Habilitation à diriger des recherches

présentée à

L'université de Nice

Spécialité :

INFORMATIQUE

Soutenue par

Manuel Serrano

le 11 Septembre 2000

Titre

Vers une programmation fonctionnelle *praticable*

Jury

MM.	Jean-Paul Rigault	Président
	Gérard Berry Pierre Cointe Daniel Ribbens Matthias Felleisen	Rapporteurs
	Xavier Leroy Christian Queinnec Michel Rueher	Examinateurs

Table des matières

Chapitre 1

Introduction

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs. – Maurice Wilkes, 1949

LA programmation est une activité terriblement difficile. Elle est tellement complexe et laborieuse qu'on finit même par accepter la piètre qualité de la plupart des réalisations informatiques. L'industrie du logiciel est la seule (avec peut-être les compagnies aériennes qui sont incapables de respecter les horaires des avions) qui soit parvenue à établir le commerce de produits aussi instables et hasardeux que sont la plupart des logiciels actuels. Personne ne sait très bien ce que les dits logiciels font ; les éditeurs informatiques se dégageant, pour leur part, de toute responsabilité en cas de dysfonctionnement. La notion même de garantie est inopérante car personne n'aurait l'« audace » de spécifier ce qu'un logiciel est supposé faire. Les déficiences de l'informatique ont des impacts de plus en plus nombreux dans notre vie quotidienne à tel point que même les journaux d'informations générales les relatent. Ainsi, dans le quotidien *Le Monde*, daté du Dimanche 26/Lundi 27 Décembre 1999, on a pu lire [4] :

« *Le bogue de l'an 2000 n'est pas seulement le plus coûteux sinistre industriel de tous les temps, il constitue également la première alerte sérieuse sur les dangers que le recours massif à une informatique mal maîtrisée fait courir aux sociétés développées... [L'informatique] est ainsi devenue simultanément l'outil le plus puissant et le plus faillible que l'homme moderne ait jamais inventé et diffusé à si grande échelle.*

Chaque année, une centaine de millions de nouvelles machines sont vendues dans le monde. Chacune d'entre elles est équipée d'une bonne dizaine de logiciels de plus en plus complexes, dont le fameux système d'exploitation Windows de Microsoft. Et pas un de ces programmes n'est exempt de bogues, de l'aveu même des informaticiens. Ces derniers s'abritent derrière la complexité extrême de leurs créations. Ils invoquent les millions de lignes de code, les arborescences aux ramifications infinies et, par-dessus tout, la contrainte de la vitesse. En informatique, peut-être plus qu'ailleurs, le temps, c'est de l'argent...

Insensiblement, l'informatique s'immisce dans toutes les fonctions vitales de la société. Pourtant, elle n'offre pas les garanties minimales de fiabilité qui sont exigées, par ailleurs, de la totalité des autres appareils, ayant un impact possible sur la sécurité des personnes. Étrangement, le principe de précaution, s'il peut s'appliquer aux « vaches folles » ou aux plantes transgéniques, épargne l'ordinateur. »

Programmer n'est pas une activité purement intellectuelle. Comme le note E. Saint-James dans *La programmation applicative (de Lisp à la machine en passant par le λ-calcul)* [190] « *...l'ordinateur* *s'affirme comme un instrument d'objectivation du raisonnement. Il s'inscrit dans la lignée des appareils qui ont permis de passer de l'expérience empirique à l'expérimentation scientifique : l'informatique se distingue des mathématiques par un outil d'expérimentation permettant d'observer, vérifier, réfuter un raisonnement».* L'informatique impose un *passage à l'acte* et c'est là que les difficultés propres apparaissent. Même l'algorithme le plus juste peut donner un programme faux, simplement parce qu'il aura été mal transcrit, ou, encore plus vulgairement, parce que des fautes de frappe se seront glissées lors de sa saisie. Les risques sont augmentés par le nombre, sans cesse grandissant, de logiciels qu'il faut déployer pour la moindre réalisation. Il faut utiliser un éditeur de textes qui permettra la saisie du programme final, un compilateur qui traduira ce texte source en une série d'instructions compréhensibles par la machine, un éditeur de liens qui permettra au programme d'utiliser des fonctionnalités déjà implantées, un système d'exploitation qui prendra en charge le lancement et l'exécution du programme et lui attribuera des ressources comme une portion de la mémoire ou bien une zone de l'écran graphique. Tous ces « méta-logiciels » sont eux-mêmes complexes (généralement très complexes même) et tous sont susceptibles d'être *le* grain de sable conduisant à un comportement erratique. Quand un dysfonctionnement se produit, il peut être réellement dantesque d'en comprendre les causes et de les corriger. Les raisons peuvent être tellement variées qu'on est très démuni et, au bout du compte, les atouts les plus sûrs restent l'expérience individuelle et l'imagination. Nous rejoignons les conclusions de P. Breton dans *La tribu informatique* [30] qui note : « *Un regard plus attentif sur les procédures mentales utilisées par les programmeurs, par exemple pour retrouver une erreur dans un logiciel, met en évidence l'utilisation systématique de l'*intuition*, quantité difficilement identifiable, mais que l'on reconnaît pourtant au premier abord... Le plus intéressant dans cette affaire est sans doute le* déni *que formulent à cet égard beaucoup d'informaticiens : alors que l'observateur les voit – si l'on peut dire – utiliser une démarche intuitive, ces derniers nient en général formellement avoir recours à autre chose qu'une démarche logique. Il est rare en effet de rencontrer des informaticiens qui osent s'avouer qu'ils ne savent pas très bien comment ils ont fait telle opération ou découvert telle erreur».* Deux anecdotes, révélatrices du quotidien de l'informaticien, illustrent assez bien l'ingéniosité dont il faut parfois faire preuve pour corriger les erreurs les plus retorses :

- La première a été reportée dans un numéro spécial de la revue *Communications of the* ACM [1]. Elle relate l'histoire d'un ingénieur confronté à un logiciel qui fonctionnait correctement tous les jours de la semaine sauf le Mercredi. En effet, uniquement ce jour de la semaine, son programme avait un comportement étrange : un menu de configuration n'était plus accessible ! Après investigation, il s'est avéré que l'espace mémoire utilisé pour enregistrer l'orthographe des jours de la semaine était trop petit. L'espace mémoire pour sauvegarder cette information était de deux mots (soit 8 lettres au maximum sur la plupart des machines actuelles). En anglais tous les jours sont des mots de moins de 8 lettres (*Monday*, *Tuesday*, ...) sauf Mercredi (*Wednesday*) qui en contient 9 ! Comme l'espace alloué au jour de la semaine était donc trop petit pour contenir le mot *Wednesday* complet, le *y* supplémentaire « écrasait » une autre information utilisée par le programme. Par hasard, cette information était utilisée pour mémoriser un choix de l'utilisateur qui était soit un *n* pour *no* soit un *y* pour *yes*. Si l'utilisateur répondait *y*, une configuration par défaut était utilisée et son menu correspondant n'était alors plus accessible. Le Mercredi seulement, la valeur *y* « écrasait » le choix de l'utilisateur et rendait donc inaccessible cette configuration. Sous ses apparences bien mystérieuses qui semblait la lier à un jour précis, il s'agit ici d'une erreur de programmation très classique. On dispose de moyens pour s'en prémunir. En particulier, le choix d'un langage de programmation de plus haut niveau aurait évité cette erreur. Elle est toutefois révélatrice de l'imagination dont il faut faire preuve pour corriger des erreurs dans des logiciels. Il n'est pas spontanément évident de faire le lien entre une erreur de programmation et le fait que Mercredi soit le seul jour de la semaine qui, écrit en anglais, compte plus de 8 lettres.
- La deuxième mésaventure nous est arrivée personnellement et dans un sens elle est encore plus extravagante. La plus volumineuse de nos réalisations est sans conteste un traducteur pour un langage de programmation. Ce programme, nommé Bigloo, traduit automatiquement des textes exprimés dans le langage Scheme en textes équivalents (c'est-à-dire, ayant

rigoureusement le même sens) exprimés dans le langage C. Bigloo est un programme qui est lui même exprimé dans le langage Scheme. Bigloo peut donc se traduire lui-même ! Cette opération se nomme l'auto-génération (ou *bootstrap* en anglais, traduction que nous utiliserons dans la suite de ce document). Un *bootstrap* est toujours particulièrement difficile à réaliser parce que si le traducteur contient des erreurs mêmes infimes, celles ci ne se manifesteront au mieux au second tour de *bootstrap* si ce n'est même lors du troisième tour. Dans un tel cas, le traducteur obtenu est finalement incapable de se traduire lui-même. La manifestation de l'erreur est alors très difficile à relier à la partie de l'implantation initiale erronée. En résumé, un *bootstrap* peut-être à lui tout seul cause d'insomnies et il entraîne fréquemment chez ses pratiquants une sorte d'attitude très craintive voire même légèrement paranoïaque. En plus des déformations de la personnalité précédemment citées, le dérèglement du comportement se manifeste chez nous par une sorte de rite qui consiste à re-compiler Bigloo plusieurs fois consécutivement à la moindre modification. Même avec nos machines modernes cette opération reste très longue (environ 1h30 sur les machines de 1998). Comme c'est devenu une réelle manie, cette opération est totalement automatisée.

Petit à petit ont commencé à apparaître des erreurs lors du *bootstrap* de Bigloo. Rien de fondamentalement « grave » mais pour une raison inexpliquée, la procédure automatique s'interrompait en signalant des erreurs incompréhensibles. Ces dernières n'étaient pas très gênantes parce qu'il suffisait de relancer la procédure là où elle s'était arrêtée, éventuellement plusieurs fois, pour que tout rentre dans l'ordre. Les semaines passant, ces interruptions impromptues, qui au départ n'étaient que sporadiques, devenaient de plus en plus fréquentes. La raison que nous avons alors envisagée pour les expliquer tenait dans l'accroissement de la taille de Bigloo. Au fil du temps, en évoluant, Bigloo est devenu un programme de plus en plus gros. Nous avons alors soupçonné le gestionnaire mémoire de Bigloo. À partir d'une certaine taille mémoire utilisée, quelque chose se passait mal. Cette hypothèse a vite été infirmée par diverses expérimentations.

À peu près tout ce qui peut être inspecté dans le traducteur l'a été. Nous avons dû nous rendre à l'évidence que l'erreur n'était pas dans Bigloo lui-même mais dans les autres logiciels employés. Nous avons alors entrepris d'adopter systématiquement les versions plus récentes de tous ses autres programmes. Opération sans succès. Les interruptions intempestives ne cessaient d'augmenter. Nous avons alors commencé à suspecter le matériel utilisé. Nous avons alors vérifié tous les composants que nous pouvions tester, en particulier mémoire et disque, jusqu'à ce que nous trouvions enfin l'explication. Le ventilateur greffé sur le processeur était insuffisant. En conséquence, lors de longs calculs, le processeur montait en température jusqu'à un seuil maximal où une alarme système se déclenchait et stoppait tout calcul en cours. Lorsque nous constations que le *bootstrap* échouait, nous le relancions. Comme le *bootstrap* est long et qu'il utilise abondamment les ressources de la machine, il est généralement lancé quand la machine est inutilisée, par exemple la nuit. Du coup, quand nous revenions le matin, la température du processeur avait eu tout le temps de baisser. Enfin, à la vue de cette explication nous avons réalisé que si les erreurs étaient de plus en plus fréquentes ce n'était pas parce que Bigloo devenait de plus en plus gros, mais simplement parce que l'été avançait. Le local où était situé notre ordinateur n'était pas climatisé et il y faisait simplement de plus en plus chaud ! Le processeur atteignait donc de plus en plus vite et de plus en plus fréquemment sa température maximale.

Encore plus que la première, cette seconde mésaventure nous semble être révélatrice de la discipline *informatique*. Parce qu'il y a « mise en application » l'informatique n'est pas une science purement théorique. Toutes les précautions possibles ne garantiront jamais la correcte exécution de tous les programmes. Même un programme produit automatiquement à partir d'une preuve par le système Coq [67] pourrait faire chauffer excessivement un processeur non climatisé ! Les recherches présentées dans ce mémoire sont globalement orientées vers un objectif : faciliter la réalisation de logiciels informatiques en proposant un environnement de développement un peu moins hostile. Il n'existera jamais de systèmes permettant de ne construire que des programmes corrects mais il peut en exister permettant d'éliminer une grande part des erreurs couramment rencontrées. Nous pensons que de tels systèmes doivent reposer sur des environnements dont les deux composants essentiels sont :

Un langage de programmation expressif. C'est pour nous la pièce maîtresse qui façonne le reste de l'environnement. De même que le cheval ne fait pas le cavalier, le langage de programmation ne fait pas les programmes...mais il y contribue grandement. Choisir un langage de haut niveau proposant un fort niveau d'abstraction est un engagement déterminant. S'il implique de perdre le contrôle très fin des exécutions il permet une écriture plus concise et souvent plus correcte. Par exemple, avec un gestionnaire mémoire automatique il est presque impossible de régler finement l'espace mémoire utilisé par les programmes mais, en contrepartie, ces programmes sont plus simples à développer et à maintenir car ils ne peuvent plus contenir d'erreurs d'allocation ni de libération de mémoire qui sont parmi les plus difficiles à corriger. L'environnement présenté dans ce mémoire utilise un langage fonctionnel (le langage Scheme, [skēm]). Nous avons choisi la famille des langages fonctionnels pour leur fondement théorique, leur concision et parce que nous pensons que plus que tout autre, ils permettent d'écrire des programmes simples et compréhensibles. Les langages fonctionnels ont généralement une sémantique simple et bien définie qui permet de pousser plus loin les analyses de programmes qu'avec des langages plus traditionnels. Enfin, parce qu'ils réduisent à l'essentiel les concepts fondamentaux (fonctions, affectations, citations, etc) ils sont sans équivalent pour l'expérimentation linguistique. Même si notre but n'est pas de proposer un laboratoire d'expérimentation mais bien un système utilisable, la possibilité d'explorer plus facilement de nouvelles constructions dans notre langage a facilité nos recherches. En général, ces nouvelles constructions (nous en verrons trois par la suite), ne sont pas restées longtemps à l'état de prototype. Très rapidement elles ont été acceptées puis intégrées par les utilisateurs du système.

Une boîte à outils fiable et adaptée. On doit y trouver en vrac :

- Un compilateur qui prend en charge la mise en correspondance des constructions du langage avec les fonctionnalités des ordinateurs. Un programme doit mettre en application des algorithmes « performants » (ayant une faible complexité), il doit consommer aussi peu de ressources que possible, etc. En revanche, il ne doit pas dépendre des caractéristiques matérielles de telle ou telle plate-forme. C'est là que réside tout l'intérêt des langages de haut niveau. Ils permettent de s'abstraire de l'ordinateur physique. C'est aux compilateurs d'assurer des traductions produisant des programmes exécutables utilisant au mieux les capacités des ordinateurs d'accueil. C'est aux compilateurs, et à eux seuls, de connaître finement les caractéristiques spécifiques de chaque machine.
- Des outils de mise au point tels que des *debuggers* ou *profilers* (nous renonçons ici à une traduction française de ces termes, tant pis pour les *debogueurs* et autres *déverminateurs*) qui aident à la compréhension des programmes. Comme nous l'avons expliqué plus avant, quelles que soient les qualités des langages de programmation il y aura toujours des programmes dont le comportement n'est pas celui attendu. C'est alors le rôle de ces outils que de permettre la compréhension de ces phénomènes. Ils procèdent tous par inspection des programmes : on exécute, on regarde ce qui se passe et on essaye de comprendre pourquoi cela ne fait pas ce que cela devrait. Les outils de mise au point sont souvent difficiles à réaliser parce que, par définition, ils sont très peu formalisables. Un *debugger* ne peut pas respecter la sémantique du langage sur lequel il opère parce que, justement, il doit permettre des entorses au modèle d'exécution normal. Les qualités d'un *debugger* sont alors très subjectives ; elles tiennent dans une large mesure à son confort d'utilisation[1].
- Des outils d'administration quotidienne. En termes imagés ce sont nos aspirateurs, lave-vaisselles et autres sèche-linge de l'informatique. Ils ne sont pas forcément indispensables, on peut faire sans, mais on vit tellement mieux avec. Ce sont, par exemple, des outils assurant la gestion de versions, un accès simple et indexé à la documentation, une navigation interactive à l'intérieur de codes source volumineux, etc.

[1] Voilà encore une spécificité de l'informatique. La notion de *confort* n'est pas très scientifique et pourtant elle est ici essentielle. Combien d'entre nous préfèrent annoter nos programmes de *print* plutôt que d'utiliser un *debugger* ?

Dans la suite de cette introduction nous détaillerons ces deux aspects essentiels. En premier lieu, nous justifierons le choix de Scheme comme langage d'étude (§ 1.1). Nous présenterons ensuite les enjeux de la compilation et le type de techniques déployées pour atteindre un haut niveau de performances (§ 1.2). Enfin, nous présenterons un survol des outils complétant notre environnement de programmation (§ 1.3).

1.1 Pourquoi Scheme ?

Ce mémoire d'habilitation traite de la programmation en utilisant les langages fonctionnels. Nous présentons un environnement qui permet le développement d'applications réelles en Scheme. Nous exposons dans cette section les raisons qui ont conduit à l'adoption de ce langage. Scheme a des qualités généralement reconnues : simplicité, formalisation, haut niveau d'abstraction et quelques autres. Toutefois, à une époque où les langages fonctionnels (LFs) connaissent une relative disgrâce, où « tout le monde *fait* du Java »[2], quel sens y a-t-il encore à se servir de Scheme ? Pour présenter notre réponse à cette question, nous devrons parfois abandonner des considérations purement informatiques au profit d'arguments plus idéologiques (voire même peut-être psychologiques) qui induisent une réflexion introspective sur les fondements d'un travail de recherche scientifique.

À l'heure où nous voyons de plus en plus de nos proches collègues se tourner vers Java, nous sentons le besoin de motiver, si ce n'est de justifier, la direction de nos recherches. En aucun cas, l'objet de cette dissertation n'est le prosélytisme. Il ne s'agit pas pour nous de révéler à la face du monde la prétendue hérésie dans laquelle il se perdrait en ne programmant pas en Scheme. La motivation est ici toute personnelle et c'est pourquoi nous nous autoriserons parfois, dans cette unique section, l'emploi de la première personne du singulier plutôt qu'un anonyme pluriel.

1.1.1 Scheme, un laboratoire technique

Les caractéristiques dynamiques de Scheme (typage, la fonction standard `call/cc` qui permet de réifier les exécutions, ...) et la simplicité de sa syntaxe qui permet à un *programme* d'être également une *donnée* font de ce langage un outil de choix pour une recherche prospective sur les langages de programmation. Plus qu'avec n'importe quel autre langage il est facile d'expérimenter de nouvelles constructions sémantiques en Scheme. Les classes larges (§ 2.6) et les champs virtuels (§ 2.8) sont deux exemples de constructions originales, développées au dessus de Scheme, que nous présentons dans ce mémoire.

Scheme permet également un travail épistémologique parce qu'il peut facilement être utilisé pour disséquer des constructions existantes dans d'autres langages. Par exemple, dans l'ouvrage *Lisp In Small Pieces*, C. Queinnec établit une taxinomie des échappements présents dans presque tous les langages de programmation. L'intérêt de cette approche est de comparer et d'expliquer, dans un formalisme unique, des constructions d'apparence dissemblable. À notre connaissance, seul Scheme permet cette approche (quel langage pourrait par exemple être utilisé pour expliquer `call/cc` ?).

1.1.2 Scheme et les langages contemporains

Si Scheme était totalement déconnecté de la *réalité* informatique contemporaine, c'est-à-dire, s'il était sans ressemblance aucune avec d'autres langages actuellement répandus alors se reposerait le problème de la pertinence d'études basées sur ce langage. Des travaux de recherche qui n'auraient aucune influence en *dehors* de leur propre communauté n'auraient qu'un intérêt très limité. Il me semble que Scheme, et plus généralement la classe de langages, les LFs à laquelle il appartient, est un précurseur important et son influence sur les langages classiques se fait petit à petit sentir.

[2]Cette affirmation est bien sûr très exagérée : tout le monde ne fait pas encore du Java. En particulier, la communauté des LFs est encore vaste.

Java est *le* langage en vogue. À ce titre il est intéressant d'étudier les influences qui ont conduit à sa conception. Hormis sa syntaxe directement influencée par C et la mise en exergue de son modèle objet inspiré de Smalltalk, les similitudes entre Java, Smalltalk et Scheme sont frappantes.

- Même si Java possède plus de formes spéciales que Scheme, les trois langages sont petits (surtout comparés à C++ ou à Dylan [12, 214] qui a le même âge que Java).
- Les trois langages reposent sur une gestion automatique de la mémoire. Bien que leur invention remonte aux temps reculés de l'informatique [145, 144, 50],a les GCs sont enfin en passe d'être reconnus comme un bienfait pour la programmation. Les GCs ont été inventés pour Lisp, un langage dont Scheme est un dialecte. Java s'est donc nourri de l'expertise développée pour Lisp et les travaux actuels qui portent sur les GCs peuvent généralement s'appliquer aux deux langages. Il est par exemple frappant de noter les similitudes entre deux travaux de recherche actuels [231, 220], le premier concernant Java et le second ML et Scheme.
- Les classes internes [153] de Java et les *blocks* de Smalltalk ressemblent aux fermetures de Scheme car comme ces dernières ils permettent la capture d'environnements lexicaux.
- Ils utilisent un modèle d'exécution similaire car, dans les deux cas, la structuration du contrôle impose assez naturellement une gestion en pile des blocs d'activation dynamiques.
- Les deux langages sont sûrs (tous les accès mémoire sont contrôlés).
- Les objets portent des informations de type qui peuvent être consultées dynamiquement. Afin d'assurer la sûreté, les exécutions peuvent parfois être amenées à vérifier les types pendant les exécutions. Sous ses airs de langage typé statiquement Java est en fait un langage typé assez dynamiquement. Le rendre plus statique est l'enjeu de la discussion actuelle sur l'ajout de types paramétriques en Java [151, 153]. L'influence viendra probablement cette fois de ML. Actuellement, les programmes Java contiennent de nombreuses opérations nécessitant des vérifications de type lors de l'exécution.

Le cas des informations dynamiques de typage sort même du cadre de Java. Pendant une période d'environ une quinzaine d'années l'accent a été mis sur des bibliothèques d'exécution (en abrégé RTS de l'anglais *Run Time System*) où les valeurs étaient dépourvues de types [232, 140, 255, 104, 7]. C'est probablement des considérations d'efficacité qui ont conduit à ces choix. Maintenant que nous disposons de plateformes matérielles plus performantes, on voit le net retour des types dans les valeurs dynamiques. C'est vrai dans Java mais aussi dans C++, qui était pourtant par le passé très *statique*, par le biais du *Run Time Type Information* (RTTI) de C++ [131].

À cause des similitudes et des influences qui viennent d'être mentionnées, il n'est pas surprenant que les thèmes classiques de recherche sur l'implantation des langages fonctionnels soient maintenant transférés à des langages tels que Java. Par exemple, le déplacement des allocations du tas vers la pile qui a été abordé à notre connaissance pour la première fois en 1988 [184] et que nous avons appliqué à Bigloo en 1996 [211] (cf. § 3.1), semble maintenant être un sujet de recherche actif pour Java [45, 23].

1.1.3 Une recherche auto-contenue

Depuis maintenant plus de sept ans, je travaille sur l'implantation d'un système de programmation : Bigloo. Il est distribué depuis 1993 et il est assez largement utilisé. Pour ses utilisateurs, Bigloo est un système praticable qui leur permet d'écrire des applications (qui, parfois même, sont industrielles). Pour moi, Bigloo est une source de motivation et un terrain d'expérimentation. Comme il s'agit d'un système complet, basé sur un compilateur optimisant, tous mes travaux d'implantation peuvent y être validés (ou parfois invalidés). Par exemple, toutes les optimisations de compilation que j'ai publiées [203, 212, 205, 213, 211, 207] ont été intégrées dans le compilateur. Elles sont *fiables*, parce que leur correction théorique est démontrée dans les dites publications et *praticables* parce que l'expérience a montré qu'elles améliorent les performances et qu'elles ont une complexité et une implantation acceptables. Seule l'expérimentation dans un système réel, pouvant traiter des programmes de taille réelles, permet la validation. Bigloo, qui joue ce rôle, est donc pour moi un outil indispensable.

L'objectif de mes travaux est d'offrir un environnement de programmation performant, c'est-à-dire, un environnement qui permet plus *rapidement* ou plus *simplement* d'écrire des applications réelles efficaces et fiables. Parce que c'est un problème complexe et d'une taille réaliste, l'application de cet environnement à son propre développement permet la validation des idées explorées. Ce travail est essentiel car il permet de se garder d'une approche trop superficielle. Diffuser un logiciel impose une finition que les prototypes n'atteignent pas. De plus les exigences liées à ce développement sont elles-mêmes autant de source d'inspiration pour s'orienter vers de nouvelles recherches. Ainsi, parmi les constructions linguistiques ou les divers outils que nous présentons dans ce mémoire, plusieurs ont été initialement conçus pour le développement même de l'environnement.

Si je suis parvenu à mener à bien le développement de Bigloo c'est parce que la réalisation d'un tel projet permet des bénéfices incomparables.

Bigloo est un investissement extrêmement lourd car le développement d'un système opérationnel est extraordinairement chronophage (par exemple, le seul temps de rédaction des scripts d'installation ou de configuration de Bigloo pour toutes les variétés de machines doit maintenant se compter en semaines, voire en mois). Si parfois ce travail est réellement enrichissant (le *bootstrap* d'un compilateur est, par exemple, une expérience assez extatique) il peut être également rébarbatif et dépourvu de tout intérêt.

Scheme n'est pas, loin sans faut, sous le monopole de quelques groupes industriels. C'est un langage qui est développé à la fois par des universitaires et, depuis peu, par quelques membres de la communauté du logiciel libre. Ces équipes ont des ressources limitées comparables à celle dont nous disposons pour Bigloo. Le vaste effort d'implantation a permis à ce compilateur d'être aujourd'hui une des principales implantations de Scheme. En conséquence, Bigloo a une influence réelle sur le reste de la communauté Scheme et sur l'évolution du langage. Cette position est bien sûr très stimulante et il me semble qu'une situation comparable ne serait pas possible avec un langage à l'origine industrielle tel que Java.

1.1.4 Scheme, un vecteur de recherche

Comme nous l'avons indiqué § 1.1.1, notre choix s'est porté sur Scheme initialement pour des critères techniques. Si aujourd'hui nous persévérons avec ce langage c'est toujours parce que nous pensons qu'il a des qualités inégalées mais c'est aussi parce que nous pensons que le choix délibéré d'un langage sans pression industrielle est un moyen de faire une recherche sans entraves et peut-être plus originale.

Tous les langages de programmation déteignent sur la façon de programmer. Choisir par exemple Java, c'est adopter la façon de programmer en Java. Ceci me semble poser le problème plus général de l'originalité et de la liberté d'esprit indispensables à tout travail de recherche. Comment alors avoir des idées originales en adoptant un carcan intellectuel ? Comment avoir des idées nouvelles et, pourquoi pas, farfelues même, en allant dans *la* direction commune ? Choisir un langage trop populaire me semble castrateur !

Comme toutes les nouvelles technologies l'informatique mobilise des ressources très importantes (rappelons que l'informatique est maintenant la première industrie mondiale devant l'automobile). Les investissements et les possibles bénéfices sont énormes. L'informatique est encore une industrie à bâtir. De nombreux domaines sont en pleine expansion, sans normes de fait. D'autres domaines sont même encore tout simplement à découvrir. L'informatique est la quintessence de la course à l'innovation. Ainsi, la frontière entre recherche et industrie est parfois assez fine. Toutefois les objectifs sont différents. Dans un cas il s'agit d'approfondir ou d'étayer un savoir. Dans l'autre il s'agit de réaliser un profit. Comme le souligne le rapport Pitac [163] commandité par le gouvernement américain en 1998, au moins en informatique, la contrainte de réaliser un profit est incompatible avec les exigences de la recherche fondamentale car cette dernière doit s'inscrire dans une optique à plus long terme sans retombées économiques directes.

Choisir Scheme plutôt qu'un langage déjà adopté par l'industrie relève de cette idée. Scheme est souvent perçu comme un langage baroque et, par ceux qui ne le connaissent pas, inutile. Soit, cela n'en fait pas un mauvais vecteur de recherche, au contraire ! Son aspect *différent*, *exotique* est, dans notre esprit un avantage. C'est peut-être justement parce qu'il est à la fois différent et déroutant pour certains (mais très pur et sémantiquement lumineux) qu'il est intéressant.

1.1.5 Du plaisir...

Aux arguments précédemment avancés, d'autres plus personnels encore se greffent pour expliquer le choix de Scheme. L'idée même de recherche est pour moi fortement liée à celle de plaisir. La recherche dans la fonction publique n'est pas une profession particulièrement lucrative. Choisir cette voie n'est pas donc motivé par des arguments pécuniaires. Il me semble qu'elle est directement liée à l'enthousiasme de la recherche. L'enthousiasme et le plaisir à pratiquer certaines recherches sont pour moi des éléments déterminants dans la qualité et la quantité des résultats que je produis. Pourquoi alors ne pas reconnaître que je trouve Scheme plus amusant que tout autre langage ? Il n'y a là rien de rationnel, que du subjectif, mais pas tant inavouable que ça ! Cultiver la différence, ne pas faire comme tout le monde n'est pas tous les jours un rôle facile à jouer mais c'est toujours assez enivrant. Je crois profondément que sans plaisir et sans passion il ne peut y avoir pour moi de recherche valable. Même si c'était sur le ton de la plaisanterie, je crois que lorsque Simon Peyton-Jones, membre du *Microsoft Cambridge Research Center*, a déclaré lors d'une présentation en 1999 à PLDI « Haskell is the love of my life », qu'il s'agissait là d'un argument sincère.

1.2 Performances : mythes et complexes

Nous avons exposé dans § 1.1 les raisons de notre préférence pour Scheme. Il nous faut alors expliquer pourquoi ce langage n'est finalement que peu utilisé. Cette section présente un premier élément de réponse.

Pendant de nombreuses années, on a cru qu'il suffirait de disposer d'implantations efficaces pour que les LFs soient largement adoptés. La majorité des travaux d'implantation ont alors porté sur l'amélioration des performances. On a presque totalement délaissé les environnements de programmation (outils de mise au point et librairies). Quelques articles lucides nous ont bien mis en garde (en particulier par Gabriel [86] en 1991 et plus tardivement par Wadler en 1998 [250]) mais pendant la décennie passée on a principalement inventé et implanté des optimisations pour les compilateurs. Il est édifiant d'étudier les actes des principales conférences traitant de programmation fonctionnelle (jusqu'en 1995 il s'agissait de *Lisp and Functional Programming* en alternance avec *Functional Programming and Computer Architecture* qui ont été fusionnées en 1996 pour donner naissance à *International Conference on Functional Programming*). Parmi environ soixante articles traitant d'implantation ou décrivant des systèmes opérationnels, à peine une petite dizaine seulement se focalisent sur les environnements (trois ou quatre de mise au point et environ le même nombre de systèmes permettant la mesure de performances). Les autres traitent tous d'amélioration des performances, soit par le biais de compilation optimisante, soit par le biais d'amélioration des bibliothèques d'exécution. Même si toutes ces recherches ont eu des résultats probants, avec le recul de quelques années, force est de reconnaître qu'une telle polarisation a été une erreur parce qu'elle nous a conduit à négliger d'autres aspects essentiels. Le succès de Java qui pourtant ne brille pas par les performances de ses implantations actuelles en est la preuve évidente. L'eau de la mare des langages fonctionnels ondule encore du pavé Java qui y a été jeté. Les programmes écrits en Java sont loin d'être plus rapides que les programmes écrits en Scheme ou ML et pourtant ils sont déjà bien plus nombreux.

La communauté des LFs a peut être attaché trop d'importance aux performances, mais il semblerait que l'attitude opposée, adoptée par les premières implantations de Java, qui consiste à négliger cet aspect des implantations, soit également excessive. Non sans malice, on remarquera que si ce langage a initialement peu souffert de ses performances assez calamiteuses, il semble maintenant connaître comme un retour de flamme. Ses utilisateurs ne semblent plus se satisfaire de sa lenteur et on ne compte plus les projets qui ont renoncé à Java pour cette raison (en particulier Netscape [55]). Améliorer les performances des programmes Java semble être devenue, depuis, une préoccupation bien partagée. De nombreux travaux portent maintenant sur l'implantation efficace de Java (en se limitant aux dernières éditions des conférences Ecoop et Oopsla on peut citer, parmi les plus significatifs, [66, 197, 45, 23, 230]) ou d'autres [166] tentent de nous convaincre que les performances de Java ne sont pas *si* catastrophiques qu'il y paraît. C'est alors avec un plaisir non dissimulé que nous constatons l'application à Java des techniques que nous avons collectivement développées ces dix ou quinze dernières années pour les LFs. On peut alors prétendre sans trop de risques que si les performances ne sont pas à elles seules une garantie de succès elles constituent néanmoins un passage obligé.

Les progrès accomplis dans la compilation des LFs sont très importants. Bien que cela soit difficilement quantifiable les LFs n'ont plus à rougir de leurs performances. Un programme écrit en Scheme ou ML est globalement toujours plus lent qu'un programme C *équivalent* (c'est-à-dire un programme C produisant le même résultat) mais dans une proportion assez faible. Diverses expérimentations semblent indiquer un écart de performance oscillant entre un ratio de 1 à 3, les cas moyens se situant quelque part entre ces deux extrêmes. Être seulement trois fois plus lent que C, dans le pire des cas, nous semble être un résultat très positif. Citons ici notre livre de chevet [156] qui fait état de l'amélioration des performances des processeurs d'un facteur multiplicatif annuel de 1.6 ; autrement dit un programme Scheme aura toutes les chances d'être plus rapide que le même programme écrit en C, tournant sur une machine de deux ans plus ancienne. S'assurer de bonnes performances permet *objectivement* une plus large utilisation des LFs. Des traitements nécessitant des temps de réaction limités peuvent être implantés au moyen des LFs. Par exemple, la construction de ce mémoire utilise plusieurs filtres écrits en Scheme. Il faut environ 2 minutes pour traiter le document dans son intégralité. Si ces filtres n'étaient pas compilés et optimisés, des expériences montrent qu'ils seraient entre 20 et 100 fois plus lent. Il faudrait alors un minimum de 40 minutes pour pouvoir visualiser le document ou l'imprimer ! Passer de 2 minutes à 40 minutes impose un changement de pratique. Il faudrait renoncer à la visualisation du document ou réécrire les filtres pour une implantation plus performante.

Comme nous l'avons largement commenté dans [204], les LFs posent des problèmes inédits dans le domaine de la compilation. Atteindre les performances des implantations actuelles a nécessité plusieurs grandes étapes.

La compilation du contrôle

La première obligation a été de compiler correctement les appels de fonctions. G. Steele et D. Kranz furent parmi les précurseurs. Ils ont été les premiers à comprendre la nécessité de ne pas *allouer* pour le contrôle [227, 127, 128]. D'autres les ont ensuite suivis ([219], [199], ...). L'enjeu était réellement d'importance, il s'agissait de parvenir à compiler les appels de fonctions de façon optimale. Le terme « optimal » est à comprendre ici dans un sens dicté par les processeurs et systèmes d'exploitation eux-mêmes. Chaque couple processeur/système d'exploitation préconise un protocole d'appel ; c'est lorsqu'il est respecté que nous parlons d'*appel optimal*. Il faut noter qu'avec cette définition l'appel optimal n'est pas, en toutes circonstances, le plus efficace. Néanmoins, il est toujours *raisonnable*. La compilation des appels de fonctions a un impact énorme dans la compilation des LFs puisque justement, ce qui caractérise ces langages, c'est l'emploi généralisé des fonctions ! Les travaux précités ont tous proposé des méthodes pour ne plus allouer de structures de données dans le tas pour représenter les fonctions. Dans la grande majorité des cas, un appel de fonction Scheme est compilé comme le serait un appel de fonction C.

D'autres travaux complémentaires [189, 198, 239, 102, 48] ont traité d'un autre problème central pour certains langages dont Scheme : l'implantation de la récursion terminale. L'habitude en Scheme est d'implanter les boucles au moyen de fonctions récursives. Si ces appels de fonction allouent des blocs d'activation, même en pile, alors il y aura consommation mémoire et des récursions trop profondes pourront faire échouer les exécutions pour cause d'épuisement des ressources de la machine. Les compilateurs et les interprètes modernes n'allouent généralement plus de bloc d'activation pour les appels en position terminale. Ce résultat est d'importance car à partir du moment où les blocs d'activation ne sont plus alloués dans le tas et où les boucles sont compilées par de simples instructions assembleur *goto*, la compilation des fonctions pour les LFs devient de qualité comparable à celles de C, qui en terme de performances, est l'étalon.

D'autres analyses et optimisations, comme les analyses de flot de contrôle ou l'intégration fonctionnelle, ont également un impact sur la qualité du code produit. Nous les présentons dans le chapitre 3.

La représentation des données et les bibliothèques d'exécution

Étant supposé un contrôle bien compilé (c'est-à-dire, aux performances comparables à celles de C), reste le problème de la représentation des données. Deux difficultés se posent à Scheme (et rigoureusement dans les mêmes termes, à Java) :

- Les objets *embarquent* des informations de types qui peuvent être testées *dynamiquement*. Elles occupent forcément un peu de place. Une synthèse des diverses techniques efficaces pour les stocker peut être trouvée dans [99]. Afin de produire de meilleurs résultats, un compilateur optimisant tentera même de ne pas construire d'objets embarquant des types en déterminant statiquement certaines propriétés des programmes (comme par exemple les « zones monomorphes » [211]).
- Le langage repose sur l'emploi d'un gestionnaire mémoire automatique. Un GC n'est pas intrinsèquement plus lent qu'une gestion mémoire manuelle. Les objets alloués dans le tas ne sont pas moins efficacement traités par les GCs, au contraire [263]. La difficulté vient de ce que si un langage repose sur un gestionnaire mémoire automatique il ne propose plus de construction permettant une gestion fine de la mémoire (ce qui serait contradictoire avec la philosophie des GCs). Par exemple, en C, l'utilisateur alloue *et* libère ses objets. L'utilisateur doit donc estimer la durée de vie des objets pour les libérer dès que possible. À partir du moment où c'est au programmeur que revient la tâche d'estimer la durée de vie, il est logique de proposer plusieurs classes d'allocations. En particulier, il est cohérent que C propose un mécanisme d'allocation en pile qui est utilisable pour tous les objets qui ne survivent pas aux activations des fonctions qui les ont créé. L'allocation en pile a un avantage principal sur l'allocation dans le tas : quoiqu'on ait pu en penser [6, 11], elle est plus rapide. En particulier, avec les protocoles d'appel de fonctions en vigueur, libérer une zone allouée en pile a un coût nul. Il est donc possible d'écrire des programmes C qui gèrent plus efficacement la mémoire que des programmes Scheme.

Ces deux problèmes sont intrinsèquement aussi complexes l'un que l'autre et bien les traiter fait intervenir des techniques similaires. Le chapitre 3 présente la solution que nous préconisons dans le cadre de Scheme.

1.3 Environnements de programmation : un réveil tardif ?

Le désintérêt que connaissent les LFs s'explique peut-être en partie par leur manque d'environnements aidant le développement et la mise au point des programmes utilisant les ressources de nos ordinateurs modernes (image, son, animation, réseau, ...) [250]. Il nous semble qu'en plus de langages simples et bien définis, trois besoins se font sentir pour programmer des applications réalistes.

Des bibliothèques. Les ordinateurs gagnent en puissance. Les programmer devient alors de plus en plus complexe parce qu'il faut utiliser des ressources sans cesse plus nombreuses, ce qui nécessite un savoir toujours plus vaste. En particulier, un langage de programmation doit permettre le développement d'interfaces graphiques (souvent abrégées *GUIs*), de tâches réseau ou bien encore de l'administration système. Tout ceci requiert des bibliothèques d'exécution qui viennent se greffer au dessus des langages.

