let loop ((blues (blue-edges t2)))
             (if (null? blues)
                 '#T
                 (let* ((current-edge (car blues))
                        (phi (operation current-edge)))
                   (and (has-op? phi t1)
                        (does-conform
                         (res (sig phi t1))
                         (res (sig phi t2)))
                        (does-conform
                         (arg (sig phi t2))
                         (arg (sig phi t1)))
                        (loop (cdr blues)))))))))
  (let ((result (does-conform t1 t2)))
    (for-each (if result greenify-red-edges! delete-red-edges!)
              nodes-with-red-edges-out)
    result))

(define (equivalent? a b)
  (and (conforms? a b) (conforms? b a)))

;; EQUIVALENCE CLASSIFICATION
; Given a list of nodes, return a list of equivalence classes

(define (classify nodes)
  (let node-loop ((classes '())
                  (nodes nodes))
    (if (null? nodes)
        (map (lambda (class)
               (sort class
                     (lambda (node1 node2)
                       (< (string-length (name node1))
                          (string-length (name node2))))))
             classes)
        (let ((this-node (car nodes)))
          (define (add-node classes)
            (cond ((null? classes) (list (list this-node)))
                  ((equivalent? this-node (caar classes))
                   (cons (cons this-node (car classes))
                         (cdr classes)))
                  (else (cons (car classes)
                              (add-node (cdr classes))))))
          (node-loop (add-node classes)
                     (cdr nodes))))))

; Given a node N and a classified set of nodes,
; find the canonical member corresponding to N

(define (find-canonical-representative element classification)
  (let loop ((classes classification))
    (cond ((null? classes) (error '() "Can't classify" element))
          ((memq element (car classes)) (car (car classes)))
          (else (loop (cdr classes))))))

; Reduce a graph by taking only one member of each equivalence
; class and canonicalizing all outbound pointers

(define (reduce graph)
  (let ((classes (classify (graph-nodes graph))))
    (canonicalize-graph graph classes)))


;; TWO DIMENSIONAL TABLES

(define (make-empty-table) (list 'TABLE))
(define (lookup table x y)
  (let ((one (assq x (cdr table))))
    (if one
        (let ((two (assq y (cdr one))))
          (if two (cdr two) '#f))
        '#f)))
(define (insert! table x y value)
  (define (make-singleton-table x y)
    (list (cons x y)))
  (let ((one (assq x (cdr table))))
    (if one
        (set-cdr! one (cons (cons y value) (cdr one)))
        (set-cdr! table (cons (cons x (make-singleton-table y value))
                              (cdr table))))))

;; MEET/JOIN
; These update the graph when computing the node for node1*node2

(define (blue-edge-operate arg-fn res-fn graph op sig1 sig2)
  (make-blue-edge op
                  (arg-fn graph (arg sig1) (arg sig2))
                  (res-fn graph (res sig1) (res sig2))))

(define (meet graph node1 node2)
  (cond ((eq? node1 node2) node1)
        ((or (any-node? node1) (any-node? node2)) any-node) ; canonicalize
        ((none-node? node1) node2)
        ((none-node? node2) node1)
        ((lookup (already-met graph) node1 node2)) ; return it if found
        ((conforms? node1 node2) node2)
        ((conforms? node2 node1) node1)
        (else
         (let ((result
                (make-node (string-append "("
                                          (name node1)
                                          " ^ "
                                          (name node2) ")"))))
           (add-graph-nodes! graph result)
           (insert! (already-met graph) node1 node2 result)
           (set-blue-edges! result
             (map
              (lambda (op)
                (blue-edge-operate join
                                   meet
                                   graph
                                   op
                                   (sig op node1)
                                   (sig op node2)))
              (intersect (map operation (blue-edges node1))
                         (map operation (blue-edges node2)))))
           result))))

(define (join graph node1 node2)
  (cond ((eq? node1 node2) node1)
        ((any-node? node1) node2)
        ((any-node? node2) node1)
        ((or (none-node? node1) (none-node? node2)) none-node) ; canonicalize
        ((lookup (already-joined graph) node1 node2)) ; return it if found
        ((conforms? node1 node2) node1)
        ((conforms? node2 node1) node2)
        (else
         (let ((result
                (make-node (string-append "("
                                          (name node1)
                                          " v "
                                          (name node2) ")"))))
           (add-graph-nodes! graph result)
           (insert! (already-joined graph) node1 node2 result)
           (set-blue-edges! result
             (map
              (lambda (op)
                (blue-edge-operate meet
                                   join
                                   graph
                                   op
                                   (sig op node1)
                                   (sig op node2)))
              (union (map operation (blue-edges node1))
                     (map operation (blue-edges node2)))))
           result))))

;; MAKE A LATTICE FROM A GRAPH

(define (make-lattice g print?)
  (define (step g)
    (let* ((copy (copy-graph g))
           (nodes (graph-nodes copy)))
      (for-each (lambda (first)
                  (for-each (lambda (second)
```

```
                              (meet copy first second)
                              (join copy first second))
                             nodes))
                  nodes)
       copy))
   (define (loop g count)
     (if print? (display count))
     (let ((lattice (step g)))
       (if print? (begin (display " -> ")
                         (display (length (graph-nodes lattice)))))
       (let* ((new-g (reduce lattice))
              (new-count (length (graph-nodes new-g))))
         (if (= new-count count)
             (begin
               (if print? (newline))
               new-g)
             (begin
               (if print? (begin (display " -> ")
                                 (display new-count) (newline)))
               (loop new-g new-count))))))
   (let ((graph
          (apply make-graph
                 (adjoin any-node
                         (adjoin none-node (graph-nodes (clean-graph g)))))))
     (loop graph (length (graph-nodes graph)))))

;; DEBUG and TEST

(define a '())
(define b '())
(define c '())
(define d '())

(define (reset)
  (set! a (make-node 'a))
  (set! b (make-node 'b))
  (set-blue-edges! a (list (make-blue-edge 'phi any-node b)))
  (set-blue-edges! b (list (make-blue-edge 'phi any-node a)
                           (make-blue-edge 'theta any-node b)))
  (set! c (make-node "c"))
  (set! d (make-node "d"))
  (set-blue-edges! c (list (make-blue-edge 'theta any-node b)))
  (set-blue-edges! d (list (make-blue-edge 'phi any-node c)
                           (make-blue-edge 'theta any-node d)))
  '(made a b c d))

(define (test)
  (reset)
  (map name
       (graph-nodes (make-lattice (make-graph a b c d any-node none-node)
                                  '#f))))

(define (go)
  (reset)
  (test))

;;; call: (go)

(define (main argv)
   (let loop ((i (string->number (cadr argv))))
      (if (= i 1)
          (begin
             (display "CONFORM")
             (display (test))
             (newline))
          (begin
             (test)
             (loop (- i 1))))))
```

earley/bigloo/earley.scm

```
; File: "earley.scm"   (c) 1990, Marc Feeley

; Earley parser.
(module early
   (main main))