Des environnements intégrés. Programmer est complexe, en partie parce que c'est une activité qui regroupe de nombreuses tâches subalternes. Programmer et, plus encore, diffuser du logiciel, c'est jongler avec une multitude d'outils. Supposons par exemple un programmeur débutant un nouveau projet dans l'environnement Unix. Sa première tâche sera de construire un *1)* `Makefile` pour compiler son programme. Pour faciliter l'inspection et l'édition de son code, il pourra s'avérer utile de produire des fichiers `tag` (par exemple en utilisant la commande *2)* `etags`). Pour conserver les étapes intermédiaires conduisant à l'aboutissement du projet il faudra utiliser un système comparable à *3)* `cvs`. Quand des erreurs se manifesteront, il faudra utiliser un outil comme *4)* `gdb`. Si les performances doivent être améliorées, il faudra alors utiliser *5)* `prof`. Afin de pouvoir utiliser ces deux derniers outils, il faut compiler ses programmes en utilisant des options de compilation *ad hoc* qui généralement varient d'un compilateur à l'autre. L'écriture de la documentation se fera dans les formats *6)* `man`, *7)* `texinfo` ou *8)* `html`. Si le produit utilise des bibliothèques, il faudra les construire avec *9)* `ar` et *10)* `ranlib` ou *11)* `ld`. Tous ces outils utilisent, comme de bien entendu, une syntaxe propre. Cette hétérogénéité les rend difficiles à utiliser pour un non expert. Le rôle d'un environnement intégré de programmation est d'harmoniser leur emploi et de rendre leur accès plus facile. En plus, l'environnement doit prendre *automatiquement* à sa charge l'accomplissement de certaines tâches parmi les plus rébarbatives comme par exemple l'archivage du projet ou la gestion des versions successives.

L'intégration dans le système d'accueil. Inspirés par Lisp, de nombreux LFs ont été implantés dans des environnements hermétiques incapables de communiquer facilement avec le système d'exploitation les accueillant. En général ces environnements étaient basés sur ce qu'on a communément appelé une boucle *read-eval-print* (abrégée Repl par la suite). Ces environnements facilitent le développement interactif [191] mais ils ont deux inconvénients majeurs : *1)* ils conduisent les utilisateurs à penser que les LFs ne peuvent pas être compilés et donc qu'ils sont lents ; *2)* une boucle Repl ne s'intègre généralement que peu harmonieusement dans les systèmes d'exploitation modernes qui appartiennent à deux catégories : ceux utilisant des *GUIs* tel que MacOs ou Microsoft Windows et ceux basés sur une interface ligne, tel que Unix. Une boucle Repl est comme un système dans le système. Elle impose sa propre vision de l'utilisation des ordinateurs qui n'est pas forcément en accord avec celle du système hôte. Par exemple il est d'un usage fréquent, sous Unix, de communiquer des informations d'un processus à l'autre par le biais de *pipe*. Cela nécessite de petits programmes qui peuvent être lancés rapidement et qui utilisent les canaux d'entrée et de sortie standard pour communiquer. Cette approche est quasi inaccessible avec une boucle Repl.

Pour combler cette lacune des LFs, nous avons entrepris la réalisation d'un environnement complet pour Scheme. Il contient plusieurs librairies comme celle présentée dans le chapitre 2 qui permet la construction d'interfaces graphiques en Scheme. Elle utilise autant que possible les constructions de haut niveau de Bigloo (fermetures, modules, déclaration de classes, fonctions génériques) afin de permettre un style déclaratif et compact. La principale contribution de ce travail est de démontrer que les LFs modernes (c'est-à-dire ceux disposant principalement de modules et de classes) ont des atouts qui permettent souvent des développements plus rapides et plus élégants que les autres langages. Les autres composants de l'environnement sont présentés dans le chapitre 4. Comme le compilateur, l'environnement complet est autogène. C'est-à-dire que tous les outils de l'environnement de Bigloo sont développés dans l'environnement même, en Scheme.

1.4 Plan

On l'aura compris, les recherches présentées dans ce mémoire s'inscrivent dans un cadre clairement défini. De près ou de loin, tous les travaux présentés contribuent à l'élaboration d'un environnement de programmation basé sur le langage Scheme dans le but de contribuer à la diffusion de ce langage. Qu'il s'agisse d'extensions linguistiques, d'implantation efficace ou même d'outils de mise au point, tous nos travaux ont pour but de proposer un environnement de programmation alternatif mais réaliste. Bee, cet environnement, est la « cause commune », Bigloo, notre compilateur, en est sa pièce maîtresse. Nous avons écarté de ce mémoire nos quelques travaux annexes qui ne s'inscrivent pas dans ce projet, même s'ils traitent directement de programmation et de langages informatiques. Qu'il s'agissent d'optimisations de compilation dans Bigloo, de systèmes de visualisation du tas dans Kbdb ou encore de réflexion sur la programmation d'interfaces graphiques, toutes les recherches ici présentées ont donné lieu à une implantation dans Bee.

Dans le chapitre 2, nous présenterons les diverses variations autour de Scheme supportées par Bigloo. En particulier, nous détaillerons sa couche objet et une construction originale, nommée les *classes larges* qui est largement utilisée dans l'implantation du compilateur lui-même. Nous montrerons comment toutes ces constructions ont été utilisées pour la réalisation de Biglook, une bibliothèque graphique. Dans le chapitre 3, nous présenterons deux nouveaux aspect des techniques de compilation utilisées dans Bigloo. Nous détaillerons deux optimisations récentes qui se sont avérées être particulièrement efficaces. Enfin, dans le chapitre 4, nous présenterons une partie plus récente de nos recherches : la construction de l'environnement de programmation intégré. Nous présenterons quelques uns des outils qui le composent. En particulier, ceux qui permettent d'évaluer et de diminuer le nombre d'allocations. Les chapitres suivants seront constitués des articles qui détaillent les aspects techniques des thèmes abordés dans ces trois chapitres.

Chapitre 2

Bigloo : le langage

A language that doesn't affect the way you think about programming, is not worth knowing.
– Alan Perlis

MÊME si quasiment tous les langages de programmation ont la même puissance théorique, on ne programme pas avec la machine de Turing ! Cet incipit est volontairement une boutade mais il permet d'appréhender l'enjeu du choix d'un langage de programmation. On ne programme pas avec la machine de Turing car même si elle permet de résoudre tous les problèmes théoriquement « calculables » elle est inutilisable parce qu'elle offre un niveau d'abstraction beaucoup trop faible. Les langages de programmation ont une grande importance parce qu'ils façonnent la pensée et déterminent pour une grande part le style d'écriture des programmes. Le choix d'un langage est donc un engagement fort.

Objectivement, même si tous les langages généralistes modernes offrent des caractéristiques communes (récursion, liaison tardive pour les langages à objets, allocation dynamique) ils induisent des styles de programmation différents. Par exemple, la vision dominante de la programmation objet [89] et les méthodes de conception telles que UML [150] prônent une organisation où les éléments centraux sont les données. Les programmes sont conçus autour des interactions liant les différentes données traitées. Cette approche cherche même à masquer l'implantation du contrôle. Par contraste, avec les langages fonctionnels, le point de départ du programme est l'algorithme, c'est-à-dire le processus de calcul. L'implantation des données n'est déterminée que dans un second temps pour permettre une écriture pratique et efficace.

Même dans le cadre d'une même famille, le choix du langage reste déterminant. Par exemple, opter pour ML plutôt que Scheme, c'est choisir la sécurité du typage. Une fois compilé un programme ML marche ! En revanche, choisir Scheme, c'est probablement accepter une plus grande part de risque mais c'est aussi, parfois, avoir accès à une écriture plus dépouillée, où l'encodage des structures de données prend encore moins le pas sur l'algorithme. Les arguments en faveur des langages peuvent être conditionnés par les problèmes à traiter (certains langages sont mieux adaptés à certaines tâches que d'autres) mais souvent il s'agit aussi d'affinités personnelles. Nous avons été séduit par la concision et l'uniformité de Scheme. Ces qualités nous semblent plus importantes que les défauts de ce langage.

Scheme est un tout petit langage. C'est à la fois une grande vertu, par exemple pour l'apprentissage du langage, et une illusion. Cinquante pages (c'est le volume du document constitutionnel : le R^5RS [120]) pour décrire un langage de programmation est insuffisant. Certains traits pourtant indispensables sont passés sous silence. On peut relever deux principaux types de lacunes : *1)* une bibliothèque d'exécution trop minimaliste *2)* l'absence de quelques constructions fondamentales.

2.1 La bibliothèque d'exécution

La première lacune n'est pas la plus importante parce qu'il est facile pour une implantation de fournir les fonctions supplémentaires indispensables. Par exemple, la bibliothèque Scheme R^5RS ne contient pas de fonctions permettant de tester l'existence ou de détruire des fichiers mais toutes les implantations de Scheme proposent ces fonctionnalités. Autre exemple, plus gênant, le R^5RS est très flou sur la description des erreurs. Scheme n'a ni exceptions ni fonction standard d'erreur. Toute implantation de Scheme propose néanmoins des constructions *ad hoc*. Bigloo a une bibliothèque d'exécution nettement plus riche que celle décrite dans le R^5RS. Nous travaillons à son extension depuis 1991 [200]. Parmi les extensions les plus significatives, on peut noter : une interface système plus raisonnable (fichiers, *sockets*, processus, ...), des fonctions de lecture structurées (analyse lexicale et syntaxique) ou encore une gestion d'erreurs basée sur des exceptions. Même si chaque implantation de Scheme propose des palliatifs, les absences du R^5RS ont une conséquence très fâcheuse : un programme Scheme est très peu portable parce qu'il dépend immanquablement des constructions idoines de tel ou tel système. De plus, pour une raison difficilement explicable (qui tient peut-être justement à la petite taille du langage, ou, allez savoir, à l'envoûtement qu'il produit sur certaines personnes), Scheme est probablement le langage qui a le plus d'implantations (au moins une vingtaine). Loin d'être un avantage, cette situation est un inconvénient. Elle dilue les efforts de la communauté et comme nous l'avons signalé, elle rend les programmes Scheme peu portables. Trop d'implantations différentes nuisent probablement à la popularité d'un langage. Il est même frappant de constater que la plupart des langages qui sont actuellement populaires (Perl, Python, Tcl) n'ont qu'une seule implantation centralisée.

Pour tenter de remédier au problème des extensions trop cacophoniques, s'est mise en place une procédure nommée *SRFI* (*Scheme Request For Implementation*) à l'image des *RFIs* Unix. Cette procédure a pour but de normaliser les extensions Scheme. Chaque implantation de Scheme déclare alors les *SRFIs* qu'elle implante. Par exemple, Bigloo implante actuellement environ la moitié des *SRFIs*. Même si les *SRFIs* engendrent beaucoup d'activité, il est encore prématuré de tirer les conclusions de cette expérience. Il n'est pas encore acquis que Scheme y gagnera en portabilité.

2.2 De nouvelles constructions

Hormis les fonctions d'interface système qui ne peuvent être écrites en Scheme si la bibliothèque proposée par une implantation est trop pauvre, un utilisateur a la possibilité de l'étendre en définissant ces propres fonctions et en les utilisant dans ces programmes. Une des grandes forces de Scheme est qu'en plus des fonctions de base, l'utilisateur peut aussi définir de nouvelles constructions par le biais des macros [69, 125, 47, 70, 120]. Les macros de Scheme (qui sont les descendantes directes de Lisp) sont, à notre connaissance, d'une puissance inégalée. Elles permettent de définir de nouvelles constructions qui peuvent être arbitrairement complexes parce que le langage pour calculer le résultat de leur expansion est Scheme lui-même et parce qu'un source Scheme est quasiment un arbre de syntaxe abstraite tant sa syntaxe est dépouillée. En général, quand un système de programmation propose des macros, c'est par le biais d'un langage dédié au calcul des expansions. Généralement ce langage est pauvre, comme par exemple Cpp qui utilise un langage très réduit et qui n'a rien de commun avec C. C'est le cumul du langage d'expansion puissant et de la simplicité de la syntaxe des codes sources qui confère aux macros de Scheme leur puissance.

Bigloo utilise des macros pour implanter un certain nombre de constructions de haut niveau. Par exemple son compilateur de filtres [168, 172, 169] est une macro, certes assez complexe. Les exceptions sont construites au dessus de `call/cc` par quelques macros. Un utilisateur pourrait lui-même définir de telles extensions.

Par contraste, il est des constructions encore plus fondamentales, encore plus étroitement liées au compilateur, que le système doit fournir de façon primitive parce qu'elles influencent tout le reste du système. Pour Bigloo, il s'agit principalement d'une *interface externe* et d'un *langage de* *modules* (un expert, et nous en connaissons, serait probablement capable d'écrire un compilateur de modules en utilisant une macro) ou encore d'une couche objet. L'interface externe de Bigloo a déjà été décrite dans [204] ; nous ne la présenterons pas ici. En revanche, nous présentons les modules qui n'ont cessé d'évoluer (§ 2.3) et notre couche objet (§ 2.5). Nous montrerons en particulier, les liens et les différences entre modules et classes de Bigloo. Par la suite, nous montrerons deux extensions que nous proposons au modèle de programmation objet classique à base de classes : les *classes larges* (§ 2.6) et les *champs virtuels* (§ 2.8). Nous présentons également dans ce chapitre, la bibliothèque graphique Biglook qui exploite toutes les constructions présentées ici (§ 2.7).

2.3 Les modules

Les modules de Bigloo ont deux fonctions essentielles : permettre la compilation séparée et augmenter le nombre d'erreurs potentiellement détectables par le compilateur. Ils sont simples car ils ont été conçus dans l'optique d'être faciles à implanter. Nous ne présentons ici que la partie des modules indispensable à la compréhension de la présentation des objets et des innovations que nous décrivons dans ce mémoire.

Un module est l'unité de compilation de Bigloo. Il est physiquement incarné par un ou plusieurs fichiers. Il a la syntaxe suivante :

```
(module module-name
   (import importation+)*
   (export exportation+)*
   (static static+)*)

optional-body
```

2.3.1 Les clauses de module

Les clauses d'*importation* servent à importer des liaisons dans le module. Lors d'une importation, on précise juste le nom (un identificateur de variable, de fonction ou de classe) à importer et dans quel module il se trouve.

Les clauses d'*exportation* et les clauses *statiques* jouent un rôle très proche. Elles signalent au compilateur que le module implante certaines liaisons et indiquent si les dites liaisons sont visibles depuis d'autres modules (exportées) ou pas (statiques). Ces clauses ne sont pas constituées d'identificateurs mais de prototypes. On peut ainsi exporter des variables (des liaisons modifiables) ou des fonctions (liaisons non modifiables). Les clauses statiques sont facultatives (les liaisons du module référencées dans aucune clause sont statiques).

Voici en exemple le module `fib` qui importe les fonctions `fib-fx` du module `fib-fixnum` et `fib-fl` du module `fib-flonum` et qui exporte la fonction `fib`.

```
(module fib
   (import (fib-fx fib-fixnum)
           (fib-fl fib-flonum))
   (export (fib x)))

(define (fib x)
   (if (fixnum? x)
       (fib-fx x)
       (fib-fl x)))
```

Lors de la compilation, Bigloo effectue un certain nombre de vérifications :

- Toutes les variables et fonctions référencées doivent être définies dans le module ou mentionnées dans une clause d'importation (il ne peut y avoir de variable libre).
- Tous les identificateurs mentionnés dans les clauses statiques ou d'exportation doivent être définis dans le module avec une valeur compatible avec leur prototype.

– Les identificateurs mentionnés dans les clauses statiques ou d'exportation dont le prototype est une fonction ne doivent pas être affectés.
– Tous les appels de fonction où l'opérateur fonctionnel est connu comme étant une fonction (l'identificateur en position de fonction a un prototype de fonction) doivent avoir un nombre d'arguments compatible avec le prototype de la fonction invoquée.

2.3.2 Les corps de module

Les expressions constituant le corps d'un module peuvent être des définitions de variables ou fonctions (au moyen de la forme Scheme `define`) ou bien des expressions Scheme quelconques. *Initialiser* un module est l'opération qui consiste à établir les liaisons variable/valeur et à évaluer les expressions qui ne sont pas des définitions. Les liaisons et les expressions sont exécutées dans leur ordre d'apparition dans le module.

Les graphes d'importation des modules ne sont pas limités à des arbres. C'est-à-dire que deux modules, ou plus, peuvent s'importer mutuellement. L'ordre d'initialisation des modules est alors non spécifié.

2.3.3 Des bibliothèques supplémentaires

Bigloo traduit des modules en fichiers objets et les lie pour produire des exécutables. Il utilise le même format que la plate-forme d'accueil pour les fichiers objet. Ainsi, sous Unix, les outils standard (`ar`, `ranlib` et `ld`) peuvent être utilisés pour construire des bibliothèques Scheme.

Pour utiliser une bibliothèque dans un code utilisateur, il suffit alors d'employer la clause de module `library` dédiée à cet usage. Ainsi :

```
(module client
   (library biglook))

(pack (instantiate::button))
```

est une application qui utilise la bibliothèque `biglook` qui est comme nous le verrons ultérieurement dans 2.7 une bibliothèque d'objets graphiques. Toutes les fonctions, variables et classes de cette bibliothèque sont alors disponibles pour l'application. L'exemple présent utilise la fonction `pack` et la classe `button` de la librairie. Lors de l'initialisation, les modules utilisés de la bibliothèque, et uniquement ceux-ci, seront initialisés comme les modules implantés dans le code de l'application.

2.4 Le typage

Le langage Scheme est très fortement ancré dans une tradition de typage dynamique. Autrement dit, en Scheme, c'est lors de l'exécution que les vérifications de type sont effectuées. On a coutume de dire qu'un programme ML peut être compilé que s'il n'existe pas d'interprétation fausse vis-à-vis des types alors qu'un programme Scheme est compilé s'il existe au moins une interprétation juste. Bigloo se place clairement dans la tradition Scheme mais contrairement aux systèmes répandus, il essaye effectivement de montrer qu'il existe au moins une interprétation juste des programmes avant de les compiler. Le résultat est que, parfois, Bigloo refuse de compiler des programmes car il détecte des erreurs de typage.

L'autre tradition de Scheme (partagée avec ML cette fois) est de ne pas écrire de types dans les programmes. Il n'y a pas moyen de déclarer qu'une variable ne contient des données que d'un type précis. Bigloo tourne le dos à cette tradition et encourage les annotations de type comme en Lisp. Elles sont facultatives mais elles permettent d'obtenir un code produit de meilleure qualité et de détecter plus d'erreurs lors de la compilation.

Un type Bigloo est soit un type Scheme atomique (par exemple, un entier Scheme, une chaîne de caractères Scheme, une liste Scheme, ...), un type externe (par exemple, un entier C, une chaîne de caractères C, une structure C, ...), ou un type associé à une classe (les classes seront présentées dans § 2.5).

Les annotations de type peuvent être insérées dans les clauses d'exportation des modules, dans les définitions de variables ou dans les blocs lexicaux. La construction syntaxique est [<*var-id*>]::<*type-id*>. Voici en exemple un module qui implante la fonction de Fibonacci sur les entiers.

```
(module fibo
   (export (fib::long ::long)))

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

La clause `(fib::long ::long)` signifie simplement que le module implante une fonction nommée `fib` qui prend un argument de type `long` et retourne un résultat lui-même de type `long`. Il est inutile de reporter les annotations de type sur les définitions lorsqu'elles sont déjà indiquées sur une clause de module ; c'est pourquoi, ici, l'annotation `::long`[1] n'est marquée que sur l'exportation de `fib`.

Le typage de Bigloo le place vraiment en marge de Scheme. D'une part, parce que, comme nous l'avons dit, la tradition n'est pas d'utiliser des annotations de type et, d'autre part, parce qu'il oblige à renoncer définitivement à la récursion terminale pour les fonctions dont le type de retour est spécifié. En effet, lorsque le résultat d'une fonction est annotée et que Bigloo ne peut démontrer que la valeur calculée par la fonction est du type annoté, il insère alors un test dans le corps de la fonction. Ainsi la fonction :

```
(define (foo::a-type x)
   (bar x))
```

doit être compilée en :

```
(define (foo::a-type x)
   (let ((tmp (bar x)))
      (if (a-type? tmp)
          tmp
          (error "foo" "Illegal type" tmp))))
```

On voit ici pourquoi les récursions terminales ne peuvent plus être garanties dans le cas général.

2.5 Une couche objet

Le paradigme objet détient une position hégémonique dans le monde moderne de la programmation. La majorité des informaticiens semble s'accorder à dire que ce style de programmation facilite le développement et la maintenance des programmes. Chaque communauté met des objets dans ses langages. C'est vrai pour les nouveaux langages tels que le déjà très célèbre Java [95] ainsi que pour les plus anciens tels que C et ses extensions, C++ [232] et Objective-C [53], ou encore O'Caml [176] et $ML_{\leq}$ [29].

« Langage objet » est un terme bien vague qui peut décrire des langages bien différents. Il existe plusieurs catégories de langages objets très différentes les unes des autres. Distinguons-en ici deux : le modèle de Smalltalk [91] qui est caractérisé par le fait que les méthodes sont associées aux classes et le modèle de Clos [24] où les méthodes sont associées à des fonctions génériques. Chacun de ces modèles peut être décliné en plusieurs variantes utilisant un *typage* statique ou dynamique, utilisant de l'*héritage* simple ou multiple et une *sélection* simple ou multiple. Notre choix s'est porté sur le modèle de Clos dans une variante basée sur de l'héritage simple et une sélection simple. La plupart des extensions linguistiques de cette section sont inspirées de celles de Meroon, la couche objet de C. Queinnec [170].

[1] Nous attirons ici l'attention du lecteur sur le fait que le type `long` dénote les entiers C et non pas les entiers Scheme. Bigloo manipule indifféremment les données Scheme et C.

2.5.1 Le modèle

Entre le modèle de Smalltalk où les méthodes sont définies dans les classes et le modèle de Clos où les méthodes sont regroupées dans des fonctions génériques, nous avons opté pour la deuxième solution. Ce choix a été motivé par la nature et l'architecture du programme, le compilateur Bigloo, que nous souhaitions réécrire avec des objets (en partie pour évaluer les apports de cette approche de la programmation dans le cadre d'un développement réel et déjà bien établi) et parce que nous pensons qu'il s'accorde mieux avec la programmation fonctionnelle classique.

Le compilateur Bigloo est un programme Scheme d'environ 45000 lignes. Il lit un programme à compiler et construit l'arbre de syntaxe abstraite représentant le programme. Cet arbre est une structure composée, dans sa version actuelle, de 23 types de nœuds différents. Il y a un nœud pour représenter les constantes, les affectations de variable, les conditionnelles, les appels de fonction, etc. Le compilateur est structuré en passes. La version actuelle contient une vingtaine de passes. Chacune d'elle peut être considérée comme une procédure modifiant ou annotant l'arbre de syntaxe abstraite. Le moteur du compilateur est donc une fonction Scheme ressemblant à :

```
(define (compilateur src)
  (let ((ast (build-ast src)))
    (macro-expand! ast)        ;; 1ère passe
    (function-inline! ast)     ;; 2ème passe
    ...
    (code-generate! ast)))     ;; 20ème passe
```

Chaque passe simple est implantée dans un fichier. Les plus complexes sont éclatées en plusieurs fichiers. Pour la nouvelle version (la version du compilateur réécrite avec des objets), nous avons impérativement souhaité conserver cette décomposition parce qu'elle nous semble être la mieux adaptée.

Cette décision écarte donc de notre choix le modèle objet de Smalltalk car la façon naturelle d'implanter notre compilateur serait d'abandonner notre structure guidée par les passes au profit d'une structure guidée par l'arbre de syntaxe abstraite. Chaque passe du compilateur serait éclatée dans toutes les classes qui implantent les nœuds de l'arbre. Il n'y aurait plus, par exemple, un fichier contenant le code de la passe `function-inline!`. Ce code aurait dû être éclaté dans les classes implantant les nœuds de l'arbre de syntaxe.

L'implantation de notre compilateur dans le modèle objet de Smalltalk, pourrait se faire au moyen du *schéma des visiteurs* (*Visitor Pattern*), décrit initialement dans [89]. Ce schéma permet l'extensibilité fonctionnelle (l'ajout de nouvelles passes, pour notre compilateur) mais, comme relevé dans l'article [129], il ne permet pas facilement l'extensibilité structurelle (l'ajout de nouveaux nœuds à notre arbre). Le schéma du *visiteur extensible* (*Extensible Visitor pattern*) permet de palier les lacunes du *Visitor Pattern*t [74, chap. 9]. Toutefois, l'emploi de ce schéma de programmation nuit à la simplicité et à la clarté du programme source. En particulier, le programmeur doit définir de nombreuses classes qui ne servent pas à modéliser les structures de données, mais qui sont simplement utiles au contrôle du flux d'exécution du programme.

Cependant pour notre compilateur, le modèle de Smalltalk aurait permis d'ajouter aisément de nouveaux nœuds à l'arbre de syntaxe abstraite. Ajouter une passe avec le modèle de Smalltalk nécessite de modifier et de recompiler tout le code déjà existant (parce qu'il faut ajouter les définitions des méthodes de la nouvelle passe) ; en revanche, ajouter un nœud n'exigerait pas de modification du code déjà présent. Ajouter un nœud revient à ajouter une nouvelle construction au langage compilé alors qu'ajouter une passe correspond bien souvent à l'implantation d'une extension du compilateur (comme une nouvelle optimisation). L'expérience nous a montré que le langage évolue bien moins vite que le compilateur lui-même. Avec le modèle de Clos, ces problèmes ne se posent pas. Il est aussi facile de rajouter des passes que de nouveaux nœuds.

Comme nous l'avons signalé pour le compilateur, placer les méthodes dans les classes fige le comportement des objets de cette classe. On peut procéder par héritage pour construire des objets ayant un comportement étendu mais souvent l'héritage introduit lui-même des limites (par exemple tous les langages ne proposent pas d'héritage multiple). En revanche, avec les fonctions génériques il est possible *à posteriori* de rajouter de nouveaux comportements aux objets définis par les classes, sans restriction. Dans § 2.7.3, page 44, nous présentons ce point en détail. Les fonctions génériques présentent l'avantage de l'extensibilité qui est une des vertus souvent mise en avant pour vanter les mérites de la programmation objet. D'autres arguments en faveur des fonctions génériques peuvent être trouvés dans [41, 147].

Puisque les classes de Bigloo ne portent pas de code, elles ne servent pas à structurer les programmes. Ce rôle est détenu *exclusivement* par les modules. Les classes ne servent qu'à modéliser les structures de données.

2.5.2 Les déclarations de classes

Les classes sont déclarées dans les clauses d'exportation ou dans les clauses statiques d'un module. Il est donc possible de rendre visible à d'autres modules une déclaration de classe ou de limiter la portée de sa déclaration à un module. La syntaxe abrégée de la déclaration d'une classe est :

```
(class class-id[::super-class-id]
   <field>*)

<field> ::= field-id[::type-id]
  | (field-id[::type-id] <option>*)

<option> ::= read-only
  | (default <s-expression>)
```

Une classe ne peut hériter que d'une seule super-classe. Les classes dont la super-classe n'est pas spécifiée héritent de la seule classe de la librairie, la classe `object`. Le type associé à une sous-classe est un sous-type du type associé à la super-classe.

Un champ peut être typé (par l'annotation d'un `::`*type-id*), ne pas être modifiable (`read-only`) et avoir une valeur par défaut (`default`). Voici de possibles déclarations pour les classiques `point` et `point-3d` :

```
(module les-petits-points
   (export (class point
              (x::double read-only (default 0.0))
              (y::double read-only (default 0.0)))
           (class point-3d::point
              (z::double read-only (default 0.0)))))
```

2.5.3 Les fonctions génériques

Les déclarations de fonctions génériques sont des déclarations de fonctions annotées du mot clé `generic`. Elles peuvent dont être exportées, ce qui signifie qu'elles peuvent être utilisées depuis d'autres modules et que des méthodes peuvent être ajoutées à ces fonctions depuis d'autres modules. Elles peuvent être statiques et donc invisibles depuis d'autres modules. Dans ce cas, aucune méthode ne peut-être ajoutée à celles introduites par le module définissant la fonction générique.

La syntaxe de définition d'une fonction générique est très proche de celle de la définition d'une fonction :

```
(define-generic (fun-id[::type-id] arg-id[::class-id] ...)
   optional-body)
```

Les fonctions génériques se distinguent des fonctions simples par le fait que leur définition peut être spécialisée par des méthodes. Dans ce cas, pour une déclaration de fonction générique, le programme comptera plusieurs déclarations de méthodes.

Les fonctions génériques doivent avoir au moins un argument puisque c'est leur premier argument qui est utilisé pour réaliser la liaison dynamique des méthodes (*dynamic dispatch*). Cet argument peut être typé et dans ce cas, le type τ doit être associé à une classe et il est impossible d'ajouter à la fonction générique des méthodes dont le premier argument n'est pas un sous-type de τ. Les fonctions génériques peuvent avoir un corps. Ce corps est évalué si lors d'un envoi de message, aucune méthode n'est définie pour le type du premier argument. Si cette situation se produit alors que la fonction générique n'a pas de corps, une erreur dynamique est signalée. Voici un exemple de définition de fonction générique dont l'argument discriminant n'est pas typé :

```
(define-generic (display-point p . port)
   (if (number? p)
       (display p port)
       (error "display-point" "Illegal argument" p)))
```

2.5.4 Les méthodes

Les méthodes permettent du polymorphisme *ad hoc* car elles surchargent, type par type, les définitions des fonctions génériques. Elles sont déclarées au moyen de la forme syntaxique :

(define-method (*fun-id*[::*type-id*] *arg-id*[::*class-id*] ...)
body)

C'est une erreur de définir une méthode s'il n'existe pas de fonction générique définie avec un prototype compatible (type des arguments et du résultat en relation de sous-typage avec ceux de la fonction générique). Voici un exemple de méthode :

```
(define-method (display-point p::point . port)
   (with-access::point p (x y)
      (print port "{" x ", " y "}")))
```

Il est possible d'invoquer à partir d'une méthode m, la méthode qui aurait été utilisée si m n'avait pas été définie (c'est-à-dire d'invoquer la méthode associée à la plus proche super-classe de l'objet). Cela se fait au moyen de la forme syntaxique `(call-next-method)`. Les arguments de la nouvelle méthode appelée sont les mêmes que les arguments de la méthode qui utilise le `call-next-method`. Cette forme n'est accessible que dans la définition d'une méthode.

2.5.5 Les instances

Lorsqu'une classe `class-id` est déclarée, Bigloo génère automatiquement un prédicat `class-id?`, un allocateur `instantiate::`*`class-id`*, un copieur `duplicate::`*`class-id`*, des accesseurs (pour un champ `x` l'accesseur se nomme `class-id-x`), des modificateurs (pour un champ `x` modifiable, le modificateur se nomme `class-id-x-set!`) et, enfin, une forme abrégée `with-access::`*`class-id`* qui permet d'accéder et de modifier les champs en les référençant seulement par leur nom. Voici un exemple d'allocation et d'accès à une instance :

```
(let ((p (instantiate::point-3d (y -3.4) (x 1.0))))
   ;; La valeur d'initialisation d'un champ peut être omise de la liste
   ;; des arguments si ce champ a une valeur par défaut comme c'est
   ;; le cas ici pour le champ z des point-3d.
   (with-access::point-3d p (x y z)
      (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

La version équivalente de cette expression sans utiliser la forme de raccourci `with-access::` est :

```
(let ((p (instantiate::point-3d (y -3.4) (x 1.0))))
   (sqrt (+ (sqr (point-3d-x p)) (sqr (point-3d-y p)) (sqr (point-3d-z p)))))
```

Les fonctions `point-3d-x`, `point-3d-y` et `point-3d-z` sont les fonctions d'accès aux champs des instances de la classe `point`. Elles sont construites automatiquement par le compilateur.

2.5.6 Introspection et sérialisation

Bigloo propose des fonctionnalités d'introspection pour les objets. Schématiquement, il est possible de déterminer lors de l'exécution le nom des champs et la classe de tout objet. Il est aussi possible d'avoir accès à la liste de ses accesseurs et de ses modificateurs. Grâce à cette possibilité, la bibliothèque d'exécution standard de Bigloo dispose d'un écrivain «raisonnable» d'objets et implante le prédicat `equal?` pour les objets.

Les objets de Bigloo peuvent être sérialisés et dé-sérialisés au moyen des fonctions de librairie `obj->string` et `string->obj`. Cette sérialisation/dé-sérialisation respecte le partage présent dans chaque objet. La complexité de la sérialisation est en $\mathcal{O}(n)$ où n est la taille de l'objet à sérialiser. La dé-sérialisation a la même complexité. Lorsqu'un objet est dé-sérialisé, une vérification est effectuée pour s'assurer que la classe de cet objet actuellement existant dans l'exécution est bien la même que celle qui existait lorsque l'objet a été sérialisé.

2.5.7 Conclusion sur les objets

Fournir une implantation efficace a été à la base de la conception de la couche objet de Bigloo. Ainsi nous avons choisi de ne proposer que de l'héritage simple et une sélection simple parce que nous savions mieux les traiter. Des méthodes pour implanter les prédicats de classes en temps constant sont connues dans le cadre de l'héritage multiple [126, 247] mais l'accès aux champs des objets est plus complexe. Des techniques pour accéder aux champs des objets qui héritent de plusieurs classes simultanément au moyen d'une seule lecture mémoire sont connues mais en général elles ne s'appliquent que si les accès ont lieu dans le corps des méthodes [140]. Dans le cas de Bigloo, les champs sont accédés depuis tous les points du programme et là les techniques efficaces ne s'appliquent pas.

Les objets pour le *bootstrap* de Bigloo

La première utilisation des objets a été la réécriture du compilateur. Ainsi nous sommes capable de présenter un bilan du passage d'une programmation fonctionnelle à une programmation objet. Nous disposons de deux compilateurs, $\text{Bigloo}_{\text{fun}}$ et $\text{Bigloo}_{\text{oo}}$, qui implantent de deux façons différentes les mêmes algorithmes.

Les deux compilateurs $\text{Bigloo}_{\text{fun}}$ et $\text{Bigloo}_{\text{oo}}$ sont très comparables. À quelques détails près, ils implantent les mêmes passes de compilation et utilisent les mêmes optimisations. Le langage de $\text{Bigloo}_{\text{oo}}$ est bien sûr plus étendu que celui de $\text{Bigloo}_{\text{fun}}$ puisque ce nouveau compilateur traite la couche objet. Malgré ce code supplémentaire, les deux compilateurs sont restés de taille très proche : 41000 lignes (33500 sans commentaires) pour $\text{Bigloo}_{\text{fun}}$ et 45000 lignes pour $\text{Bigloo}_{\text{oo}}$ (34000 sans commentaires). Les objets semblent donc permettre un code plus compact. Ce résultat doit être considéré avec précaution car, pour $\text{Bigloo}_{\text{oo}}$, nous n'avons que réimplanté un programme déjà existant : il est donc, à priori, un peu mieux écrit que $\text{Bigloo}_{\text{fun}}$.

Le principal apport de la technologie objet a été pour nous le sous-typage. C'est à notre sens ce qui permet économie et réutilisation du code. En revanche, nous ne pensons pas que la liaison tardive (le sélection dynamique) représente une avancée majeure. Pour un langage n'offrant pas de structures de contrôle évoluées, comme c'est par exemple le cas de C, l'ajout de la liaison tardive (*sélection* dynamique comme nous l'avons jusqu'à présent nommée ou encore *dispatch* ou *late binding* en Anglais) aide vraiment le programmeur (dans ce sens C++ apporte vraiment *un* plus à C). En revanche, pour un langage offrant du filtrage, l'intérêt d'une sélection dynamique est beaucoup plus faible tant ces deux constructions sont proches.

Le couple fonction générique/méthode possède l'avantage de pouvoir être éclaté sur plusieurs fichiers (les méthodes peuvent être déclarées dans d'autres fichiers que leur fonction générique) et d'être extensible sans modification du code déjà existant. En revanche, il est un peu plus fastidieux à utiliser. Examinons les deux styles dans un fragment de code implantant une descente dans un arbre :

```
(define-generic (walker node::node))

(define-method (walker node::leaf)
   (with-access::leaf node (value)
      value))

(define-method (walker node::binary)
   (with-access::binary node (left right)
      (cons (walker left) (walker right))))

(define-method (walker node::ternary)
   (with-access::ternary node (left middle right)
       (list (walker left)
             (walker middle)
             (walker right))))
```

Le même programme avec des structures et du filtrage serait :

```
(define (walker node::node)
   (match-case node
      (#{leaf ?value}
       value)
      (#{binary ?left ?right}
       (cons (walker left) (walker right)))
      (#{ternary ?left ?middle ?right}
       (list (walker left)
             (walker middle)
             (walker right)))))
```

Le code utilisant du filtrage (`match-case`) est un peu plus dense et donc un peu moins fastidieux à écrire. De plus, même si ce n'est pas apparent dans cet exemple, les filtres ne sont ni obligatoirement contraints par une seule variable ni forcément linéaires. Le fait que Bigloo ne propose que de la sélection dynamique simple pour les fonctions génériques augmente cette différence. Le filtrage est très utile pour implanter des transformations source à source. Par exemple, la forme Scheme `let` n'est pas une forme primitive. Elle peut être implantée par le biais d'une macro qui la transforme en une application directe d'une λ-expression (c'est exactement de la sorte que procède l'interprète de Bigloo). Toutefois, lorsqu'on compile ces applications directes de fonction, le compilateur a tout intérêt à y reconnaître le plus tôt possible la construction de blocs lexicaux. Une telle transformation peut être implantée par un seul filtrage. À titre d'exemple nous montrons également une autre transformation opérée par Bigloo qui consiste à supprimer des constructions `let` dont le seul objet est de lier une fonction. Ainsi on opère la transformation :

$$\mathcal{S}\left[\!\!\left[\begin{array}{l}\ddots\\ \texttt{(let ((}\mathit{fun}\ \boxed{\texttt{(lambda}\ \mathit{formals\ body}\texttt{)}}\ \texttt{))}\\ \quad \mathit{fun}\texttt{)}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\ddots\\ \boxed{\texttt{(lambda}\ \mathit{formals\ body}\texttt{)}}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

Le filtrage pour ces deux transformations peut être :

```
(define (source-to-source s-exp)
   (match-case s-exp
      (((lambda ?formals . ?body) ?actuals)
       (source-to-source `(let ,(map list formals actuals) ,@body)))
      ((let ((?fun (lambda ?formals . ?body))) ?fun)
       (source-to-source `(lambda ,formals ,@body)))
      ...
      (else
       s-exp)))
```

On voit ici l'intérêt d'un filtre non linéaire (la variable de filtre `?fun` apparaît deux fois dans le second filtre). Une telle transformation serait plus laborieuse à réaliser avec des objets.

Malgré le passage à une programmation objet, la taille des deux versions du compilateur est comparable. La taille de $\text{Bigloo}_{\text{fun}}$ (comprenant le compilateur et l'interprète) est de 1.5MB sur Sparc pour 1.8MB pour $\text{Bigloo}_{\text{oo}}$. Il y a donc un accroissement d'environ 20%. On peut penser que cet accroissement de la taille de l'exécutable est normal puisque le langage compilé est plus vaste. Ce n'est pas totalement vrai puisque les deux compilateurs ont sensiblement le même nombre de lignes de code source. Néanmoins, on peut affirmer que le passage aux objets n'a pas entraîné une explosion de la taille des codes compilés.

Pour mesurer les performances du compilateur en terme de vitesse d'exécution, nous nous sommes limités à une unique mesure comparative. Il n'est pas très juste de mesurer le temps passé par $\text{Bigloo}_{\text{fun}}$ pour compiler un fichier et de le comparer au temps de $\text{Bigloo}_{\text{oo}}$ car la nouvelle version applique plusieurs fois certaines passes. C'est, par exemple, le cas de l'intégration fonctionnelle qui est appliquée en début de la compilation et en fin, après l'insertion des tests de type. Il est donc normal que $\text{Bigloo}_{\text{oo}}$ soit un peu plus lent. Nous avons mesuré le temps passé par Bigloo pour compiler son lecteur. C'est un assez bon test, puisque c'est un des plus longs fichiers à compiler. Le lecteur contient une grammaire lexicale qui est compilée vers un automate déterministe qui est lui-même expansé vers une fonction Scheme. Sur une Sparc 5 avec 64 Mega de mémoire, la compilation du lecteur prend 10.3 s avec la version fonctionnelle et 11.8 s avec la version objet. La nouvelle version est donc un peu plus lente d'environ 15%. Il est difficile d'établir la cause de cette perte de performances. D'une part, les applications multiples des optimisations ralentissent le compilateur, d'autre part, puisqu'il est auto-compilé, le compilateur résultant devrait bénéficier d'une compilation plus optimisée et il devrait donc être plus performant. Il est très difficile de savoir si le fait d'appliquer deux fois une passe suffit à améliorer le compilateur produit d'un facteur qui lui permet de récupérer le temps passé dans la deuxième application de la passe. Nous n'essayerons pas de savoir à quoi correspondent exactement les 15% perdus. Nous nous contenterons seulement de remarquer que même si 15% ne sont pas négligeables, les performances « générales » du compilateur ont été préservées.

2.6 Les classes larges

Comme nous l'avons expliqué dans § 2.5.1, le compilateur Bigloo utilise un arbre de syntaxe abstraite (AST) qui transite de passe en passe. Chaque passe modifie l'AST ; certains nœuds sont supprimés (par exemple par la passe qui *élimine le code mort*), d'autres sont ajoutés (par exemple par la passe d'optimisation qui *déroule les boucles*) mais la structure générale de l'AST change peu pendant toute la compilation.

2.6.1 Le problème : implantation du compilateur Bigloo

L'élaboration de l'architecture du compilateur s'est articulée autour de deux exigences principales : *1)* assurer l'indépendance de chaque passe et *2)* pouvoir changer arbitrairement l'ordre d'application des passes. La première est motivée par le fait de rendre plus facile le développement et la maintenance du compilateur. Si chaque passe est indépendante alors chaque passe peut être élaborée, puis testée séparément. La deuxième s'explique par le fait que Bigloo a été un laboratoire d'expérimentation. En particulier nous souhaitions pouvoir changer l'ordre d'application des passes afin de déterminer, par la pratique, le plus efficace (celui donnant les exécutables les plus rapides). Au cours de ces expériences il nous est apparu utile d'appliquer plusieurs fois une même passe d'optimisation à divers moments de la compilation. Par exemple, la *propagation de copies* permet d'améliorer les résultats de l'*élimination de sous-expressions communes* et cette deuxième optimisation crée des copies qui peuvent être factorisées par la première optimisation. Il est alors judicieux pour une compilation optimisante d'appliquer la *propagation de copies* suivie de l'*élimination de sous-expressions communes* puis de réappliquer la *propagation de copies*.

S'il convient principalement de juger un compilateur par la qualité du code qu'il produit, le temps de compilation et l'encombrement mémoire sont d'autres critères importants. Parce que Bigloo produit du code C, la traduction Scheme jusqu'au code objet est assez lente (ce point est discuté dans [204]). Toutefois, nous avons essayé de diminuer au maximum l'encombrement du compilateur. Pour cela, nous avons eu comme préoccupation de choisir une structure de donnée aussi compacte que possible pour représenter l'AST et implanter les traitements qui lui sont appliqués.

En plus des informations *globales* portées par l'AST, chaque passe a besoin d'informations spécifiques. Par exemple l'allocation de fermetures a besoin de connaître pour chaque fonction du programme l'ensemble des variables libres. Ces variables sont copiées dans des environnements associés aux fonctions ou bien elles sont ajoutées comme paramètres supplémentaires. Après l'allocation de fermetures il n'y a plus de fonctions qui utilisent des variables libres. Seule la passe d'allocation de fermetures a donc besoin d'être renseignée sur les variables libres. La représentation de l'AST doit alors satisfaire les contraintes suivantes :

- Permettre l'encodage d'informations éphémères tout en isolant les passes les unes par rapport aux autres.
- Être aussi compacte que possible afin de diminuer l'espace mémoire utilisé par le compilateur.

Nous avons recensé trois façons classiques de procéder pour traiter ce problème.

L'AST global. Dans cette stratégie l'AST est composé de nœuds qui ont autant de champs qu'il y a d'informations à enregistrer tout au long de la compilation. Si le compilateur contient n passes et que chacune de ces passes p utilise s_p champs alors les nœuds de l'arbre auront tous $\sum_{i=1}^{n} s_i$ champs.

Ce schéma est simple à mettre en œuvre mais il est inefficace parce qu'il consomme plus de mémoire que nécessaire. En effet, à un moment t, chaque nœud de l'arbre a de nombreux champs inutilisés (tous ceux qui ne sont pas utiles pour enregistrer les informations de la passe courante). De plus, avec un langage comme Scheme reposant sur un gestionnaire mémoire automatique, le programme doit *explicitement* détruire les champs au fur et à mesure qu'ils deviennent inutiles au risque de voir le GC être dans l'incapacité de récupérer les zones mémoires mortes. Cette nécessité est en contradiction avec la philosophie des GCs.

L'AST *global* ne permet pas de garantir la séparation entre les passes. En effet, les calculs locaux utilisés par des passes antérieures sont, à priori, toujours disponibles dans l'AST. Notre expérience nous a montré que la tentation de les utiliser est trop forte et que la séparation entre les passes est difficilement respectée. Il devient alors difficile de changer l'ordre d'application des passes ou d'en appliquer plusieurs fois certaines.

Enfin, l'AST *global* présente l'inconvénient de ne pas être extensible. Si une passe est ajoutée au compilateur, alors il faut changer la représentation globale de l'arbre et, probablement, recompiler tout le code existant.

Les copies en entrée. Lorsqu'on débute le calcul d'une nouvelle passe, c'est-à-dire, lorsqu'on « entre » dans une nouvelle passe, l'AST est entièrement recopié dans une structure plus large, où les champs sont assez grands pour recueillir les informations globales de l'arbre et celles spécifiques de la passe.

Ce procédé a l'avantage de préserver la nette séparation entre les passes et il maintient l'AST aussi compact que possible. À un moment t, seules les informations utiles au calcul courant sont contenues dans l'arbre. Cette solution présente toutefois l'inconvénient d'être inefficace en temps. Elle nécessite un grand nombre d'allocations et de désallocations. Il est donc probable qu'une application utilisant ce schéma « passera » du temps dans le GC. De plus, la copie de l'AST n'est pas simple à mettre en œuvre car elle doit respecter les propriétés de partage existantes dans les nœuds. Il est donc probable que la copie de l'arbre sera également une opération relativement lente.

Les champs utilisateur. Ici, chaque noeud de l'arbre possède un champ additionnel : un champ utilisateur. Chaque passe du compilateur l'utilise pour « accrocher » à l'arbre une structure locale contenant les informations utiles pour la passe courante. Cette solution est largement répandue parce qu'elle cumule plusieurs avantages. L'arbre n'est pas recopié, mais il est néanmoins aussi compact qu'en utilisant de la *copie en entrée*. Cet encodage est extensible car l'ajout d'une nouvelle passe ne requiert aucune modification de l'arbre global.

Les champs utilisateur présentent toutefois deux inconvénients :

– Ils ne permettent que difficilement de garantir une programmation sûre. En effet, il n'est pas possible de typer les champs utilisateur car, par construction, le compilateur ne sait pas, initialement à quoi ils seront utilisés. Le type des champs utilisateur doit donc être le type le plus général que propose le langage (par exemple, en C, on utilisera `void *`). Cette absence de contrôle par le système conduit à une erreur fréquente : une passe P_i utilise les champs utilisateur alors qu'ils contiennent encore les données de la passe P_{i-1}. Ceci déclenche alors, pour les langages qui en sont pourvus, une exception qui stoppe l'exécution.
– Les champs utilisateur ne sont pas très pratiques à utiliser car ils exigent un niveau d'indirection supplémentaire dans le code source. Il faut lire ce champ, puis de là, effectuer une seconde lecture pour trouver la valeur recherchée. Cet inconvénient devient plus important dans le cadre d'un langage objet. En général, pour ces langages, les programmes sont essentiellement constitués de méthodes qui surchargent des définitions globales. Cette méthodologie n'est pas applicable au niveau de l'AST même, car le type des nœuds ne changent pas si les champs utilisateur sont modifiés. Il n'est donc pas possible d'implanter simplement une méthode qui s'appliquerait uniquement si un champ utilisateur est de tel ou tel type.

Ces trois procédés sont indépendants du langage de programmation sous-jacent. On peut les mettre en application dans les langages objets ou dans des langages classiques mais il nous est apparu qu'aucune de ces constructions traditionnelles ne permettait d'implanter de façon satisfaisante notre compilateur. Comme Bigloo$_{\text{oo}}$ utilise des objets pour représenter son AST et des fonctions génériques pour l'implantation de ses passes, nous avons alors élaboré une nouvelle construction pour les langages objets qui permet de traiter économiquement et efficacement les données éphémères : les *classes larges* et les opérations d'*élargissement* et de *rétrécissement*.

On voit ici le double rôle joué par Bigloo. D'une part, c'est notre outil d'implantation et d'expérimentation où les constructions linguistiques y sont implantées puis testées. D'autre part, comme Bigloo est le plus gros logiciel que nous développons, de nombreuses extensions linguistiques sont motivées par son propre développement. Ainsi, les classes larges ont été conçues pour alléger son architecture logicielle. Par la suite, le compilateur est devenu la plate-forme d'expérimentation de leur implantation. C'est ici qu'on mesure l'intérêt de l'auto-génération du compilateur. De la même façon que les classes larges ont été instantanément mises en application, toute nouvelle optimisation peut être évaluée sur le programme « réel » que constitue le compilateur.

2.6.2 Les classes larges

Les *classes larges* et les opérations d'*élargissement* et de *rétrécissement* sont des extensions du modèle objet des langages à classes. L'élargissement permet à un objet (par la suite une *instance*) d'être temporairement *élargi*, c'est-à-dire, transformé en une instance d'une sous-classe de sa classe d'origine, une *classe large*, puis, ultérieurement d'être *rétréci* pour redevenir une instance directe de sa classe d'origine. Les classes larges partagent les propriétés principales des classes traditionnelles. Elles ont un nom, une super-classe, elles peuvent être instanciées, elles ont des prédicats qui permettent de tester l'appartenance des instances, et enfin, elles peuvent être utilisées pour surcharger des fonctions génériques. L'élargissement se distingue des constructions qui parfois existent dans d'autres langages (le `change-class` de CLOS [24] et le `become:` de Smalltalk [91]) parce que c'est une opération typée. En particulier, l'élargissement qui est une sorte d'héritage dynamique préserve les relations de sous typage. Cette propriété est importante car elle permet une implantation simple et efficace des classes larges et parce qu'elle permet également au compilateur de détecter des erreurs de type.

Une classe large se déclare presque comme une classe simple :

```
(wide-class class-id::super-class-id <field>*)
```

La super classe d'une classe large peut être une classe simple ou une autre classe large. En revanche, la super-classe d'une classe simple ne peut pas être une classe large. Cette restriction est expliquée dans § 9.3.6, page 171. À titre d'exemple, voici la déclaration d'une classe représentant des points nommés et des points en trois dimensions nommés :

```
(module named-point
   (export (wide-class named-point::point
              name::string)
           (wide-class named-point-3d::named-point
              (z::double (default 0.0)))))
```

2.6.3 Les instances larges et l'élargissement

Les *instances larges*, c'est-à-dire les instances des classes larges peuvent être utilisées dans tout contexte où un objet simple peut l'être. En particulier, les instances larges peuvent être créées par instanciation directe des classes larges et elles peuvent être utilisées pour choisir dynamiquement les méthodes à utiliser lors d'un appel à une fonction générique. Ainsi, on peut écrire :

```
(define-method (display-point::obj p::named-point)
   (with-access::named-point p (x y name)
      (print "<point(" name "): x:" x ", y:" y ">")))

(define (display-instance x)
   (if (point? x)
       (display-point x)
       (display x)))

(let ((p (instantiate::named-point
            (name "org")
            (x 0)
            (y 0))))
   (display-instance x))
```

L'élargissement et le rétrécissement sont les principales nouveautés. Elles permettent à une instance, d'être temporairement élargie en une instance d'une sous-classe large de sa classe courante puis, plus tard, d'être rétrécie vers sa classe d'origine. Par ailleurs, l'élargissement et le rétrécissement préservent les propriétés de partage. Autrement dit, l'élargissement et le rétrécissement n'introduisent ni recopies ni reconstructions des instances.

L'élargissement

La syntaxe de l'élargissement ressemble à la syntaxe de l'instanciation :

```
(widen!::class-id instance (field-id0 val0) (field-id1 val1) ...)
```

`Class-id` est le nom de la classe large vers laquelle l'instance est élargie. Les identificateurs *field-id*$_i$ sont les noms des champs non hérités de `class-id`. Les valeurs *val*$_i$ sont des valeurs d'initialisation pour les champs de la classe large. Voici un exemple complet d'instanciation et d'élargissement :

```
(let ((pt (instantiate::point (x 0.0) (y 5.3))))
   (widen!::named-point pt (name "dummy"))
   (display-point pt))
```

Instancier une classe large est équivalent à créer une instance simple puis à l'élargir autant de fois que nécessaire pour atteindre la classe large. Ainsi :

```
(instantiate::named-point-3d (x 0) (y 0) (z 0) (name "gee"))
```

est équivalent à :

```
(let ((p (instantiate::point (x 0) (y 0))))
   (widen!::named-point p (name "gee"))
   (widen!::named-point-3d p (z 0))
   p)
```

L'élargissement préserve l'identité des objets, autrement dit, en termes Scheme, si et seulement si deux instances sont `eq?` alors elles le seront toujours après l'élargissement. Ainsi, la valeur de :

```
(let* ((p     (instantiate::point))
       (new-p (widen!::named-point p (name "new"))))
   (eq? new-p p))
```

est « vraie ». Cette propriété est essentielle car c'est elle qui permet d'utiliser très simplement les classes larges dans notre compilateur. Puisque les classes larges préservent les propriétés de partage, il est possible, au cours de la compilation, d'élargir une portion de l'Ast sans modifier d'autres parties de l'arbre qui peuvent toujours pointer librement sur les parties qui sont élargies.

Le rétrécissement

Le rétrécissement est l'opération inverse de l'élargissement. Il permet à une instance large de redevenir une instance d'une classe simple. Sa syntaxe est :

```
(shrink! instance)
```

Ainsi, le code suivant :

```
(let ((p (intantiate::point)))
   (widen!::named-point p (name "dummy2"))
   (widen!::named-point-3d p (z 3.0))
   (display-point p)
   (shrink! p)
   (display-point p)
   (shrink! p)
   (display-point p))
```

une fois compilé, produira lors de son exécution :

```
<point(dummy2): x: 0.0 y: 0.0 z: 3.0>
<point(dummy2): x: 0.0 y: 0.0>
<point: x: 0.0 y: 0.0>
```

Un exemple complet d'utilisation des classes larges peut être trouvé dans § 9.3.5, page 169.

2.6.4 Les classes larges et le typage

Les classes larges supposent un typage dynamique, au moins partiellement. Il ne nous semble guère possible d'appliquer l'inférence de types de langages comme O'Caml [176] aux classes larges. Les algorithmes d'inférence de types basés sur l'algorithme W [58] ne peuvent gérer le fait qu'à deux endroits d'un programme une variable puisse avoir des types différents (même si le programme n'utilise aucune affectation). Ainsi, très probablement, des algorithmes d'inférence ne seront pas capable de détecter l'incorrection du programme :

```
(let ((p (instantiate::point)))
   (widen!::point-3d p (z 45))
   (+ (point-3d-z p)
      (begin
         (shrink! p)
         (point-3d-z p))))
```

Pour un algorithme d'inférence classique, `p` doit impérativement être un `point` ou un `point-3d` mais il ne peut être l'un ou l'autre en fonction de l'endroit où on se situe dans le code. C'est pourtant ce qui se passe avec l'élargissement et le rétrécissement. Ligne *1* la variable `p` est de type `point`, ligne *2*, `p` est élargie en un `point-3d` puis ligne *5* elle est rétrécie en un `point`. Ainsi, l'évaluation de l'expression ligne *6* produira une erreur lors de l'exécution.

S'il nous semble difficile d'intégrer les classes larges pour des langages comme O'Caml, en revanche, une telle intégration nous semble plus réaliste pour des langages comme Java qui disposent d'opérations de conversion de type vérifiées lors de l'exécution (les *downward casts*).

2.6.5 Les classes larges et le compilateur Bigloo

Les classes larges offrent une nouvelle possibilité pour représenter l'Ast de notre compilateur. Plutôt que d'utiliser une des trois solutions que nous avons présenté dans §2.6.1, nous avons opté pour l'introduction des classes larges dans Bigloo. Le schéma d'implantation d'une passe $\mathcal{P}$ est alors devenu :

1. Élargir les nœuds de l'Ast qui sont utilisés dans $\mathcal{P}$.
2. Procéder au calcul de $\mathcal{P}$ sur l'arbre ainsi élargi.
3. Rétrécir les nœuds une fois $\mathcal{P}$ terminée.

Ce schéma réduit la mémoire utilisée parce que l'arbre est aussi compact que possible. Il isole les passes parce que les informations temporaires ne sont plus accessibles dès que les passes sont terminées. Il est sûr parce que si une passe tente d'utiliser une instance dans un mauvais contexte, soit le compilateur pourra détecter l'erreur, soit lors de l'exécution une erreur de type sera clairement identifiée. Enfin, il permet une implantation simple des passes parce qu'une fois élargies, les instances sont utilisables comme tout objet. Il est alors possible d'utiliser directement les champs appartenant aux composantes « larges » des instances ou de surcharger les méthodes en utilisant les classes larges.

2.6.6 Conclusion sur les classes larges

Les classes larges ressemblent à d'autres mécanismes déjà proposés pour les langages objets. Toutefois, le « cahier des charges » qui est à la base de leur conception leur a donné des caractéristiques singulières. Clos et Smalltalk proposent des mécanismes pour changer dynamiquement le type des objets (`change-class` et `become:`) mais ces deux opérations ne sont pas contraintes (tout objet peut être transformé arbitrairement en une instance de n'importe qu'elle autre classe). Ainsi `change-class` et `become:` sont beaucoup plus difficiles à implanter que les classes larges car ces deux constructions ont un impact sur tout le modèle d'exécution et la représentation des objets. Les classes larges acceptent une implantation très simple (cf. § 9.5). D'autres extensions du modèle objet à classes qui permettent aux objets de changer dynamiquement de comportement ont déjà été étudiées. Par exemple, le langage Cecil [39] offre des *predicate classes* [40]. Un objet appartient à une telle classe s'il vérifie dynamiquement un certain prédicat. Les objets peuvent donc appartenir temporairement à une classe ou plusieurs classes. À la différence des classes larges, les *predicate classes* ne permettent pas d'ajouter simplement de nouveaux champs aux objets. En effet, ces dernières ne constituent pas un moyen de changer la représentation mémoire des objets. On peut faire la même remarque à propos de la programmation objet à base de *modes* [238]. Enfin, les travaux qui semblent les plus proches des classes larges sont les *champs dynamiques* [196]. Un objet peut dynamiquement être étendu par des champs dynamiques. Contrairement aux classes larges, la conception des champs dynamiques présuppose que les instances soient initialement construites de telle façon qu'elles puissent accueillir les champs dynamiques. C'est-à-dire que lors de l'instanciation un espace mémoire est réservé aux champs dynamiques. Par ailleurs, une fois qu'une instance a été étendue, il n'est plus possible de la ramener à son état initial. Cette différence majeure avec les classes larges rend les champs dynamiques inapplicables pour une application telle que Bigloo où il est fondamental qu'en sortie de toutes les passes, les objets formant l'Ast soient rétrécis.

	Qualités				
Techniques	Extensible	Économe	Efficace	Sûr	Remarques
AST **global**	Non	Non	Oui	Oui	Doublement gourmand à moins de faire du GC à la main. Confusion entre les passes.
Copies en entrée	Oui	Oui	Non	Oui	Coûteux en temps de reconstruction de l'arbre et en temps de GC. Complexe à implanter pour respecter les propriétés de partage entre les nœuds.
Champs utilisateur	Oui	Oui	Oui	Non	Code plus fastidieux car il faut passer par des indirections pour accéder aux informations locales.
Classes larges	Oui	Oui	Oui	Oui	Même sobriété et efficacité que les *champs utilisateur* mais sans le problème de sûreté et plus agréable à manipuler.

FIG. 2.1 – Récapitulatif : les avantages des classes larges

La figure 2.1 synthétise les avantages des classes larges par rapport aux solutions déployées dans les langages traditionnels. Pour un faible coût d'implantation elles nous ont permis de réécrire le compilateur de façon plus compacte, plus sûre et en utilisant moins de mécanismes fondamentaux. Maintenant toutes les passes du compilateur peuvent réellement être implantées par le biais de fonctions génériques et de méthodes puisque les classes larges peuvent être utilisées pour le *dispatch* dynamique. Elles sont implantées depuis la version 1.9 de Bigloo, c'est-à-dire la version diffusée en 1997.

2.7 La bibliothèque graphique Biglook

Biglook est une bibliothèque d'objets graphiques (nommés par la suite *widgets*) qui permet la réalisation d'interfaces graphiques (*GUIs*) complexes. Biglook se distingue par son interface de programmation (*API*) originale qui utilise intensivement les modules, les objets et les champs virtuels de Bigloo. Biglook est apparue avec la version 2.0 de Bigloo en 1998. Nous présentons cette librairie dans ce mémoire parce qu'elle constitue un exemple *grandeur nature* d'application de nos extensions, parce qu'elle démontre notre volonté de faire de Bigloo un système réaliste, permettant la programmation d'applications modernes (c'est-à-dire des applications possédant *également* des interfaces graphiques) et enfin, parce que nous utilisons cette bibliothèque abondamment pour la construction de l'environnement de programmation Bee présenté dans le chapitre 4.

2.7.1 L'*API* de Biglook

L'*API* de Biglook est originale parce qu'elle repose uniquement sur les constructions de Bigloo, ses modules et ses classes. Elle permet une programmation déclarative des interfaces et une nette séparation entre les parties qui implantent les interfaces et les parties qui implantent les algorithmes du programme. Avec Biglook, on construit des *widgets* par « instanciation » des classes de la bibliothèque puis on les place à l'écran. Toutes les informations relatives aux *widgets* sont contenues dans les champs des classes les décrivant. Voici alors un premier exemple d'un programme complet créant deux boutons :

```
(module biglook-button-example
   (library biglook))
```

```
(instantiate::button
   (text "This is the first button")
   (foreground "gray")
   (font "lucida-12")
   (background "red"))

(instantiate::button
   (text "This is a second button")
   (background "blue")
   (balloon "A tip message"))
```

La majorité des *widgets* ont des configurations facultatives. Par exemple, le premier de nos deux boutons utilise une police de caractères spécifique alors que le second utilise la police par défaut et seul le second bouton est agrémenté d'une bulle d'aide (le champ `balloon`).

Biglook utilise un modèle de *callback* pour spécifier les actions à exécuter lorsque l'interface est utilisée. Les *widgets* réactifs ont un champ `command`, qui spécifie une fonction Scheme à exécuter lorsque l'objet graphique est manipulé par l'utilisateur de l'interface :

```
(let ((num 0))
   (instantiate::button
      (text "This is another button")
      (command (lambda ()
                  (print "Click num: " num)
                  (set! num (+ 1 num))))))
```

On voit sur cet exemple que les *callbacks* peuvent être n'importe quelle fonction Scheme et en particulier des fermetures arbitraires.

Le placement

Une fois les objets créés, il reste à les disposer à l'écran. Pour cela Biglook propose plusieurs gestionnaires de placement. Chacun suit une politique différente faite de contraintes graphiques. En général, on ne place pas les *widgets* Biglook à des positions absolues à l'écran mais on ordonne les composants les uns par rapport aux autres. Cette démarche permet d'avoir des interfaces que l'utilisateur peut retailler dynamiquement. Le principal gestionnaire est implanté par la fonction `pack`. En plus des objets à afficher, on peut fournir quelques options à cette fonction pour affiner le placement (par exemple, la distance aux bords des fenêtres, les espaces que doivent occuper les composants, l'expansion ou le rétrécissement des objets lors d'un retaillage, etc.). Pour l'exemple suivant, on construit une interface faite de lignes horizontales contenant un message et un bouton.

```
 1 :  (define (make-row num)
 2 :     (let* ((msg (string-append "Select: " (number->string num)))
 3 :            (fr  (instantiate::frame))
 4 :            (lbl (instantiate::label
 5 :                    (parent fr)
 6 :                    (text msg)))
 7 :            (but (instantiate::check-button
 8 :                    (parent fr))))
 9 :        (pack lbl but :expand #t :side 'left)
10 :        fr))
```

Pour construire nos lignes horizontales trois *widgets* doivent être instanciés : un conteneur (ligne *3*), un bouton (ligne *7*) et un texte (ligne *4*). Les conteneurs permettent le placement respectif des textes et des boutons. C'est-à-dire qu'on spécifie que les textes et les boutons doivent être inscrits à l'*intérieur* des conteneurs (ligne *5* et *8*). Ensuite, il ne reste plus qu'à indiquer la disposition graphique des textes et boutons dans leur conteneur (ligne *9*). Dans notre exemple, le texte est placé à l'extrême gauche et le bouton à l'extrême droite. Cette contrainte sera toujours vérifiée même si l'interface est retaillée dynamiquement.

```
11 :  (define (main argv)
12 :     (let ((num (string->integer (cadr argv))))
13 :        (let loop ((num num))
14 :           (if (>fx num 0)
15 :               (let ((row (make-row num)))
16 :                  (pack row :expand #t)
17 :                  (loop (-fx num 1)))))))
```

Le programme principal pour notre application construit autant de lignes qu'il le désire (ligne *15*), qu'il place alors à l'écran (ligne *16*) en spécifiant que si l'interface graphique est retaillée alors les lignes devront occuper tout le nouvel espace ainsi créé (`:expand #t`). Comme nous le montrons dans le chapitre 11, nous avons opté pour une architecture logicielle qui nous a permis de ne pas implanter les fonctionnalités de bas niveau nécessaires à Biglook. Ainsi, dans sa version actuelle, Biglook repose sur la bibliothèque primitive Tk. Le mérite de l'implantation des solveurs de contraintes graphiques revient uniquement à J. Ousterhout, l'auteur de cette librairie [155]. Même si Biglook utilise Tk, Biglook se distingue de Tk par une interface de programmation totalement différente et par un ensemble de *widgets* différent.

Les événements

Biglook repose sur le principe déclaratif de *callback*. Lors de l'instanciation des *widgets*, par le biais de leur champ `command`, on peut spécifier des actions à exécuter lorsqu'ils sont manipulés dans l'interface. Il existe une trop grande quantité d'événements auxquels les *widgets* peuvent réagir (button-1, button-2, shift-button-1, ctrl-shift-button-1, ...) pour que Biglook puisse proposer un champ pour chacun. En remplacement, la bibliothèque contient la forme déclarative `event-case` qui permet d'ajouter *à posteriori* des actions à des *widgets*. Par exemple, si on souhaite ajouter une action lorsque le pointeur de la souris entre ou sort d'un *widget*, il suffira d'écrire :

```
(event-case w
   (("<Enter>")
    (print "Entering [" (the-event-x) ", " (the-event-y) "]"))
   (("<Leave>")
    (print "Leaving " (the-event-widget))))
```

Cette forme « installe » des *callbacks* sur l'objet `w` pour deux types d'événements graphiques. Il s'agit donc d'une déclaration. Les chaînes de caractères `<Enter>` et `<Leave>` sont des descripteurs d'événements. Tous ces descripteurs peuvent être combinés les uns avec les autres pour décrire des événements composés. Les fonctions Biglook `the-event-x`, `the-event-widget`, ... permettent d'obtenir des informations sur les objets actifs.

2.7.2 Quelques exemples d'applications

Dans cette section, nous illustrons les capacités de Biglook en présentant trois applications et leur implantation complète. Nous avons choisi trois exemples permettant d'illustrer la qualité principale de Biglook : le code consacré aux interfaces est compact.

```
(module hierarchy-class
   (library biglook))

(define (get-class-children data)
   (map (lambda (x)
           (if (null? (class-subclasses x)) x (list x)))
        (class-subclasses data)))

(define t
   (instantiate::hierarchy-tree
      (balloon '("A balloon" "for the example"))
      (get-node-children get-class-children)
      (root object)
      (get-node-label class-name)
      (node-hook (lambda (data text icon)
                    (event-case text
                       (("<Button-3>")
                        (print (class-subclasses data)))
                       (("<Button-2>")
                        (print (class-name data)))))))))

(pack t :expand #t :fill 'both)
```

FIG. 2.3 – Le navigateur de classes

Un navigateur de classes

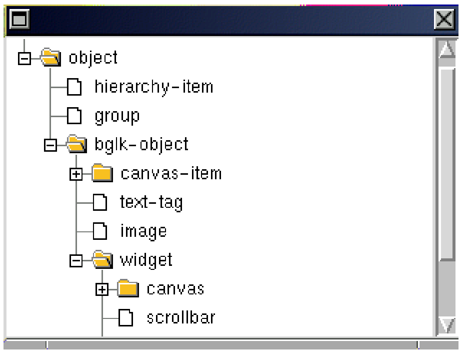


FIG. 2.2: Un navigateur de classes

Le premier exemple implante un navigateur de classes Bigloo qui repose sur les mécanismes d'introspection du langage et sur la classe `hierarchy-tree` pour l'affichage graphique des structures arborescentes. Le premier paramètre de configuration de ces objets graphiques est `root`. Il indique l'objet qui est la « racine » de l'arbre à afficher. La fonction affectée au champ `get-node-children` indique comment accéder aux fils d'un nœud de l'arbre. La fonction `get-node-label` permet elle d'associer un label aux nœuds affichés. La classe `hierarchy-tree` prend en charge l'« ouverture » et la « fermeture » des nœuds de l'arbre lors de clic souris sur les icônes présentes. Pour que les objets graphiques soient réactifs, il faut pouvoir leur associer des *callbacks*. La difficulté provient du fait que c'est la hiérarchie elle-même qui ajoute ou supprime dynamiquement des objets lors des pliages et dépliages de nœuds. Pour permettre au programme de « poser » des *callbacks*, si le champ `node-hook` de la hiérarchie instanciée est une fonction, alors cette fonction est appelée par la hiérarchie dès qu'un nouvel objet graphique est dessiné à l'écran. Cette fonction prend trois paramètres : l'objet Scheme pour lequel des *widgets* sont dessinés à l'écran, le texte et l'icône Biglook qui seront dessinés pour représenter graphiquement l'objet. La figure 2.3 contient le code complet de cette application.

```
(module hierarchy-file
   (library biglook))

(define (get-file-children data)
   (map (lambda (x)
           (let ((f (string-append data "/" x)))
              (if (directory? f) (list f) f)))
        (directory->list data)))

(define t
   (instantiate::hierarchy-tree
      (get-node-children get-file-children)
      (balloon '("" "" "Edit"))
      (root ".")
      (node-hook (lambda (data text icon)
                    (event-case text
                       (("<Button-3>") (system "xedit" text "&")))))))

(pack t :expand #t :fill "both")
```

FIG. 2.5 – Le sélecteur de fichiers

Un sélecteur de fichiers

Il faut noter que l'interface des hiérarchies de Biglook permet de dessiner des arbres sans allouer de structures de données isomorphes à celles représentées. Ce sont directement les structures de données déjà existantes qui servent à la construction des arbres. Ainsi, pour utiliser les hiérarchies dans un autre contexte, il suffit de changer la valeur de la racine de l'arbre et la façon de calculer les fils d'un nœud. Pour représenter un arbre de classes nous avions placé les classes elles-mêmes dans l'arbre et utilisé la fonction `class-subclasses` qui retourne la liste des sous-classes d'une classe mère et nous avions utilisé `object`, la mère de toutes les classes comme racine. Pour réaliser un sélecteur de fichiers, il suffit d'utiliser une fonction qui lit sur le disque les fichiers d'un répertoire (`get-file-children`) et de choisir `"."` comme racine de notre arbre. La figure 2.5 contient le code de cette application.

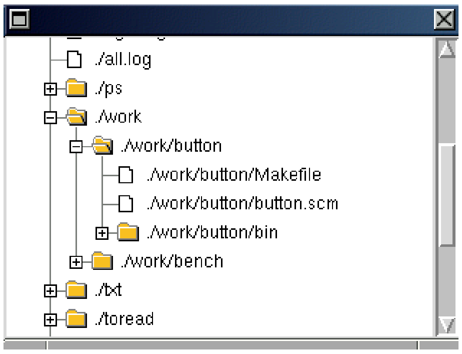


FIG. 2.4: Un sélecteur de fichiers

Une animation graphique

La dernière application que nous présentons dans cette section permet d'illustrer l'emploi de la récursivité pour réaliser une animation graphique simple. Dès que l'utilisateur presse sur le bouton `Start new runner` un petit dessin représentant un sprinter apparaît à l'écran. Sur la copie d'écran quatre sprinters différents sont dessinés dans leur course. Chacun de ces personnages est indépendant et, en particulier, chacun est animé d'une vitesse personnelle. La course de chaque personnage est décomposée en 6 étapes graphiques qui, affichées les unes après les autres, donnent l'illusion du mouvement. Cette application illustre également la possibilité, depuis le code utilisateur, de définir des sous-classes des classes standard de la bibliothèque de Biglook. Ainsi, les sprinters seront représentés par des instances de la classe `runner` qui est construite à partir de la classe standard `image-item`. C'est cette dernière qui permet l'insertion d'images dans les interfaces. Une instance de `runner` est donc un objet ayant

FIG. 2.6: Une animation

une image, un nombre, qui représente le dossard du sprinter, une vitesse et une liste d'images qui décompose son pas. Cette classe est définie par :

```
 1 :  (module biglook-demo-runner
 2 :     (library biglook)
 3 :     (static  (class runner::image-item
 4 :                 (number (default (gensym 'runner)))
 5 :                 (images (default (get-image-list)))
 6 :                 (speed  (default (random))))))
```

Les images des pas des sprinters sont implantées sous forme de listes Scheme contenant les `images` des pas :

```
 7 :  (define *runner-images*
 8 :     `#(,(instantiate::image (file "runner1.xpm"))
 9 :        ,(instantiate::image (file "runner2.xpm"))
10 :        ,(instantiate::image (file "runner3.xpm"))
11 :        ,(instantiate::image (file "runner4.xpm"))
12 :        ,(instantiate::image (file "runner5.xpm"))
13 :        ,(instantiate::image (file "runner6.xpm"))))

14 :  (define (get-image-list)
15 :     (let ((l (vector->list *runner-images*)))
16 :        (set-cdr! (last-pair l) l)
17 :        l))
```

Le terrain qui constitue le parcours des `runners` est représenté par un `canvas` Biglook. Ce *widget* permet la définition de zones de dessin à l'intérieur d'une interface complète :

```
18 :  (define *ground*
19 :     (instantiate::canvas
20 :        (height 30)
21 :        (width  150)
22 :        (background "white")))
```

La ligne qui matérialise le sol sur lequel les sprinters courent est alors une simple ligne insérée dans le canvas `*ground*` :

```
23 :  (instantiate::line
24 :     (parent *ground*)
25 :     (fill "red")
26 :     (init-coords (list 10 20 (- (canvas-width *ground*) 10) 20)))
```

Le bouton qui permet de « lancer » un nouveau sprinter est :

```
27 :  (define *start*
28 :     (instantiate::button
29 :        (text    "Start new runner")
30 :        (command (let ((num 1))
31 :                    (lambda ()
32 :                       (let ((n (instantiate::runner
33 :                                   (init-coords '(10 15))
34 :                                   (parent *ground*))))
35 :                          ((make-runner-run n))))))))
```

La fonction exécutée à chaque fois que l'utilisateur presse sur ce bouton crée un nouveau sprinter, construit une fermeture (ligne *35*) et l'applique. Les champs `number`, `images` et `speed` de la classe `runner` ont des valeurs par défaut (ligne *4*, *5* et *6*). En particulier, on peut constater que la vitesse de chaque sprinter est déterminée aléatoirement à chaque fois qu'un nouveau sprinter est créé.

```
36 : (define (make-runner-run r)
37 :    (define (run)
38 :       (with-access::runner r (coords image images speed)
39 :          (match-case coords
40 :             ((?x ?y)
41 :              (if (< x (- (canvas-width *ground*) 13))
42 :                  (with-access::runner r (image images speed)
43 :                     (set! coords (list (+ 1 x) y))
44 :                     (set! image (car images))
45 :                     (set! images (cdr images))
46 :                     (after speed run))
47 :                  (destroy-canvas-item! r)))
48 :             (else
49 :              (error "run" "runner lost" coords)))))
50 :    run)
```

Cette fonction sélectionne la prochaine image représentant le pas du sprinter, calcule la nouvelle position du sprinter et l'affiche à l'écran. Pour implanter la vitesse du sprinter, nous utilisons la fonction `after` qui applique une fonction après un certain délai. Ainsi, pour chaque sprinter, il suffit d'attendre un temps calculé en fonction de sa vitesse puis d'appeler récursivement `run` (ligne *46*) qui affichera la prochaine image du pas du sprinter à sa nouvelle position. Le programme principal de cette application consiste seulement à placer à l'écran le terrain puis le bouton permettant de lancer de nouveaux sprinters.

```
51 : (pack *ground*)
52 : (pack *start* :expand #t :fill 'x)
```

Les applications que nous présentons dans cette section sont suffisamment simples pour que nous puissions donner leurs implantations complètes. Biglook possède les fonctionnalités suffisantes pour construire des interfaces plus réalistes. Certaines de ces interfaces sont exposées dans le chapitre 4 qui contient la présentation de l'environnement de programmation de Bigloo puisque tous les outils graphiques de Bee sont écrits intégralement en Bigloo(k).

2.7.3 À propos de la programmation des *GUIs*

En implantant Biglook et en réalisant des interfaces graphiques, il nous est apparu que trois constructions de Bigloo influencent plus que toutes les autres le style des applications graphiques. Nous les présentons dans cette section.

Fonctions d'arité variable et arguments nommés

Sans fonctions acceptant un nombre variable d'arguments *nommés* il ne serait pas possible d'adopter un style déclaratif pour la programmation des interfaces. Ce trait est indispensable pour la création des *widgets*. Par exemple, un bouton Biglook est caractérisé par une classe qui ne compte pas moins de 24 champs ! Comme nous l'avons vu, certains définissent la représentation graphique des boutons et d'autres décrivent leur état interne. Presque tous ces champs ont des valeurs par défaut acceptables pour l'utilisateur. Par exemple, à moins de le spécifier, les textes des boutons sont visualisés dans une police de caractères standard. Ainsi, la forme permettant la création des boutons doit accepter un nombre variable de paramètres, une police pouvant ou non être passée en argument lors de l'instanciation. L'utilisateur doit pouvoir fournir certaines valeurs et utiliser les défauts pour d'autres. Comme il y a beaucoup de paramètres qui peuvent être potentiellement passés lors de la création d'un *widget*, il est utile de pouvoir les nommer et de les fournir dans un ordre quelconque. Comme nous l'avons présenté dans § 2.5.5 et § 2.7.1 la forme qui permet d'instancier des *widgets* (la forme `instantiate::`) offre ces deux possibilités. De plus, comme cette forme est implantée par le biais d'une macro Scheme, le surcoût associé à la recherche et au tri des paramètres fournis disparaît intégralement à la compilation.

Pour un langage ne disposant pas simultanément de fonctions d'arité variable et de paramètres nommés, les *widgets* doivent être créés puis, dans un second temps, configurés. Même la surcharge et les constructeurs de classes tels que pratiqués en C++ et en Java ne constituent pas une alternative aux fonctions d'arité variable. Imaginons l'implantation de nos boutons en Java. Pour permettre un style purement déclaratif, il faudrait, comme le fait Bigloo, offrir aux utilisateurs la possibilité de créer des boutons en fournissant n'importe quelle valeur pour les champs de cette classe. C'est-à-dire qu'il faudrait fournir pour les boutons autant de constructeurs qu'il y a de combinaisons possibles de paramètres d'initialisation. Comme les boutons ont 24 champs, il faudrait donc 2^{24} constructeurs. Bien sûr cela n'est pas réaliste. En plus, même si nos classes étaient plus petites, la solution adoptée en C++ et Java ne serait de toute façon pas applicable parce que le moyen utilisé dans ces langages pour résoudre la surcharge est le typage. Plusieurs champs de la classe des boutons ont le même type (par exemple, la hauteur et la largeur d'un bouton sont deux valeurs numériques entières). Le typage ne permet pas alors de les distinguer. Il n'y a donc pas de moyen dans ces langages de spécifier *précisément* quelles sont les valeurs fournies lors de la création des widgets.

Les fermetures pour les *callbacks*

Biglook, à l'instar des autres bibliothèques graphiques modernes, utilise des *callbacks*. L'utilisateur associe des commandes à des événements graphiques (un événement de la souris, du clavier, ...). Lorsqu'un événement se produit une commande est déclenchée. Il nous est apparu que les fermetures constituent le moyen le plus simple pour représenter ces commandes. On peut classer les langages qui ne proposent pas de fermetures dans deux catégories :

Les langages avec fonctions sans environnement. C'est le cas de langages comme C [115] qui n'offrent que des fonctions globales. Une fonction C ne peut accéder qu'à ses paramètres et aux variables globales du programme. Contrairement aux fonctions Scheme et ML une fonction C n'est pas définie dans un environnement lexical qui lui est propre. En C, les *callbacks* sont modélisées par des fonctions mais sans environnement elles sont forcément très limitées. Pourtant, en général, les *callbacks* ont besoin de connaître le *widget* auquel elles sont associées. Pour pallier cette insuffisance de C, GTK+ [158], une bibliothèque pour ce langage, permet d'associer aux événements graphiques un couple composé d'une fonction f et d'une structure de données e. Quand l'événement se produit, f est appelée en lui passant en paramètre e. GTK+ simule les fermetures des LFs avec ce paramètre supplémentaire qui est en fait un environnement. Il faut noter qu'avec cette pratique il incombe aux utilisateurs d'allouer et de gérer leurs environnements alors qu'avec les fermetures des LFs cela est géré automatiquement par le système.