; (make-parser grammar lexer) is used to create a parser from the grammar
; description 'grammar' and the lexer function 'lexer'.
;
; A grammar is a list of definitions.  Each definition defines a non-terminal
; by a set of rules.  Thus a definition has the form: (nt rule1 rule2...).
; A given non-terminal can only be defined once.  The first non-terminal
; defined is the grammar's goal.  Each rule is a possibly empty list of
; non-terminals.  Thus a rule has the form: (nt1 nt2...).  A non-terminal
; can be any scheme value.  Note that all grammar symbols are treated as
; non-terminals.  This is fine though because the lexer will be outputing
; non-terminals.
;
; The lexer defines what a token is and the mapping between tokens and
; the grammar's non-terminals.  It is a function of one argument, the input,
; that returns the list of tokens corresponding to the input.  Each token is
; represented by a list.  The first element is some 'user-defined' information
; associated with the token and the rest represents the token's class(es) (as a
; list of non-terminals that this token corresponds to).
;
; The result of 'make-parser' is a function that parses the single input it
; is given into the grammar's goal.  The result is a 'parse' which can be
; manipulated with the procedures: 'parse->parsed?', 'parse->trees'
; and 'parse->nb-trees' (see below).
;
; Let's assume that we want a parser for the grammar
;
;  S -> x = E
;  E -> E + E | V
;  V -> V y |
;
; and that the input to the parser is a string of characters.  Also, assume we
; would like to map the characters 'x', 'y', '+' and '=' into the corresponding
; non-terminals in the grammar.  Such a parser could be created with
;
; (make-parser
;   '(
;      (s (x = e))
;      (e (e + e) (v))
;      (v (v y) ())
;    )
;   (lambda (str)
;     (map (lambda (char)
;            (list char ; user-info = the character itself
;                  (case char
;                    ((#\x) 'x)
;                    ((#\y) 'y)
;                    ((#\+) '+)
;                    ((#\=) '=)
;                    (else (error "lexer error")))))
;          (string->list str)))
; )
;
; An alternative definition (that does not check for lexical errors) is
;
; (make-parser
;   '(
;      (s (#\x #\= e))
;      (e (e #\+ e) (v))
;      (v (v #\y) ())
;    )
;   (lambda (str) (map (lambda (char) (list char char)) (string->list str)))
; )
;
; To help with the rest of the discussion, here are a few definitions:
;
; An input pointer (for an input of 'n' tokens) is a value between 0 and 'n'.
; It indicates a point between two input tokens (0 = beginning, 'n' = end).
; For example, if 'n' = 4, there are 5 input pointers:
;
;   input                  token1     token2     token3     token4
;   input pointers       0          1          2          3          4
;
; A configuration indicates the extent to which a given rule is parsed (this
; is the common 'dot notation').  For simplicity, a configuration is
; represented as an integer, with successive configurations in the same
; rule associated with successive integers.  It is assumed that the grammar
; has been extended with rules to aid scanning.  These rules are of the
; form 'nt ->', and there is one such rule for every non-terminal.  Note
; that these rules are special because they only apply when the corresponding
; non-terminal is returned by the lexer.
;
; A configuration set is a configuration grouped with the set of input pointers
; representing where the head non-terminal of the configuration was predicted.
;
; Here are the rules and configurations for the grammar given above:
;
;  S -> .         \
;       0          |
;  x -> .          |
;       1          |
;  = -> .          |
;       2          |
;  E -> .          |
;       3           > special rules (for scanning)
;  + -> .          |
;       4          |
;  V -> .          |
;       5          |
;  y -> .          |
;       6         /
;  S -> .  x  .  =  .  E  .
;       7     8     9     10
;  E -> .  E  .  +  .  E  .
;       11    12    13    14
;  E -> .  V  .
;       15    16
```

```
;  V -> .  V  .  y  .
;       17    18    19
;  V -> .
;       20
;
; Starters of the non-terminal 'nt' are configurations that are leftmost
; in a non-special rule for 'nt'.  Enders of the non-terminal 'nt' are
; configurations that are rightmost in any rule for 'nt'.  Predictors of the
; non-terminal 'nt' are configurations that are directly to the left of 'nt'
; in any rule.
;
; For the grammar given above,
;
;   Starters of V   = (17 20)
;   Enders of V     = (5 19 20)
;   Predictors of V = (15 17)

(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 (index 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 (index 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
              (vector-set! steps i (- i nb-nts))
              (vector-set! names i (list (vector-ref nts i) 0))
              (vector-set! enders i (list i))
              (nt-loop (- i 1)))))

        (let def-loop ((defs grammar) (conf (vector-length nts)))
          (if (pair? defs)
            (let* ((def (car defs))
                   (head (car def)))
              (let rule-loop ((rules (cdr def)) (conf conf) (rule-num 1))
                (if (pair? rules)
                  (let ((rule (car rules)))
                    (vector-set! names conf (list head rule-num))
                    (add-conf conf head nts starters)
                    (let loop ((l rule) (conf conf))
                      (if (pair? l)
                        (let ((nt (car l)))
                          (vector-set! steps conf (index nt nts))
                          (add-conf conf nt nts predictors)
                          (loop (cdr l) (+ conf 1)))
                        (begin
                          (vector-set! steps conf (- (index head nts) nb-nts))
                          (add-conf conf head nts enders)
                          (rule-loop (cdr rules) (+ conf 1) (+ rule-num 1))))))
                  (def-loop (cdr defs) conf))))))))

    ; Now, for each non-terminal, compute the starters, enders and predictors and
    ; the names and steps tables.

    (setup-tables grammar nts starters enders predictors steps names)

    ; Build the parser description

    (let ((parser-descr (vector lexer
                                nts
                                starters
                                enders
                                predictors
                                steps
                                names)))
      (lambda (input)

        (define (index 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 (comp-tok tok nts) ; transform token to parsing format
          (let loop ((l1 (cdr tok)) (l2 '()))
            (if (pair? l1)
              (let ((i (index (car l1) nts)))
                (if i
                  (loop (cdr l1) (cons i l2))
                  (loop (cdr l1) l2)))
              (cons (car tok) (reverse l2)))))

        (define (input->tokens input lexer nts)
          (list->vector (map (lambda (tok) (comp-tok tok nts)) (lexer input))))

        (define (make-states nb-toks nb-confs)
          (let ((states (make-vector (+ nb-toks 1) #f)))
            (let loop ((i nb-toks))
              (if (>= i 0)
                (let ((v (make-vector (+ nb-confs 1) #f)))
                  (vector-set! v 0 -1)
                  (vector-set! states i v)
                  (loop (- i 1)))
                states))))

        (define (conf-set-get state conf)
          (vector-ref state (+ conf 1)))

        (define (conf-set-get* state state-num conf)
          (let ((conf-set (conf-set-get state conf)))
            (if conf-set
              conf-set
              (let ((conf-set (make-vector (+ state-num 6) #f)))
                (vector-set! conf-set 1 -3) ; old elems tail (points to head)
                (vector-set! conf-set 2 -1) ; old elems head
                (vector-set! conf-set 3 -1) ; new elems tail (points to head)
                (vector-set! conf-set 4 -1) ; new elems head
                (vector-set! state (+ conf 1) conf-set)
                conf-set))))

        (define (conf-set-merge-new! conf-set)
          (vector-set! conf-set
            (+ (vector-ref conf-set 1) 5)
            (vector-ref conf-set 4))
          (vector-set! conf-set 1 (vector-ref conf-set 3))
          (vector-set! conf-set 3 -1)
          (vector-set! conf-set 4 -1))

        (define (conf-set-head conf-set)
          (vector-ref conf-set 2))

        (define (conf-set-next conf-set i)
          (vector-ref conf-set (+ i 5)))
```

```
(define (conf-set-member? state conf i)
  (let ((conf-set (vector-ref state (+ conf 1))))
    (if conf-set
      (conf-set-next conf-set i)
      #f)))

(define (conf-set-adjoin state conf-set conf i)
  (let ((tail (vector-ref conf-set 3))) ; put new element at tail
    (vector-set! conf-set (+ i 5) -1)
    (vector-set! conf-set (+ tail 5) i)
    (vector-set! conf-set 3 i)
    (if (< tail 0)
      (begin
        (vector-set! conf-set 0 (vector-ref state 0))
        (vector-set! state 0 conf)))))

(define (conf-set-adjoin* states state-num l i)
  (let ((state (vector-ref states state-num)))
    (let loop ((l1 l))
      (if (pair? l1)
        (let* ((conf (car l1))
               (conf-set (conf-set-get* state state-num conf)))
          (if (not (conf-set-next conf-set i))
            (begin
              (conf-set-adjoin state conf-set conf i)
              (loop (cdr l1)))
            (loop (cdr l1))))))))

(define (conf-set-adjoin** states states* state-num conf i)
  (let ((state (vector-ref states state-num)))
    (if (conf-set-member? state conf i)
      (let* ((state* (vector-ref states* state-num))
             (conf-set* (conf-set-get* state* state-num conf)))
        (if (not (conf-set-next conf-set* i))
          (conf-set-adjoin state* conf-set* conf i))
        #t)
      #f)))

(define (conf-set-union state conf-set conf other-set)
  (let loop ((i (conf-set-head other-set)))
    (if (>= i 0)
      (if (not (conf-set-next conf-set i))
        (begin
          (conf-set-adjoin state conf-set conf i)
          (loop (conf-set-next other-set i)))
        (loop (conf-set-next other-set i))))))

(define (forw states state-num starters enders predictors steps nts)