Les langages à classes. Pour les langages n'offrant pas de fermetures mais proposant des classes, une autre stratégie peut être utilisée. Les *callbacks* peuvent être réalisées au moyen de fonctions membre. Ces fonctions ont accès aux attributs des objets qui les portent. Si sous cet éclairage les fonctions membre ressemblent aux fermetures elles ne sont pas aussi générales parce qu'elles sont associées aux *classes* et non pas aux *instances*. Toutes les instances directes d'une même classe partagent exactement les mêmes fonctions membre. Cette restriction empêche de les utiliser pour les *callbacks*. Imaginons l'implantation de deux boutons, le premier réagissant à la souris en imprimant un message et le second en émettant un son. Deux classes doivent être définies pour permettre deux surcharges de fonctions membre ! S'il faut définir une classe par *callback*, il est apparent que si les *widgets* peuvent réagir simultanément à plusieurs événements il y a un risque d'explosion combinatoire. Il ne sera pas possible de définir une classe pour les boutons réagissant aux événements bouton-1, une autre pour bouton-2, une troisième pour shift-bouton-1, une quatrième pour ctrl-shift-bouton-1, etc.

Pour éviter ces déclarations de classes, la seconde version de Java a introduit la notion de classes internes. Une classe interne est définie à l'intérieur d'une classe mère et elle peut être anonyme. Les classes internes sont alors utilisées pour modéliser les *callbacks*. Les *widgets* ont maintenant, comme avec Biglook, un attribut qui contient un objet décrivant une action à exécuter si un événement graphique se produit. Cet attribut est d'une classe générale (l'interface `ActionListener` dans AWT [96]). Le principe consiste alors à définir des sous-classes de cette classe générale à chaque fois qu'on souhaite implanter une nouvelle commande. Comme cette opération est très fréquente, Java a introduit une nouvelle syntaxe qui permet une déclaration simplifiée des classes internes. Ainsi, il est possible d'écrire :

```
button.addActionListener(new ActionListener () {
   public void actionPerformed(ActionEvent e) {
      a user code that may reference current lexical bindings
   }
})
```

L'expression `new ActionListener` nécessite quelques explications : *1)* elle déclare une classe interne anonyme *2)* qui implante l'interface `ActionListener` et *3)* elle crée une instance unique de cette nouvelle classe qui est passée en paramètre à `addActionListener`, fonction membre de la classe `Button`. En fait, cet usage des classes internes de Java correspond *exactement* aux fermetures des LFs. Toutefois, il nous apparaît évident que la syntaxe de ces dernières est plus harmonieuse et concise que celle de Java.

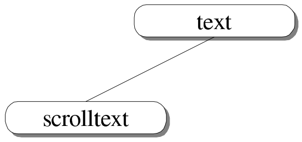
Les fermetures jouent un rôle central dans la programmation des interfaces graphiques parce qu'elles représentent le mécanisme le plus simple pour l'implantation des *callbacks*. Comme nous l'avons sous-entendu, toutes les bibliothèques qui utilisent des *callbacks* proposent des constructions qui permettent d'utiliser ou de simuler les fermetures. Toutefois, lorsqu'une simulation est nécessaire, elle est généralement moins agréable à employer pour l'utilisateur que la forme `lambda` de Scheme parce qu'elle nécessite sa collaboration (comme pour GTK+) ou bien l'ajout d'une nouvelle syntaxe qui ne s'intègre pas forcément très facilement dans le langage (comme pour Java).

Les fonctions génériques

Une fonction générique est un regroupement de méthodes. Pour les langages qui les emploient, le comportement des objets n'est plus spécifié dans les classes mais dans les fonctions génériques. Il n'est plus alors utile de prévoir, dès la conception d'une classe, tous les comportements des instances. Les fonctions génériques permettront d'associer des comportements aux instances *après* la définition des classes. Ainsi, les fonctions génériques permettent une extensibilité impossible à envisager pour les autres modèles objet.

Étendre les *widgets* publiques. Étudions les possibilités qui se présentent à un utilisateur pour étendre les *widgets* standard d'une bibliothèque dans les divers modèles objet. Nous présentons trois solutions : *1)* la première basée sur les fonctions génériques, *2)* la seconde utilisant l'héritage multiple de certains langages et enfin, *3)* la dernière, en n'utilisant que de l'héritage simple.

Biglook dispose de plusieurs *widgets* de textes. Parmi eux, on compte les textes simples, représentés par la classe `text` et les textes avec une barre de déplacement représentés par la classe `scrolltext` qui est une sous-classe de `text`.

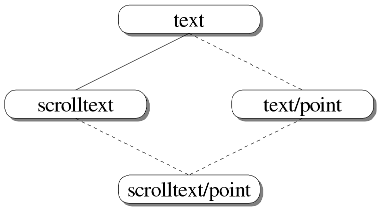


Chaque texte a un curseur dont la position est définie par des coordonnées en X et en Y. Imaginons maintenant une application pour laquelle il serait plus simple de représenter la position du curseur comme dans Emacs, c'est-à-dire par un nombre unique qui est le numéro du caractère sous le curseur. Passer de la représentation $X \times Y$ à la représentation linéaire et réciproquement est algorithmiquement très simple. Avec des fonctions génériques il suffit d'écrire :

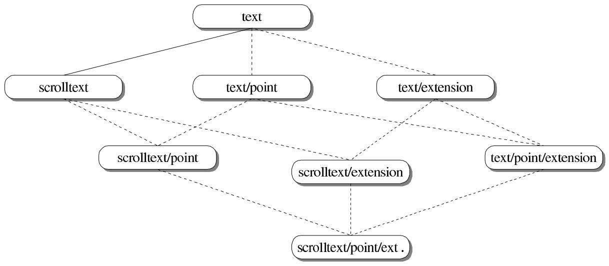
```
(define-generic (point::long t::text)
   (with-access::text t (cursor-x cursor-y line-width)
      ;; this is an over simplified implementation, because in
      ;; practice not all lines are supposed of same width
       (+ cursor-x (* line-width cursor-y))))
```

Cette définition est valide pour la classe `text` et toutes ses sous-classes, en particulier pour la classe `scrolltext`. Sans fonctions génériques, comment parvenir au même résultat ?

– Si comme C++ ou O'Caml le langage permet l'héritage multiple, un utilisateur pourra définir une nouvelle sous-classe de `text`, par exemple la classe `text/point`, qui étend la classe mère avec une nouvelle fonction membre `point`. Il sera alors ensuite possible de définir les `scrolltext/point` en dérivant simultanément de `scrolltext` et de `text/point` :

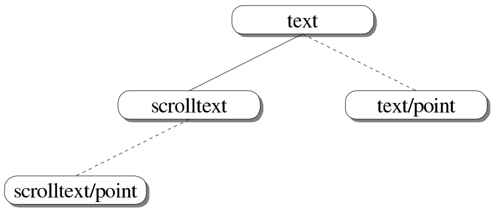


Si cette solution permet l'extension voulue, il faut noter que deux classes doivent être introduites pour *chaque* extension. Si nous voulions étendre une nouvelle fois les textes il nous faudrait alors définir quatre nouvelles classes. Nous serions alors amener à introduire le nouveau graphe d'héritage :



On voit clairement que de tels graphes d'héritage peuvent rapidement devenir impraticables !

– Si le langage ne dispose que d'héritage simple ou, comme Java, que d'une forme restreinte d'héritage multiple, notre extension serait à peine réalisable. En Java, parce que `text` et `scrolltext` doivent être de vraies classes (par opposition aux interfaces), il n'est pas possible de construire des classes qui en héritent simultanément. Ainsi, il faut avoir recours à l'arbre d'héritage suivant :



Avec cette solution, les classes `text/point` et `scrolltext/point` ne sont plus liées par des relations de sous typage, ce qui signifie qu'il n'y a pas moyen de fournir une instance de `scrolltext/point` lorsque une fonction attend une instance de `text/point`. Clairement cette solution n'offre pas la même puissance que les fonctions génériques.

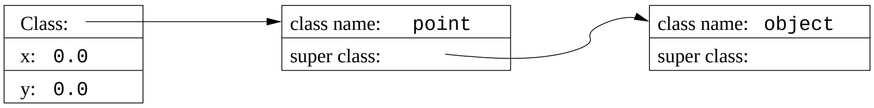
Étendre les *widgets* embarqués. La supériorité des fonctions génériques est encore plus évidente lorsqu'on souhaite implanter de nouveaux services pour les *widgets* embarqués. Par exemple, Biglook fournit un *widget* composite qui permet de créer des fenêtres de texte *à la* Emacs. Ces objets sont formés de plusieurs *widgets* primitifs (une barre de tâches, une entrée pour le *minibuffer*, des labels pour la *modeline*, un texte avec barre de déplacement pour la zone d'édition). Pour accéder à la position du curseur de la zone d'édition, une application doit, dans un premier temps, lire le `scrolltext` qui est embarqué dans l'instance de la classe `emacs`, puis, dans un second temps, extraire la position du curseur. La classe `emacs` ne permet pas aux utilisateurs de changer la classe de la zone de texte. La création d'une instance de `emacs` crée automatiquement une instance de `scrolltext`. Comme cette classe ne peut pas être remplacée par une classe utilisateur, sans fonction générique, il n'est pas possible d'implanter l'adressage linéaire pour les instances d'`emacs`.

2.8 Les champs virtuels

Pour un lecteur familier des langages à objets de la veine d'Eiffel [146] ou C++, notre usage des objets peut sembler paradoxal. Les objets, et plus particulièrement les classes, ne constituent pas pour nous une construction de « génie logiciel ». Les classes ne servent pas à délimiter et à isoler les différentes composantes d'un programme. Pour nous, ce rôle est tenu par les modules. Ainsi, il n'existe pas dans Bigloo le moyen de déclarer que des champs sont privés dans l'acceptation C++ du terme. Ceci s'explique par le fait que le *comportement* n'est pas défini dans les classes mais dans les fonctions génériques. La structuration en terme de disponibilité et visibilité se fait donc au niveau des fonctions génériques et des modules. Dans Bigloo, une classe est en fait un modèle mémoire et elle est associée à un et un seul type. Ainsi, la déclaration :

```
(class point
   (x::double (default 0.0))
   (y::double (default 0.0)))
```

est principalement la définition d'une « carte mémoire » qui est :



Comme nous l'avons signalé dans §2.5.2, le compilateur définit automatiquement des fonctions d'accès (lecteurs et écrivains) pour tous les champs (hérités ou définis) d'une classe. En plus d'être un moyen de déclarer des cartes mémoire, la programmation classique utilisant des classes possède plusieurs avantages :

- Bigloo associe un nouveau type à chaque classe déclarée. Ces types ainsi construits peuvent être utilisés arbitrairement par le programme. Les classes sont donc un moyen « officiel » pour déclarer de nouveaux types agrégés dans Bigloo.
- La couche objet présente toute une panoplie d'aide à la programmation. Plusieurs syntaxes permettent de créer simplement de nouvelles instances (`instantiate::` et `duplicate::`), d'autres permettent d'accéder aux données contenues dans les objets (`with-access::`).
- Les types associés aux classes peuvent être utilisés pour définir des surcharges au cas par cas au moyen de méthodes.
- Les classes peuvent être construites par héritage (par extension) ce qui facilite l'écriture des programmes en permettant de supprimer des répétitions de code. De plus, les classes définies par extension introduisent des « sous-types » des types de leur classe mère. Cela permet souvent une programmation facilement extensible.

Ainsi, même en renonçant aux fonctionnalités qu'on pourrait qualifier de purement objet (liaison tardive, héritage et sous typage) les classes ont un intérêt patent car elles peuvent jouer le rôle de « super-enregistrements ».

2.8.1 Les champs virtuels

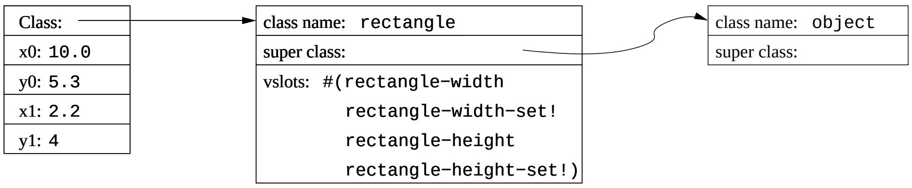
Les champs virtuels ont les trois caractéristiques suivantes :
- On accède à un champ virtuel comme à un champ traditionnel.

– Les champs virtuels ne sont pas alloués en mémoire. Autrement dit, ils ne sont pas associés à une zone mémoire.
– Les fonctions d'accès aux champs virtuels ne sont pas définies par le compilateur. C'est le rôle du programmeur de les définir.
– Les champs virtuels sont déclarés au moyen des nouvelles options de déclaration de champ de classe : `get` et `set`.

Dans Biglook, nous utilisons les champs virtuels pour donner une interface « objet » à des structures de données primitives qui n'utilisent pas cette technologie. Les champs virtuels peuvent également être utilisés pour implanter plusieurs vues simultanées d'une même donnée. Supposons une classe rectangle où chaque instance est caractérisée par son point d'origine (`x0`, `y0`), soit son point supérieur (`x1`, `y1`), soit ses dimensions (`width`, `height`). Les champs virtuels permettent d'implanter une telle classe. Dans l'implantation présentée ci-dessous `height` et `width` sont des champs virtuels.

```
(class rectangle
   (x0 (default 0.0)) (y0 (default 0.0))
   (x1 (default 0.0)) (y1 (default 0.0))
   (width (get (lambda (rect)
                  (with-access::rectangle rect (x0 x1)
                     (- x1 x0))))
          (set (lambda (rect v)
                  (with-access::rectangle rect (x0 x1)
                     (set! x1 (+ x0 v))))))


   (height (get (lambda (rect)
                  (with-access::rectangle rect (y0 y1)
                     (- y1 y0))))
           (set (lambda (rect v)
                  (with-access::rectangle rect (y0 y1)
                     (set! y1 (+ y0 v)))))))
```

Modifier la valeur du champ virtuel `width` (ou `height`) ajuste automatiquement la valeur de `x1` et *réciproquement*. Dans notre modèle à objets, seuls les champs virtuels permettent d'assurer une telle synchronisation entre les champs. Pour la classe `rectangle` il n'y a pas de zone mémoire consacrée à l'enregistrement de `width` et `height` car ces valeurs sont recalculées à chaque fois qu'elles sont consultées. Ainsi la carte mémoire pour les instances de la classe `rectangle` est :



2.8.2 Biglook et les champs virtuels

Biglook, la bibliothèque graphique de Bigloo présentée dans § 2.7, est à l'origine de la création des champs virtuels. Biglook propose une interface de programmation (*API*) originale utilisant exclusivement les objets de Bigloo. Les objets graphiques sont tous modélisés par des classes et les

interfaces graphiques sont donc des conglomérats d'instances.
Afin de limiter notre effort, nous avons choisi d'appuyer le développement de Biglook sur des bibliothèques graphiques de plus bas niveau. Ainsi, toute la gestion des opérations graphiques de base (les pixels, les événements, les recouvrements graphiques, etc.) est déléguée à la bibliothèque de base. La figure 2.7 schématise l'architecture logicielle de Biglook. Plusieurs bibliothèques de bas niveau peuvent être utilisées par Biglook. Actuellement deux implantations existent : une utilisant Tk [155], l'autre basée sur GTK+ [158]. Ces deux bibliothèques primitives offrent des possibilités graphiques comparables, mais elles se distinguent par une interface de programmation totalement différente. Tk est conçue pour être couplée à Tcl, un langage de scripts manipulant principalement des chaînes de caractères alors que GTK+ propose une interface C plus neutre. Comme nous le verrons ultérieurement, une des caractéristiques majeures de Biglook est qu'elle ne dépend pas de la bibliothèque de base. En particulier, son *API* est identique pour la version Tk et la version GTK+. Ni Tk ni GTK+ n'utilise une *API* objet. Nous voulions cependant un modèle objet parce que cela semble parfaitement convenir pour la programmation d'interfaces graphiques. Nous avons alors conçu une nouvelle construction permettant de réifier des structures (enregistrements) primitives en des instances Bigloo : les champs virtuels.

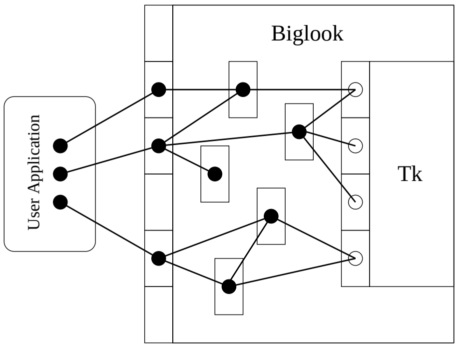
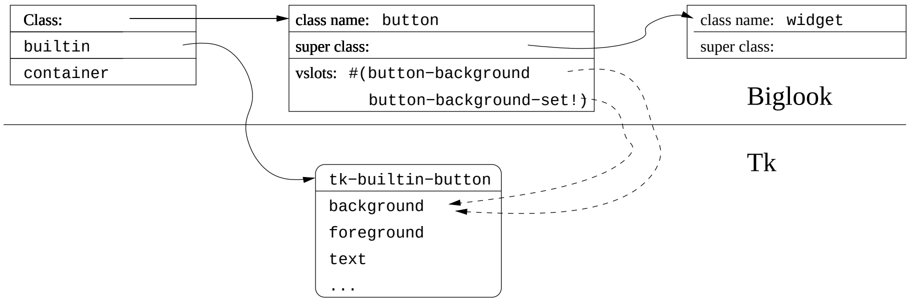


Fig. 2.7: L'architecture de Biglook

Les champs virtuels sont au cœur de l'implantation de Biglook. Les objets graphiques de Biglook (les *widgets*) sont tous représentés par des classes qui contiennent deux champs traditionnels : *1)* un pointeur vers une structure native associée au *widget* Biglook et *2)* un pointeur vers un container natif. Tous les champs additionnels qui décrivent les caractéristiques graphiques des objets sont représentés par des champs virtuels. Par exemple la couleur de fond d'un *widget*, son champ `background`, est implantée au moyen d'un champ virtuel dont les fonctions d'accès se contentent en fait d'appeler les fonctions d'accès primitives de la bibliothèque de base. Ainsi la déclaration des boutons dans Biglook ressemble à :

```
(class widget
   (builtin read-only)
   (container read-only))
(class button::widget
   (background
    (get (lambda (o)
           (<builtin-background-getter> (button-builtin o))))
    (set (lambda (o v)
           (<builtin-background-setter> (button-builtin o) v)))))
```

La carte mémoire pour les instances de la classe `button` est :



Comme pour les méthodes, il est parfois indispensable de référencer le champ virtuel d'une des super-classes d'une instance. Ceci est réalisé par la forme `(call-next-slot)` qui peut uniquement être utilisée à l'intérieur de la définition d'une fonction d'accès. Cette forme est utile dans Biglook pour l'implantation des *widgets* composites. Ces *widgets* particuliers sont présentés § 11.2.6, page 224. Étudions seulement un exemple simple. Un *widget* composite est formé de plusieurs *widgets* élémentaires qui, regroupés ensemble forment un nouvel objet complexe. Par exemple, Biglook propose des `labeled-entry` qui sont formés d'une juxtaposition entre un `label` (un petit texte) et une `entry` (une sélection textuelle). Le schéma classique de construction d'une telle classe dans Biglook consiste à construire une nouvelle classe qui hérite du *widget* principal, c'est-à-dire du *widget* qui donne son comportement au composite (ici l'`entry`) et qui « embarque » des widgets supplémentaires correspondants aux attributs graphiques du composite (ici le `label`). Lorsque l'utilisateur accède à la couleur de fond d'un tel *widget*, il accède indifféremment à celle de l'`entry` ou celle du `label`. En revanche, lorsqu'il change cette couleur, la modification doit être répercutée sur les deux *widgets* natifs réellement construits. Ainsi la déclaration d'un `labeled-entry` peut-être :

```
(class labeled-entry::entry
   (%label::label read-only)
   (background
    (get (lambda (o)
            (call-next-slot)))
    (set (lambda (o v)
            (call-next-slot)
            (label-background-set! (labeled-entry-%label o) v)))))
```

Sans champ virtuel la construction de tels *widgets* serait impossible car pour les réaliser, il est indispensable de disposer de champ dont les accès opèrent des traitements plus complexes que de simple lectures ou écritures mémoire.

2.8.3 L'implantation des champs virtuels

L'implantation des champs virtuels est très simple et elle ressemble à l'implantation de méthodes dans un langage comme Smalltalk. Chaque classe contient un vecteur de champs virtuels. L'accès à un de ces champs est alors implanté par :

1. Un accès mémoire à la classe pour lire le *vecteur de champs virtuels*.
2. Dans ce vecteur lire la fonction d'accès recherchée. La position qu'elle occupe est connue dès la compilation.
3. Appeler la fonction d'accès.

Ainsi, accéder à un champ virtuel pour une instance `i` revient à écrire :

```
(let ((vvec (class-virtual-vector (object-class i))))
   ((vector-ref vvec <a-static-offset>) i))
```

Il est possible de faire l'économie d'un accès mémoire si les instances contiennent directement un pointeur vers le vecteur de champs virtuels de leur classe (on rappelle ici que Bigloo utilise de l'héritage simple). Toutefois avec cette solution, les instances grossissent d'un mot.

Comme Bigloo exige d'avoir accès à toutes les définitions de classes qui sont utilisées dans un programme, il peut détecter les mauvaises utilisations de la forme `(call-next-slot)`. Comme cette forme ne peut être utilisée que dans la définition des fonctions d'accès aux champs virtuels, le compilateur a les moyens de tester que les super-classes (directes ou indirectes) de la classe courante (celle qui contient la définition du champ virtuel qui référence `call-next-slot`) définissaient déjà ce champ.

2.8.4 Les champs virtuels et les fonctions membres

Les champs virtuels ne sont pas les fonctions membres des langages comme Smalltalk, C++ ou Java. Pour les langages qui mettent en œuvre l'encapsulation, les fonctions constituent traditionnellement le moyen standard d'accès aux données membres (pour adopter la terminologie C++). Parce que la sélection des fonctions d'accès aux champ virtuels est retardée jusqu'à l'exécution (on parle de liaison tardive) et parce qu'elles sont définies par l'utilisateur, les champs virtuels peuvent s'apparenter aux fonctions membres. Toutefois pour un langage avec introspection tel que Java, il n'y a pas moyen de distinguer si une fonction membre est utilisée pour implanter une fonction d'accès ou si elle implante un service de l'utilisateur. Cette distinction est pourtant importante dans certaines applications. Par exemple, les constructeurs d'interfaces doivent connaître la liste des configurations possibles pour les *widgets*. Si les champs sont accédés au moyen de fonctions membre ordinaires, il faut alors *explicitement* indiquer au constructeur celles qui jouent le rôle de fonctions d'accès. Cela réduit l'intérêt de l'introspection parce qu'une description externe supplémentaire est requise.

2.8.5 Le paradoxe des champs virtuels

On peut s'étonner d'un paradoxe apparent des champs virtuels. Là où d'autres langages, en particulier C++ et Java, « masquent » au maximum les champs (données membres) des classes en ne proposant comme interface que des fonctions d'accès, les champs virtuels constituent un dispositif qui permet de masquer des fonctions d'accès en les faisant passer pour des champs ! Il y a deux explications à ce paradoxe :

- Comme nous l'avons signalé en § 2.8, les classes de Bigloo n'ont pas la vocation d'être des opérateurs de portée. Ce rôle est assumé par les modules et toute redondance est évitée.
- Les comportements auxquels les objets réagissent sont définis par le biais de fonctions génériques et non pas par les classes qui ne font que donner un type aux instances.

Ainsi, dans notre modèle, une classe n'est fondamentalement que la définition des informations que portent ses instances. Les champs virtuels permettent alors uniquement d'avoir une vision uniformisée de l'accès à ces informations. Le fait que la couleur d'un bouton ne soit pas directement sauvée dans la zone mémoire qui décrit l'instance de la classe `button` manipulée par l'utilisateur, depuis son programme Scheme, est sans importance. Il ne s'agit que d'un détail de l'implantation de Biglook qui fait qu'à l'instance `button` on associe une structure Tk et que c'est cette structure qui contient la couleur de fond.

2.8.6 Conclusion sur les champs virtuels

Les *champs virtuels* jouent un rôle essentiel dans le modèle objet de Bigloo parce qu'ils permettent de donner une interface objet uniforme à des structures de données hétéroclites. Les champs virtuels sont à la base de l'implantation de Biglook, la bibliothèque permettant la réalisation de *GUIs* avec Bigloo.

Avant d'être ajoutés à Bigloo, les champs virtuels ont été expérimentés dans le système STk [88] qui repose sur un interprète Scheme étendu d'une couche objet très proche de CLOS. En particulier, STk implante le *Meta Object Protocol* (abrégé par la suite, MOP) de CLOS. L'intérêt d'un tel mécanisme pour l'exploration des caractéristiques objet est évident. Toutefois la méta-programmation peut être incompatible avec la compilation et à fortiori la compilation séparée telle que prônée par Bigloo. Pour Bigloo, nous avons extrait de STk les fonctionnalités que nous sommes parvenus à compiler efficacement. On voit une sorte de compromis apparaître entre expressivité (le MOP) et une implantation raisonnable (ce qui est « compilable »).

2.9 Conclusion

Nous avons présenté dans ce chapitre quatre des constructions linguistiques que nous avons ajoutées à Scheme : *1)* les modules, *2)* les objets, *3)* les classes larges et *4)* les champs virtuels. Toutes ces constructions ont dépassé le stade du prototype puisqu'elles sont implantées et utilisées dans Bigloo(k). Les trois premières servent directement au *bootstrap* du compilateur et la dernière est au cœur de la bibliothèque graphique Biglook.

Ces quatre constructions découlent d'une démarche similaire. À plusieurs reprises, nous avons opté pour des restrictions sémantiques parce que cela simplifiait notablement l'implantation. Par exemple, nous avons renoncé à l'héritage multiple et au *dispatch* multiple pour les classes et fonctions génériques de Bigloo parce que nous savions mieux implanter le modèle actuellement proposé. Dans le même ordre d'idée, Bigloo ne propose pas de Mop [123] parce que cela complique énormément la compilation. En revanche, Bigloo propose des extensions aux classes traditionnelles (les classes larges et les champs virtuels) qui ne sont pas des obstacles à la compilation. La perte d'expressivité induite par nos restrictions est, nous le pensons, contrebalancée par de meilleures performances. Pour nos extensions linguistiques nous avons, en toutes circonstances, cherché le point d'équilibre entre expressivité et performance.

L'objet de nos recherches depuis 1991 est de concevoir un système de programmation réaliste basé sur le langage Scheme. Après avoir motivé le choix de ce langage dans l'introduction de ce mémoire (cf. chapitre 1, § 1.1), nous avons expliqué dans § 2 que la version « brute » de Scheme est trop restreinte pour être réaliste. Nous avons alors présenté quelques unes des constructions que nous avons ajoutées au langage. Chacune de ces constructions a été motivée par la résolution d'un problème qui nous a été posé, soit lors de l'implantation du compilateur Scheme Bigloo, soit lors de la conception de la bibliothèque graphique Biglook. Toutes ces constructions ont donc été conçues pour apporter une solution à un problème réel et elles ont été intégrées dans le système de façon à pouvoir être utilisées dans d'autres contextes.

2.10 Perspectives

Nous avons présenté tout au long de ce chapitre le langage de programmation, fortement ancré dans la tradition de Scheme, sur lequel nous avons bâti le système Bigloo. Nous présentons dans cette section un panorama des avancées qui nous semblent les plus significatives dans la conception des langages de programmation depuis l'apparition de Scheme. L'objectif de cette section est de mieux situer nos travaux dans le contexte global de la programmation.

2.10.1 Les langages fonctionnels depuis Scheme

Scheme, apparu en 1975 [228], est un des derniers représentants de la lignée Lisp. En reprenant les idées développées par C. Hewitt dans les langages Planner et Plasma [109], il a généralisé la liaison lexicale au détriment de la liaison dynamique largement employée dans les précédents dialectes de Lisp et fait des fonctions des objets de première classe. À ce titre, Scheme est le fruit d'un travail de simplification sémantique et d'harmonisation qui a été entrepris autour des variations de Lisp. Scheme n'a renié aucune de ses racines fondamentales : la syntaxe de Scheme (qui, quoi qu'on puisse en penser est une caractéristique très forte) est celle de Lisp et il utilise le typage dynamiquement. Depuis son apparition, Scheme a été le lieu privilégié de recherches sur les continuations (par exemple avec la présentation du *continuation passing style*) et sur la macro expansion.

Si les continuations de Scheme constituent un formidable outil pour modéliser et expliquer les diverses sortes d'échappements existantes dans les langages de programmation [171], programmer réellement avec des continuations nous semble moins défendable. À notre connaissance, `call/cc` est la forme jamais inventée qui permet le plus facilement l'écriture de programmes parfaitement incompréhensibles [173] (nous ne considérons pas ici la programmation distribuée). Même si Bigloo implante `call/cc` [2], nous préconisons plutôt l'emploi d'opérateurs de contrôle plus limités que sont les échappements lexicaux (`bind-exit` et `unwind-protect`). Nous sommes convaincu que les continuations de Scheme doivent être utilisées comme laboratoire d'expérimentation [174] mais qu'elles ne constituent pas un outil de programmation quotidien.

Très rapidement, après Scheme, sont apparus les premiers langages fonctionnels paresseux avec Sasl [243] suivi de Miranda [244] puis Haskell [112]. Bien que cela soit assez inattendu pour nous, nous devons reconnaître que vingt ans plus tard, la branche paresseuse de la programmation fonctionnelle est la plus active. Pour s'en convaincre il suffit de feuilleter les actes des dernières éditions de la conférence majeure du domaine (ICFP). Les langages paresseux ont, certes, une

[2] Mais à quel prix !

expressivité inégalée, ils nous semblent cependant difficiles à pratiquer. Par exemple, la mise au point dans ces langages relève du pire cauchemar, car afficher une valeur force son évaluation et donc change l'exécution du programme. Il est ainsi quasiment impossible de rejouer sous le *debugger*, ou même simplement en instrumentant, une exécution posant problème. Néanmoins, au vue des réalisations de cette communauté, nous devons reconnaître que cela ne semble pas être, pour ses pratiquants, un obstacle majeur.

Bien qu'il s'en défende, ML, apparu en 1978 [94] est très proche de Scheme. Toutefois, avec son système de types il a présenté une innovation majeure. ML est un langage fonctionnel (les fonctions qui sont des objets de première classe utilisent un mécanisme de liaison lexicale), polymorphe, fortement typé. De plus, il n'impose pas à ses utilisateurs d'annoter le code source avec des informations de type car, en ML, ils sont inférés automatiquement [58]. Il semblerait même que l'inférence (par contraste à la simple vérification) de types ait été le point de ralliement des « MLiens ». Jusqu'à une date assez récente, il semblait inconcevable pour cette communauté de devoir introduire manuellement des informations de types dans les programmes (au moins dans les définitions des fonctions). Leur passage à la technologie objet a visiblement affaibli cette position.

En plus du typage, la communauté ML s'est emparée des modules. Depuis l'article fondateur de D. MacQueen en 1984 [143] on ne compte plus les articles traitant des modules de ML (pas moins d'une vingtaine). Le principe est de proposer un langage pour construire les applications à partir de modules élémentaires. Ces recherches semblent arriver maintenant à maturité avec des versions un peu plus simples que les propositions initiales [135]. Avec le peu de recul historique dont nous disposons actuellement il est encore trop tôt pour évaluer l'impact de ces travaux. Néanmoins, il semble que les modules à la ML aient du mal à sortir du cadre de ce langage. Aucun autre langage apparu depuis 1996 ne semble s'inspirer des modules de ML. Sans réelle surprise, seul Scheme semble légèrement influencé [81]. Les modules de Bigloo, pour leur part, n'ont rien de commun avec les travaux entrepris pour ML parce que, par opposition, à ML, il n'est pas possible de calculer avec les modules de Bigloo. Probablement les modules de Bigloo sont plus proches des paquetage de Modula-2 [259].

L'apparition la plus récente pour les langages fonctionnels s'est faite en deux temps. En 1992, Apple a présenté le langage Dylan à l'occasion de la conférence *Lisp and Functional Programming* [12]. Dans sa première version le langage avait une syntaxe parenthésée qu'il a perdu dans son développement ultérieur en 1996 [214]. Dylan est un langage un peu hybride mêlant à des traits empruntés à Scheme, une couche objet ressemblant à CLOS avec une syntaxe *à la* C. Dylan est apparu sensiblement en même temps que Java avec quelques points communs (langage objet, GC, bibliothèque d'exécution typée et sûre) mais avec une philosophie opposée. Là ou Java est concis et relativement simple à implanter, Dylan est gros et son implantation efficace est un sujet de recherche à lui tout seul ! Le sort que subit Dylan (le langage est presque « mort-né ») est alors peu surprenant.

2.10.2 Les langages orientés objets

Depuis l'apparition de Smalltalk-80 en 1980 [91] (précédé de Smalltalk-72 et Smalltalk-78) les objets n'ont cessé de prendre de l'importance dans la programmation à tel point que nous vivons maintenant une période du « tout à l'objet ». Plus un langage n'apparaît, plus un ouvrage ne s'écrit, plus un compilateur ne s'implante, sans qu'il soit question d'objets. Le terme programmation objet est assez galvaudé et il décrit pour nous deux modèles assez différents. D'un côté le modèle de Smalltalk où les classes ont le double rôle de décrire les données manipulées et de structurer les programmes. De l'autre le modèle de CLOS où les classes servent uniquement de structures de données. Comme nous l'avons expliqué dans § 2.5 notre préférence va vers ce deuxième courant qui, nous devons le reconnaître, est très minoritaire. Même à l'intérieur de la communauté fonctionnelle, ce choix ne s'impose pas à tout le monde alors même que les fonctions génériques s'inscrivent harmonieusement dans ce modèle de programmation. Toutefois, nous ne pouvons nous empêcher de penser que si un système comme O'Caml [176] (le premier dialecte de ML à avoir supporté les objets) a opté pour l'approche Smalltalk de la programmation objet c'est par un choix négatif plus que positif. Il semblerait que l'inférence de types pour les fonctions génériques soit très difficile à réaliser [29, 33, 147] et comme le typage est au centre de la philosophie ML il est en fait peu surprenant que cette communauté se soit rabattue sur la solution « typable » avec les techniques alors connues. Force est de constater que finalement O'Caml a fait le choix le moins fonctionnel, un peu comme si dans le dilemme opposant fonctions et types, O'Caml avait choisi ces derniers. Il en résulte une certaine redondance dans le langage car deux constructions peuvent être utilisées pour structurer les programmes : les modules et les classes [137].

Les principaux langages objets qui ont marqué les années 80 et 90 de leurs empreintes sont assurément C++ [232] et Eiffel [146]. Y a-t-il des points communs entre Scheme et ces deux langages ? S'ils existent, ce ne sont en tout cas pas des points communs immédiats comme on peut en voir avec Java qui est apparu environ dix ans après C++ [234]. En termes linguistiques, Scheme n'a probablement rien apporté à C++ et nous n'avons pas eu connaissance d'extension Scheme influencée par C++. Seul ML a eu une influence notable sur C++ [233]. Peut-être même que C++ et Scheme (on peut généraliser ici à tous les LFs) ont été conçus de manière diamétralement opposée. C++ a été pensé, probablement comme C, dans l'optique d'une compilation performante. Cela a conditionné la plupart des traits de ce langage comme cela est expliqué dans [233]. Il en résulte que C++ est complexe, rempli de règles particulières qui ne s'expliquent que par l'implantation même du langage. Scheme et ML, au contraire, ont été basés sur un fondement théorique clairement défini : le λ-calcul. La performance n'a jamais été présente dans l'esprit des concepteurs de Scheme et ML[3] au moment de les créer. Il en a résulté des langages difficiles à compiler mais simples à apprendre et à comprendre.

Il semblerait toutefois qu'on puisse déceler une petite influence des langages de haut niveau dont les LFs font partie sur C++. Il nous semble, par exemple, détecter une influence des *foncteurs* de ML dans la conception de la STL [130]. Si initialement C++ était très fortement statique, l'introduction d'extensions comme le *run-time type identification* adoptée en 1993, diminue cette forte connotation initiale. Cela nous semble être assez révélateur d'une tendance plus générale ou après plus de dix années de règne des langages statiques, on voit depuis peu la renaissance des langages plus dynamiques dont l'exemple le plus populaire est bien sûr Java qui propose du typage dynamique et même de l'introspection. Des travaux comparables, dans l'esprit, pour C++ ont également été menés [44]. Il faut ici noter que Bigloo se situe à mi-chemin entre les deux extrêmes. Il n'offre pas tout le dynamisme que permet un MOP afin de permettre une implantation raisonnablement efficace, mais il permet l'introspection et le typage dynamique. Les deux principales extensions au modèle objet que nous avons présentées, les classes larges et les champs virtuels, illustrent ce compromis. Ces deux constructions utilisent un trait dynamique du langage (le typage) mais elles sont assez efficaces et surtout, elles ne compromettent pas le reste de l'implantation du système. Un programme qui ne les utiliserait pas, ne verrait pas ses performances diminuer par leur simple existence. Les classes larges et les champs virtuels ne coûtent (un peu) que si on les utilise !

2.10.3 Les langages de *scripts*

En marge des langages généralistes est apparue une multitude de petits langages spécialisés. Parmi les plus répandus on trouve Perl [252], Tcl [154] et Python [141]. Ces derniers ne sont que les représentants d'une deuxième génération de langages. La première datant de la fin des années soixante dix (AWK, Bourne Shell, ... [122]). Que dire de ces langages sinon qu'ils sont tous apparus en même temps que les premières machines RISC au début des années 1990 ? Il s'agit, pour la plupart, de langages de scripts qui ne sont pas adaptés au développement d'applications complexes et volumineuses. Ils semblent souvent avoir été construits avec des principes assez soigneusement évités dans les langages de haut niveau. Lorsque dans un langage généraliste, on cherche, à tout prix, à éviter les constructions ambiguës et celles qui masquent la complexité réelle ou qui ont des effets non apparents, leurs auteurs semblent avoir fait les choix inverses. Par exemple un langage comme Perl érige quasiment en dogme la notion de « règle implicite ». Il existe plusieurs syntaxes pour écrire le même programme en Perl. La plus compacte, probablement celle des experts Perl, étant la plus irrégulière. Tcl apporte lui aussi son lot d'excentricités. Son modèle

[3]Elle aurait peut-être dû l'être un peu plus, par exemple lorsque les créateurs de Scheme ont pensé à `call/cc`.

d'évaluation et sa syntaxe étant proches de ceux des *shells* d'Unix, les caractères " et $ jouent un rôle assez obscur. Pour ces raisons, il nous plus d'une ou deux pages dans ces langages. Pour conclure, nous nous contenterons de mentionner que lors d'une querelle assez animée, R. Stallman s'est fait l'avocat de Scheme. La *Free Software Fundation* l'a choisi comme langage de scripts à la place de Perl ou Tcl.

2.10.4 Prospective

L'informatique évolue si vite qu'il est difficile de spéculer sur les transformations à venir. Par exemple, l'évolution soudaine du réseau a fait naître des besoins nouveaux qu'on ne sait pas traiter convenablement avec les langages de programmation classiques. Il semblerait qu'une part de l'informatique s'oriente vers des systèmes éclatés où les programmes migreront d'une machine à l'autre et où la sécurité des informations manipulées sera déterminante (par exemple pour le commerce électronique). Nous avons montré que la sûreté des informations est liée aux constructions élémentaires des langages de programmation [248]. La démarche consistant à tenter de rendre sûr un langage de programmation qui n'aurait pas intégré *initialement* cette contrainte dans sa conception est vouée à l'échec. Nous ne disposons pas encore de ceux qui assureraient de façon satisfaisante l'intégrité des données manipulées.

L'évolution semble également aller vers des systèmes plus dynamiques (peut-être faudrait-il dire, *plus interactifs*). De plus en plus souvent on « télécharge » des programmes déjà compilés et on les exécute sur sa machine. Pourquoi ne pas en télécharger simplement des morceaux ? C'est le but des « applets » qui permettent d'ajouter dynamiquement des services à une application. Cette pratique nouvelle fait naître de nouveaux besoins linguistiques (chargement dynamique de code compilé, vérification dynamique que les programmes ainsi chargés font bien ce qu'ils doivent en respectant les règles de fonctionnement des machines hôtes, etc). Parallèlement, avec la miniaturisation, il devient envisageable de mettre des processeurs généraux et programmables dans presque tous les appareils électriques (comme les appareils électroménagers ou même les cartes à puce). Cette pratique définit de nouveaux besoins en terme de programmation en mettant, par exemple, l'accent sur la compacité du code et des données par opposition à la vitesse d'exécution. Ces dix dernières années n'ont apporté que des avancées modestes pour les langages de programmation généralistes. Probablement, les principales innovations à venir seront le fruit de réflexions autour des langages spécialisés.

Chapitre 3

Bigloo : l'implantation

A CONS is an object which cares.
– Bernie Greenberg

COMME nous l'avions noté dans [204] « *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)* ». Si cette volonté parfois minimaliste complique la compilation c'est aussi probablement ce qui fait l'élégance de ces langages. Depuis 1994, nous avons exploré deux nouvelles directions pour encore améliorer celle de Bigloo. La première présentée dans ce chapitre (§ 3.1) consiste en une analyse des données des programmes [211]. Elle permet d'économiser des allocations mémoire et d'utiliser une représentation des valeurs plus efficace. Notre seconde préoccupation (§ 3.2) a été d'améliorer l'intégration fonctionnelle déjà implantée dans le compilateur. Cette optimisation diminue le coût des appels de fonctions. Nous présentons dans ce chapitre un procédé nouveau qui s'est avéré plus efficace que celui précédemment mis en œuvre [207].

3.1 L'analyse des structures de données

Sur toutes les architectures contemporaines, les lectures et les écritures mémoire sont des opérations plus coûteuses que les opérations arithmétiques. L'accès à la mémoire est le goulet d'étranglement. En conséquence, un compilateur doit produire du code qui alloue le moins de mémoire possible et il doit judicieusement choisir ses zones d'allocations. Pour y parvenir, les deux principales difficultés sont :

– Le polymorphisme des langages comme Scheme et ML est difficile à bien implanter. Dans ceux-ci, les fonctions peuvent accepter des arguments de plusieurs types. La plus représentative est l'*identité* qui accepte indifféremment toutes les valeurs que le langage supporte, par exemple, des caractères, des nombres exacts ou des nombres flottants. Le polymorphisme complique la compilation car les différentes valeurs du langage ne peuvent pas forcément être traitées de la même façon. Par exemple, un nombre exact et un nombre flottant n'ont pas la même taille et ils ne doivent pas résider dans les mêmes registres physiques. La solution généralement employée pour traiter le polymorphisme est alors de ne pas directement manipuler de valeurs mais d'utiliser des pointeurs vers ces valeurs. Cette technique se nomme le *boxing* parce qu'elle nécessite l'allocation de « boîtes » mémoire. On parle alors de représentation *uniforme* car toutes les valeurs manipulées ont la même taille : celle des pointeurs. Cet encodage est malheureusement inefficace car il augmente la quantité d'allocations opérées

par le programme. Parvenir à l'éviter permet d'atteindre un meilleur niveau de performance. En adoptant une représentation *mixte*, c'est-à-dire, une représentation où toutes les valeurs n'ont pas obligatoirement la même taille, les allocations sont moins fréquentes et les accès aux valeurs plus efficaces.
– La majorité des bibliothèques d'exécution (RTS) utilise une pile pour allouer les blocs d'activation des fonctions. Pour ces systèmes, allouer des données utilisateur, en pile plutôt que dans le tas est plus efficace parce que leur désallocation est automatiquement réalisée dès que les fonctions se terminent. Certains langages comme C ou C++ permettent aux utilisateurs d'explicitement allouer dans la pile. En général les langages qui utilisent des GCs n'offrent pas cette possibilité. C'est alors au compilateur de détecter les données ayant une durée de vie coïncidant avec celle des fonctions qui les allouent et donc de les allouer en pile plutôt que dans le tas.

Dans cette section nous présentons une nouvelle analyse, la SUA, qui permet à la fois d'utiliser une représentation mixte et de déplacer des allocations du tas vers la pile. L'algorithme complet de l'analyse ainsi que des notes d'implantation sont présentées § 7.2, page 126. Dans cette section nous exposons seulement le principe de l'algorithme et nous montrons son application sur quelques exemples.

3.1.1 L'analyse

La SUA s'inspire de la 0cfa (0th order control flow analysis) de O. Shivers [219, 218]. Cette dernière est une interprétation abstraite qui calcule des approximations des valeurs fonctionnelles. Son résultat est un ensemble d'approximations. Pour chaque variable dans chaque fonction d'un programme, la 0cfa indique la liste des fonctions qu'elle peut contenir. La 0cfa appartient à une famille d'analyses, la famille des Ncfa (1cfa, 2cfa, ...), qui se distinguent par la précision des approximations qu'elles calculent. Ainsi, la 1cfa est plus précise que la 0cfa mais elle a aussi une complexité supérieure (la complexité de la 0cfa est entre $\mathcal{O}(n^2)$ et $\mathcal{O}(n^3)$ alors que la 1cfa est exponentielle ; il s'agit de complexité dans le pire des cas qui sont en fait rarement atteintes). Ces analyses ont été conçues dans l'optique d'améliorer la compilation du contrôle (c'est au moyen d'une 0cfa que Bigloo fait l'allocation de fermetures [206]). Nous avons étendu la 0cfa pour calculer des approximations composées non plus seulement de fonctions mais de toutes les valeurs que peuvent manipuler les programmes, dans le même esprit que les *set bases analysis* [107]. Ainsi le résultat de la SUA est un ensemble d'approximations pour toutes les variables et toutes les fonctions du programme. Chaque approximation est formée des constructeurs (les formes du programme qui construisent les valeurs ; les constructeurs étant généralement des fonctions comme `CONS` qui construit des paires ou même `+` qui construit des nombres) qui représentent toutes les valeurs qui peuvent *théoriquement* transiter par cette variable du programme. Pour être correcte, l'analyse est conservative. C'est-à-dire que les approximations calculées sont strictement plus grandes que les valeurs dynamiques que peuvent prendre les variables lors de *toutes* les exécutions possibles.

La SUA procède par itérations de point fixe où chaque itération consiste en un parcours en profondeur de l'AST du programme à compiler. Les itérations sont stoppées dès qu'un parcours complet de l'arbre n'ajoute plus d'informations aux ensembles d'approximations. La terminaison de ce processus est garantie par le fait que les programmes contiennent un nombre fini de variables, que les ensembles d'approximations contiennent un nombre fini d'éléments et qu'un élément n'est jamais supprimé d'un ensemble.

Dans les ensembles calculés par la SUA, les valeurs sont représentées par leur constructeur. Ainsi, une paire ou un vecteur est représenté par le nœud de l'AST qui correspond à leur allocation. Pour les littéraux, on utilise des constructeurs génériques qui ont le nom des types des valeurs manipulées. Ainsi, on suppose l'existence de deux constructeurs uniques nommés `fixnum` et `flonum` qui permettent de représenter tous les nombres utilisés dans les programmes. Étudions le déroulement de la SUA sur un premier exemple très simple :

```
1 :  (define (id x)
2 :     x)
3 :
4 :  (define (plus a b)
5 :     (+ a b))
6 :
7 :  (define (start)
8 :     (plus
9 :      (id 4)
10 :     (id 5.0)))
```

L'analyse collecte les ensembles des approximations pour les variables `x`, `a` et `b` et pour les résultats des fonctions `id`, `plus` et `start`. Les itérations commencent leur descente dans l'arbre par la fonction `start` que nous considérons ici comme le point d'entrée du programme. La Sua atteint donc la ligne *9*. La valeur 4 est un entier, elle est représentée par le constructeur `fixnum` qui est ajouté à l'ensemble décrivant le paramètre formel de la fonction `id`. À ce stade, l'ensemble d'approximations de `x` est alors composé uniquement de {`fixnum`}. Comme `id` retourne `x`, l'approximation pour la valeur de la fonction `id` est alors également {`fixnum`}. Après avoir traité le corps `id`, la Sua reprend sa descente à la ligne *10*. L'argument est cette fois un réel et alors l'approximation de `x` est étendue à {`fixnum`, `flonum`}. Afin d'éviter que l'analyse boucle en présence de définitions mutuellement récursives, le corps de chaque fonction n'est parcouru qu'*une* seule fois au cours de chaque itération. Ainsi, l'appel ligne *10* ne conduit pas la Sua à un second parcours du corps de la fonction `id` et la première itération conclut donc que l'ensemble des approximations de la valeur de retour de `id` est uniquement formé de {`fixnum`}. Les arguments ayant été tous examinés, la Sua entreprend alors l'examen de la fonction `plus` appelée ligne *8*. Comme à ce stade de l'analyse, le résultat de `id` est {`fixnum`}, l'analyse attribue aux variables `a` et `b` le même ensemble d'approximations, c'est-à-dire {`fixnum`}. Bien sûr ce résultat temporaire n'est pas correct et il sera complété lors des itérations suivantes.

Au cours de la seconde itération, lorsque l'analyse examine le corps de `id`, ligne *2*, `flonum` est ajouté à l'ensemble des valeurs de retour de cette fonction puisque l'ensemble d'approximations de `x` est {`fixnum, flonum`}. Cette approximation est propagée jusqu'aux variables `a` et `b`. Le point fixe est atteint dès la seconde itération et l'analyse conclut que `a` et `b` peuvent contenir soit `fixnum` soit `flonum`. Nous présentons dans § 3.1.2 et § 3.1.5 comment ces résultats peuvent être utilisés pour améliorer la compilation.

La Sua et les fonctions d'ordre supérieur

La Sua accepte en entrée une variante du langage Scheme où les fermetures doivent être explicitées. Ainsi, le langage traité contient trois constructions : `make-closure` qui alloue des fonctions d'ordre supérieur, `closure-ref` qui permet d'accéder aux variables libres d'une fonction et `closure-call` qui permet d'appeler une fonction. La transformation qui permet de traduire un programme Scheme dans le langage de la Sua est connue sous le nom de λ-lifting et elle est présentée dans [118]. Ainsi, le programme :

```
(define (curry-plus x)
   (lambda (y) (+ x y)))

(define (add a b)
   ((curry-plus a) b))

(define (start)
   (add 3.4 5.6))
```

se réécrit par λ-lifting :

```
1 :  (define (curry-plus x)
2 :     (make-closure_i <anonymous-1> x))
3 :
4 :  (define (<anonymous-1> p y)
5 :     (+ (closure-ref p 0) y))
6 :
7 :  (define (add a b)
8 :     (closure-call
9 :      (curry-plus a) b))
10 :
11 :  (define (start)
12 :     (add 3.4 5.6))
```

Comme pour l'exemple précédent la Sua commence par l'examen du point d'entrée `start`, ligne *11*. L'appel ligne *12* ajoute {`flonum`} aux approximations de `a` et `b` (paramètres de `add`, ligne *7*). L'appel ligne *9* ajoute à son tour {`flonum`} à l'approximation de `x` (paramètre de `curry-plus`, ligne *1*). La fonction `curry-plus` retourne une fonction, l'approximation de sa valeur de retour est alors $\{\texttt{make-closure}_i\}$, c'est-à-dire, un ensemble formé d'un élément qui est un nœud de l'Ast. Les indices des constructeurs permettent dans ce texte de les distinguer les uns des autres. En fait, la valeur enregistrée dans l'approximation de `curry-plus` est suffisante pour accéder aux données enregistrées dans la fonction construite. Ainsi, il serait plus juste de représenter l'approximation de `curry-plus` par $\{\texttt{make-}\widehat{\texttt{closure}}\texttt{(<anonymous-1>, \{flonum\})}\}$. La notation $\texttt{make-}\widehat{\texttt{closure}}$ nous sert ici à représenter une « méta » structure de données isomorphe à celle construite par le programme mais qui n'est utilisée que par la Sua. Pour accéder à cette donnée, les fonctions $\texttt{closur}\widehat{\texttt{e-function}}(a)$ et $\texttt{clos}\widehat{\texttt{ure-ref}}(a, i)$ seront utilisées. Lorsque la Sua examine l'appel d'ordre supérieur ligne *8* en utilisant $\texttt{closur}\widehat{\texttt{e-function}}$, elle détermine donc que c'est la fonction `<anonymous-1>` qui est appelée. En utilisant la seconde fonction d'accès, ligne *5*, la Sua conclut alors que la fonction `<anonymous-1>` retourne une valeur qui est l'addition de deux nombres réels (un pour `y` et un pour `(closure-ref p 0)`). Ainsi, l'ensemble d'approximations pour `<anonymous-1>` est {`flonum`}.

Bigloo permet la compilation séparée et la Sua doit donc pouvoir traiter des programmes incomplets. Pour cela, on ajoute une valeur particulière dans les ensembles d'approximations notée $\top$ qui dénote l'ensemble de toutes les valeurs possibles des programmes. Un ensemble qui contient cette valeur contient en fait toutes les valeurs possibles. Une fonction importée a comme approximation de son résultat $\{\top\}$. Les paramètres d'une fonction exportée valent tous $\{\top\}$[1].

La Sua et les listes

La Sua permet de calculer des approximations de valeurs placées dans des listes. Pour cela, elle utilise des méta-listes qui sont des couples d'approximations. La forme $\widehat{\texttt{cons}}$ dénote une liste telle que représentée de façon interne par la Sua. Il s'agit d'un couple d'approximations qui peut être accédé au moyen de $\widehat{\texttt{car}}$ et $\widehat{\texttt{cdr}}$. Comme les listes Scheme, les listes de la Sua sont mutables. Étudions ainsi le traitement du programme :

[1] L'analyse présentée dans 7 calcule deux ensembles d'approximations par variable : un pour les types et un pour les valeurs. Ainsi, lorsqu'une fonction est exportée, si elle porte des annotations de type pour ses paramètres ou sa valeur de retour, la Sua est encore capable de calculer une approximation *utilisable* pour les types des variables.

```
(define lst
   (let ((p1 (cons_i 1 0)))
      (let ((p2 (cons_ii 2 p1)))
         p2)))

(define (length l)
   (if (pair? l)
       (+ 1 (length (cdr l)))
       0))

(define (start)
   (length lst))
```

Ce programme utilise des nombres exacts (`fixnum`), deux paires ($\texttt{cons}_i$ et $\texttt{cons}_{ii}$) et les approximations calculées pour ce programme sont :

$$\begin{array}{lcl} \texttt{p1} & \equiv & \{\texttt{cons}_i\} \\ \texttt{p2} & \equiv & \{\texttt{cons}_{ii}\} \\ \texttt{lst} & \equiv & \{\texttt{cons}_{ii}\} \\ \texttt{l} & \equiv & \{\texttt{fixnum}, \texttt{cons}_i, \texttt{cons}_{ii}\} \end{array}$$

Et les deux paires sont représentées par :

$$\begin{array}{lcl} \texttt{cons}_i & \equiv & \widehat{\texttt{cons}}(\{\texttt{fixnum}\}, \{\texttt{fixnum}\}) \\ \texttt{cons}_{ii} & \equiv & \widehat{\texttt{cons}}(\{\texttt{fixnum}\}, \{\texttt{cons}_i\}) \end{array}$$

Bigloo, en pratique, ne calcule pas exactement ces approximations. Comme nous l'expliquons § 7.4.5, page 142, les paires doivent contenir des valeurs utilisant la représentation uniforme, autrement les fonctions standard de la bibliothèque ne pourraient pas être utilisées. Pour forcer une représentation uniforme pour les paires, les fonctions d'accès $\widehat{\texttt{car}}$ et $\widehat{\texttt{cdr}}$ incluent toujours dans les approximations qu'elles retournent un constructeur particulier, nommé $\boldsymbol{obj}$, qui joue un rôle de *marqueur*.

La Sua et les vecteurs

Les vecteurs sont traités différemment des listes et des fonctions. En revanche, toutes les autres données agrégées (principalement les structures et les objets) peuvent être traitées de la même manière que les trois cas présentés (fermetures, listes et vecteurs). Les vecteurs se distinguent des fermetures et des listes parce qu'en général, la valeur de l'indice qui est utilisé pour les accéder n'est pas connue lors de la compilation. Ainsi, il est généralement impossible de considérer que les approximations seront représentées par une structure de données isomorphe au vecteur dynamiquement construit comme c'était le cas pour les fonctions et les paires. Pour les vecteurs, toutes les approximations de toutes les valeurs possibles contenues sont fusionnées. Par exemple, si un vecteur est composé de caractères et de nombres entiers, alors la Sua considérera que chaque accès à ce vecteur retourne une valeur qui est soit un caractère soit un nombre. C'est comme si la Sua ne disposait que de vecteur à un seul champ ! Si ce procédé pour calculer les approximations peut sembler grossier, il n'en est pas moins efficace. Comme nous le verrons par la suite, la principale utilisation de la Sua est la détection des zones et des données monomorphes. Fusionner toutes les valeurs contenues dans un vecteur n'est pas incompatible avec ce calcul.

3.1.2 La représentation uniforme des données

Les approximations calculées par la Sua peuvent être utilisées pour supprimer des tests de type dynamiques. Ce genre de test peut être supprimé si la Sua parvient à démontrer que toutes les valeurs possibles de l'objet du test sont (ou ne sont pas) du type testé. Cette idée est à la base du *type-recovery* tel que présenté par Shivers dans [219, chapitre 9]. Plus que les tests de type, nous pensons (cf. chapitre 7, § 7.4.5) que c'est la représentation uniforme des données qui provoque une importante dégradation des performances. Dans ce modèle, tous les objets ont exactement la même taille (généralement la taille d'un mot mémoire de la machine utilisée). Les objets qui requièrent plus d'espace, comme les nombres flottants, sont alloués dans le tas et manipulés au travers de pointeurs. Cette technique, nommée *boxing*, permet de masquer les disparités entre les données.

Le polymorphisme généralise l'emploi de la représentation uniforme parce que les types dynamiques *exacts*, c'est-à-dire les types des valeurs dynamiques *réelles* auxquelles les variables polymorphes sont liées lors des exécutions, ne peuvent généralement pas être connus statiquement. Ainsi, lorsqu'une fonction polymorphe est compilée, le compilateur ne dispose ni de la taille qu'auront les paramètres effectifs ni des informations lui permettant de choisir le protocole d'appel le plus approprié. Cela conduit à une grande dégradation des performances.

Il n'est pas nécessaire que toutes les données aient la même taille avec une représentation mixte. Certaines valeurs peuvent être allouées alors que d'autres peuvent être manipulées directement. Les travaux les plus significatifs utilisant une représentation mixte sont dus principalement à X. Leroy [132, 134], Peyton-Jones et Launchbury [161] et Shao et Appel [216, 215]. Malheureusement aucune des techniques présentées ne peut être appliquée à Scheme parce que toutes imposent un typage statique. La méthode décrite par X. Leroy mélange des « zones » spécialisées où les types utilisés sont monomorphes et des zones « uniformes » qui correspondent aux parties polymorphes des programmes. Des conversions entre les deux types de représentations sont introduites quand un objet polymorphe est utilisé dans un contexte monomorphe, par exemple en « entourant » le corps d'une fonction polymorphe. Cette solution permet la compilation séparée, mais leur inconvénient est que plusieurs conversions peuvent, potentiellement, être appliquées au même objet.

Également pour permettre la compilation séparée nous adoptons une technique similaire à celle de Leroy en introduisant dans les programmes des conversions entre représentations. La principale originalité de notre travail est alors de ne pas utiliser des informations de type mais d'utiliser les résultats de la Sua pour sélectionner les représentations utilisées.

3.1.3 L'élection des types dans Bigloo

Lors de la compilation, immédiatement après la Sua, Bigloo opère l'élection des types. Cette opération consiste à attribuer un type à chaque variable et à chaque fonction du programme. Elle se base uniquement sur les approximations de la Sua, à l'exclusion de toute analyse supplémentaire. Étudions l'élection des types avec l'exemple ci-dessous (ici `-fx` et `=fx` sont les fonctions arithmétiques spécialisées pour les entiers) :

```
(define (bcopy! dst src size)
   (let loop ((i (-fx size 1)))
      (if (=fx i -1)
          0
          (let ((c (string-ref src i)))
             (string-set! dst i c)
             (loop (-fx i 1))))))

(define (copy-string str)
   (let* ((len (string-length str))
          (new (make-string len)))
      (bcopy! new str len)
      new))

(copy-string "foo")
```

La Sua parvient à montrer que `str`, `new`, `dst` et `src` sont des chaînes de caractères alors que `len` et `i` sont des nombres entiers, `c` étant un caractère. La Sua parvient à calculer ces approximations parce que les fonctions de la bibliothèque standard (`string-length`, `make-string`, `string-ref` et `string-set!`) sont connues du compilateur. Chacune des variables ne contient qu'un seul type de valeur dans son ensemble d'approximations et ce type leur est attribué. Les variables qui contiennent plus d'un type dans leur approximation se voient attribuer le type *obj* qui est le type le moins spécifique (tous les autres types sont des sous-types de *obj*).

La SUA fusionne toutes les approximations d'une variable dans un même ensemble. Ainsi, si pour une variable v un ensemble d'approximations ne contient qu'un seul type de données, cela implique que v n'est utilisée que dans des zones monomorphes du programme. Par exemple, si la SUA parvient à démontrer que les paramètres formels d'une hypothétique fonction identité sont des entiers, cela démontre que cette fonction, dont la définition est polymorphe, n'est utilisée que sur un seul type de valeur et elle est donc *ipso facto* monomorphe. La SUA isole les usages monomorphes des programmes polymorphes. Les algorithmes de typage *à la ML* (algorithme W [58]) calculent pour chaque fonction le type le plus général alors que la SUA calcule le type le plus restreint possible. L'algorithme W type en examinant uniquement le corps des fonctions alors que la SUA type en n'examinant que les usages qui en sont faits. Pour la détection du polymorphisme, la SUA produit des informations plus fines.

3.1.4 Les conversions de type

L'étape suivant l'élection des types dans la compilation est l'introduction des conversions. Il s'agit comme pour Leroy d'introduire des fonctions de conversions aux frontières des zones monomorphes et des zones polymorphes. Les données peuvent être indifféremment mises sous une représentation mixte ou sous une représentation uniforme. Les conversions sont introduites au cours d'un simple passage de l'AST. Ainsi, supposons le programme :

```
(define (id x)
   x)

(define (foo y)
   (id (+fl 1.0 (id y))))
```

Et supposons que la fonction `foo` soit exportée, son argument et son type de retour sont donc *obj*. La fonction identité `id` est appliquée une fois sur un nombre flottant (le résultat de la fonction `+fl`) et une fois sur un objet de type *obj*. Le paramètre `x` se voit donc attribuer le type *obj*. Les conversions suivantes sont alors introduites dans le programme :

```
(define (id x)
   x)

(define (foo y)
   (id (float-box (+fl 1.0 (float-unbox (id y))))))
```

Tout comme pour les travaux de Leroy, ce procédé peut conduire à convertir plusieurs fois une même valeur. Dans certains cas pathologiques, rares dans les faits (cf. § 3.1.6), cela peut dégrader les performances des programmes.

3.1.5 L'allocation en pile

Le but de l'optimisation de l'allocation en pile est de détecter automatiquement parmi les données allouées de l'utilisateur celles qui peuvent être déplacées du tas vers la pile. Pour cela nous supposons deux zones mémoire : *1)* une zone gérée par un GC et *2)* une zone gérée comme une pile. Cette seconde zone fait partie des racines du GC. Les blocs d'activation des fonctions sont alloués en pile lorsque l'exécution atteint le début d'une fonction et ils sont automatiquement recyclés lorsque l'exécution de la fonction prend fin. Avec ce schéma, l'allocation en pile pour les structures de données de l'utilisateur est possible. Les données ainsi allouées sont automatiquement libérées à la fin des fonctions. En général, l'allocation en pile est très rapide (comme l'allocation dans un GC copiant) et la récupération des zones allouées est gratuite parce qu'elle est assurée par le code de retour des fonctions qui est obligatoirement exécuté, même s'il n'y a pas eu d'allocations en pile. Illustrons au moyen de trois exemples les différentes configurations d'allocations de nos programmes :

```
1 :  (define (foo)
2 :     (let ((x (cons_i 1 2)))
3 :        (car (id x))))
4 :
5 :  (define (id z)
6 :     z)
```

Pour ce premier exemple, la paire cons$_i$ liée à la variable x ligne *2* peut être allouée en pile puisqu'elle n'est jamais utilisée au delà de la construction let qui définit la portée de x.

```
1 :  (define (bar)
2 :     (let ((x (cons_ii 1 2)))
3 :        (let ((y (cons_iii 3 x)))
4 :           (cdr y))))
```

Dans ce second exemple, la paire allouée ligne *2*, cons$_{ii}$ est vivante lorsque bar termine son exécution (c'est la valeur de retour de bar). Elle ne peut donc pas être allouée en pile. En revanche, cons$_{iii}$ n'est plus accessible en dehors de la portée de x. Elle peut donc être allouée en pile.

```
1 :  (define (gee a b)
2 :     (set-cdr! a b))
3 :
4 :  (define (hux)
5 :     (let ((p0 (cons_iv 1 2)))
6 :        (let ((p1 (cons_v 3 4)))
7 :           (let ((p2 (gee p0 p1)))
8 :              p0))))
```

Enfin, dans ce dernier exemple, aucune paire ne peut être allouée en pile parce qu'elles sont toutes vivantes au retour de hux.

L'allocation en pile et la SUA

En faisant abstraction des contraintes spécifiques de Scheme qui imposent un traitement sans consommation de pile des appels récursifs en position terminale (ce point est discuté dans § 7.3.3, page 138) une allocation peut être effectuée en pile si la donnée allouée n'est plus utilisée à partir du moment (dynamique) où la fonction qui a conduit à son exécution se termine. Autrement dit, si la donnée n'est plus vivante au point de retour de la fonction. Une donnée est vivante si elle est accessible depuis la valeur de retour d'une fonction ou si elle est accessible via une variable globale du programme. Les approximations calculées par la SUA permettent statiquement d'estimer la durée de vie des données. Les informations que nous utilisons sont : *1)* parmi les valeurs que peut prendre une variable, celles qui correspondent à des données allouées, *2)* la liste des allocations qui peuvent être pointées par une structure de données elle-même allouée et *3)* l'ensemble des allocations qui peuvent être retournées par une fonction.

Pour l'allocation en pile, chaque allocation porte une estampille qui est un nombre entier. L'optimisation de l'allocation en pile nécessite un parcours supplémentaire de l'AST. Elle utilise un générateur d'estampilles dont la valeur est incrémentée à chaque fois qu'un nœud let de l'AST est atteint. Quand l'algorithme d'optimisation atteint la définition d'une fonction f, l'estampille courante est sauvée dans une variable h, le corps de f est alors parcouru. Parmi les allocations qui sont contenues dans l'ensemble d'approximations de la valeur de retour de f, celles qui portent une estampille plus récente (un nombre plus grand) que h ne peuvent pas être allouées en pile. De plus, lors de l'inspection d'un nœud représentant l'affectation d'une variable globale, toutes les allocations contenues dans l'approximation de la variable sont marquées comme ne pouvant pas être allouée en pile. L'algorithme complet est présenté § 7.3.2, page 136.

3.1.6 Les mesures de performances

	Caractéristiques	
prgm	Taille en nb de lignes	Ressources consommées
Ttak	20	Appels de fonctions.
Bcopy	43	Chaînes, caractères, nombres entiers, boucles.
Mbrot	46	Nombres flottants, boucles.
Fft	127	Arithmétique en nombres flottants, boucles.
Puzzle	390	Nombres entiers, tableaux.
Beval	548	Ordre supérieur, conditionnelles.
Boyer	606	Ordre supérieur, calcul symbolique.
Maze	800	Tableaux, nombres entiers, boucles
Slatex	2821	IO, chaînes, listes.
Nucleic	3496	Arithmétique en nombres flottants.

FIG. 3.1 – Les programmes de test

La SUA, l'élection des types et l'allocation en pile sont implantées dans Bigloo. Ainsi, nous avons pu effectuer des mesures de performances qui nous permettent d'estimer la pertinence de ces optimisations. Des mesures plus complètes peuvent être trouvées § 7.5, page 143. Nous n'en montrons ici que quelques extraits. Pour toutes les mesures présentées ici, nous comparons deux versions du compilateur. La version 1.8 distribuée en 1996 qui a été la première à implanter la SUA et la version 1.7 de 1995 qui est, hormis la SUA, en tout point identique à la version 1.8.

Les mesures ont été relevées sur architecture DEC Alpha (Alpha 21064, 150 MHz, avec 160 MBytes) et sur un processeur Sparc (Sparc 2, avec 48 MBytes). Nous avons utilisé divers programmes pour nos mesures. Ils ont des tailles variables et ils utilisent différentes ressources. La figure 3.1 les décrit succinctement.

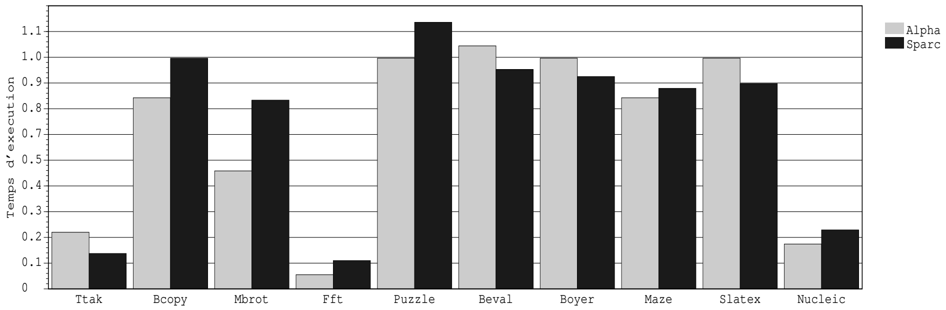


FIG. 3.2 – L'accélération occasionnée par la SUA sur deux architectures

Afin d'estimer l'impact de la SUA sur les performances des programmes compilés nous présentons figure 3.2 les ratios des temps d'exécution entre les programmes compilés avec et sans la SUA. Les chiffres présentés sont calculés par la formule :

$$\delta_e = \frac{\text{exécution optimisée (v1.8)}}{\text{exécution normale (v1.7)}}$$

C'est-à-dire que plus les chiffres sont petits, plus l'accélération due à la SUA est grande. Seul le programme **Puzzle** sur Sparc subit une dégradation des performances ($\delta_e = 1.13$). Tous les autres sont sensiblement plus rapides. On voit ainsi que l'introduction des conversions n'a que rarement un effet négatif. Les accélérations les plus significatives sont relevées pour les programmes opérant des calculs flottants. Par exemple l'accélération notée sur Alpha pour le test **Fft** est presque d'un facteur 20 ($\delta_e = 0.06$) et celle du programme **Nucleic** sur Sparc est environ d'un facteur 4 ($\delta_e = 0.25$). Ces accélérations spectaculaires s'expliquent par le fait que sans la Sua ces programmes allouent énormément de structures de données pour représenter les nombres. Les exécutions optimisées ne passent plus alors leur temps à allouer des structures inutiles.

La figure 3.3 est composée des ratios d'allocation mémoire. Elle permet d'estimer, sur nos programmes de test, les réductions d'allocations autorisées par la Sua. Les chiffres présentés sont calculés par les formules :

$$\delta_{mt} = \frac{\text{allocations dans le tas avec Sua}}{\text{allocations dans le tas sans Sua}}, \delta_{ms} = \frac{\text{allocations en pile avec Sua}}{\text{allocations dans le tas sans Sua}}$$

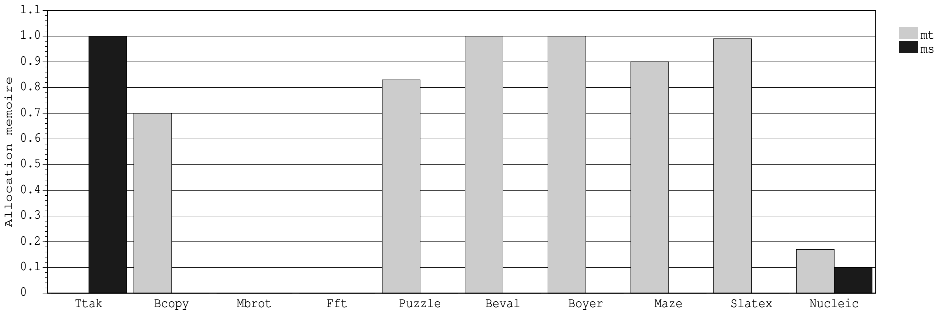


Fig. 3.3 – Les allocations mémoire

Plus les mesures δ_{mt} sont petites, moins les programmes optimisés allouent de mémoire dans le tas. À l'opposé, plus les mesures δ_{ms} sont grandes, plus les versions optimisées allouent en pile. On voit que les programmes qui connaissent la plus forte accélération sont également ceux dont la quantité d'allocations a été la plus réduite par la Sua. Par exemple, sans optimisation, **Ftt** alloue environ 415 Mo alors qu'avec la Sua le même programme n'alloue plus que 174 Ko. Pour le programme **Nucleic** on mesure $\delta_m t = 0.17$ (les allocations sont réduites de 730 Mo à 124 Mo). En revanche, l'allocation en pile semble être très peu appliquée. Le seul test pour lequel elle a un effet est **Ttak**. Ce programme est écrit dans un style « ML » en utilisant des tuples pour passer les arguments. Ce style qui est fantaisiste en Scheme correspond à une réalité en ML car dans ce langage les fonctions n'ont toutes qu'un argument à cause de la currification automatique. Pour passer plusieurs valeurs, il faut soit utiliser des fonctions currifiées soit passer des tuples. L'allocation en pile permet de déplacer du tas vers la pile toutes les constructions des tuples. Cela explique pourquoi **Ttak** est grandement amélioré par la Sua ($\delta_e = 5$ sur Alpha). Si les autres programmes sont très peu sensibles à l'allocation en pile, c'est peut-être à cause du style employé en Scheme. Le style usuel de programmation en Scheme consiste à écrire des fonctions d'allocation qui retournent les valeurs nouvellement allouées. Ainsi, on trouvera fréquemment des programmes comme :

```
1 :  (define (make-user-type x)
2 :     (cons x x))
3 :
4 :  (define (user-type-ref o)
5 :     (car x))
6 :
7 :  (define (foo x)
8 :     (let ((val (make-user-type x)))
9 :        (user-type-ref x)))
```

L'allocation ligne *2* ne peut pas être faite en pile parce que la paire allouée survit à la fonction qui la crée. Probablement l'allocation en pile doit être fortement connectée à l'intégration fonctionnelle présentée § 3.2.

La figure 3.4 permet d'estimer la qualité de l'inférence de types de la SUA. On a mesuré ici le pourcentage de variables et fonctions qui se voient attribuer le type générique *obj*. Plus l'influence est bonne, plus le pourcentage est faible. Sans réelle surprise ces mesures « confirment » les chiffres relevés pour les allocations. Là où la SUA parvient à élire des types spécifiques, elle réussie, en général, à diminuer les allocations et les exécutions sont alors plus rapides.

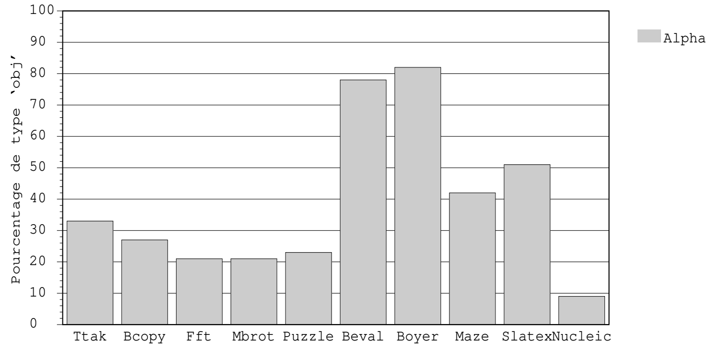


FIG. 3.4 – Le typage

Enfin, les dernières mesures, figure 3.5, que nous avons effectuées permettent d'évaluer l'incidence de la SUA sur les temps de compilation. Nous présentons ici deux mesures qui sont les fractions de temps consacrées à la SUA lors d'une compilation.

$$\delta_{cc} = \frac{\text{temps de la SUA}}{\text{temps de production de C}}, \delta_{co} = \frac{\text{temps de la SUA}}{\text{temps de production de fichier objet}}$$

Comme nous l'avons sous-entendu dans § 3.1.1 la complexité dans le pire des cas de la SUA est élevée. Le nombre maximum d'itérations pour atteindre le point fixe est le produit de la plus grande des tailles des ensembles d'approximations par le nombre de ces ensembles, c'est-à-dire n^2 pour un programme de taille n. Chaque itération est en $\mathcal{O}(n^2)$ car la descente dans le graphe est en $\mathcal{O}(n)$ et les opérations effectuées pour chaque nœud sont aussi en $\mathcal{O}(n)$. Ainsi, la complexité dans le pire des cas, est en $\mathcal{O}(n^4)$! En dépit de cette complexité qui peut sembler impraticable, les mesures montrent que la SUA est raisonnablement rapide. Dans la plus mauvaises configuration ici mesurées, la SUA n'occupe que 20 % du temps de compilation global (le programme **Slatex**).

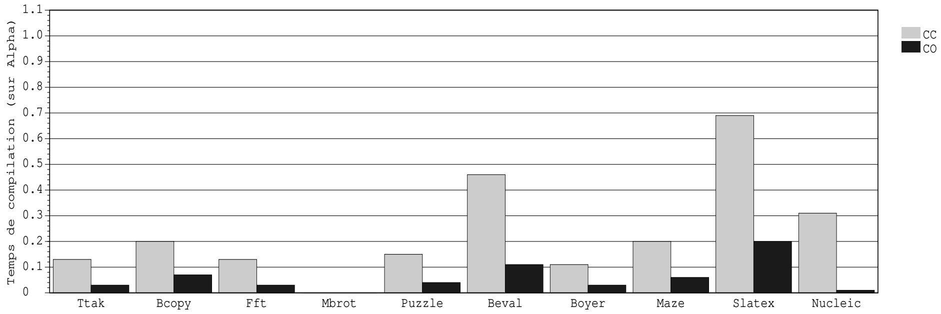


FIG. 3.5 – Temps de compilation

3.1.7 Conclusion sur la SUA

La SUA a été implantée dans Bigloo depuis la version 1.8. Elle a permis d'améliorer les performances du code produit, en particulier pour les applications numériques. Un de nos programmes de test est même 20 fois plus rapide s'il est traité par la SUA ! Par ailleurs, même si la complexité théorique de cette optimisation est élevée nous avons pu constater empiriquement qu'elle est utilisable (le *bootstrap* du compilateur l'utilise). La SUA permet deux optimisations : *1)* la représentation mixte des objets et *2)* l'allocation en pile. L'intérêt de ces deux optimisations dépasse le cadre de Scheme et même celui des langages fonctionnels. Tous les langages permettant des définitions polymorphes et reposant sur une gestion automatique de la mémoire peuvent tirer partie de cette analyse.

3.2 L'intégration fonctionnelle

L'intégration fonctionnelle (abrégée *inlining*) est, tous langages confondus, l'optimisation la plus largement répandue et étudiée. En se limitant aux cinq dernières années et aux principales conférences traitant d'implantation on peut citer les travaux [162, 66, 164, 16, 13, 117]. La pierre que nous avons apportée à cet édifice est présentée dans [207]. Nous avions déjà eu l'occasion d'étudier l'intégration fonctionnelle dans [204] et [205] mais, en poursuivant nos investigations, nous sommes parvenus à de meilleurs résultats.

3.2.1 Les enjeux et les risques de l'intégration fonctionnelle

L'intégration fonctionnelle remplace un appel de fonction par une version modifiée du corps de la fonction appelée. Puisque l'appel de fonction est au cœur même des programmes fonctionnels l'*inlining* est une optimisation très importante pour cette classe de langage. De précédents travaux ont montré que c'est également une optimisation efficace pour les langages à objets [65, 38] et même pour les langages impératifs tels que C [113, 149]. L'*inlining* améliore la qualité du code produit pour trois raisons :

- Le coût d'un appel de fonction varie d'une architecture à l'autre. Sur certaines architectures et dans certains contextes il peut être très important (par exemple pour les machines Sparc dès qu'il y a débordement des fenêtres de registres [251]). L'*inlining* supprime les instructions qui implantent l'appel de fonction (sauvegarde/restitution des registres et les deux branchements). Grâce à l'*inlining*, potentiellement moins de registres devront être copiés en pile (moins de *spill code*) pour chaque fonction.
- Sur les architectures modernes [156, 157, 221] les instructions de branchement peuvent être très coûteuses parce qu'elles peuvent occasionner des ruptures de *pipe-line*. Ainsi, la documentation des constructeurs conseillent d'opérer de l'*inlining* pour les « petites fonctions » [5]. Un des principaux problèmes posés par l'*inlining* est justement l'établissement de critères permettant de définir ce que sont les « petites » fonctions.
- L'*inlining* améliore la portée des autres optimisations. Par exemple, si un appel de fonction est remplacé par une copie du corps de la fonction appelée, alors les paramètres formels de la dite fonction seront remplacés par les valeurs effectives de l'appel de fonction. Si une de ces valeurs est une constante, il est alors probable qu'une optimisation suivant l'*inlining*, telle que la propagation de copie, permettra d'améliorer la version copiée de la fonction.