  (define (predict state state-num conf-set conf nt starters enders)

    ; add configurations which start the non-terminal 'nt' to the
    ; right of the dot

    (let loop1 ((l (vector-ref starters nt)))
      (if (pair? l)
        (let* ((starter (car l))
               (starter-set (conf-set-get* state state-num starter)))
          (if (not (conf-set-next starter-set state-num))
            (begin
              (conf-set-adjoin state starter-set starter state-num)
              (loop1 (cdr l)))
            (loop1 (cdr l))))))

    ; check for possible completion of the non-terminal 'nt' to the
    ; right of the dot

    (let loop2 ((l (vector-ref enders nt)))
      (if (pair? l)
        (let ((ender (car l)))
          (if (conf-set-member? state ender state-num)
            (let* ((next (+ conf 1))
                   (next-set (conf-set-get* state state-num next)))
              (conf-set-union state next-set next conf-set)
              (loop2 (cdr l)))
            (loop2 (cdr l)))))))

  (define (reduce states state state-num conf-set head preds)

    ; a non-terminal is now completed so check for reductions that
    ; are now possible at the configurations 'preds'

    (let loop1 ((l preds))
      (if (pair? l)
        (let ((pred (car l)))
          (let loop2 ((i head))
            (if (>= i 0)
              (let ((pred-set (conf-set-get (vector-ref states i)
                                            pred)))
                (if pred-set
                  (let* ((next (+ pred 1))
                         (next-set (conf-set-get* state state-num
                                                  next)))
                    (conf-set-union state next-set next pred-set)))
                (loop2 (conf-set-next conf-set i)))
              (loop1 (cdr l))))))))

  (let ((state (vector-ref states state-num))
        (nb-nts (vector-length nts)))
    (let loop ()
      (let ((conf (vector-ref state 0)))
        (if (>= conf 0)
          (let* ((step (vector-ref steps conf))
                 (conf-set (vector-ref state (+ conf 1)))
                 (head (vector-ref conf-set 4)))
            (vector-set! state 0 (vector-ref conf-set 0))
            (conf-set-merge-new! conf-set)
            (if (>= step 0)
              (predict state
                       state-num
                       conf-set
                       conf
                       step
                       starters
                       enders)
              (let ((preds (vector-ref predictors (+ step nb-nts))))
                (reduce states state state-num conf-set head preds)))
            (loop)))))))

(define (forward starters enders predictors steps nts toks)
  (let* ((nb-toks (vector-length toks))
         (nb-confs (vector-length steps))
         (states (make-states nb-toks nb-confs))
         (goal-starters (vector-ref starters 0)))
    (conf-set-adjoin* states 0 goal-starters 0) ; predict goal
    (forw states 0 starters enders predictors steps nts)
    (let loop ((i 0))
      (if (< i nb-toks)
        (let ((tok-nts (cdr (vector-ref toks i))))
          (conf-set-adjoin* states (+ i 1) tok-nts i) ; scan token
          (forw states (+ i 1) starters enders predictors steps nts)
          (loop (+ i 1)))))
    states))

(define (produce conf i j enders steps toks states states* nb-nts)
  (let ((prev (- conf 1)))
    (if (and (>= conf nb-nts) (>= (vector-ref steps prev) 0))
      (let loop1 ((l (vector-ref enders (vector-ref steps prev))))
        (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)))
                (if (>= k 0)
                  (begin
                    (and (>= k i)
                         (conf-set-adjoin** states states* k prev i)
                         (conf-set-adjoin** states states* j ender k))
                    (loop2 (conf-set-next ender-set k)))
                  (loop1 (cdr l))))
              (loop1 (cdr l)))))))))

(define (back states states* state-num enders steps nb-nts toks)
  (let ((state* (vector-ref states* state-num)))
    (let loop1 ()
      (let ((conf (vector-ref state* 0)))
        (if (>= conf 0)
          (let* ((conf-set (vector-ref state* (+ conf 1)))
                 (head (vector-ref conf-set 4)))
            (vector-set! state* 0 (vector-ref conf-set 0))
            (conf-set-merge-new! conf-set)
            (let loop2 ((i head))
              (if (>= i 0)
                (begin
                  (produce conf i state-num enders steps
                           toks states states* nb-nts)
                  (loop2 (conf-set-next conf-set i)))
                (loop1)))))))))

(define (backward states enders steps nts toks)
  (let* ((nb-toks (vector-length toks))
         (nb-confs (vector-length steps))
         (nb-nts (vector-length nts))
         (states* (make-states nb-toks nb-confs))
         (goal-enders (vector-ref enders 0)))
    (let loop1 ((l goal-enders))
      (if (pair? l)
        (let ((conf (car l)))
          (conf-set-adjoin** states states* nb-toks conf 0)
          (loop1 (cdr l)))))
    (let loop2 ((i nb-toks))
      (if (>= i 0)
        (begin
          (back states states* i enders steps nb-nts toks)
          (loop2 (- i 1)))))
```

```
              states*))

          (define (parsed? nt i j nts enders states)
            (let ((nt* (index nt nts)))
              (if nt*
                (let ((nb-nts (vector-length nts)))
                  (let loop ((l (vector-ref enders nt*)))
                    (if (pair? l)
                      (let ((conf (car l)))
                        (if (conf-set-member? (vector-ref states j) conf i)
                          #t
                          (loop (cdr l))))
                      #f)))
                #f)))

          (define (deriv-trees conf i j enders steps names toks states nb-nts)
            (let ((name (vector-ref names conf)))

              (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* (index 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* (index 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)))

        (define (test)
          (let ((p (make-parser '( (s (a) (s s)) )
                                (lambda (l) (map (lambda (x) (list x x)) l)))))
            (let ((x (p '(a a a a a a a a a))))
              (length (parse->trees x 's 0 9)))))

        (define (main argv)
          (let loop ((i (string->number (cadr argv))))
            (if (= i 1)
                (begin
                  (display "EARLY")
                  (display (test))
                  (newline))
                (begin
                  (test)
                  (loop (- i 1))))))
```

peval/bigloo/peval.scm

```
;------------------------------------------------------------------------------
;
;                           A simple partial evaluator
;
;                             Marc Feeley (05/15/88)
;
;------------------------------------------------------------------------------

(module peval
   (static (last-pair l))
   (main main))

; Utilities

(define (every? pred? l)
  (let loop ((l l))
    (or (null? l) (and (pred? (car l)) (loop (cdr l))))))

(define (some? pred? l)
  (let loop ((l l))
    (if (null? l) #f (or (pred? (car l)) (loop (cdr l))))))

(define (map2 f l1 l2)
  (let loop ((l1 l1) (l2 l2))
    (if (pair? l1)
      (cons (f (car l1) (car l2)) (loop (cdr l1) (cdr l2)))
      '())))

(define (last-pair l)
  (let loop ((l l))
    (let ((x (cdr l))) (if (pair? x) (loop x) l))))

(define (proper-list? l) (list? l))

;------------------------------------------------------------------------------
;
; The partial evaluator.

(define (partial-evaluate proc args)
  (peval (alphatize proc '()) args))

(define (alphatize exp env) ; return a copy of 'exp' where each bound var has
  (define (alpha exp)       ; been renamed (to prevent aliasing problems)
    (cond ((const-expr? exp)
           (quot (const-value exp)))
          ((symbol? exp)
           (let ((x (assq exp env))) (if x (cdr x) exp)))
          ((or (eq? (car exp) 'if) (eq? (car exp) 'begin))
           (cons (car exp) (map alpha (cdr exp))))
          ((or (eq? (car exp) 'let) (eq? (car exp) 'letrec))
           (let ((new-env (new-variables (map car (cadr exp)) env)))
             (list (car exp)
                   (map (lambda (x)
                          (list (cdr (assq (car x) new-env))
                                (if (eq? (car exp) 'let)
                                  (alpha (cadr x))
                                  (alphatize (cadr x) new-env))))
                        (cadr exp))
                   (alphatize (caddr exp) new-env))))
          ((eq? (car exp) 'lambda)
           (let ((new-env (new-variables (cadr exp) env)))
             (list 'lambda
                   (map (lambda (x) (cdr (assq x new-env))) (cadr exp))
                   (alphatize (caddr exp) new-env))))
          (else
           (map alpha exp))))
  (alpha exp))

(define (const-expr? expr) ; is 'expr' a constant expression?
  (and (not (symbol? expr))
       (or (not (pair? expr))
           (eq? (car expr) 'quote))))

(define (const-value expr) ; return the value of a constant expression
  (if (pair? expr) ; then it must be a quoted constant
    (cadr expr)
    expr))

(define (quot val) ; make a quoted constant whose value is 'val'
  (list 'quote val))

(define (new-variables parms env)
  (append (map (lambda (x) (cons x (new-variable x))) parms) env))

(define *current-num* 0)

(define (new-variable name)
  (set! *current-num* (+ *current-num* 1))
  (string->symbol
    (string-append (symbol->string name)
                   "_"
                   (number->string *current-num*))))

;------------------------------------------------------------------------------
;
; (peval proc args) will transform a procedure that is known to be called
; with constants as some of its arguments into a specialized procedure that
; is 'equivalent' but accepts only the non-constant parameters.  'proc' is the
; list representation of a lambda-expression and 'args' is a list of values,
; one for each parameter of the lambda-expression.  A special value (i.e.
; 'not-constant') is used to indicate an argument that is not a constant.
; The returned procedure is one that has as parameters the parameters of the
; original procedure which are NOT passed constants.  Constants will have been
; substituted for the constant parameters that are referenced in the body
; of the procedure.
;
; For example:
;
;   (peval
;     '(lambda (x y z) (f z x y)) ; the procedure
;     (list 1 not-constant #t))   ; the knowledge about x, y and z
;
; will return: (lambda (y) (f '#t '1 y))

(define (peval proc args)
  (simplify!
    (let ((parms (cadr proc))  ; get the parameter list
          (body (caddr proc))) ; get the body of the procedure
      (list 'lambda
            (remove-constant parms args) ; remove the constant parameters
            (beta-subst ; in the body, replace variable refs to the constant
              body      ; parameters by the corresponding constant
              (map2 (lambda (x y) (if (not-constant? y)
                                      '(()) (cons x (quot y))))
                    parms
                    args))))))

(define not-constant (list '?)) ; special value indicating non-constant parms.

(define (not-constant? x) (eq? x not-constant))

(define (remove-constant l a) ; remove from list 'l' all elements whose
  (cond ((null? l)            ; corresponding element in 'a' is a constant
         '())
        ((not-constant? (car a))
         (cons (car l) (remove-constant (cdr l) (cdr a))))
        (else
         (remove-constant (cdr l) (cdr a)))))

(define (extract-constant l a) ; extract from list 'l' all elements whose
  (cond ((null? l)             ; corresponding element in 'a' is a constant
         '())
        ((not-constant? (car a))
         (extract-constant (cdr l) (cdr a)))
        (else
         (cons (car l) (extract-constant (cdr l) (cdr a))))))

(define (beta-subst exp env) ; return a modified 'exp' where each var named in
  (define (bs exp)           ; 'env' is replaced by the corresponding expr (it
    (cond ((const-expr? exp) ; is assumed that the code has been alphatized)
           (quot (const-value exp)))
          ((symbol? exp)
           (let ((x (assq exp env)))
             (if x (cdr x) exp)))
          ((or (eq? (car exp) 'if) (eq? (car exp) 'begin))
           (cons (car exp) (map bs (cdr exp))))
          ((or (eq? (car exp) 'let) (eq? (car exp) 'letrec))
           (list (car exp)
                 (map (lambda (x) (list (car x) (bs (cadr x)))) (cadr exp))
                 (bs (caddr exp))))
          ((eq? (car exp) 'lambda)
           (list 'lambda
                 (cadr exp)
                 (bs (caddr exp))))
          (else
           (map bs exp))))
  (bs exp))

;------------------------------------------------------------------------------
;
; The expression simplifier.

(define (simplify! exp)     ; simplify the expression 'exp' destructively (it
                            ; is assumed that the code has been alphatized)
  (define (simp! where env)

    (define (s! where)
      (let ((exp (car where)))

        (cond ((const-expr? exp))  ; leave constants the way they are

```

```
                ((symbol? exp))      ; leave variable references the way they are

                ((eq? (car exp) 'if) ; dead code removal for conditionals
                 (s! (cdr exp))      ; simplify the predicate
                 (if (const-expr? (cadr exp)) ; is the predicate a constant?
                   (begin
                     (set-car! where
                       (if (memq (const-value (cadr exp)) '(#f ())) ; false?
                         (if (= (length exp) 3) ''() (cadddr exp))
                         (caddr exp)))
                     (s! where))
                   (for-each! s! (cddr exp)))) ; simplify consequent and alt.

                ((eq? (car exp) 'begin)
                 (for-each! s! (cdr exp))
                 (let loop ((exps exp)) ; remove all useless expressions
                   (if (not (null? (cddr exps))) ; not last expression?
                     (let ((x (cadr exps)))
                       (loop (if (or (const-expr? x)
                                     (symbol? x)
                                     (and (pair? x) (eq? (car x) 'lambda)))
                                 (begin (set-cdr! exps (cddr exps)) exps)
                                 (cdr exps))))))
                 (if (null? (cddr exp)) ; only one expression in the begin?
                   (set-car! where (cadr exp))))

                ((or (eq? (car exp) 'let) (eq? (car exp) 'letrec))
                 (let ((new-env (cons exp env)))
                   (define (keep i)
                     (if (>= i (length (cadar where)))
                       '()
                       (let* ((var (car (list-ref (cadar where) i)))
                              (val (cadr (assq var (cadar where))))
                              (refs (ref-count (car where) var))
                              (self-refs (ref-count val var))
                              (total-refs (- (car refs) (car self-refs)))
                              (oper-refs (- (cadr refs) (cadr self-refs))))
                         (cond ((= total-refs 0)
                                (keep (+ i 1)))
                               ((or (const-expr? val)
                                    (symbol? val)
                                    (and (pair? val)
                                         (eq? (car val) 'lambda)
                                         (= total-refs 1)
                                         (= oper-refs 1)
                                         (= (car self-refs) 0))
                                    (and (caddr refs)
                                         (= total-refs 1)))
                                (set-car! where
                                  (beta-subst (car where)
                                              (list (cons var val))))
                                (keep (+ i 1)))
                               (else
                                (cons var (keep (+ i 1))))))))
                   (simp! (cddr exp) new-env)
                   (for-each! (lambda (x) (simp! (cdar x) new-env)) (cadr exp))
                   (let ((to-keep (keep 0)))
                     (if (< (length to-keep) (length (cadar where)))
                       (begin
                         (if (null? to-keep)
                           (set-car! where (caddar where))
                           (set-car! (cdar where)
                             (map (lambda (v) (assq v (cadar where))) to-keep)))
                         (s! where))
                       (if (null? to-keep)
                         (set-car! where (caddar where)))))))

                ((eq? (car exp) 'lambda)
                 (simp! (cddr exp) (cons exp env)))

                (else
                 (for-each! s! exp)
                 (cond ((symbol? (car exp)) ; is the operator position a var ref?
                        (let ((frame (binding-frame (car exp) env)))
                          (if frame ; is it a bound variable?
                            (let ((proc (bound-expr (car exp) frame)))
                              (if (and (pair? proc)
                                       (eq? (car proc) 'lambda)
                                       (some? const-expr? (cdr exp)))
                                (let* ((args (arg-pattern (cdr exp)))
                                       (new-proc (peval proc args))
                                       (new-args (remove-constant (cdr exp)
                                                                  args)))
                                  (set-car! where
                                    (cons (add-binding new-proc frame (car exp))
                                          new-args)))))
                            (set-car! where
                              (constant-fold-global (car exp) (cdr exp))))))
                       ((not (pair? (car exp))))
                       ((eq? (caar exp) 'lambda)
                        (set-car! where
                          (list 'let
                                (map2 list (cadar exp) (cdr exp))
                                (caddar exp)))
                        (s! where)))))))

    (s! where))

  (define (remove-empty-calls! where env)

    (define (rec! where)
      (let ((exp (car where)))