Comme l'*inlining* duplique des portions de code elle a la contrepartie de potentiellement augmenter la taille du code produit, ce qui a deux effets néfastes :

- Cela allonge les temps de compilation.
- Dans certains cas, cela peut rendre le code produit moins efficace :
  - Avoir des fonctions trop grandes peut, par exemple, mettre en échec l'allocateur de registres. En effet, généralement l'allocation de registres est faite fonction par fonction. La barrière de l'appel de fonction est donc un lieu privilégié par le compilateur pour sauver les registres utilisés en pile. Souvent, les fonctions sont utilisées pour séparer le programme en unités logiques. Sauver les registres avant un appel de fonction peut alors être un choix judicieux (indépendamment des contraintes imposées par le protocole d'appel des plateformes d'accueil). Après intégration fonctionnelle, le compilateur perd la structuration sémantique imposée par les fonctions. Si trop de registres sont utilisés il se peut donc que

le compilateur ne soit plus capable de bien utiliser ceux dont il dispose. Certains de ces effets néfastes sont étudiés avec précision dans [59, 51].
– Avec le décalage sans cesse grandissant entre les performances des processeurs et les accès mémoire, il devient de plus en plus important de placer le code et les données dans les caches mémoire du processeur. De façon évidente, plus le code est gros moins il a de chance de « tenir » dans le cache.

En conséquence, pour les mêmes raisons qui lui permettent d'obtenir de meilleures performances, l'intégration fonctionnelles peut également les dégrader ! On comprend alors pourquoi cette optimisation est fondamentalement heuristique. Il s'agit de trouver un bon compromis, ou, en d'autres termes, de *savoir s'arrêter d'intégrer au bon moment.* Voilà pourquoi l'*inlining*, même si elle est très largement étudiée, reste un sujet ouvert.

Nous allons présenter succinctement dans § 3.2.2 les procédés généralement déployés pour mettre en œuvre l'*inlining*. Pour terminer, dans § 3.2.3, nous présenterons l'intégration fonctionnelle telle qu'elle est actuellement réalisée dans Bigloo. Par la suite, dans § 3.2.4, nous présenterons des travaux plus prospectifs qui, s'ils ne sont pas encore finalisés, illustrent bien les difficultés dues à l'aspect intrinsèquement heuristique de l'*inlining*.

3.2.2 Un survol des techniques d'intégration fonctionnelle

On compte principalement quatre types de critères utilisés pour décider si une fonction doit être intégrée sur un site d'appel.

Les annotations utilisateur. Certains systèmes (par exemple le compilateur C Gcc) ou certains langages (par exemple C++) permettent d'annoter les fonctions qui doivent être intégrées. La décision d'intégration est alors prise par l'utilisateur et on ne peut pas parler d'*optimisation*. Cette technique a le même inconvénient que les annotations d'allocation de registres que permettent les compilateurs C. Si les utilisateurs ne connaissent pas très bien les arcanes du système qu'ils utilisent, alors il est probable que leurs annotations seront plus néfastes que bénéfiques[2]. De plus les annotations d'*inlining* ne peuvent généralement être que portées par les définitions de fonctions et non par les sites d'appels. Soit une fonction est tout le temps intégrée, soit elle ne l'est jamais !

Ils nous semble que les annotations utilisateur ne sont utiles que pour certaines fonctions très particulières. Par exemple, elles permettent au concepteur de bibliothèques d'affiner son implantation mais elles ne sont pas suffisantes. Le compilateur doit être capable de prendre l'initiative d'intégrer des fonctions sur certains sites d'appels privilégiés.

Les critères de taille. C'est une des façons les plus répandues pour décider d'intégrer une fonction. Si le corps d'une fonction est suffisamment petit, c'est-à-dire, s'il est plus petit qu'une constante $\mathcal{K}$ arbitrairement fixée par le compilateur, alors les appels à cette fonction sont intégrés. Cette règle, malgré sa simplicité a un inconvénient majeur. Elle permet de limiter l'accroissement de la taille du code pour les définitions non récursives mais pour ces dernières elle peut faire échouer le compilateur. En général, ceux-ci se munissent d'une règle supplémentaire qui leur permet d'intégrer une fonction si elle est suffisamment petite et si l'appel n'est pas inscrit dans un cycle de récursion. Si cette approche est correcte elle ne donne pas de très bons résultats. En effet, imaginons une fonction définie par :

```
(define (plus x1 x2 x3 ... xn)
   (+ x1 x2 x3 ... xn))
```

Dès que n est supérieure à $\mathcal{K}$ notre fonction `plus` n'est plus intégrée alors qu'évidemment cette intégration serait bénéfique.

Ainsi, certains systèmes adoptent une version améliorée de la précédente règle : si intégrer un appel de fonction produit une augmentation *limitée* de la taille globale du code à compiler, alors

[2] C'est pour cette raison précise que le compilateur gcc ignore les annotations `register` dès qu'on utilise l'option `-O2`.

la décision d'intégration est prise. Nous verrons par la suite dans § 3.2.3 que pour Bigloo nous utilisons une variante de cette règle.

Une estimation des gains. Une technique assez différente est parfois mise en œuvre (par exemple dans les systèmes SML/NJ [8] et Self [65]) qui consiste à estimer les gains que produira l'intégration d'une fonction. C'est-à-dire qu'on compare une estimation du coût de l'appel de fonction cumulé au coût de l'évaluation du corps de la fonction, à l'estimation d'une possible compilation de la copie du corps de la fonction si elle était intégrée. Pour cette deuxième estimation, la difficulté est d'évaluer correctement l'impact des autres optimisations du compilateur appliquées sur la copie. Si cette technique semble intéressante nous doutons qu'elle donne de très bons résultats. De nombreuses optimisations sont globales et ont une complexité assez importante (par exemple l'optimisation décrite dans § 3.1). Il est irréaliste de les appliquer à chaque fois qu'on a besoin de calculer une estimation de gain d'intégration. En conséquence cette technique d'estimation ne peut concerner que les optimisations simples qui malheureusement ne sont généralement pas celles qui améliorent le plus sensiblement les performances du code produit.

Les décisions retardées. Avant de décider d'intégrer des fonctions, l'utilisateur doit compiler une première version de son programme et l'exécuter sur un jeu de données. Ensuite, dans une seconde étape, il lui faut procéder à une nouvelle compilation qui prend en paramètre supplémentaire le relevé d'exécution collecté. Cette solution peut donner d'excellents résultats puisqu'elle ne travaille pas à *l'aveugle*. Le compilateur décide d'intégrer une fonction sur un site précis parce que lors de l'exécution d'échantillonnage, ce site a été souvent utilisé ou parce que le ratio entre le temps passé à appeler la fonction et le temps passé à exécuter son corps s'avère favorable à son intégration.

Nous n'avons pas retenu cette solution pour Bigloo parce qu'elle nous semble nécessiter trop de collaboration de la part de l'utilisateur. Tout d'abord, celui-ci doit concevoir un jeu de données représentatif des exécutions. Cette tâche peut être arbitrairement compliquée. Ensuite, parce que plusieurs compilations sont requises. D'après notre expérience ces optimisations qui nécessitent plusieurs compilations sont très rarement utilisées[3].

3.2.3 L'intégration fonctionnelle de Bigloo

Afin de décider s'il doit intégrer une fonction sur un site d'appel, Bigloo se base uniquement sur des informations statiques. La décision dépend de la taille de la fonction appelée, de la taille de l'appel (c'est-à-dire du nombre de paramètres effectifs) et enfin, de l'endroit où est situé l'appel. La technique utilisée dans Bigloo n'emploie ni annotation de l'utilisateur ni informations collectées lors d'une exécution préalable. Elle s'inspire des travaux d'A. Appel [8, page 92]. Son principe consiste donc à tolérer une expansion limitée par site d'appel et à appliquer récursivement l'*inlining* aux corps intégrés. Lorsque l'algorithme est appliqué récursivement sur des corps de fonctions intégrées alors le facteur de tolérance devient de plus en plus petit. Plus l'algorithme est appliqué profondément, plus il est réticent à intégrer une nouvelle fonction.

Voici un exemple permettant d'illustrer le fonctionnement de l'algorithme :

[3] Par exemple, quels utilisateurs de machines à base de processeurs Alpha ont déjà utilisé `cord` et `ftoc` qui pourtant peuvent faire économiser, d'après nos mesures, jusqu'à 20 % du temps d'exécution ?

```
1 :  (define (inc-fx x)
2 :     (+ x 1))
3 :
4 :  (define (inc-fl x)
5 :     (inc-fx (inexact->exact x)))
6 :
7 :  (define (inc x)
8 :     (if (fixnum? x)
9 :         (inc-fx x)
10 :        (inc-fl x)))
11 :
12 : (define (foo x)
13 :    (inc x))
14 :
15 : (foo 4)
```

Supposons qu'initialement, c'est-à-dire à une profondeur de récursion de 0, l'algorithme permette à un site d'appel de devenir quatre fois ($\mathcal{K}_0 = 4$) plus gros après intégration et qu'à chaque appel récursif de l'algorithme (à chaque fois qu'on procède à l'*inlining* des corps des fonctions intégrées) on divise ce facteur d'expansion par deux. L'appel situé ligne *13* a une taille de 2 (1 pour la fonction appelée, 1 pour l'argument). La taille du corps de `inc` est 7 (1 pour la construction conditionnelle (ligne *8*), 2 pour le test et 2 pour chaque branche (ligne *9* et *10*)). Le corps de `inc` est donc suffisamment petit ($7 \leq 2 \times \mathcal{K}_0$) et la fonction est donc intégrée ligne *13*. Avant de remplacer l'appel par une copie du corps de `inc`, le corps de cette fonction est à son tour intégré mais avec un nouveau facteur d'expansion $\mathcal{K}_1 = \mathcal{K}_0/2$. C'est-à-dire que lorsque l'algorithme d'intégration inspecte la copie de la ligne *9*, $\mathcal{K}_1 = 2$. Le corps de `inc-fx` a une taille de 3 (1 pour l'opérateur `+` ligne *2*, et 1 pour chaque argument). L'appel a, lui, une taille de 2 et il est donc intégré. Aucune intégration ne peut plus avoir lieu sur le corps de `inc-fx` puisque `+` est un opérateur primitif. L'algorithme inspecte alors la copie de la ligne *10* avec le facteur d'expansion $\mathcal{K}_1$. La fonction `inc-fl` est intégrée puisque son corps a une taille de 3 et l'algorithme est donc invoqué récursivement sur son corps. Le nouveau facteur d'expansion $\mathcal{K}_2$ vaut à ce moment 2. Ainsi l'appel à `inc-fx` ligne *5* ne peut plus être intégré parce qu'il dépasserait l'expansion tolérée ($3 > \mathcal{K}_2$). Une fois l'intégration totalement appliquée notre exemple sera transformé en :

```
(define (inc-fx x)
   (+ x 1))

(define (foo x)
   (if (fixnum? x)
       (+ x 1)
       (inc-fx (inexact->exact x))))

(foo 4)
```

Les fonctions inutilisées (`inc` et `inc-fl`) ont été supprimées et ainsi, la taille totale de notre programme *après* intégration fonctionnelle est plus petite qu'*avant*. Les mesures expérimentales que nous avons opérées (cf. § 8.5, page 155) montrent que ce phénomène est en fait fréquent. Dans de nombreuses situations, l'*inlining* diminue la taille des programmes à compiler.

Le calcul du facteur d'expansion

De façon évidente les résultats de l'algorithme sont fortement influencés par la fonction de calcul de la régression du facteur d'expansion. Dans l'exemple du § 3.2.3 nous avons utilisé une régression linéaire $\mathcal{K}_n = \mathcal{K}_{n-1}/2$. Pour Bigloo nous avons procédé à plusieurs expériences pour déterminer une « bonne » fonction de régression. En particulier, nous avons testé quatre fonctions :

décrément : $\mathcal{K}_n = \mathcal{K}_{n-1} - \mathcal{N}$

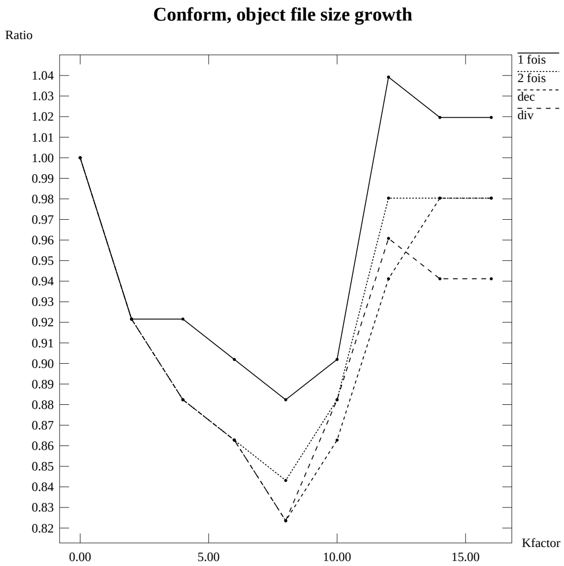
division : $\mathcal{K}_n = \mathcal{K}_{n-1}/\mathcal{N}$

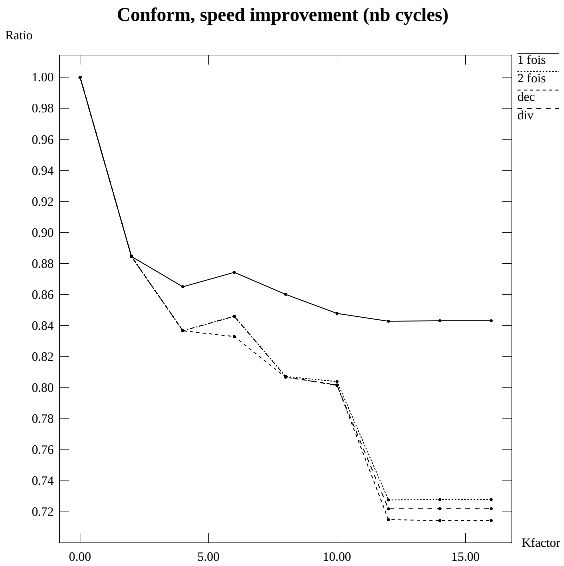
1 fois : $\mathcal{K}_n =$
si $n = 0$ *alors* $\mathcal{K}_i$ *sinon* 0

2 fois : $\mathcal{K}_n =$
si $n = 0$ *alors* $\mathcal{K}_i$ *sinon* *si* $n = 1$ *alors* $\mathcal{K}_i/\mathcal{N}$ *sinon* 0

Ces deux dernières permettent d'évaluer l'intérêt des applications récursives de l'intégration sur les fonctions intégrées. En effet, la solution **1 fois** n'intègre que les fonctions au premier niveau et la solution **2 fois** n'intègre que les fonctions du premier et second niveau de récursion.

Pour chacune de ces fonctions de régression nous avons mesuré l'augmentation de la taille du code après compilation complète et l'accélération qu'elles permettent. D'après nos mesures, la valeur de $\mathcal{N}$ n'est pas déterminante et donc nous ne présentons les mesures que pour $\mathcal{N} = 2$ car empiriquement cette valeur s'est avérée satisfaisante. Les mesures complètes pour divers programmes de tests sont présentées § 8.5, page 155 ; étudions ici seulement les mesures montrant l'influence des fonctions de régression sur le programme test **Conform** :





En abscisse sont présentées les variations de la valeur initiale de $\mathcal{K}$ (noté $\mathcal{K}_i$ dans **1 fois** et **2 fois**). En ordonnée sont représentées les tailles et les temps d'exécution obtenus.

Il est particulièrement visible sur cet exemple que la solution **1 fois** produit les moins bons résultats. Cela montre qu'un algorithme récursif (un algorithme qui tente d'intégrer le résultat d'une intégration) donne de meilleurs résultats. Par contre, on voit assez nettement sur cet exemple (qui est révélateur) que les 3 autres solutions produisent des résultats très proches. Cela démontre le résultat assez prévisible que seuls les premiers « pas » d'intégration sont importants. Les fonctions utilisées pour **décrément** et **division** permettent une expansion assez large même si nous n'avons jamais noté expérimentalement ce phénomène (dans le pire des cas, l'expansion de **division** est $2^{\frac{1}{\mathcal{N}}*\log_2 k^2}$). L'expansion occasionnée par la solution **2 fois** est beaucoup plus limitée ($k^2/\mathcal{N}$ dans le pire des cas). Ainsi elle semble représenter le meilleur compromis.

Comment opérer les intégrations ?

Comme nous l'avions montré dans [204] et [213], décider *quand* intégrer une fonction est une première étape mais il reste alors à décider comment procéder à cette intégration. Nous avions proposé deux schémas d'intégration, l'un pour l'intégration des fonctions simples et l'autre pour les fonctions récursives. Le premier schéma est largement répandu dans les divers compilateurs. Il consiste en une β-réduction :

$$\textit{inline-plain}\left[\!\!\left[\begin{array}{l}\texttt{(define (}\mathit{fun}\ a_0\ \ldots\ a_n\texttt{)}\\ \quad \mathit{Exp}\texttt{)}\\ \\ \ddots\\ \quad\boxed{\texttt{(}\mathit{fun}\ v_0\ \ldots\ v_n\texttt{)}}\\ \qquad\quad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{(define (}\mathit{fun}\ a_0\ \ldots\ a_n\texttt{)}\\ \quad \mathit{Exp}\texttt{)}\\ \\ \ddots\\ \quad\boxed{\begin{array}{l}\texttt{(let ((}a_0\ v_0\texttt{)}\ \ldots\ \texttt{(}a_n\ v_n\texttt{))}\\ \quad\mathit{Exp}\texttt{)}\end{array}}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

Pour les fonctions récursives nous avons proposé un schéma différent qui ne repose pas sur une β-réduction mais sur une α-conversion :

$$\textit{inline-rec}\left[\!\!\left[\begin{array}{l}\texttt{(define (}\mathit{fun}\ a_0\ \ldots\ a_n\texttt{)}\\ \quad \mathit{Exp}\texttt{)}\\ \\ \ddots\\ \quad\boxed{\texttt{(}\mathit{fun}\ v_0\ \ldots\ v_n\texttt{)}}\\ \qquad\quad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{(define (}\mathit{fun}\ a_0\ \ldots\ a_n\texttt{)}\\ \quad \mathit{Exp}\texttt{)}\\ \\ \ddots\\ \quad\boxed{\begin{array}{l}\texttt{(letrec ((}\mathit{fun}\ \texttt{(lambda (}a_0\ \ldots\ a_n\texttt{)}\ \mathit{Exp}\texttt{)))}\\ \quad\texttt{(}\mathit{fun}\ \ v_0\ \ldots\ v_n\texttt{))}\end{array}}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

L'intérêt de cette deuxième transformation est qu'elle permet l'application ultérieure d'autres optimisations. En particulier, juste après l'intégration suivant le schéma *inline-rec*, il est facile de faire une analyse pour déterminer si certains paramètres de la fonction sont invariants. Le compilateur effectue une propagation de constante pour ceux-ci. 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))
```

Étudions l'*inlining* de la fonction récursive `map` dans la fonction `map-succ`. La transformation *inline-rec* 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 soumis de nouveau 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 : *1)* 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), *2)* 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.

L'intégration fonctionnelle et le dépliage des boucles

Dans [207] nous avons tenté d'étendre ce schéma d'intégration fonctionnelle au dépliage des boucles (*loop unrolling*). L'idée est assez simple : pour déplier une boucle, il suffit d'intégrer une première fois la fonction qui implante la boucle en suivant le schéma *inline-rec* puis d'appliquer récursivement l'intégration fonctionnelle sur le corps ainsi dupliqué en ne s'autorisant que le schéma *inline-plain* sur ce code. La transformation appliquée peut alors être schématisée par :

```
                 [[ (define (fun a0 ... an)   ]]     [[ (define (fun a0 ... an)
                 [[    ⋱                      ]]     [[    ...)
                 [[      (fun w0 ... wn)      ]]     [[
                 [[          ⋱)               ]]     [[ ⋱
inline-unroll    [[                           ]]  =  [[   (letrec ((fun (lambda (a0 ... an)
                 [[ ⋱                         ]]     [[                    ⋱
                 [[   (fun v0 ... vn)         ]]     [[                      (let ((a0 w0) ... (an wn))
                 [[      ⋱                    ]]     [[                         ⋱
                                                     [[                           (fun w'0 ... w'n)
                                                     [[                               ⋱)
                                                     [[                                          ⋱)))
                                                     [[     (fun  v0 ... vn))
                                                     [[                    ⋱                          ]]
```

Appliquée sur l'exemple précédent de la fonction `map-succ`, cette transformation produit :