        (cond ((const-expr? exp))
              ((symbol? exp))
              ((eq? (car exp) 'if)
               (rec! (cdr exp))
               (rec! (cddr exp))
               (rec! (cdddr exp)))
              ((eq? (car exp) 'begin)
               (for-each! rec! (cdr exp)))
              ((or (eq? (car exp) 'let) (eq? (car exp) 'letrec))
               (let ((new-env (cons exp env)))
                 (remove-empty-calls! (cddr exp) new-env)
                 (for-each! (lambda (x) (remove-empty-calls! (cdar x) new-env))
                            (cadr exp))))
              ((eq? (car exp) 'lambda)
               (rec! (cddr exp)))
              (else
               (for-each! rec! (cdr exp))
               (if (and (null? (cdr exp)) (symbol? (car exp)))
                 (let ((frame (binding-frame (car exp) env)))
                   (if frame ; is it a bound variable?
                     (let ((proc (bound-expr (car exp) frame)))
                       (if (and (pair? proc)
                                (eq? (car proc) 'lambda))
                         (begin
                           (set! changed? #t)
                           (set-car! where (caddr proc))))))))))))

    (rec! where))

  (define changed? #f)

  (let ((x (list exp)))
    (let loop ()
      (set! changed? #f)
      (simp! x '())
      (remove-empty-calls! x '())
      (if changed? (loop) (car x)))))

(define (ref-count exp var) ; compute how many references to variable 'var'
  (let ((total 0)           ; are contained in 'exp'
        (oper 0)
        (always-evaled #t))
    (define (rc exp ae)
      (cond ((const-expr? exp))
            ((symbol? exp)
             (if (eq? exp var)
               (begin
                 (set! total (+ total 1))
                 (set! always-evaled (and ae always-evaled)))))
            ((eq? (car exp) 'if)
             (rc (cadr exp) ae)
             (for-each (lambda (x) (rc x #f)) (cddr exp)))
            ((eq? (car exp) 'begin)
             (for-each (lambda (x) (rc x ae)) (cdr exp)))
            ((or (eq? (car exp) 'let) (eq? (car exp) 'letrec))
             (for-each (lambda (x) (rc (cadr x) ae)) (cadr exp))
             (rc (caddr exp) ae))
            ((eq? (car exp) 'lambda)
             (rc (caddr exp) #f))
            (else
             (for-each (lambda (x) (rc x ae)) exp)
             (if (symbol? (car exp))
               (if (eq? (car exp) var) (set! oper (+ oper 1)))))))
    (rc exp #t)
    (list total oper always-evaled)))

(define (binding-frame var env)
  (cond ((null? env) #f)
        ((or (eq? (caar env) 'let) (eq? (caar env) 'letrec))
         (if (assq var (cadar env)) (car env) (binding-frame var (cdr env))))
        ((eq? (caar env) 'lambda)
         (if (memq var (cadar env)) (car env) (binding-frame var (cdr env))))
        (else
         (error '() '() "ill-formed environment"))))

(define (bound-expr var frame)
  (cond ((or (eq? (car frame) 'let) (eq? (car frame) 'letrec))
         (cadr (assq var (cadr frame))))
        ((eq? (car frame) 'lambda)
         not-constant)
        (else
         (error '() '() "ill-formed frame"))))
```

```
(define (add-binding val frame name)
  (define (find-val val bindings)
    (cond ((null? bindings) #f)
          ((equal? val (cadar bindings)) ; *kludge* equal? is not exactly what
           (caar bindings))              ; we want...
          (else
           (find-val val (cdr bindings)))))
  (or (find-val val (cadr frame))
      (let ((var (new-variable name)))
        (set-cdr! (last-pair (cadr frame)) (list (list var val)))
        var)))

(define (for-each! proc! l) ; call proc! on each CONS CELL in the list 'l'
  (if (not (null? l))
    (begin (proc! l) (for-each! proc! (cdr l)))))

(define (arg-pattern exps) ; return the argument pattern (i.e. the list of
  (if (null? exps)         ; constants in 'exps' but with the not-constant
    '()                    ; value wherever the corresponding expression in
    (cons (if (const-expr? (car exps)) ; 'exps' is not a constant)
              (const-value (car exps))
              not-constant)
          (arg-pattern (cdr exps)))))

;-------------------------------------------------------------------------
;
; Knowledge about primitive procedures.

(define *primitives*
  (list
    (cons 'car (lambda (args)
                 (and (= (length args) 1)
                      (pair? (car args))
                      (quot (car (car args))))))
    (cons 'cdr (lambda (args)
                 (and (= (length args) 1)
                      (pair? (car args))
                      (quot (cdr (car args))))))
    (cons '+ (lambda (args)
               (and (every? number? args) (quot (apply + args)))))
    (cons '* (lambda (args)
               (and (every? number? args) (quot (apply * args)))))
    (cons '- (lambda (args)
               (and (> (length args) 0)
                    (every? number? args)
                    (quot (apply - args)))))
    (cons '/ (lambda (args)
               (and (> (length args) 1)
                    (every? number? args)
                    (quot (apply / args)))))
    (cons '< (lambda (args)
               (and (= (length args) 2)
                    (every? number? args)
                    (quot (< (car args) (cadr args))))))
    (cons '= (lambda (args)
               (and (= (length args) 2)
                    (every? number? args)
                    (quot (= (car args) (cadr args))))))
    (cons '> (lambda (args)
               (and (= (length args) 2)
                    (every? number? args)
                    (quot (> (car args) (cadr args))))))
    (cons 'eq? (lambda (args)
                 (and (= (length args) 2)
                      (quot (eq? (car args) (cadr args))))))
    (cons 'not (lambda (args)
                 (and (= (length args) 1)
                      (quot (not (car args))))))
    (cons 'null? (lambda (args)
                   (and (= (length args) 1)
                        (quot (null? (car args))))))
    (cons 'pair? (lambda (args)
                   (and (= (length args) 1)
                        (quot (pair? (car args))))))
    (cons 'symbol? (lambda (args)
                     (and (= (length args) 1)
                          (quot (symbol? (car args))))))
    (cons 'length (lambda (args)
                    (and (= (length args) 1)
                         (proper-list? (car args))
                         (quot (length (car args))))))
  )
)

(define (reduce-global name args)
  (let ((x (assq name *primitives*)))
    (and x ((cdr x) args))))

(define (constant-fold-global name exprs)

  (define (flatten args op)
    (cond ((null? args)
           '())
          ((and (pair? (car args)) (eq? (caar args) op))
           (append (flatten (cdar args) op) (flatten (cdr args) op)))
          (else
           (cons (car args) (flatten (cdr args) op)))))

  (let ((args (if (or (eq? name '+) (eq? name '*)) ; associative ops
                  (flatten exprs name)
                  exprs)))
    (or (and (every? const-expr? args)
             (reduce-global name (map const-value args)))
        (let ((pattern (arg-pattern args)))
          (let ((non-const (remove-constant args pattern))
                (const (map const-value (extract-constant args pattern))))
            (cond ((eq? name '+) ; + is commutative
                   (let ((x (reduce-global '+ const)))
                     (if x
                       (let ((y (const-value x)))
                         (cons '+
                               (if (= y 0) non-const (cons x non-const))))
                       (cons name args))))
                  ((eq? name '*) ; * is commutative
                   (let ((x (reduce-global '* const)))
                     (if x
                       (let ((y (const-value x)))
                         (cons '*
                               (if (= y 1) non-const (cons x non-const))))
                       (cons name args))))
                  ((eq? name 'cons)
                   (cond ((and (const-expr? (cadr args))
                               (null? (const-value (cadr args))))
                          (list 'list (car args)))
                         ((and (pair? (cadr args))
                               (eq? (car (cadr args)) 'list))
                          (cons 'list (cons (car args) (cdr (cadr args)))))
                         (else
                          (cons name args))))
                  (else
                   (cons name args))))))))

;-------------------------------------------------------------------------
;
; Examples:

(define (my-try proc args)
  (partial-evaluate proc args))

; . . . . . .

(define example1
  '(lambda (a b c)
     (if (null? a) b (+ (car a) c))))

;(my-try example1 (list '(10 11) not-constant '1))

; . . . . . .

(define example2
  '(lambda (x y)
     (let ((q (lambda (a b) (if (< a 0) b (- 10 b)))))
       (if (< x 0) (q (- y) (- x)) (q y x)))))

;(my-try example2 (list not-constant '1))

; . . . . . .