```
(define (map-succ l)
   (letrec ((map (lambda (l1)
                    (if (null? l1)
                        '()
                        (cons (+ 1 (car l1))
                              (let ((l2 (cdr l1)))
                                 (if (null? l2) '()
                                     (cons (+ 1 (car l2))
                                           (map (cdr l2))))))))))
      (map l)))
```

Nous avons alors mesuré expérimentalement la pertinence des trois transformations *inline-plain*, *inline-rec* et *inline-unroll* en les appliquant à une grande série de tests (cf. § 8.5, page 155). Il nous est apparu que seules les transformations *inline-plain* et *inline-rec* produisent de bons résultats. Visiblement la transformation *inline-rec* est importante pour minimiser la taille du code produit par l'*inlining*. C'est même l'intérêt principal de cette transformation : elle diminue plus la taille du code qu'elle n'améliore les performances. En revanche, la transformation *inline-unroll* est décevante parce qu'elle n'améliore les performances que pour deux tests seulement. Ce phénomène est difficile à expliquer mais on peut mettre en avant deux hypothèses :

- Généralement les boucles ont un seul paramètre (la variable de boucle) et donc seules les très petites boucles peuvent être déroulées (les autres dépassent très vite le facteur d'expansion toléré). Si cette hypothèse était juste cela signifierait que l'*inlining* et le dépliage sont des transformations différentes qui nécessitent des critères différents, et peut-être même des temps d'application différents dans le compilateur.
- Le dépliage est une optimisation assez mal comprise. Comme le notent les concepteurs de gcc dans [225], « *[general loop unrolling] usually makes programs run more slowly* ». Les raisons sont diffuses mais on peut avancer que le dépliage a probablement un effet néfaste sur l'allocateur de registres. Un dépliage intempestif augmente le nombre de registres vivants à l'intérieur d'une boucle et donc il peut avoir tendance à augmenter le *spill code* à l'intérieur de ces boucles. Ce risque est moins grand pour l'intégration. Comme explication supplémentaire, il est probable que les architectures modernes sont très sensibles aux branchements des programmes parce qu'ils cassent les traitements en cours dans les *pipelines*. Pour corriger ce problème tous les processeurs modernes implantent de la prédiction de branchement. C'est-à-dire que le processeur met en œuvre un arsenal de techniques pour prédire l'adresse de l'instruction suivant un branchement. En général, ces techniques utilisent de petits caches qui prédisent convenablement un nombre assez petit de sauts. Déplier les boucles augmente le nombre statique de branchements différents qu'elles contiennent (même si leur nombre exécutés est le même). Par exemple, celui qui indique la poursuite de la boucle ne sera plus implanté par *un* saut mais par plusieurs. Il apparaît donc que plus de ressources du processeur devront être mobilisées pour assurer une bonne prédiction pour tous les branchements des boucles dépliées.

Conclusion sur l'intégration fonctionnelle dans Bigloo

Les algorithmes de décision et les techniques d'intégration utilisés dans Bigloo permettent une amélioration sensible des performances. Sur les programmes tests que nous avons utilisés nous avons noté, en moyenne, une diminution de 15 % des temps d'exécution. Cette efficacité ne se fait pas au détriment de la taille du code produit. Au contraire, après *inlining* elles sont souvent plus petites. Enfin, les algorithmes utilisés sont simples à mettre en œuvre puisque la taille totale de cette optimisation représente moins de 4 % du code complet du compilateur.

3.2.4 L'intégration fonctionnelle de bas niveau

Si la solution actuellement utilisée dans Bigloo produit des résultats satisfaisants avec des exécutables optimisés plus petits et plus rapides, il subsiste une zone d'ombre gênante pour l'esprit. La version courante permet de gagner 15 % mais est-ce une limite intrinsèque de l'intégration ou est-il possible d'obtenir de meilleurs résultats ? Comment évaluer la pertinence de notre processus d'*inlining* ? Pour cela nous avons conduit des recherches qui se sont avérées être décevantes *in fine* mais qui permettent néanmoins d'avoir une compréhension plus fine des phénomènes mis en jeu par l'*inlining*.

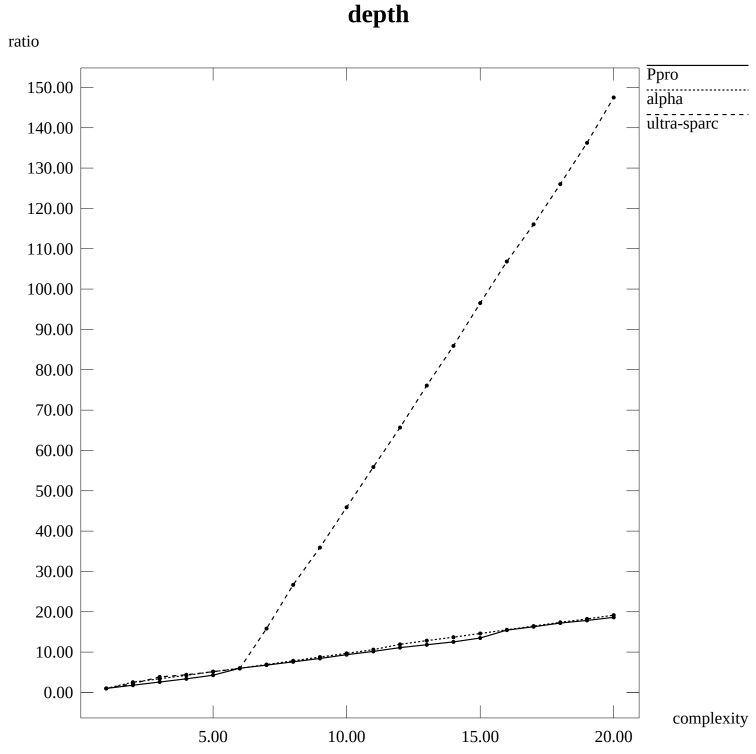


FIG. 3.6 – L'impact de la profondeur sur trois architectures

Nous avons avancé dans § 3.2.3 des explications mettant en lumière la raison pour laquelle le dépliage de boucles ne donne pas de résultats satisfaisants. Nous avons mis en avant l'idiosyncrasie des processeurs. L'intégration, si elle semble devoir toujours améliorer les performances puisqu'elle supprime des instructions à exécuter, peut en fait ralentir l'exécution des programmes parce que ceux-ci, après *inlining*, sont moins bien adaptés aux machines dont nous disposons à ce jour. Si c'est effectivement les caractéristiques matérielles des machines qui sont déterminantes pour l'*inlining*, alors très probablement cette optimisation doit tenir compte des caractéristiques des dites machines. Nous avons tenté d'estimer empiriquement l'importance de l'architecture cible pour l'*inlining*.

Le coût d'un appel de fonction

La première de nos investigations a porté sur l'évaluation du coût de l'appel de fonction. Ce coût dépend de quatre critères principaux :

- La plate-forme matérielle utilisée.
- La position du site d'appel (par exemple sa profondeur d'appel).
- Le nombre d'arguments.
- Le contre coup de l'appel (par exemple, plus il y a de variables vivantes autour d'un appel de fonction, plus il « coûte » cher).

Pour estimer l'importance de la position *dynamique* du site, nous avons utilisé un programme C dont le principe est d'appeler un nombre constant de fois une fonction récursive dont la profondeur d'appel varie. Si le temps d'exécution augmente linéairement et proportionnellement à la profondeur d'appel, alors nous dirons que le coût d'un appel de fonction est indifférent à sa position *dynamique*. La figure 3.6 présente un relevé de mesures sur trois architectures différentes et révélatrices des machines modernes : un Compaq Alpha, un Sun Sparc et un Intel PentiumPro. La profondeur d'appel varie de 0 à 20. Hormis pour les processeurs Sparc, la profondeur *dynamique* d'un appel est sans incidence. Cette première série de mesures montre de façon définitive qu'un algorithme d'intégration fonctionnelle doit être réglé différemment pour Sparc. En effet, sur cette architecture, un appel de fonction a un coût exorbitant s'il occasionne un débordement des fenêtres de registres [235]. Cette figure est d'ailleurs tellement flagrante qu'elle permet même de mesurer le nombre de fenêtres de registres que compte la machine utilisée pour ces mesures. La courbe du Sparc présente une cassure nette pour une profondeur de 6, c'est-à-dire, en fait, lors du $7^{\text{ème}}$ appel (6 appels récursifs plus l'appel au point d'entrée `main`). Un algorithme d'*inlining* pour Sparc a alors probablement intérêt à tolérer un plus grand facteur d'expansion que sur les autres plateformes.

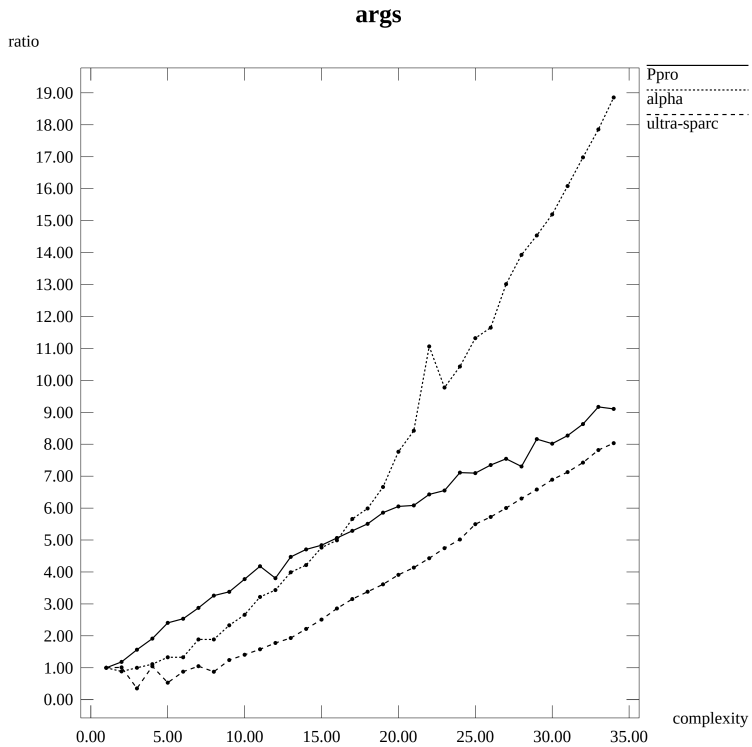


FIG. 3.7 – L'impact du nombre d'arguments sur trois architectures

Pour notre deuxième série de mesures, nous avons tenté d'évaluer l'impact du nombre d'arguments des appels de fonctions. C'est-à-dire que nous avons cherché à savoir si le coût d'un appel de fonction croît linéairement par rapport au nombre de paramètres de la fonction. Nous avons donc conçu un petit générateur de programmes C qui produit des programmes C appelant un nombre fixe de fois une fonction qui compte de plus en plus d'arguments. La figure 3.7 présente ces mesures où le nombre d'arguments varie de 0 à 35. Comme pour la position dynamique, le nombre d'arguments n'a pas la même incidence sur toutes les architectures. Ici, c'est l'Alpha qui est le plus sensible au nombre d'arguments, de façon d'ailleurs relativement inexplicable. Le protocole d'appel des Alpha stipule que les 6 premiers arguments doivent être placés dans des registres. La courbe de l'Alpha semble en effet s'infléchir au delà de 6 arguments sans que cela soit linéaire. Les Sparc utilisent, pour leur part, 7 registres, pour passer les premiers arguments des fonctions. La courbe pour cette architecture est alors un peu chaotique pour des petites valeurs du nombre d'arguments puis elle croît de façon uniforme dès que la pile est utilisée.

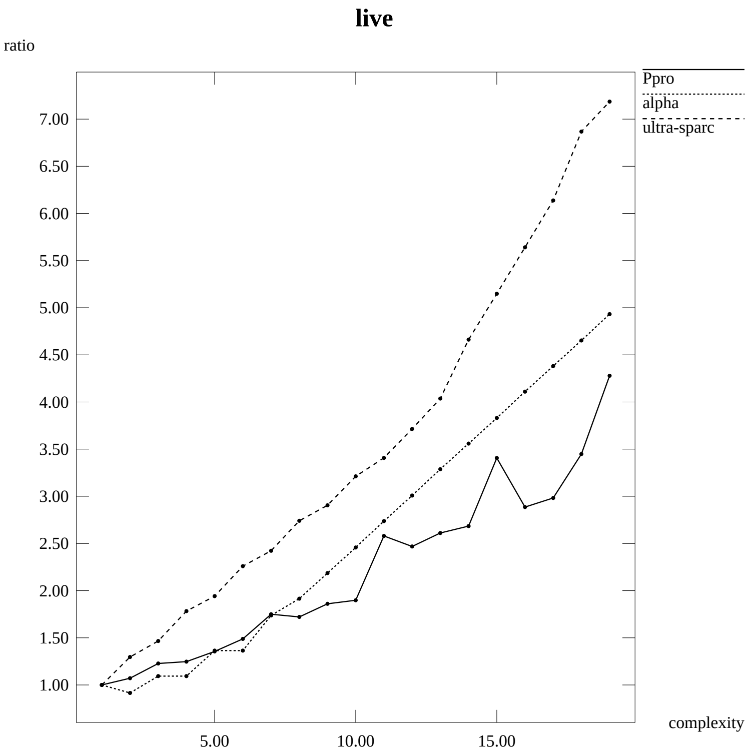


FIG. 3.8 – L'impact du nombre de variables vivantes sur trois architectures

Enfin, nous avons cherché à évaluer l'impact du nombre de variables vivantes lors d'un appel. Pour cela nous avons écrit un dernier générateur de programmes C qui produit des programmes appelant un nombre fixe de fois une fonction qui se trouve au milieu d'un calcul utilisant de plus en plus de temporaires. La figure 3.8 présente ces mesures où le nombre de temporaires vivants varie de 0 à 20. Pour cette dernière série de mesures, les résultats sont plus uniformes même si on peut noter que c'est pour le Sparc que la quantité de variables vivantes a le plus d'importance. Là non plus nous ne savons pas expliquer ce résultat parce que le Sparc est le processeur qui dispose du plus grand nombre de registres pour sauvegarder les temporaires. Normalement les Sparc devraient être moins sensibles que les autres architectures à la quantité de variables vivantes !

Le but de ces mesures n'est pas de classer les machines les unes par rapport aux autres mais de démontrer l'influence de l'architecture sur le coût de l'appel de fonction. Comme la diminution de ce coût est l'objet de l'*inlining*, nous pouvons conclure que cette optimisation ne peut pas se faire sans connaissances fines de la machine cible.

L'intégration fonctionnelle exhaustive

La deuxième étape de notre expérimentation a été d'estimer les maxima et minima qu'on peut attendre de l'*inlining*. C'est-à-dire, à partir d'un programme donné, quelle est la meilleure combinaison d'*inlining* pour ce programme (celle qui donnera le temps d'exécution minimal) et quelle est celle qui donnera le plus mauvais résultat. Parce que l'*inlining* dépend des caractéristiques de bas niveau des machines nous avons décidé de mener ces investigations en C plutôt qu'en Scheme afin de diminuer la distance entre le langage source de nos programmes de test et les instructions assembleur exécutées. Nous avons alors conçu un *inliner* C qui prend en entrée deux paramètres : *1)* un programme à optimiser et *2)* un fichier de configuration décrivant précisément parmi tous les appels de fonctions que contient le programme quels sont ceux qui doivent être intégrés. Ensuite, nous avons écrit un second utilitaire qui, partant d'un source C, produit des fichiers de configuration pour toutes les configurations d'*inlining* possibles. Ainsi, muni de nos deux outils, nous avons pu, pour un source C donné, mesurer l'impact de tous les *inlining* possibles de ce programme. Nous en avons sélectionné quelques un assez simples car les gros programmes ont un nombre trop important de combinaisons possibles. Pour notre programme d'énumération de configurations il est possible de régler extérieurement la profondeur maximale des *inlining*. Si cette profondeur est assez grande, alors même des programmes très simples peuvent avoir un grand nombre de configurations parce que les fonctions récursives sont intégrées en profondeur. La figure 3.9 présente les programmes de test utilisés ainsi que leur taille en nombre de lignes, la profondeur maximale d'*inlining* choisie et le nombre de configurations calculées par notre générateur.

	Caractéristiques		
prgm	Taille en lignes	Profondeur des récursions	Nombre de combinaisons
fib	19	8	82
fact	37	25	27
queens	187	30	187
permute	103	1	545
ackerman	30	8	730

FIG. 3.9 – Les programmes de test et leurs caractéristiques

Puisque les configurations peuvent décrire des chaînes d'intégration récursives, il faut noter que même les petits programmes tels que **fib** ou **ackerman** sont révélateurs car en faisant varier la profondeur maximale d'*inlining* on peut les faire grossir arbitrairement (plus on tolère des intégrations profondes, plus on duplique du code source et plus le programme à compiler devient gros). Ainsi, avec une profondeur de 8, **ackerman** qui est un petit programme nous permet de tester 730 combinaisons ! Nous avons alors compilé puis exécuté toutes les combinaisons de tous ces programmes sur nos trois architectures de référence[4].

La figure 3.10 présente pour chacun des programmes les meilleurs et les plus mauvais résultats obtenus. Les valeurs présentées sont des ratios par rapport au temps de référence sans *inlining*. Ainsi un ratio de 0.5 signifie un temps d'exécution 2 fois plus court et un ratio de 2.0 dénote un temps d'exécution 2 fois plus long.

	Processeurs					
	Alpha (21164)		Sparc (Ultra)		Pentium II (Klamath)	
prgm	*min*	*max*	*min*	*max*	*min*	*max*
fib	0.79	1.19	0.63	1.00	0.62	1.00
fact	0.95	1.05	0.22	1.00	0.48	1.00
queens	0.98	1.53	0.98	1.82	0.99	1.31
permute	0.65	1.00	0.50	1.01	0.56	1.17
ackerman	0.57	1.53	0.09	1.00	0.57	1.00

FIG. 3.10 – Les maxima et minima d'*inlining*.

Sans surprise, les gains produits par l'*inlining* sont très dépendants du source sur lequel on applique l'optimisation. Nous ne savons d'ailleurs ni prévoir ni expliquer pourquoi certains programmes « réagissent » mieux que d'autres. Par exemple, **ackerman** et **fib** semblent être assez proches et pourtant l'amélioration produite par l'*inlining* est très différente. Le deuxième résultat est la confirmation de ce que nous avions avancé dans § 3.2.1 : l'*inlining* a des effets très différents d'une architecture à l'autre. Par exemple, pour **ackerman**, le gain maximal sur un processeur Sparc est environ d'un facteur d'accélération 10 alors qu'il n'est que légèrement inférieur

[4]Ce qui fait un total de 4632 compilations et, comme chaque configuration est exécutée trois fois consécutives afin de minimiser les risques d'erreurs de mesures dus à la charge de la machine, un total de 13896 exécutions ! Nous remercions au passage l'Institut Pasteur de Paris qui nous a permis un *squat* en règle de ses machines à base de processeurs Alpha.

à 2 sur un processeur Alpha. Par ailleurs, un mauvais *inlining* peut largement allonger le temps d'exécution d'**ackerman** sur Alpha alors que dans le pire des cas sur Sparc il est sans effet.

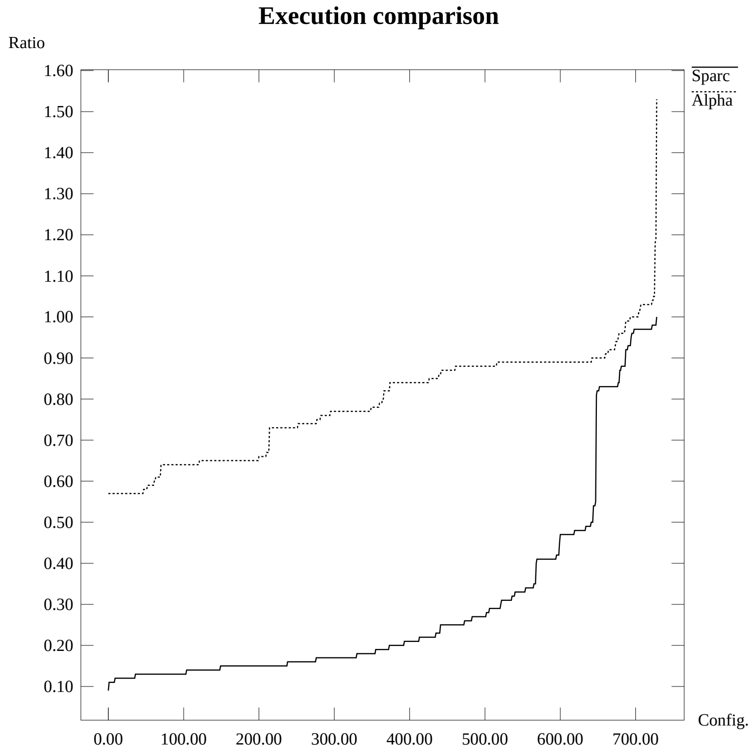


FIG. 3.11 – L'impact des différentes configurations sur Alpha et Sparc

Pour déterminer l'exécution la plus courte et la plus longue, nous avons compilé et exécuté $\mathcal{N}$ versions de nos programmes source, où $\mathcal{N}$ est le nombre de configurations d'*inlining*. Nous avons alors trié par ordre décroissant les temps d'exécution obtenus pour chaque configuration. La figure 3.11 présente ainsi les temps d'exécution des 730 configurations d'**ackerman** pour les processeurs Alpha et Sparc. Ces deux courbes sont dessinées en triant séparément les temps des Alphas et des Sparcs. Supposons que les configurations sont nommées k_0, k_1, k_2, ..., k_n. La figure 3.11 présente, pour les processeurs Alpha et Sparc deux tris différents de ces configurations. C'est-à-dire qu'elle présente les courbes $\mathcal{K}^{sparc} = (k^{sparc}_{min}, k^{sparc}_{min+1}, ..., k^{sparc}_{max})$ et $\mathcal{K}^{alpha} = (k^{alpha}_{min}, k^{alpha}_{min+1}, ..., k^{alpha}_{max})$. Par exemple, le tri décroissant sur Sparc pourrait être ($\mathcal{K}^{sparc} = k_{345}, k_{234}, k_{123}, k_{99}, ...$). Nous avons alors « croisé » nos configurations. La figure 3.12 présente les temps sur les processeurs Alpha correspondant aux différentes configurations d'*inlining* triées selon l'ordre décroissant obtenu pour les processeurs Sparc. C'est-à-dire qu'au lieu de montrer les temps d'exécution des processeurs Alpha pour le tri des configurations $\mathcal{K}^{alpha}$, nous présentons les temps d'exécution des processeurs Alpha pour le tri $\mathcal{K}^{sparc}$. On voit alors nettement que les « bonnes » configurations du Sparc ne sont pas celles de l'Alpha et réciproquement. Nous avons noté que ce phénomène est général. Les « bonnes » configurations pour une architecture X ne sont pas forcément les bonnes configurations pour une architecture Y. La figure 3.13 montre les temps sur l'Alpha pour le programme **fib** trié en fonction de l'ordre des Pentiums.

Ces résultats permettent d'affirmer que l'*inlining* est principalement une optimisation de bas niveau, c'est-à-dire, très liée à l'architecture de la machine cible. Cette conclusion nous laisse penser que dans un contexte comme celui de Bigloo où l'accent est mis sur les optimisations de haut niveau, il faut surtout s'attacher lors de l'intégration fonctionnelle à ne pas augmenter la taille du code à compiler. Une seconde passe d'intégration plus fine et utilisant les caractéristiques matérielles de la machine cible peut éventuellement intervenir en fin de compilation. Si ces travaux sur l'*inlining*

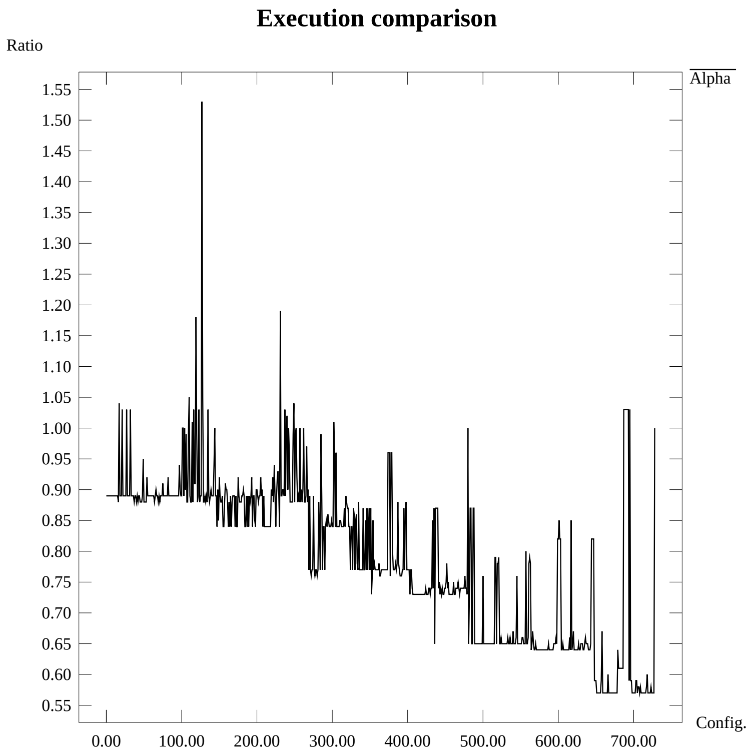


FIG. 3.12 – L'ordre optimal des Sparcs appliqué aux Alpha

nous ont permis de démontrer que cette optimisation doit tenir compte des architectures cibles, nous sommes en revanche dans l'incapacité de dégager un critère expliquant pourquoi certaines configurations d'*inlining* donnent de meilleurs résultats que d'autres. En utilisant le système de mesures dynamiques Atom disponible sur les Alpha [224] nous n'avons pu trouver d'explication aux bons ou mauvais comportements de certaines configurations. En particulier, nous n'avons pu estimer l'importance de l'allocation de registres, de l'utilisation des caches mémoire de données et de code ni les problèmes de prédiction des branchements sur l'*inlining*. Nos tentatives d'analyses sur processeurs Pentium n'ont pas été plus concluantes. Nous ne sommes donc toujours pas capables d'avancer des critères objectifs indiquant comment l'*inlining* doit être appliquée. Comme tous les précédents travaux sur le sujet nous devons nous en remettre à des approches heuristiques.

3.3 Conclusion

Bigloo est un compilateur Scheme qui produit des exécutables aux performances « raisonnables », c'est-à-dire du même ordre de grandeur que C. Pour atteindre ce niveau de performances il a fallu étudier deux aspects de la compilation :

La compilation du contrôle. On cherche en priorité à ne pas allouer de structures dans le tas pour encoder les fonctions. On procède à de l'intégration fonctionnelle pour ne pas pâtir du style de programmation en vigueur dans les langages fonctionnels qui multiplie l'usage de petites fonctions [101].

La compilation des données. Scheme, et dans une moindre mesure ML, comme quelques langages à objets (Smalltalk, Java, ...) permettent des fonctions polymorphes et l'inspection dynamique du type des objets. Ces deux traits compromettent l'efficacité des représentations de données pour ces langages. Il nous a donc fallu concevoir une analyse. Elle détecte les zones des programmes sur lesquelles une représentation efficace est possible et les zones

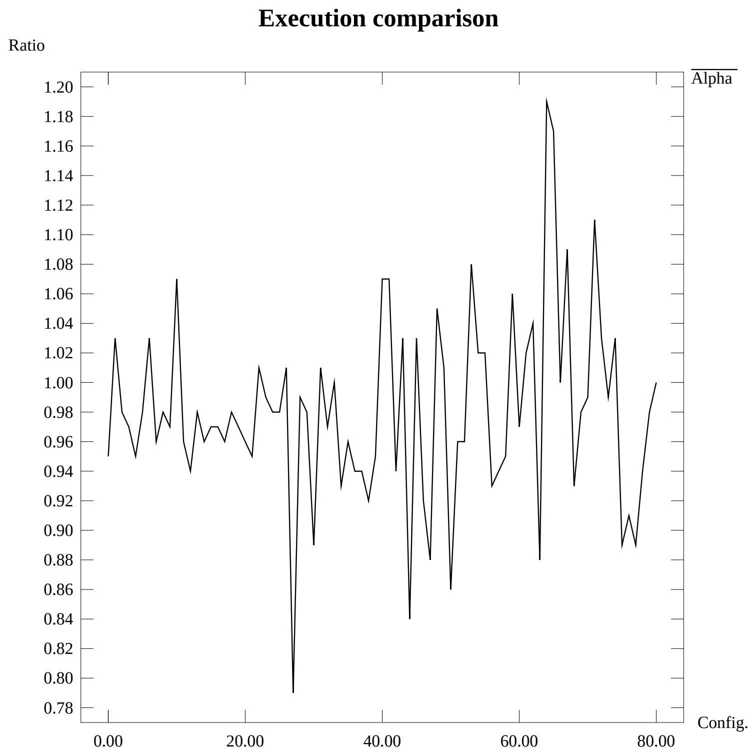


FIG. 3.13 – L'ordre optimal des Pentiums appliqué aux Alphas

sur lesquelles une compilation standard doit être appliquée. Dans ce cas, les données doivent contenir assez d'informations et être représentées dans un format unique pour pouvoir être utilisées sans restriction par des fonctions polymorphes ou par des fonctions inspectant les types dynamiques.

Ces deux types d'optimisations se retrouvent dans quelques-uns des compilateurs optimisants pour LFs (par exemple Caml Special Light renommé O'Caml ou Bigloo) et elles permettent aux LFs de disposer d'implantations compétitives.

3.4 Perspectives

Dès 1978 G. Steele avait jeté les bases de la compilation moderne des langages fonctionnels avec son compilateur Rabbit [227]. La voie ainsi ouverte a été poursuivie dès le début des années 1980 par J.A. Rees et N.I. Adams avec le système T, qui débouchera quelques années plus tard sur le premier compilateur Scheme optimisant [127, 128]. C'est dans le milieu des années 1980 apparaît une implantation MIT de Scheme [182, 183]. Ces travaux ont révélé l'importance d'une bonne compilation du contrôle et en particulier d'allouer le moins possible dans le tas les blocs d'activation des fonctions. C'est là le principe fondateur de la compilation moderne et dans ce sens, on peut penser que tous les compilateurs modernes pour langages fonctionnels stricts (Scheme et ML) sont les fils spirituels de Rabbit.

À partir de la fin des années 1980 et jusqu'à nos jours un système s'est toutefois singularisé en choisissant une approche radicalement différente. Il s'agit du compilateur SML/NJ développé conjointement aux Bell Labs et à Princeton University [8], principalement par A. Appel. Ce système peut être qualifié de paradoxal, puisque justement, il repose sur un schéma uniforme de compilation des fonctions où tous les blocs d'activation sont alloués dans le tas. Le dogme fondateur de ces vingt années d'efforts est exposé dans un article de 1987 [6] puis repris dans divers articles dont [11].

Le principe développé est que les GCs copiant ont une complexité qui ne dépend que des objets vivants et non pas des objets alloués, par opposition aux GCs dits *mark & sweep*[5]. Le coût de la libération des objets avec ce type de GC est donc nul. De plus, le coût de l'allocation peut quasiment être ramené à un simple incrément de pointeur si on dispose d'un tas suffisamment grand. En effet, plus le tas est grand, moins on copiera les objets et plus le coût de leur allocation est faible. Tout le système SML/NJ est donc bâti sur cet axiome : il suffit d'augmenter la taille du tas pour accélérer le système. Ce postulat présuppose que l'accès à la mémoire a un coût uniforme, ce qui semble être faux (il suffit de voir l'importance que jouent les caches mémoire dans la conception des processeurs modernes [222]) malgré les justifications d'Appel [93].

À la fin des années 1980 un phénomène a bouleversé le monde de la compilation : l'apparition des processeurs Risc. Cet âge d'or des stations de travail a été caractérisé par la prolifération de plateformes hétéroclites. Les techniques de compilation alors en vigueur se sont avérées être mal adaptées à ces nouvelles architectures[6]. Certains compilateurs qui étaient très performants sur des machines Cisc n'ont jamais pu être adaptés aux nouvelles contraintes matérielles (par exemple T déjà cité, mais aussi Le_Lisp [37]). S'est alors posé de façon cruciale le rôle de la compilation *portable*. Ce nouvel objectif a conduit à la réalisation de compilateurs produisant du C à la place de code machine. C est une sorte d'assembleur portable qui permet de l'arithmétique de pointeurs c'est-à-dire des manipulations sans restrictions de la mémoire. De plus, C a été largement popularisé par le système d'exploitation Unix qui l'a imposé comme *le* langage de la programmation système. C n'est pas un langage très compliqué à compiler parce qu'il est très proche du modèle d'exécution des machines. Les premiers compilateurs C étaient d'ailleurs capables de produire du code à la volée. De plus, les constructeurs comprenant l'enjeu de C, l'ont toujours soutenu et en général ils se sont attachés à fournir des compilateurs C produisant du code de bonne facture pour leurs machines. C est alors devenu la référence en terme de performances alliant ainsi deux qualités essentielles : portabilité *et* performance. Le précurseur de la compilation des LFs stricts vers C est J. Bartlett avec son compilateur Scheme->C qui est apparu en 1989 [22]. Il a ensuite été suivi par CeML dû à E. Chailloux [34] et sml2c dû à D. Tarditi, A. Acharya et P. Lee [239]. Bartlett a cherché à projeter les codes source Scheme vers C. Dans son compilateur il y a une correspondance directe entre les fonctions et variables Scheme et les fonctions et variables C. En particulier, Scheme->C exploite directement le contrôle de C. Cela pose alors trois problèmes :

- Si les fonctions Scheme sont compilées comme des fonctions C et les variables Scheme en des variables C, alors la bibliothèque d'exécution de Scheme doit être compatible avec celle de C. En particulier, le GC utilisé doit être compatible avec C. Les GCs pouvant s'acquitter de cette tâche sont dits « à racines ambiguës ». Ils sont depuis cette époque un sujet de discorde. Les travaux initiaux sur cette catégorie de GCs sont dûs à Bartlett [20, 21] et Boehm [27]. Ce dernier en est d'ailleurs le chantre moderne [26]. On reproche principalement à ces GCs leur lenteur par rapport à des GCs exacts. B. Zorn a démontré que s'ils sont moins efficaces que des GCs copiant générationnels [138], ils ont des performances comparables aux allocateurs explicites [263].
- C ne met pas l'accent sur la récursion terminale. Pour tous les compilateurs C répandus, même si un appel de fonction est placé en position terminale, il consommera de la mémoire en pile. Ceci est en contradiction avec Scheme qui impose un traitement sans consommation de mémoire pour de tels appels de fonctions. Bartlett a éludé ce problème en ne considérant comme important que les appels terminaux servant à implanter des boucles. Tarditi et *al* ont introduit une technique C nommée le *trampoline* qui permet de corriger cette défaillance des compilateurs C. Diverses variantes de trampoline verront alors le jour [72]. Si cette technique permet effectivement de traiter convenablement les récursions terminales, elle a toutefois l'inconvénient d'être globalement inefficace.

[5] Plusieurs textes synthétisent les principales techniques de GCs. Parmi les plus significatives on trouve [258, 119]. Les performances respectives de deux familles de GCs ont été mesurées dans [262].

[6] Le *Dragon* [3] qui est *le* livre de référence de la compilation des années 1970-80 est devenu obsolète presque instantanément. Même si ses premiers chapitres continuent de faire référence, sa description de la génération de code est en décalage avec ce qu'implantent les compilateurs modernes. Par exemple, l'allocation de registres y est décrite en à peine une demi page !

– Toujours en respectant les principes de Bartlett, si la pile d'exécution de C est utilisée pour les appels de fonctions Scheme, alors l'opérateur de contrôle Scheme `call/cc` est épouvantablement complexe à implanter.

Bigloo respecte les principes de Scheme->C. Il adopte le principe de compilation que nous avons nommé la *projection naturelle* (cf. chapitre 6) qui consiste à mettre en correspondance les constructions Scheme avec des constructions équivalentes C (lorsqu'elles existent). La projection naturelle impose l'usage d'un GC à racines ambiguës et nous utilisons celui développé par H. Boehm depuis la version 1.3 de Bigloo. Ce choix ne lui permet pas de respecter scrupuleusement la norme de Scheme car certains appels récursifs terminaux requièrent de l'allocation en pile (principalement les appels de fonctions « extra-module ») et il a grandement compliqué l'implantation de `call/cc`. Toutefois, nous pensons que les avantages priment sur les inconvénients. En particulier, les GCs à racines ambiguës permettent la réalisation d'interfaces externes particulièrement simples et puissantes. Puisque le GC explore la pile C, les valeurs Scheme peuvent être détenues par des variables C. Cela permet naturellement de faire transiter les valeurs du monde C au monde Scheme et réciproquement sans aucune technique d'enregistrement *ad hoc* pour les valeurs allouées dans le tas du GC.

Dans le début des années 1990, les travaux sur la compilation du contrôle ont été étendus. En particulier, sont apparues les premières tentatives d'application d'interprétations abstraites à la compilation des langages fonctionnels. Les premières publications sont dues à O. Shivers [218, 219] et G. Rozas [183]. L'objectif était alors d'aller un peu plus « loin » dans la compilation du contrôle. C'est-à-dire de repousser le moment où un compilateur pour Scheme ou ML alloue les blocs d'activation dans le tas. Même si les techniques proposées sont séduisantes ces travaux ont finalement eu peu de succès. D'une façon générale, l'interprétation abstraite a, jusqu'à présent, eu peu d'influence pratique sur la compilation car rares sont les compilateurs qui l'utilisent. Bigloo est même un des rares compilateurs à utiliser une interprétation abstraite à des fins d'optimisation (cf. § 3.1). Le manque d'applications réelles des techniques d'interprétations abstraites s'explique probablement par la complexité élevée des analyses et de la difficulté de leur mise en application.

Ces travaux sur la compilation du contrôle de Scheme ont largement amélioré celle des LFs stricts. Avec quelques adaptations (cf. chapitre 6) ML a directement profité des avancées de Scheme [54, 35, 212]. Empiriquement nous avons mesuré que dans environ 85 % des cas, l'efficacité de l'appel des fonctions Scheme est la même que celle de C [206]. Les 15 % restants correspondent aux programmes d'ordre supérieur que les compilateurs ne parviennent pas à optimiser et qui sont sans contrepartie en C. On peut donc penser que la compilation du contrôle des LFs a été « acquise » un peu avant le milieu des années 90. Dès lors, peu de travaux ont été publiés sur ce sujet. Il faut toutefois faire une exception pour le cas de l'intégration fonctionnelle qui, probablement à cause de son caractère heuristique, concentre encore beaucoup d'attentions.

Le contrôle bien compilé, on a alors commencé à améliorer la compilation des données. Hormis les travaux sur les GCs qui sont bien antérieurs et toujours autant d'actualité (par exemple avec deux décrits dans [230]) la principale avancée est une meilleure représentation des objets immédiats dans des bibliothèques d'exécution polymorphes et sûres. Comme nous l'avons montré dans § 3.1, les langages polymorphes utilisent généralement une représentation uniforme pour les structures de données qui implique que toutes les valeurs qui sont plus grandes que les mots de la machine sont allouées. Ceci est source d'inefficacité parce que les exécutions sont ralenties par des allocations, désallocations, écritures et lectures mémoire qui ne servent qu'à manipuler ces données. Le cas le plus criant est celui des nombres flottants. Si ces valeurs sont allouées, alors il n'est tout simplement pas possible de faire du calcul numérique intensif. Les premiers à avoir soulevé ce problème et proposé des solutions sont X. Leroy [132, 134] et S. Peyton Jones et J. Launchbury [161]. Ces travaux ne s'appliquent qu'aux langages fortement typés (c'est-à-dire aux langages typés statiquement comme ML). Les premiers travaux utilisant des techniques *globales* pour les langages typés dynamiquement sont ceux de J. Goubault [97] et ceux présentés dans § 3.1.

L'une des raisons de l'inefficacité des langages utilisant des GCs est qu'ils ne permettent généralement pas aux utilisateurs de porter des annotations de durée de vie sur les données des programmes. En particulier, ces langages ne permettent pas aux utilisateurs de spécifier que les objets peuvent être alloués en pile. Pourtant, l'allocation en pile est particulièrement efficace parce qu'elle a un coût très faible, que la désallocation est gratuite et parce qu'elle assure une bonne localité des données. Si l'utilisateur n'a pas le moyen d'exprimer des propriétés de durée de vie, le compilateur doit pouvoir, sans aide extérieure, les calculer afin d'allouer en pile les données qui peuvent l'être. La préoccupation d'allocation en pile est assez ancienne. Il semblerait que les premières études pour les LFs stricts soient dues à C. Ruggieri et T. Murtagh [184], D. Chase [43], suivis de B. Goldberg et G. Park [92] puis M. Tofte et J-P. Talpin [242]. L'analyse présentée § 3.1 permet de déplacer des allocations du tas vers la pile.

Les travaux sur les représentations uniformes ont donné des résultats très satisfaisants parce qu'il ont permis d'améliorer énormément les performances de certains programmes (en particulier ceux faisant du calcul flottant intensif). L'optimisation d'allocation en pile a eu un impact un peu plus marginal pour les LFs, peut-être parce qu'elle n'est pas réellement plus efficace qu'un GC copiant à générations. À notre connaissance, peu de compilateurs utilisent effectivement de l'allocation en pile. Ce sujet semble toutefois connaître un regain d'intérêt pour le langage Java [45, 23, 257]. Le modèle d'exécution de ce langage étant tellement proche de celui de Scheme (polymorphisme et typage dynamique), qu'il n'est pas surprenant que des techniques comparables s'appliquent indifféremment aux deux langages.

On peut remarquer que dès le début des années 1980 jusqu'au milieu milieu des années 1990 environ, la majorité des recherches portant sur l'implantation des langages se sont orientées vers l'efficacité et la compilation. Nous vivons sous le règne de C depuis son apparition en 1972. Hormis quelques dissidents notoires (Smalltalk, Clos, Self et quelques autres qui ont mis l'accent sur des aspects *dynamiques* qui sont assez contradictoires avec l'approche *statique* qu'impose la compilation) presque toutes les implantations ont cherché, jusqu'à l'obsession, à atteindre ou à se rapprocher des performances de C. Le handicap est pourtant de taille car à l'opposé de langages comme C++ dont la volonté d'efficacité est perceptible dans les mécanismes les plus fondamentaux du langage [233], les LFs sont de présentation assez éloignées du modèle d'exécution des ordinateurs contemporains. Pour compiler efficacement ces langages, il faut donc déployer un arsenal d'analyses et d'optimisations assez vaste.

Le cas du langage Self est suffisamment particulier pour qu'il mérite ici une mise en lumière. Au début des années 1990, l'équipe « Self » s'est livré à une étude systématique d'optimisation pour parvenir à posséder une implantation efficace de ce langage pourtant très dynamique. Parmi les très nombreux ouvrages alors publiés citons [42, 110, 38, 65, 63]. Ce sont principalement les aspects liés à la programmation qui ont été étudiés (envoie de message, tests de types ou bien des optimisations plus générales telle que l'intégration fonctionnelle). Grâce à toutes les optimisations déployées, la vitesse d'exécution des programmes Self a été largement améliorée. Cependant, leur consommation mémoire les rendaient peut utilisables sur les machines alors disponibles.

Depuis quelques années les recherches sur l'implantation des langages fonctionnels semblent moins actives. La fin des années 1980 aura donc été le temps de la compilation du contrôle avec notamment des allocations de fermetures de plus en plus efficaces. Le milieu des années 90 aura vu l'amélioration de la compilation des données avec principalement l'introduction de la représentation mixte et de l'allocation en pile. En revanche, il ne se dégage pas une tendance aussi nette aujourd'hui. La principale orientation que semble prendre les concepteurs de compilateurs consiste à utiliser des langages intermédiaires fortement typés [240, 217, 260, 254]. Cette technique n'est accessible que pour les langages qui sont eux-mêmes fortement typés comme ML. Il nous est assez difficile d'évaluer l'intérêt de cette approche. D'abord parce que nous ne pouvons pas l'expérimenter (Scheme étant typé dynamiquement), ensuite parce que les publications ne sont pas encore assez convaincantes.

Renoncer, du moins provisoirement aux performances, a été la principale audace de Java. Sun a étroitement lié ce langage à un modèle d'exécution basé sur la machine abstraite JVM [139]. Cette technique n'a rien de nouveau puisqu'elle a déjà été mise en application dans de nombreux systèmes réels parmi lesquels ont peut citer [36, 91, 160, 133], mais son application pour Java a marqué un tournant dans l'implantation des langages. Pour la première fois depuis plus de vingt ans, les performances n'ont pas été au cœur de l'implantation d'un langage généraliste. Depuis, de nouveaux langages semblent délibérément opter pour un modèle d'exécution favorisant l'emploi de machines virtuelles. Citons ici simplement le langage Limbo [121], développé aux

Bell Labs, qui est totalement lié au système Inferno et à sa machine virtuelle *Dis*. Toutefois ces langages, en particulier Java, souffrent maintenant de leurs mauvaises performances, et de nombreux travaux sont menés pour y remédier, par exemple, par le biais de compilation *Just In Time* [165, 2]. Il convient d'ailleurs de noter ici que l'effort réalisé par la communauté objet pour implanter efficacement cette catégorie de langages est comparable à celui entrepris par la communauté fonctionnelle. À titre d'exemple, on peut citer des travaux aussi divers que ceux réalisés par C. Chambers et J. Dean sur des langages très dynamiques comme Self [38, 64], ceux de Vitek et Horspool sur l'envoi de messages [246] ou encore les techniques plus statiques employées pour des langages comme C++ décrites par S. Lippman dans [140] ou G. Ramalingam et H. Srinivasan dans [175].

Chapitre 4

Bigloo : l'environnement de programmation Bee

The software isn't finished until the last user is dead.
– anonymous

UN environnement de programmation est un agrégat d'outils qui facilitent le développement, la maintenance, l'archivage et la documentation d'un logiciel. Unix et sa batterie de programmes (cc, ld, emacs, make, gdb, prof, grep, man, texinfo, trace, autoconf, cvs, diff, patch, strings, nm, awk, sed, tar, |, ...) est presque un environnement de programmation pour le langage C avant d'être un système d'exploitation. Les environnements de programmation intégrés (abrégé *IDEs*) constituent une forme évoluée d'environnements de programmation. Les deux différences entre Unix et les *IDEs* résident dans ce que ces derniers facilitent l'utilisation de divers outils en proposant une interface uniforme, généralement graphique, d'accès aux divers composants qu'ils contiennent mais, en revanche, les *IDEs* ne peuvent généralement pas être utilisés par le biais de scripts. Historiquement, les premiers *IDEs* sont apparus dès 1972 avec InterLisp [241] puis en 1975 avec les environnements des machines Lisp [191], suivi par l'environnement de Smalltalk qui dès les années 80 [90] en proposait un complet. Ces travaux furent alors relayés par ceux du constructeur Apple pour les machines Lisa et Macintosh et plus tard par ceux de Microsoft pour le système d'exploitation Windows 95.

Unix est un environnement sous lequel on n'a de cesse de programmer. Le système d'exploitation se fond dans l'environnement de développement. Les deux forment un tout cohérent basé sur une interface « texte » d'une puissance sans égale. Unix souffre cependant d'un réel ésotérisme. Le simple fait de connaître l'existence même de la moitié seulement des outils disponibles d'Unix dénote une compétence certaine. Être capable de les utiliser à bon escient est le fait indiscutable d'un expert. Unix requiert une pratique quotidienne car il impose presque un savoir encyclopédique. Si un utilisateur ne connaît pas l'existence de tel ou tel outil, il a fort peu de chance de le découvrir tout seul. Afin de gommer cet inconvénient majeur, nous avons conçu et implanté un environnement de développement intégré pour Bigloo. Cet *IDE* est nommé Bee et sa présentation est l'objet de ce chapitre. Dans § 4.1 nous présentons un survol de l'environnement et quelques unes de ses fonctionnalités simples. Dans § 4.2 nous présentons le navigateur de code source. Dans § 4.3 et § 4.4 nous présentons le *profiler* et le *debugger*. Dans § 4.5 nous présenterons les travaux en cours et ceux prévus.

Ce chapitre présente de façon superficielle les outils de Bee. L'objectif est de montrer leurs possibilités et leur incorporation au sein d'un environnement intégré sans détailler leurs implantations.

Une description de ces dernières peut être trouvée dans les articles [209, 210].

4.1 Bee, un environnement intégré

Comme nous avons initialement conçu Bee pour l'environnement Unix, nous avons respecté ses préceptes. Bee permet la compilation séparée, l'utilisation de bibliothèques pré-compilées et la production de petits exécutables autonomes. Le cycle de développement de Bee suit le schéma traditionnel *édition/compilation/mise au point*. Facultativement, lorsque les programmes sont corrects et qu'ils donnent de bons résultats, leurs performances peuvent être améliorées par le biais de *profilers* qui produisent des informations sur les temps d'exécution ainsi que sur la consommation de mémoire.

Bee est un environnement de programmation Scheme ouvert sur le monde C. C'est-à-dire qu'il y est facile de connecter Scheme et C. Pour cela, on a porté tout notre effort sur la minimisation des différences entre le modèle d'exécution de Scheme et celui de C. En particulier, dans Bee, il n'y a aucun surcoût associé à l'appel d'une fonction C depuis Scheme et *vice versa*. Comme le GC utilisé dans la bibliothèque d'exécution de Bigloo est compatible avec les structures et la pile d'exécution de C, les valeurs Scheme peuvent même transiter par des variables C sans que le GC les collecte de façon intempestive.

4.1.1 Le gestionnaire de projet et la compilation séparée

Comme nous l'avons indiqué dans § 2.3, l'unité de compilation de Bigloo, le compilateur de Bee, est le module. Les modules structurent les programmes et ils permettent au compilateur la détection d'erreurs. Par exemple, si une variable non déclarée est utilisée, le compilateur stoppera la compilation. Les modules ont été conçus dans l'optique d'un traitement simple. En particulier, ce dernier n'impose aucun ordre quant à leur compilation car il n'utilise que les clauses des modules importées pour connaître la liste des variables disponibles. En conséquence, pour pouvoir compiler un module, il suffit que les clauses des modules qu'il importe soient définies.

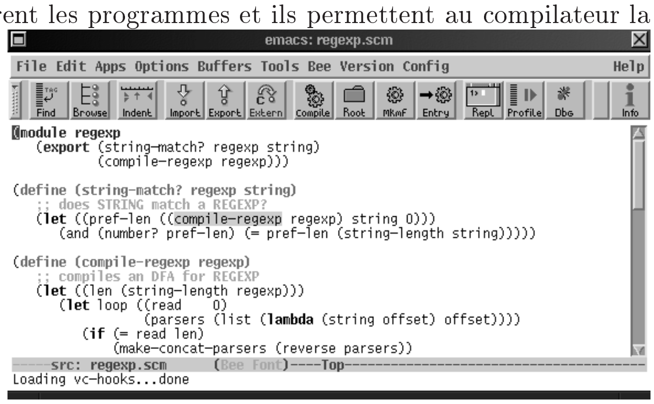


Fig. 4.1: La fenêtre principale de Bee

Le but du gestionnaire de projet de Bee est de faciliter la compilation et la navigation dans les modules. Une fois le premier module d'une nouvelle application partiellement implanté, cliquer sur l'icône Mkmf (cf. figure 4.1) enregistre le projet et crée trois fichiers auxiliaires. Le premier est une table d'association entre nom de module et nom de fichier, nommé `afile`. Comme nous l'avions noté dans § 2.3 les clauses d'importation des modules ne font référence qu'à des noms symboliques et Bigloo n'impose aucune règle de nommage permettant d'associer module et fichier. La table d'association `afile` permet au compilateur de trouver les définitions des modules dans les fichiers du système d'exploitation. Le second fichier généré, `btags`, contient une description de toutes les définitions de tous les modules qui composent le projet. Cette base de données est utilisée pour la navigation dans le code source. Enfin, le troisième fichier, `Makefile` est un fichier d'entrée pour l'utilitaire standard `make`. La construction de ces trois fichiers nécessite une exploration complète de l'arborescence de fichiers Unix afin de trouver tous ceux qui contiennent des définitions de module. Une fois le projet enregistré, toutes les fonctionnalités de Bee peuvent être utilisées. À titre d'illustration imaginons un petit projet composé de deux modules, `main-module` :

```
(module main-module
   (import bar-module)
   (main   main))

(define (main argv)
   (let ((name (car argv))
         (args (cdr argv)))
      (bar args)))
```

et `bar-module` :

```
(module bar-module
   (export (bar x y)))

(define (bar x y)
   '...)
```

Ce projet peut-être compilé en cliquant sur l'icône Compile (figure 4.1). Comme le code source est erroné, une nouvelle fenêtre affichant l'erreur qui se produit pendant la compilation apparaît. Les erreurs de l'utilisateur sont reportées directement dans le code source. La fenêtre de compilation peut alors être utilisée pour éditer, dans le code, la ligne incriminée. Cette fonctionnalité est usuelle dans les environnements de langages classiques comme C mais elle est beaucoup plus rare pour Scheme. Elle impose au compilateur d'associer aux nœuds de son arbre de syntaxe abstraite des positions dans les fichiers. Comme en Scheme, cet arbre peut être directement manipulé par les macros de l'utilisateur, l'association est parfois complexe à assurer. Dans § 10.6.3, page 206, nous présentons la technique utilisée dans Bigloo.

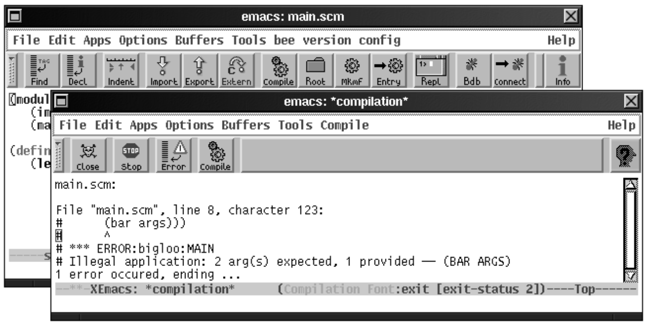


Fig. 4.2: Une fenêtre de compilation

Comme le gestionnaire de projet produit une table spécifiant dans quel fichier physique sont implantés les modules, il est inutile de préciser cette information depuis le code source. Ainsi, `main-module` importe `bar-module` mais nulle part ne sont spécifiés, dans les modules eux-mêmes, les noms des fichiers.

Bee ne recompile pas l'intégralité d'un projet lorsqu'une partie seulement de ce dernier est modifiée. Poursuivons notre session, sur l'exemple précédent. Supposons que le `bar-module` soit corrigé. Recompiler le projet lancera la compilation de `bar-module` et, éventuellement, celle de `main-module`. Si nous supposons que seul le corps de `bar-module` a été modifié, alors la reconstruction du projet nécessite seulement la recompilation de `bar-module`. En revanche, si une des clauses d'exportation de `bar-module` a été changée alors `main-module` devra également être recompilé afin d'assurer la cohérence entre les modules. Nous montrons dans § 10.2.1, page 184, comment la détection des fichiers à compiler est assurée.

4.1.2 La documentation en ligne

Parce qu'une documentation en ligne permet un accès indexé simple, elle est supérieure à une documentation imprimée. L'accès à la documentation en ligne depuis Bee est dépendante du contexte. Lorsque le curseur d'édition se trouve sur un identificateur, cliquer sur l'icône Info fait apparaître une nouvelle fenêtre qui affiche l'aide en ligne pour cet identificateur. Si le curseur est positionné sur un autre type de symbole terminal de la grammaire de Bigloo, alors une documentation idoine est présentée. Par exemple, si le curseur est positionné sur un nombre la documentation générale des nombres sera choisie. La figure 4.3 propose un exemple de session où la documentation en ligne présente la rubrique consacrée aux clauses de module *main*.

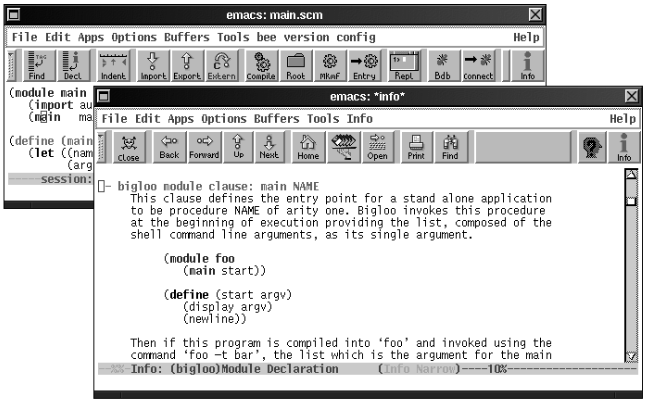


Fig. 4.3: La documentation en ligne de Bee

4.1.3 L'interface externe

Scheme est un petit langage. La dernière version du rapport le définissant compte à peine 50 pages [120]. Comme nous l'avons mentionné dans le chapitre 2 une telle concision est parfois un avantage (par exemple pour l'enseignement) mais c'est aussi un inconvénient parce que la librairie standard ne contient pas assez de fonctions. Bigloo en propose une plus riche que celle du R^{5}RS et il supporte des librairies additionnelles comme par exemple la bibliothèque de *widgets* présentée en § 2.7. Toutefois, même ces librairies sont insuffisantes parce qu'elles ne pourront jamais être exhaustives. Bigloo permet alors une connexion avec le langage C pour pouvoir contourner les limites de Scheme. Une présentation complète de la connexion externe peut être trouvée dans [204]. Un survol est présenté dans § 10.4, page 198. Dans ce chapitre nous ne montrons de l'interface externe que son utilisation depuis Bee.

Bee dispose d'un outil dédié, nommé Cigloo, qui permet l'extraction automatique du prototype des déclarations C (fonctions, variables et types). Sous certaines réserves, Cigloo parvient également à déterminer le prototype des macros de Cpp. Pour cela, il faut que ces macros n'introduisent pas d'irrégularité dans la syntaxe C (par exemple, Cigloo ne serait pas capable de reconnaître une macro qui laisserait une accolade ouverte). C est un langage typé mais Cpp ne l'est pas. Les macros de Cpp sont polymorphes. Cigloo n'a, en général, pas les moyens de déterminer statiquement le type d'une macro. Par défaut, il suppose alors que tous les arguments et la valeur de retour sont de type `int`. Bien sûr cette hypothèse est fréquemment fausse et Bee offre, au cas par cas, la possibilité de valider d'autres choix. Cigloo est invoqué lorsque l'icône Extern est cliquée. Un sélecteur de fichiers est alors présenté à l'utilisateur. Étudions les effets de Cigloo sur un exemple :

```
(module hash
   (export (hash x)))

(define (hash x)
   (cond
      ((symbol? x)
       (hashstring (symbol->string x)))
      ((string? x)
       (hashstring x))
      (else
       ...)))
```

Et supposons une fonction `hashstring` implantée en C, dans le fichier `hash.c` :

```
long hashstring( char *string ) {
   char c;
   unsigned long result = 0;

   while( c = *string++ )
      result += (result << 3) + (long)c;

   return result;
}
```

Comme nous l'avons mentionné dans § 2.4, Bigloo peut compiler des fonctions Scheme qui manipulent indifféremment des valeurs Scheme ou des valeurs C. Par exemple, Bigloo peut traiter des fonctions Scheme dont les valeurs de retour ont des types externes, comme par exemple des `char *` C. Lorsque c'est nécessaire, le compilateur introduit des conversions entre les valeurs suivant les principes énoncés dans § 3.1.4. Ainsi, si après avoir lancé Cigloo, l'utilisateur sélectionne le fichier `hash.c` alors le module `hash` sera modifié en :

```
(module hash
   (extern
    (hashstring::long (::char*) "hashstring")
    (type char*->long "long $(char *)"))
   (export (hash x)))

(define (hash x)
   (cond
      ((symbol? x)
       (hashstring (symbol->string x)))
      ((string? x)
       (hashstring x))
      (else
       ...)))
```

Pour illustrer les limites de Cigloo, étudions un nouvel exemple :

```
typedef struct pt {
   double x, y;
} *point;

#define PT_EGAL( pt1, pt2 ) \
   (((pt1)->x == (pt2)->x) && (((pt1)->y == (pt2)->y)))
```

`point.h` contient les définitions d'un type composé et d'une macro. Pour ce fichier, Cigloo produit :