(define example3
  '(lambda (l n)
     (letrec ((add-list
               (lambda (l n)
                 (if (null? l)
                   '()
                   (cons (+ (car l) n) (add-list (cdr l) n))))))
       (add-list l n))))

;(my-try example3 (list not-constant '1))

;(my-try example3 (list '(1 2 3) not-constant))

; . . . . . .

(define example4
  '(lambda (exp env)
     (letrec ((eval
               (lambda (exp env)
                 (letrec ((eval-list
                           (lambda (l env)
                             (if (null? l)
                               '()
                               (cons (eval (car l) env)
                                     (eval-list (cdr l) env))))))
                   (if (symbol? exp) (lookup exp env)
                     (if (not (pair? exp)) exp
                       (if (eq? (car exp) 'quote) (car (cdr exp))
```

```
                          (apply (eval (car exp) env)
                                 (eval-list (cdr exp) env)))))))))
        (eval exp env))))

;(my-try example4 (list 'x not-constant))

;(my-try example4 (list '(f 1 2 3) not-constant))

; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(define example5
  '(lambda (a b)
     (letrec ((funct
               (lambda (x)
                 (+ x b (if (< x 1) 0 (funct (- x 1)))))))
       (funct a))))

;(my-try example5 (list '5 not-constant))

; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(define example6
  '(lambda ()
     (letrec ((fib
               (lambda (x)
                 (if (< x 2) x (+ (fib (- x 1)) (fib (- x 2)))))))
       (fib 10))))

;(my-try example6 '())

; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(define example7
  '(lambda (input)
     (letrec ((copy (lambda (in)
                      (if (pair? in)
                        (cons (copy (car in))
                              (copy (cdr in)))
                        in))))
       (copy input))))

;(my-try example7
;       (list '(a b c d e f g h i j k l m n o p q r s t u v w x y z)))

; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(define example8
  '(lambda (input)
     (letrec ((reverse (lambda (in result)
                         (if (pair? in)
                           (reverse (cdr in) (cons (car in) result))
                           result))))
       (reverse input '()))))

;(my-try example8
;       (list '(a b c d e f g h i j k l m n o p q r s t u v w x y z)))

; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(define (test)
  (list (my-try example1 (list '(10 11) not-constant '1))
        (my-try example2 (list not-constant '1))
        (my-try example3 (list not-constant '1))
        (my-try example3 (list '(1 2 3) not-constant))
        (my-try example4 (list 'x not-constant))
        (my-try example4 (list '(f 1 2 3) not-constant))
        (my-try example5 (list '5 not-constant))
        (my-try example6 '())
        (my-try example7
            (list '(a b c d e f g h i j k l m n o p q r s t u v w x y z)))
        (my-try example8
            (list '(a b c d e f g h i j k l m n o p q r s t u v w x y z)))))

(define (main argv)
   (let loop ((i (string->number (cadr argv))))
      (if (= i 1)
          (begin
             (display "PEVAL")
             (display (test))
             (newline))
          (begin
             (test)
             (loop (- i 1))))))
```

pp/bigloo/pp.scm

```
;*=====================================================================*/
;*    serrano/these/src/pp/bigloo/pp.scm ...                           */
;*    -------------------------------------------------------------    */
;*    Author      :  Manuel Serrano                                    */
;*    Creation    :  Thu Oct 20 16:47:49 1994                          */
;*    Last change :  Thu Oct 20 16:48:33 1994 (serrano)                */
;*    -------------------------------------------------------------    */
;*    Un petit pretty-printer                                          */
;*=====================================================================*/

;*---------------------------------------------------------------------*/
;*    Le module                                                        */
;*---------------------------------------------------------------------*/
(module pp
   (main main))

;*---------------------------------------------------------------------*/
;*    *width* ...                                                      */
;*---------------------------------------------------------------------*/
(define *width* 80)

;*---------------------------------------------------------------------*/
;*    *output*                                                         */
;*---------------------------------------------------------------------*/
(define *output* (current-output-port))
(define *output-name* '())

;*---------------------------------------------------------------------*/
;*    the *case* ...                                                   */
;*---------------------------------------------------------------------*/
(define *case* 'respect)

;*---------------------------------------------------------------------*/
;*    Les commentaires                                                 */
;*---------------------------------------------------------------------*/
(define *ignore-comment* #f)

;*---------------------------------------------------------------------*/
;*    main ...                                                         */
;*---------------------------------------------------------------------*/
(define (main argv)
   (let ((files (parse-args! (cdr argv))))
      ;; on ouvre le fichier de sortie
      (if (string? *output-name*)
          (set! *output* (open-output-file *output-name*)))
      ;; on test si c'est bien un port de sortie
      (if (not (output-port? *output*))
          (error "pp" "Can't open output file" *output-name*))
      ;; on pretty-print
      (if (null? files)
          (pp-console)
          (pp-files files))
      ;; on ferme le port de sortie (si on l'a ouvert)
      (if (string? *output-name*)
          (close-output-port *output*))))

;*---------------------------------------------------------------------*/
;*    parse-args! ...                                                  */
;*---------------------------------------------------------------------*/
(define (parse-args! args)
   (let loop ((args args))
      (cond
         ((null? args)
          '())
         ((string=? (car args) "-help")
          (print "usage: pp [-comment] [-o output-file]
                            [-upper] [-lower] [-wWIDTH] [file1] ... [fileN]")
          (exit -1))
         ((and (> (string-length (car args)) 2)
               (string=? (substring (car args) 0 2) "-w"))
          (set! *width* (string->integer
                         (substring (car args)
                                    2
                                    (string-length (car args)))))
          (loop (cdr args)))
         ((string=? (car args) "-o")
          (if (null? (cdr args))
              (error "pp" "-o require one argument" args)
              (begin
                 (set! *output-name* (cadr args))
                 (loop (cddr args)))))
         ((string=? (car args) "-upper")
          (set! *case* 'upper)
          (loop (cdr args)))
         ((string=? (car args) "-lower")
          (set! *case* 'lower)
          (loop (cdr args)))
         ((string=? (car args) "-comment")
          (set! *ignore-comment* #t)
          (loop (cdr args)))
         ((null? (cdr args))
          (if (not (file-exists? (car args)))
              (begin
                 (set! *output-name* (car args))
                 '())
              (list (car args))))
         (else
          (cons (car args) (loop (cdr args)))))))
```

```
99
100
101  ;*---------------------------------------------------------------------*/
102  ;*    pp-console ...                                                   */
103  ;*---------------------------------------------------------------------*/
104  (define (pp-console)
105     (let loop ((exp (read (current-input-port))))
106        (if (eof-object? exp)
107            #T
108            (begin
109               (generic-write exp
110                              #f
111                              *width*
112                              (lambda (s) (display s *output*) #t))
113               (flush-output-port *output*)
114               (loop (read (current-input-port)))))))
115
116  ;*---------------------------------------------------------------------*/
117  ;*    pp-files ...                                                     */
118  ;*---------------------------------------------------------------------*/
119  (define (pp-files list)
120     (let ((banner (not (null? (cdr list)))))
121        (let loop ((list list))
122           (if (null? list)
123               #T
124               (begin
125                  (if (file-exists? (car list))
126                      (begin
127                         (if banner
128                             (begin
129                                (print "::::::::::::::")
130                                (print (car list))
131                                (print "::::::::::::::")))
132                         (let ((port (open-input-file (car list))))
133                            (let loop ((exp (read port)))
134                               (if (eof-object? exp)
135                                   (close-input-port port)
136                                   (begin
137                                      (generic-write exp
138                                                     #f
139                                                     *width*
140                                                     (lambda (s)
141                                                        (display s *output*)
142                                                        #t))
143                                      (loop (read port)))))))
144                  (loop (cdr list)))))))
145
146  ; 'generic-write' is a procedure that transforms a Scheme data value (or
147  ; Scheme program expression) into its textual representation.  The interface
148  ; to the procedure is sufficiently general to easily implement other useful
149  ; formatting procedures such as pretty printing, output to a string and
150  ; truncated output.
151  ;
152  ; Parameters:
153  ;
154  ;   OBJ       Scheme data value to transform.
155  ;   DISPLAY?  Boolean, controls whether characters and strings are quoted.
156  ;   WIDTH     Extended boolean, selects format:
157  ;               #f = single line format
158  ;               integer > 0 = pretty-print (value = max nb of chars per line)
159  ;   OUTPUT    Procedure of 1 argument of string type, called repeatedly
160  ;               with successive substrings of the textual representation.
161  ;               This procedure can return #f to stop the transformation.
162  ;
163  ; The value returned by 'generic-write' is undefined.
164  ;
165  ; Examples:
166  ;
167  ;   (write obj)   = (generic-write obj #f #f display-string)
168  ;   (display obj) = (generic-write obj #t #f display-string)
169  ;
170  ; where display-string = (lambda (s) (for-each write-char (string->list s)) #t)
171
172  (define (generic-write obj display? width output)
173
174    (define (read-macro? l)
175      (define (length1? l) (and (pair? l) (null? (cdr l))))
176      (let ((head (car l)) (tail (cdr l)))
177        (case head
178          ((QUOTE QUASIQUOTE UNQUOTE UNQUOTE-SPLICING) (length1? tail))
179          (else                                        #f))))
180
181    (define (read-macro-body l)
182      (cadr l))
183
184    (define (read-macro-prefix l)
185      (let ((head (car l)) (tail (cdr l)))
186        (case head
187          ((QUOTE)            "'")
188          ((QUASIQUOTE)       "`")
189          ((UNQUOTE)          ",")
190          ((UNQUOTE-SPLICING) ",@"))))
191
192    (define (out str col)
193      (and col (output str) (+ col (string-length str))))
194
195    (define (wr obj col)
196
197      (define (wr-expr expr col)
198        (if (read-macro? expr)
199          (wr (read-macro-body expr) (out (read-macro-prefix expr) col))
200          (wr-lst expr col)))
201
202      (define (wr-lst l col)
203        (if (pair? l)
204          (let loop ((l (cdr l)) (col (wr (car l) (out "(" col))))
205            (and col
206                 (cond ((pair? l) (loop (cdr l) (wr (car l) (out " " col))))
207                       ((null? l) (out ")" col))
208                       (else      (out ")" (wr l (out " . " col)))))))
209          (out "()" col)))
210
211      (cond ((match-case obj
212                ((COMMENT (? integer?) (? string?))
213                 #t)
214                (else
215                 #f))
216             (let ((add (- *width* (string-length (caddr obj)) 3)))
217                (if (<= add 0)
218                    (out (caddr obj) col)
219                    (out (string-append (caddr obj) (make-string add #\space))
220                         col))))
221            ((pair? obj)        (wr-expr obj col))
222            ((null? obj)        (wr-lst obj col))
223            ((vector? obj)      (wr-lst (vector->list obj) (out "#" col)))
224            ((boolean? obj)     (out (if obj "#t" "#f") col))
225            ((number? obj)      (out (number->string obj) col))
226            ((symbol? obj)      (case *case*
227                                  ((respect)
228                                   (out (symbol->string obj) col))
229                                  ((upper)
230                                   (out (string-upcase (symbol->string obj))
231                                        col))
232                                  (else
233                                   (out (string-downcase (symbol->string obj))
234                                        col))))
235            ((procedure? obj)   (out "#<procedure>" col))
236            ((string? obj)      (if display?
237                                    (out obj col)
238                                    (let loop ((i 0) (j 0) (col (out "\"" col)))
239                                      (if (and col (< j (string-length obj)))
240                                        (let ((c (string-ref obj j)))
241                                          (loop i (+ j 1) col))
242                                        (out "\""
243                                             (out (substring obj i j) col))))))
244            ((char? obj)        (if display?
245                                    (out (make-string 1 obj) col)
246                                    (out (case obj
247                                           ((#\space)   "space")
248                                           ((#\newline) "newline")
249                                           (else        (make-string 1 obj)))
250                                         (let ((s (make-string 2)))
251                                           (string-set! s 0 #\#)
252                                           (string-set! s 1 #\\)
253                                           (out s col)))))
254            ((input-port? obj)  (out "#[input-port]" col))
255            ((output-port? obj) (out "#[output-port]" col))
256            ((eof-object? obj)  (out "#[eof-object]" col))
257            (else               (out "#[unknown]" col))))
258
259    (define (pp obj col)
260
261      (define (spaces n col)
262        (if (> n 0)
263          (if (> n 7)
264            (spaces (- n 8) (out "        " col))
265            (out (substring "        " 0 n) col))
266          col))
267
268      (define (indent to col)
269        (and col
270             (if (< to col)
271               (and (out (make-string 1 #\newline) col) (spaces to 0))
272               (spaces (- to col) col))))
273
274      (define (pr obj col extra pp-pair)
275        (if (or (pair? obj) (vector? obj)) ; may have to split on multiple lines
276          (let ((result '())
277                (left (min (+ (- (- width col) extra) 1) max-expr-width)))
278            (generic-write obj display? #f
279              (lambda (str)
280                (set! result (cons str result))
281                (set! left (- left (string-length str)))
282                (> left 0)))
283            (if (> left 0) ; all can be printed on one line
284              (out (reverse-string-append result) col)
```

```
            (if (pair? obj)
              (pp-pair obj col extra)
              (pp-list (vector->list obj) (out "#" col) extra pp-expr))))
        (wr obj col)))

    (define (pp-expr expr col extra)
      (if (read-macro? expr)
        (pr (read-macro-body expr)
            (out (read-macro-prefix expr) col)
            extra
            pp-expr)
        (let ((head (car expr)))
          (if (symbol? head)
            (let ((proc (style head)))
              (if proc
                (proc expr col extra)
                (if (> (string-length (symbol->string head))
                       max-call-head-width)
                  (pp-general expr col extra #f #f #f pp-expr)
                  (pp-call expr col extra pp-expr))))
            (pp-list expr col extra pp-expr)))))

    ; (head item1
    ;       item2
    ;       item3)
    (define (pp-call expr col extra pp-item)
      (let ((col* (wr (car expr) (out "(" col))))
        (and col
             (pp-down (cdr expr) col* (+ col* 1) extra pp-item))))

    ; (item1
    ;  item2
    ;  item3)
    (define (pp-list l col extra pp-item)
      (let ((col (out "(" col)))
        (pp-down l col col extra pp-item)))

    ; (item1 item2 item3
    ;    item4)
    (define (pp-defun-form expr col extra pp-item)
       (let* ((col  (out "({}" col))
              (col2 (wr (car expr) col))
              (col3 (wr (cadr expr) col2)))
          (pp-down (cddr expr) col3 (+ col3 1) extra pp-item)))

    (define (pp-down l col1 col2 extra pp-item)
      (let loop ((l l) (col col1))
        (and col
             (cond ((pair? l)
                    (let ((rest (cdr l)))
                      (let ((extra (if (null? rest) (+ extra 1) 0)))
                        (loop rest
                              (pr (car l) (indent col2 col) extra pp-item)))))
                   ((null? l)
                    (out ")" col))
                   (else
                    (out ")"
                         (pr l
                             (indent col2 (out "." (indent col2 col)))
                             (+ extra 1)
                             pp-item)))))))

    (define (pp-general expr col extra named? pp-1 pp-2 pp-3)

      (define (tail1 rest col1 col2 col3)
        (if (and pp-1 (pair? rest))
          (let* ((val1 (car rest))
                 (rest (cdr rest))
                 (extra (if (null? rest) (+ extra 1) 0)))
            (tail2 rest col1 (pr val1 (indent col3 col2) extra pp-1) col3))
          (tail2 rest col1 col2 col3)))

      (define (tail2 rest col1 col2 col3)
        (if (and pp-2 (pair? rest))
          (let* ((val1 (car rest))
                 (rest (cdr rest))
                 (extra (if (null? rest) (+ extra 1) 0)))
            (tail3 rest col1 (pr val1 (indent col3 col2) extra pp-2)))
          (tail3 rest col1 col2)))

      (define (tail3 rest col1 col2)
        (pp-down rest col2 col1 extra pp-3))

      (let* ((head (car expr))
             (rest (cdr expr))
             (col* (wr head (out "(" col))))
        (if (and named? (pair? rest))
          (let* ((name (car rest))
                 (rest (cdr rest))
                 (col** (wr name (out " " col*))))
            (tail1 rest (+ col indent-general) col** (+ col** 1)))
          (tail1 rest (+ col indent-general) col* (+ col* 1)))))

    (define (pp-expr-list l col extra)
      (pp-list l col extra pp-expr))

    (define (pp-expr-defun l col extra)
      (pp-defun-form l col extra pp-expr))

    (define (pp-DEFiNE expr col extra)
      (pp-general expr col extra #f pp-expr-list #f pp-expr)
      (out "\n" 0))

    (define (pp-DEFUN expr col extra)
      (pp-general expr col extra #t pp-expr-defun #f pp-expr)
      (out "\n" 0))

    (define (pp-LAMBDA expr col extra)
      (pp-general expr col extra #f pp-expr-list #f pp-expr))

    (define (pp-COMMENT expr col extra)
       (match-case expr
          ((COMMENT (and (? integer?) ?column) (and (? string?) ?string))
           (let ((add (- *width* (string-length string) 3)))
              (if (=fx column 0)
                  (if (> add 0)
                      (out (string-append string (make-string add #\space))
                           0)
                      (out string 0))
                  (if (> add 0)
                      (out (string-append string (make-string add #\space))
                           col)
                      (out string col)))))
          (else
           (pp-general expr col extra #f pp-expr #f pp-expr))))

    (define (pp-IF expr col extra)
      (pp-general expr col extra #f pp-expr #f pp-expr))

    (define (pp-COND expr col extra)
      (pp-call expr col extra pp-expr-list))

    (define (pp-CASE expr col extra)
      (pp-general expr col extra #f pp-expr #f pp-expr-list))

    (define (pp-AND expr col extra)
      (pp-call expr col extra pp-expr))

    (define (pp-LET expr col extra)
      (let* ((rest (cdr expr))
             (named? (and (pair? rest) (symbol? (car rest)))))
        (pp-general expr col extra named? pp-expr-list #f pp-expr)))

    (define (pp-BEGIN expr col extra)
      (pp-general expr col extra #f #f #f pp-expr))

    (define (pp-DO expr col extra)
      (pp-general expr col extra #f pp-expr-list pp-expr-list pp-expr))

    ; define formatting style (change these to suit your style)

    (define indent-general 2)

    (define max-call-head-width 5)

    (define max-expr-width 50)

    (define (style head)
      (case (if (eq? *case* 'respect)
                (string->symbol (string-upcase (symbol->string head)))
                head)
        ((LAMBDA LET* LETREC) pp-LAMBDA)
        ((DEFINE)             pp-DEFINE)
        ((DEFUN DE)           pp-DEFUN)
        ((IF SET!)            pp-IF)
        ((COND)               pp-COND)
        ((CASE)               pp-CASE)
        ((AND OR)             pp-AND)
        ((LET)                pp-LET)
        ((BEGIN)              pp-BEGIN)
        ((DO)                 pp-DO)
        ((COMMENT)            pp-COMMENT)
        (else                 #f)))

    (pr obj col 0 pp-expr))

  (if width
    (out (make-string 1 #\newline) (pp obj 0))
    (wr obj 0)))

; (reverse-string-append l) = (apply string-append (reverse l))

(define (reverse-string-append l)

  (define (rev-string-append l i)
    (if (pair? l)
```

```
        (let* ((str (car l))
               (len (string-length str))
               (result (rev-string-append (cdr l) (+ i len))))
          (let loop ((j 0) (k (- (- (string-length result) i) len)))
            (if (< j len)
              (begin
                (string-set! result k (string-ref str j))
                (loop (+ j 1) (+ k 1)))
              result)))
        (make-string i)))

  (rev-string-append l 0))

; (object->string obj) returns the textual representation of 'obj' as a
; string.
;
; Note: (write obj) = (display (object->string obj))

(define (object->string obj)
  (let ((result '()))
    (generic-write obj #f #f (lambda (str) (set! result (cons str result)) #t))
    (reverse-string-append result)))

; (object->limited-string obj limit) returns a string containing the first
; 'limit' characters of the textual representation of 'obj'.

(define (object->limited-string obj limit)
  (let ((result '()) (left limit))
    (generic-write obj #f #f
      (lambda (str)
        (let ((len (string-length str)))
          (if (> len left)
            (begin
              (set! result (cons (substring str 0 left) result))
              (set! left 0)
              #f)
            (begin
              (set! result (cons str result))
              (set! left (- left len))
              #t)))))
    (reverse-string-append result)))

; (pretty-print obj port) pretty prints 'obj' on 'port'.  The current
; output port is used if 'port' is not specified.

(define (pretty-print obj . opt)
  (let ((port (if (pair? opt) (car opt) (current-output-port))))
    (generic-write obj #f 79 (lambda (s) (display s port) #t))))

; (pretty-print-to-string obj) returns a string with the pretty-printed
; textual representation of 'obj'.

(define (pretty-print-to-string obj)
  (let ((result '()))
    (generic-write obj #f 79 (lambda (str) (set! result (cons str result)) #t))
    (reverse-string-append result)))
```

Résumé

Cette thèse est consacrée à la compilation portable et performante des langages fonctionnels modernes tels que Scheme et ML. Pour obtenir la portabilité, nous avons opté pour un schéma de compilation qui utilise C comme langage d'assemblage. Le choix de ce langage cible interdit l'utilisation de certaines des techniques traditionnellement efficaces. Nous avons comblé cet handicap en multipliant les analyses et les optimisations de haut niveau.

Les travaux présentés dans ce document ont donné lieu à la réalisation d'un compilateur très performant : Bigloo. Distribué depuis plus d'un an sur ftp anonyme, il est maintenant largement utilisé. Toutes les optimisations présentées dans ce document y sont implantées. Nous avons donc pu mesurer l'impact de chacune d'entre elles. Nous présentons en conclusion de cette thèse des mesures de performances qui nous permettent d'affirmer que Bigloo est, actuellement, le compilateur Scheme le plus rapide.

En plus de sa rapidité, notre compilateur est extensible à deux niveaux : *(i)* au niveau du langage source : il est possible d'insérer des passes supplémentaires dans le processus de compilation. Plusieurs extensions ont déjà été réalisées. Bigloo est capable, à ce jour, de compiler, en plus de Scheme, Caml-light ou MEROON (la couche objet de Scheme conçue par C. Queinnec). *(ii)* au niveau des langages externes : Bigloo possède une interface qui permet au programmeur d'avoir accès à un langage externe depuis Scheme (ou ML). Grâce à elle, plusieurs langages peuvent être mélangés pour produire des applications autonomes. L'utilisation conjointe de nos deux niveaux d'extensibilité nous a permis, à titre d'expérience, de réaliser des programmes exécutables mélangeant Scheme, Caml-light et C.

Mots Clés

Langages fonctionnels, langages impératifs, Scheme, ML, C, compilation, optimisation, analyse de programmes, portabilité.

Abstract

This PhD thesis is devoted to the portable and efficient compilation of modern functional languages such as Scheme and ML. Portability is achieved by adopting a compilation scheme which uses the C programing language as portable assembly language. This choice has a drawback, namely that it forbids to use efficient techniques, traditionally known. This handicap was filled by introducing multiple analysis and high-level optimizations.

The results presented in this document have led to the development of a very efficient compiler : Bigloo. Available by anonymous ftp, for a year, it is now widely used. All optimizations presented have been implemented. We have been able then to measure the impact of each of them. Performances measures are given at the end of the document which allow to conclude that Bigloo is, for the moment, the fastest Scheme compiler.

In addition to its speed, our compiler is extensible with respect to two distinct levels : *(i)* the source language level : additional passes can be inserted within the compilation process. Various extensions have been already developed. Bigloo can compile, in addition to Scheme Caml-light or MEROON (an object oriented layer of Scheme introduced by C. Queinnec) *(ii)* the foreign languages level : Bigloo has an interface which allows the programer to have access to a foreign language from Scheme source code (or ML). Thanks to this interface, various languages can be mixed to produce autonomous applications.

The simultaneous utilization of this two levels of extensibility has allowed us to develop executable programs mixing Scheme Caml-light and C.

Keywords

Functional languages, imperative languages, Scheme, ML, C, compilation, optimization, programme analyses, portability.