```
 (extern
   ;; beginning of src/session/point.h
File "point.h", line 5, character 48:
##define PT_EGAL( pt1, pt2 ) \
#^
# *** WARNING:bigloo:define:
Unknown type expression -- Using 'int' type
   (macro PT_EGAL::int (::int ::int) "PT_EGAL")
   (type s-pt (struct (x::double "x") (y::double "y")) "struct pt")
   (type point s-pt* "point")
  ;; end of src/session/point.h
   )
```

Comme on peut le voir, les types de la macro `PT_EGAL` ont été mal interprétés par Cigloo (les argument de `PT_EGAL` sont des `points`). Ces erreurs devront être corrigées manuellement. Déclarer une structure C dans une clause `extern` de Bigloo force le compilateur à produire des créateurs et des fonctions d'accès pour les objets de ce type. En Scheme, ces structures C seront manipulées par l'intermédiaire de pointeurs.

4.1.4 Divers

Bee dispose de quelques outils supplémentaires comme un formateur Scheme, un interprète Scheme qui permet de tester des fonctions ou mêmes, des modules entiers, interactivement depuis l'éditeur. Il dispose également d'un outil permettant de visualiser les effets de la macro expansion sur un code source. Cette dernière possibilité est particulièrement utile lors de la mise au point de macros assez complexes. De plus, Bee offre des modes d'édition avancés où par un simple clic de souris il est possible d'ajouter une déclaration dans une clause d'exportation ou encore, toujours par un simple clic, d'importer automatiquement le module qui exporte une variable utilisée.

4.2 Kbrowse : le navigateur de Bigloo

Kbrowse est un navigateur de programmes Bigloo. Il permet d'inspecter tous les modules et les déclarations qui composent un programme. Kbrowse est intégré dans l'environnement Bee. Cela lui permet d'émettre des ordres d'édition. Ainsi, il est, par exemple, possible d'inspecter toutes les définitions contenues dans un module et, par un simple clic de souris, d'éditer une de ces définitions (une classe, une variable, une fonction, une méthode, une macro ou même une déclaration externe).
L'utilité des navigateurs de programmes a déjà été démontrée par le passé, en particulier par le système Smalltalk. Le navigateur de Bee ne joue toutefois pas un rôle aussi important que dans cet environnement précédent. Bee suppose une structuration du programme par fichiers. Un programme est composé de modules qui sont édités dans des fichiers. Le système d'accueil Unix offre déjà des outils permettant l'édition et la navigation dans les fichiers. Le navigateur de Bee n'est donc qu'un moyen supplémentaire qui est parfois plus pratique que celui du système hôte parce qu'il exploite la syntaxe des programmes Bigloo pour offrir un meilleur confort de navigation.

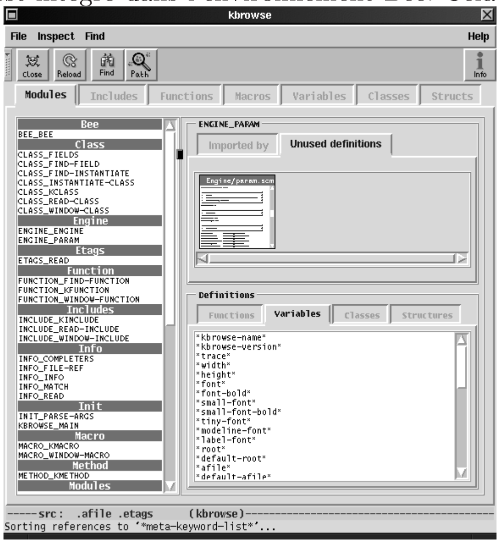


Fig. 4.4: La fenêtre principale de Kbrowse

En plus des fonctionnalités standard des navigateurs, Kbrowse fournit des informations sur l'usage fait des variables, fonctions et classes des programmes. Ainsi, par exemple, il est possible de connaître tous les sites où une fonction est directement appelée (c'est-à-dire tous les sites non calculés où une fonction est appelée), ou encore, d'exhiber tous les sites d'un programme qui instancie telle ou telle classe 4.5. Pour implanter cette fonctionnalité, Kbrowse utilise les grammaires lexicales de Bigloo [200]. Parce que la syntaxe de Scheme est très simple et très régulière, un appel de fonction ou l'instanciation d'une classe sont des expressions qui peuvent être décrites par des langages rationnels. Rechercher de telles expressions est alors très rapide.

Pour chaque classe, Kbrowse rapporte la liste des sous-classes, toutes les positions dans le source où la classe est instanciée et également la liste de toutes les méthodes qui surchargent des fonctions génériques pour cette classe. En quelque sorte, Kbrowse rétablit une structure par classe du programme.

En plus de Kbrowse, l'environnement offre un arsenal de fonctionnalités permettant la navigation dans le code source. Depuis l'éditeur, on peut inspecter la définition associée à tout symbole par un simple clic de souris. Il est de plus possible de placer des liens *hyper-texte* dans le code source. Des morceaux d'implantation peuvent alors faire des références explicites à d'autres parties d'un programme. Basé sur le même principe, on peut également faire des références explicites depuis le code source d'un programme au code source de sa documentation. Toutes les tables indispensables à une navigation rapide sont générées automatiquement lors de la déclaration puis lors de la compilation des projets. C'est-à-dire que la navigation est toujours accessible à l'utilisateur

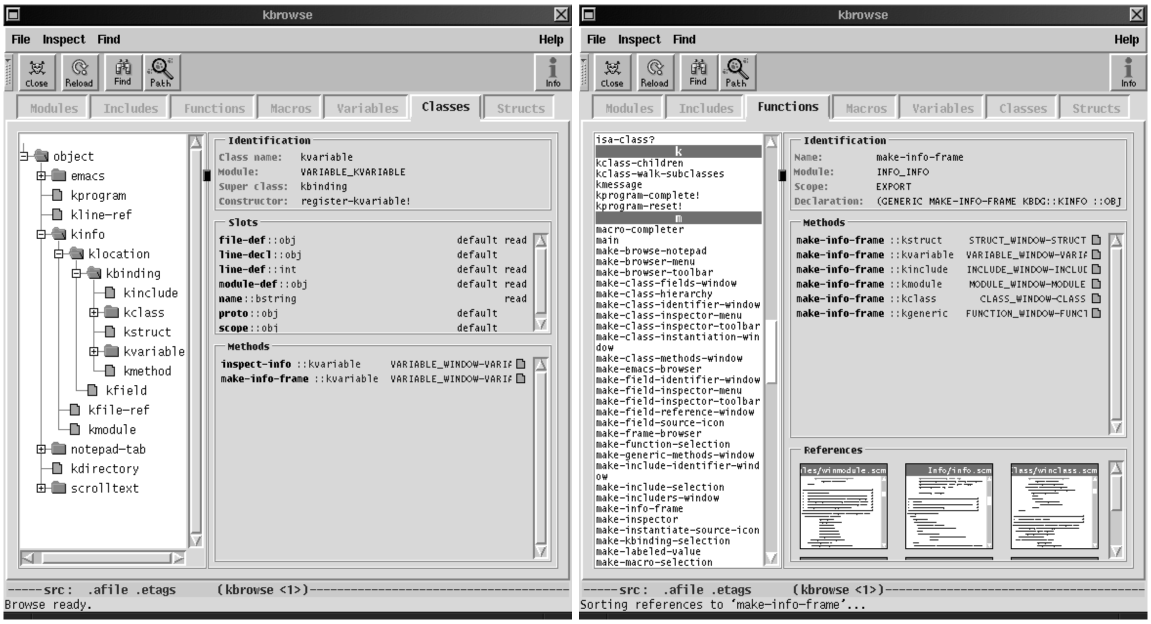


FIG. 4.5 – Les fonctions et les classes dans Kbrowse

sans que ce dernier n'ait besoin, à la main, de pratiquer aucune opération. C'est l'environnement qui prend entièrement à sa charge la navigation.

4.3 Kprof : le *profiler* de Bigloo

Les *profilers* sont des outils indispensables pour l'évaluation des performances des programmes et pour déterminer les portions d'un code source qui sont particulièrement responsables de consommation de ressources. À notre sens, plus la distance entre le langage source et le modèle d'exécution de la machine exécutant le programme est grande, plus les *profilers* jouent un rôle important. Dans le cadre de Scheme la distance est grande entre les constructions de ce langage et les instructions assembleur réellement exécutées ; de plus Bigloo a parfois recours à des encodages complexes, même pour des constructions Scheme d'apparence anodine. Il est alors parfois presque impossible pour un utilisateur qui ne connaîtrait pas les détails internes du compilateur d'évaluer, *à priori*, l'efficacité de ses programmes.

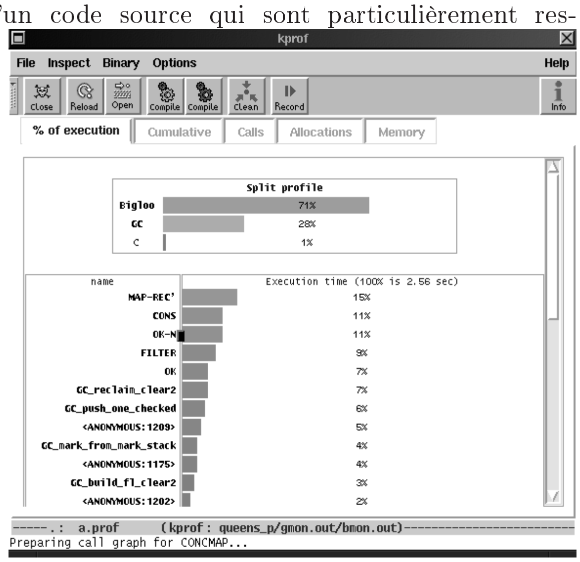


FIG. 4.6: La fenêtre principale de Kprof

Kprof, le *profiler* de Bigloo, présente deux types d'informations différentes. Comme tous les *profiler* classiques, il fournit des informations sur les fonctions (par exemple le temps passé dans chacune d'elles). En plus, Kprof présente des informations sur les allocations mémoire. Il est intégré dans l'environnement. Comme le navigateur, il est possible depuis le *profiler* d'invoquer l'éditeur et réciproquement. Il est par exemple possible, depuis l'éditeur, d'examiner les temps d'exécution de la fonction qui est en cours

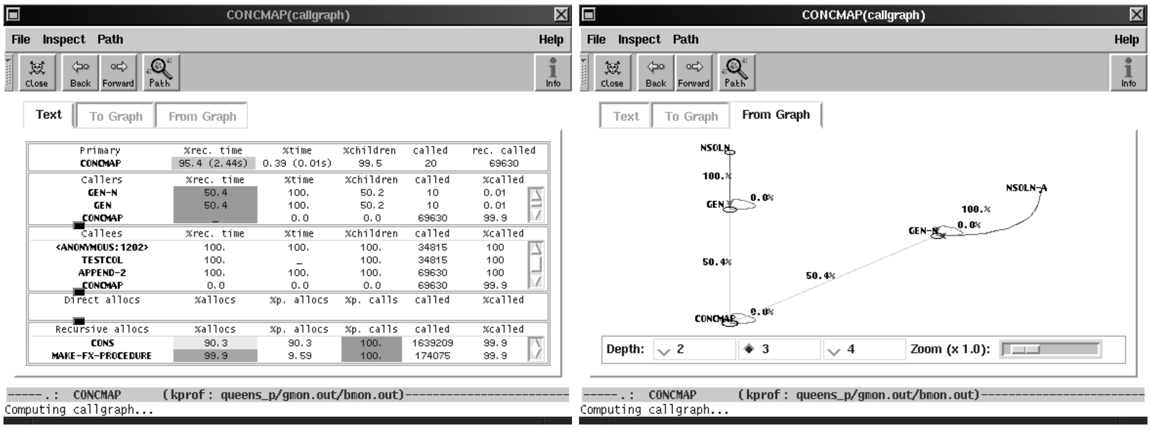


FIG. 4.7 – Le graphe d'appel de la fonction `CONCMAP`

d'édition (bien sûr, sous réserve, qu'une exécution en mode *profiling* ait déjà été exécutée).

La figure 4.6 est une copie d'écran de la fenêtre principale de Kprof. On y voit une première série d'informations affichées pour un programme nommé `queens` (dont le code complet est donné dans § 12.7, page 258). Ce programme implante une résolution du problème des 8 reines. La méthode utilisée ici consiste à énumérer tous les placements possibles des reines sur un échiquier et, parmi ses configurations, à ne retenir que celles qui sont des solutions. Chaque combinaison est représentée par une liste. À chaque fois qu'il est exécuté, `queens` résout deux fois le problème des reines. Ce programme contient deux fonctions d'énumération des configurations, équivalentes quant à leur résultat mais différentes quant à leur implantation : `sol1` et `sol2`. `Queens` nous sert souvent pour évaluer les performances des GCs. En effet, `queens` alloue énormément de listes ! D'ailleurs, comme le montre la figure 4.6, Kprof nous apprend que 28 % du temps complet de l'exécution a été passé dans le gestionnaire mémoire (principalement pour collecter les listes devenues inutiles) et plus de 11 % du temps a été passé dans la fonction Scheme qui alloue les paires, `CONS`.

Kprof permet d'obtenir d'autres informations générales telle que le nombre de fois où chaque fonction a été appelée. Il permet également de connaître des informations plus fines sur la décomposition du temps passé fonction par fonction. La figure 4.7 présente deux vues, une textuelle et une graphique du graphe d'appel de la fonction `CONCMAP` du programme `queens`. On y découvre, par exemple, que 50 % (aux erreurs d'arrondi près) du temps global passé dans `CONCMAP` l'a été lorsque cette fonction a été appelée depuis `GEN`. On y apprend également que `CONCMAP` est responsable de tous les appels à la fonction `TESTCOL`.

En plus de ces informations, Kprof rapporte des mesures liées aux allocations mémoire effectuées par les programmes. La première de ces informations est la quantité de mémoire allouée par un programme et la seconde est la décomposition, par pourcentage d'appel, des constructeurs. L'inspection des allocations mémoire du programme `queens` nous apprend que l'exécution complète nécessite 11.6 Mo, que parmi tous les appels de fonctions exécutés environ 40 % concerne des allocateurs et enfin, que 90 % des allocations concernent des paires. Kprof peut également présenter les diverses configurations mémoire de l'exécution de `queens` au fil du temps ; le temps étant mesuré ici en GC (c'est-à-dire que les informations présentées ont été mesurées lors de chaque GC). La lecture de la figure 4.8 nous apprend ainsi que la taille maximale qu'atteint le tas pour `queens` est de 1.2 Mo, qu'il a fallu exécuter 18 récupérations mémoire (collections) pour l'exécution complète. La courbe la plus basse nous indique la taille de la pile d'exécution telle qu'elle a été mesurée lors de chaque GC. Enfin, les deux dernières courbes permettent de connaître les tailles mémoire allouées entre deux collections et la taille totale des objets qui survivent aux GCs. On voit ainsi que entre le GC 10 et 11, environ 0.95 Mo ont été alloués et seulement 0.20 Mo ont survécu au $11^{\text{ème}}$ GC. Les lignes verticales de la figure 4.8 dénotent les évaluations d'expressions annotées du code source. Ces annotations sont introduites par la nouvelle syntaxe :

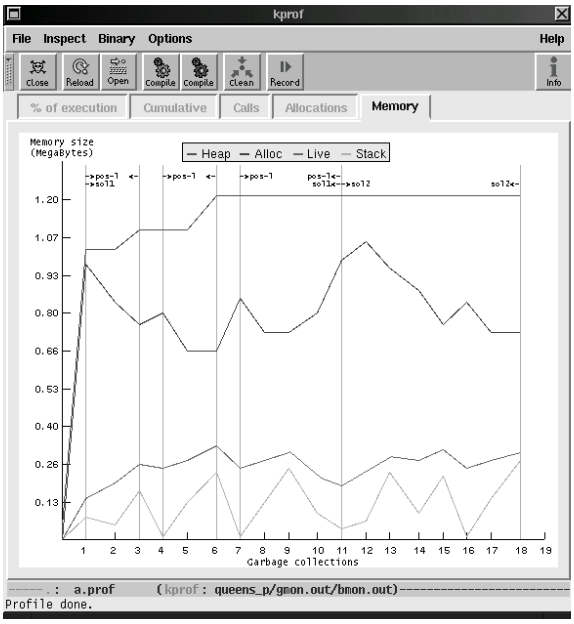


Fig. 4.8: La mémoire présentée par Kprof

(profile *<label>* *<s-expression>***)**

L'évaluation d'une forme `(profile <lbl> <expr>)` est équivalente à `(begin <expr>)` mais elle force Kprof à introduire un label représentant l'évaluation de l'expression (dans la figure 4.8, trois expressions, `pos-l`, `sol1` et `sol2`, ont été annotées). Il est alors possible de connaître le nombre d'allocations qui sont nécessaires à l'évaluation de *expr* ou même le temps passé dans cette évaluation et les fonctions appelées.

Dans le programme `queens`, la définition de `CONCMAP` est :

```
(define (concmap f l)
   (if (null? l)
       '()
       (append (f (car l)) (concmap f (cdr l)))))
```

Comme cette définition le montre, la fonction `CONCMAP` ne fait directement appel à aucun allocateur. La rubrique *Direct allocs* de la figure 4.7 est donc vide. En revanche, les fonctions appelées par `CONCMAP` (`APPEND` et surtout la fonction passée en paramètre, `F`) font des allocations. La rubrique *Recursive allocs* de la figure 4.7 nous apprend que 90.3 % des appels à `CONS` sont en fait réalisés par des fils de `CONCMAP`. Kprof est capable d'afficher les graphes d'appels dynamiques qui permettent d'aller d'un point du programme à un autre. La figure 4.9 présente ainsi les chemins qui existent entre `CONCMAP` et `CONS`. Chaque arc est étiqueté par son pourcentage d'appel. Ainsi, 33.3 % des appels de `CONCMAP` sont orientés vers la fonction `APPEND-2` alors que 100 % des appels de `APPEND-2` sont orientés vers `CONS`.

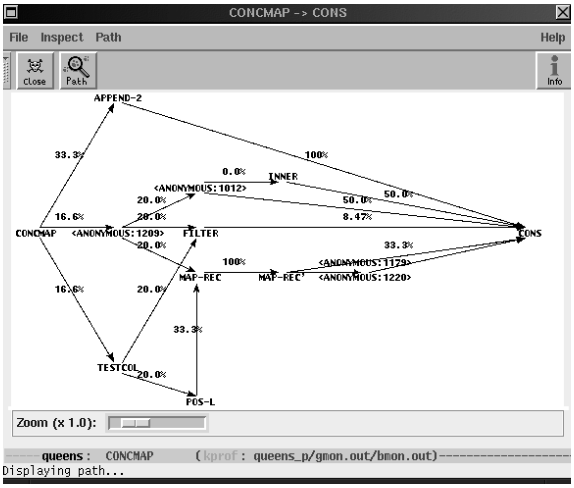


Fig. 4.9: Les graphes d'appels dans Kprof

4.3.1 Kprof et le *bootstrap* de Bigloo

La figure 4.10 présente l'évolution complète de la compilation d'un des modules du compilateur. Chaque passe de la compilation étant enrobée dans une forme `profile`, on peut mesurer pour chacune d'elle ses allocations. En appliquant ainsi Kprof sur le compilateur nous avons été capable de diminuer grandement ses allocations. Nous avons développé Bigloo dans le but d'obtenir un programme aussi économe que possible mais, sans outil spécialisé, il est dans la pratique très difficile d'estimer les parties qui allouent réellement beaucoup de mémoire. Dans le compilateur, c'était la dernière phase de la compilation (celle qui prend en charge la génération de code) qui était la principale responsable des allocations. Pourtant, cette dernière passe n'opère qu'un simple parcours de l'Ast afin de produire une représentation en C de cet arbre. En examinant les chemins menant de la fonction principale de la génération de code de Bigloo jusqu'aux allocateurs, il est apparu que les allocations étaient consacrées à l'élaboration de listes utilisées par les fonctions d'affichage. Dans la définition standard de Scheme, toutes les fonctions d'affichage acceptent un argument optionnel qui est le flux de sortie sur lequel les objets doivent être imprimés. En Scheme, les arguments optionnels doivent être placés dans des listes fraîchement allouées. Ainsi, l'évaluation d'une expression comme `(display obj` *`a-port`*`)` alloue une paire pour placer le paramètre *`a-port`* dans une nouvelle liste. Ce mode de passage d'arguments optionnels peu économe était la seule cause des allocations de la passe de génération de code. Nous avons alors implanté une optimisation *ad hoc* très simple qui traite de façon particulière les fonctions d'affichage qui, à elle seule, a réduit de plus de 20 % les allocations *globales* du *bootstrap* du compilateur (comme on connaît les définitions des fonctions standard d'affichage, on sait qu'elles ne font aucun usage des listes d'arguments) ! Cet exemple illustre parfaitement l'intérêt d'outils tels que les *profilers*. Ils permettent parfois de mettre en évidence des phénomènes insoupçonnés.

4.3.2 Kprof et les fuites mémoire

La visualisation de l'évolution de l'occupation mémoire telle que présentée par Kprof permet un contrôle assez grossier de l'espace mémoire occupé par l'évaluation d'une expression. Comme nous venons de le voir, ces informations peuvent être suffisantes pour diminuer le taux d'allocations mémoire des programmes. Elles peuvent également mettre en évidence une fuite mémoire. Dans un système utilisant un GC, une fuite mémoire se produit dès que le collecteur ne parvient pas à récupérer des zones mémoires qui pourtant sont inutiles pour tout le reste de l'exécution. Dans de tels systèmes, deux types d'erreurs *classiques* produisent des fuites mémoire :

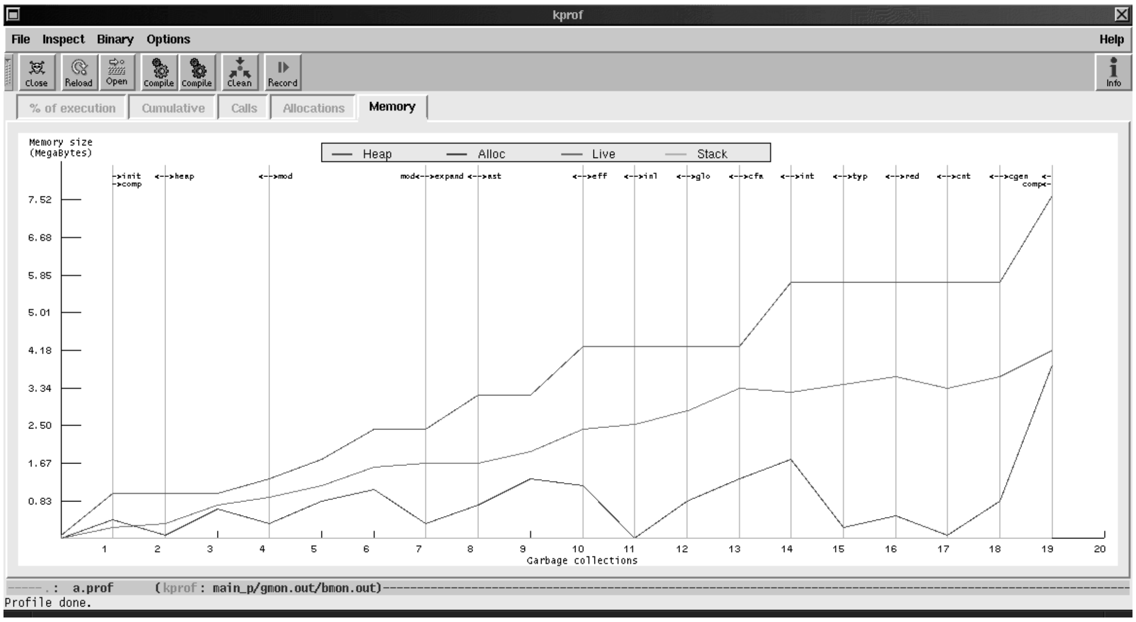


FIG. 4.10 – Le *profiling* de Bigloo

- Une structure de données est pointée par une variable globale ou par une structure globale (telle une table de hachage). Lorsque cette structure devient inutile la variable (ou la table) continue à la pointer et le GC ne peut pas la collecter.
- Une structure est encore utilisée mais certains de ses champs deviennent inutiles. Le GC ne peut pas collecter l'espace qu'ils occupent parce que la structure qui les pointe est, elle, toujours vivante.

Sans outil spécialisé il est presque impossible de détecter et de corriger les fuites mémoire. Kprof permet de déceler les fuites mais il ne permet pas de les corriger. En effet, Kprof ne fournit d'informations que sur les allocations qui sont déconnectées des fuites elles-mêmes. Pour corriger une fuite mémoire il faut disposer d'un outil permettant d'étudier la durée de vie des objets. Dans notre environnement, ce rôle est assuré par Kbdb, notre *debugger*.

4.4 Kbdb : le *debugger* de Bigloo

Kbdb, le *debugger* de Bigloo, est doté des attributs standard des *debuggers*. Il permet la pose de points d'arrêt (*breakpoints*), l'affichage des valeurs des paramètres, des variables globales et locales et d'une représentation de la pile d'exécution. Kbdb rend possible l'examen de tous les blocs d'activation en suspens. On peut « remonter » la pile des appels pour, par exemple, comprendre le fil d'une exécution. La figure 4.11 présente la fenêtre principale de Kbdb. On y voit plusieurs zones d'affichages : *1)* la console qui sert à la saisie et à l'affichage des résultats des commandes internes, *2)* l'affichage de la pile d'exécution, *3)* les paramètres effectifs de la fonction courante et *4)* les variables locales visibles au point courant de l'exécution.

Comme les précédents outils que nous avons présentés (Kbrowse et Kprof), Kbdb est intégré dans l'environnement Bee. Ceci facilite son utilisation. Par exemple, les points d'arrêt peuvent être directement posés dans l'éditeur de code source, « à la souris », dans lequel ils sont représentés graphiquement. Il est également possible d'exécuter pas à pas le programme, la ligne courante étant, elle aussi, visualisée dans l'éditeur.

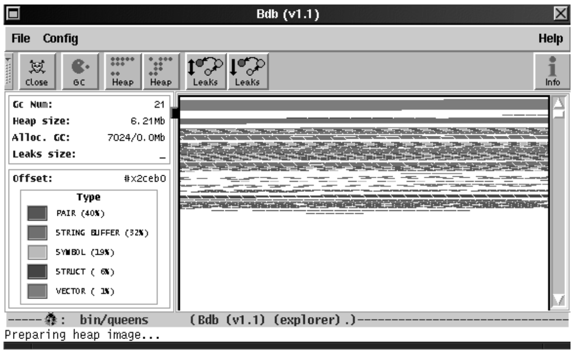


Fig. 4.11: La fenêtre principale de Kbdb

Bigloo permet de mélanger du code C et du code Scheme au sein d'une même application. Kbdb est utilisable pour les deux langages. On peut exécuter pas à pas aussi bien du code Scheme que du code C et que l'on peut afficher les valeurs des variables Scheme et C.

4.4.1 Kbdb et les fuites mémoire

Kbdb permet également une visualisation du tas. C'est cette possibilité qui est utilisée pour corriger les fuites mémoire. Lorsqu'une exécution est suspendue (soit parce qu'un point d'arrêt a été atteint, soit parce que l'exécution est faite pas à pas, soit, enfin, parce qu'une erreur dynamique, telle que le débordement des bornes d'un tableau, a été détectée), Kbdb peut produire une image bidimensionnelle du tas. Dans cette image, les objets sont représentés par des traits horizontaux. La taille des traits est directement proportionnelle à celle des objets visualisés. Les espaces inutilisés du tas sont affichés en blanc. La figure 4.12 présente une visualisation du tas lors de l'exécution du programme `queens` présenté en § 4.3. Dans le cadre supérieur gauche, on voit le nombre de GCs déjà exécutés à ce stade de l'exécution (ici 22), la taille du tas (4.65 Mo) et la taille des allocations mémoires effectuées depuis le dernier GC (0.02 Mo). Pour la figure 4.12 les objets ont été classés par type. C'est-à-dire que chaque type a été associé à une couleur et donc tous les objets de même type sont représentés par des traits de même couleur dans la partie droite de la figure. D'autres modes de sélection peuvent être utilisés comme, par exemple, l'âge des objets.

Kbdb dispose d'une loupe qui permet l'agrandissement de zones spécifiques de l'image. Il devient alors possible d'inspecter individuellement tous les objets du tas en cliquant sur les traits représentant les objets vivants. On obtient alors des informations comme :

```
Holder     : FILTER, frame 4 (local L) ;; Le détenteur de l'objet
Type       : PAIR   (0)                ;; Le type de l'objet
Producer   : FILTER                    ;; La fonction qui l'a allouée
Generation: 5                          ;; La date de naissance de l'objet
Size ytes : 12                         ;; Sa taille en octets
   [PAIR]      (1)                     ;; La chaîne de pointeurs
   [PAIR]      (2)
   [PAIR]      (3)
(bdb:MAIN) display (0)                 ;; L'affichage de l'objet sélectionné
$63 = (8)
(bdb:MAIN) display (3)                 ;; L'affichage du 3ème objet de la chaîne
$63 = (6 7 8)
```

On y apprend que l'objet sélectionné était une paire, qu'il a été alloué après le 4$^{\text{ème}}$ GC par la fonction `FILTER`, que sa taille est de 12 octets[1]. Kbdb indique également que cette paire est détenue (pointée) par la variable `L` de la fonction `FILTER`. La *chaîne d'objets* est la liste de pointeurs qui mène de l'objet sélectionné jusqu'à une racine du GC(la variable `L`). Tous les objets appartenant à cette chaîne peuvent être à leur tour inspectés ou leur valeur peut être simplement affichée. La chaîne d'objets permet de comprendre pourquoi un objet est vivant aux yeux du GC.

[1]Comme nous le montrons dans § 12.4.3, page 251, pour que Kbdb puisse afficher des informations relatives au tas, un mode spécial de compilation doit être utilisé. Dans ce mode, les objets sont un peu plus gros que pour les exécutions optimisées.

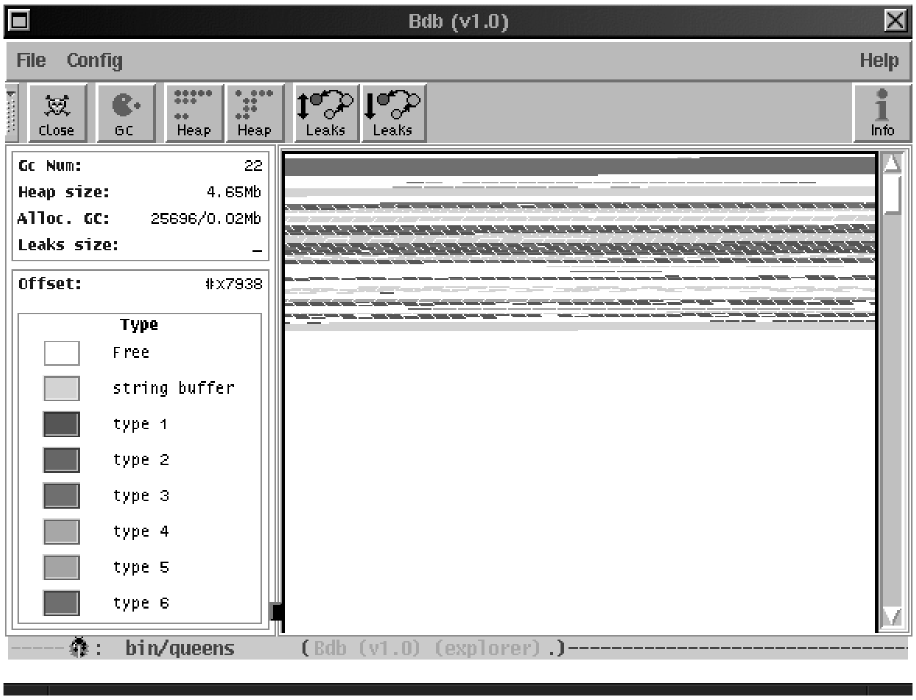


Fig. 4.12 – Une visualisation du tas par type d'objets

La sélection des objets du tas est à la base de la recherche des fuites mémoire. Une fuite est toujours caractérisée par des objets dont la durée de vie est trop longue. C'est-à-dire que certains objets survivent anormalement à l'évaluation d'une expression $\mathcal{E}$. Pour les démasquer il suffit de déterminer parmi tous ceux qui survivants à l'évaluation de $\mathcal{E}$ ceux qui ont été alloués pendant ce calcul. Kbdb peut, à la demande, afficher une image du tas, où seuls sont présentés les objets qui ont été alloués pendant l'évaluation d'une expression arbitraire et qui sont encore vivants lors de la requête de visualisation.

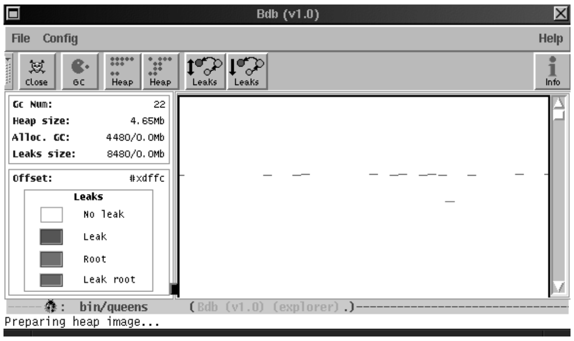


Fig. 4.13: Une fuite mémoire exhibée

Comme nous l'avons mentionné, `Queens` résout deux fois le problème des reines. Une première fois au moyen de la fonction `sol1` et une seconde fois au moyen de la fonction `sol2`. Chacune de ces deux fonctions allouent des listes pour représenter les configurations des reines sur l'échiquier puis les consomme. Le résultat de `sol1` et `sol2` indique seulement le nombre de placements qui résolvent le problème, pas les configurations des reines. Ainsi, aucune des listes allouées pendant l'évaluation de `sol1` ou `sol2` ne doit survivre. En examinant la figure 4.8 on peut toutefois noter que le tas, avant le calcul de `pos1`, est environ de 0.15 Mo alors qu'à la fin du calcul il est d'environ 0.20 Mo. Cela semblerait indiquer une fuite dans cette fonction. La figure 4.13 est une visualisation du tas où ne sont affichés que les objets alloués pendant l'évaluation de `sol1` et qui sont toujours vivants après le retour de cette fonction. Bien que cela ne soit pas visible avec une impression en noir et blanc, deux types d'objets sont affichés, chacun désigné par une couleur. En rouge, sont affichés les objets qui composent la fuite et en bleu sont affichées les racines du GC qui maintiennent en vie les objets de la fuite. Bien que cela ne soit pas le cas sur cet exemple, les racines peuvent faire directement partie de la fuite, par exemple, si une donnée « évadée » est directement pointée par une variable globale. Il est alors possible de sélectionner les objets de la fuite :

```
Holder    : COUNT
Type      : PAIR   (0)
Producer  : NSOLN
Generation: 2
Size      : 12
   [PAIR]      (1)
   [CELL]      (2)
   [PROCEDURE] (3)
```

Cet objet peut être affiché :

```
(bdb:MAIN) display (0)
(2 3 4 5 6 7 8)
```

Il s'agit donc d'une petite liste. Bien que cela ne soit pas visible dans l'impression de ce texte, grâce aux couleurs des objets, Kbdb indique qu'une seule racine est la cause de cette fuite. En cliquant sur cet objet, on obtient :

```
(bdb:MAIN) explore (5)
Holder    : COUNT
Type      : PROCEDURE   (0)
Producer  : COUNT
Generation: 2
Size      : 20
```

C'est donc une fonction qui est à la base de la fuite. Sa définition peut être chargée sous l'éditeur par un clic de souris :

```
1 :  (define count
2 :     (let ((memo '()))
3 :        (lambda (from to)
4 :           (define (inner from to)
5 :              (if (>fx from to)
6 :                  '()
7 :                  (cons from (inner (+fx 1 from) to))))
8 :           (let* ((key (cons from to))
9 :                  (cell (assoc key memo)))
10 :             (if (pair? cell)
11 :                 (cdr cell)
12 :                 (let ((new (inner from to)))
13 :                    (set! memo (cons (cons (cons from to) new) memo))
14 :                    new))))))
```

`Count` construit les listes utilisées pour les configurations. C'est une mémo-fonction, c'est-à-dire qu'elle garde trace de toutes les valeurs qu'elle construit. La variable `memo` (ligne *2*, libre dans le corps de la fonction), détient une table qui contient les configurations construites. Comme `count` est une fonction globale, sa valeur est toujours considérée par le GC comme vivante. Ainsi, la fonction définie ligne *3* n'est jamais collectée et donc la valeur de la table non plus.

4.5 Perspectives

L'environnement de programmation de Bigloo est actuellement l'objet de nos attentions. Il est en évolution constante. Nous prévoyons par exemple l'extension de Kbdb. Dans sa version courante cet outil de visualisation est capable de représenter le tas en utilisant trois critères : *1)* une classification par type, *2)* une classification par âge et *3)* la détection des fuites mémoire. Kbdb pourrait être utilisé pour représenter le tas *à la* Haskell. Kbdb dispose des informations nécessaires pour une représentation des producteurs des allocations mémoire plutôt que des allocations elles-mêmes. Cette visualisation, présentée dans l'article [187], permet parfois de mieux comprendre si une fuite mémoire a lieu et de déterminer la fonction qui en est responsable. Cette représentation nous semble être complémentaire avec celles actuellement implantées. Il serait peut-être pertinent de présenter des informations statistiques sur la durée de vie des objets. Par exemple, le nombre moyen de GCs auxquels les objets survivent. Nous pourrions également exploiter directement le GC dans l'environnement. Puisqu'un GC, par construction, est capable de parcourir tous les objets vivants à un instant précis des exécutions, nous pourrions l'utiliser pour calculer dynamiquement des propriétés de partage et d'*aliasing* des données. Il serait aussi possible d'incorporer dans le *debugger* une nouvelle commande permettant de connaître, sur un point d'arrêt, et pour une donnée spécifiée par l'utilisateur, la liste des autres données qui la référencent. Pour tous les langages utilisant des GCs (les LFs, Smalltalk ou même Java), très peu d'environnements de programmation sont capables d'afficher des informations liées au tas. Notre brève, car récente, expérimentation de Kprof et Kbdb nous laisse penser que ces outils sont pourtant utiles, voire indispensables, pour améliorer les performances de certains programmes. Seule une pratique intensive permettra de concevoir des outils réellement utilisables et pertinents. C'est dans cette direction que nous pensons poursuivre nos recherches.

4.5.1 Les futurs composants de Bee

Parallèlement à l'extension des outils dont nous disposons déjà dans Bee, nous prévoyons l'adjonction de nouveaux composants. Le principal outil qui est actuellement en cours de développement est un constructeur graphique d'interfaces graphiques. Ce nouvel outil nommé Buildoo est encore à l'état de prototype mais les résultats déjà obtenus sont encourageants. Buildoo permet de programmer graphiquement des interfaces utilisateur. C'est un composant classique des nouveaux systèmes nommés *RAD* (pour *Rapid Application Development*). La figure 4.14 présente une copie d'écran d'une session Buildoo. Même s'il n'est pas terminé, Buildoo est suffisamment avancé pour pouvoir être autogénéré. C'est-à-dire que les interfaces graphiques utilisées dans Buildoo sont elles-mêmes produites par Buildoo. Son intégration dans Bee est déjà prévue et en partie réalisée. Ainsi, tous les outils de Bee tels `bmake` et `bdepend` qui produisent et mettent à jour les `Makefile` ou `btags` et `afile`, qui produisent les tables de modules et tables d'identificateurs acceptent en entrée les fichiers de description d'interfaces.

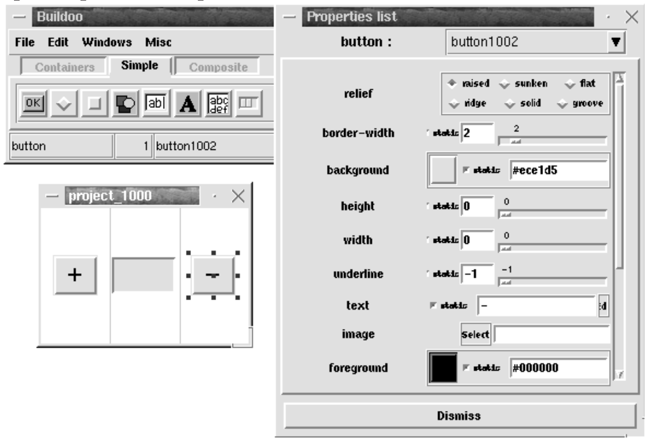


Fig. 4.14: Buildoo

Les fichiers de Buildoo contiennent des représentations binaires des interfaces qu'ils décrivent. Son objectif est de dispenser totalement les utilisateurs de modifier manuellement le code Scheme généré par Buildoo. Comme nous l'avons mentionné dans le § 2.7, la librairie graphique Biglook favorise un style déclaratif ce qui permet une séparation nette entre les parties du code qui implantent les interfaces graphiques et celles qui implantent les calculs effectués par le programme. Buildoo produit alors des modules Scheme qui sont autonomes. Ils ne contiennent pas de code utilisateur. Ils se contentent de référencer les identificateurs pour mentionner les *callbacks* des *widgets* qui doivent être définis dans d'autres modules qui sont, eux, totalement à la charge de l'utilisateur. Dans la construction d'un projet, la production de code Scheme est retardée le plus possible. Le format binaire de Buildoo, caractérisé par l'extension `.bld`, n'est traduit vers un format acceptable par Bigloo que lorsque la compilation du projet est lancée. Dans la période d'édition, seul le format binaire est utilisé. En d'autres termes, Buildoo n'a jamais besoin de relire les fichiers Scheme générés à partir de ses fichiers binaires. Il en serait d'ailleurs incapable. Buildoo, son utilisation, son format binaire et son intégration dans Bee sont décrits dans l'article [236].

4.5.2 Les environnements de programmation sous Unix

Depuis quelques années apparaissent des environnements de programmation intégrés sous Unix. Parmi ces systèmes on peut citer Code Crusader, Code Medic, Kdevelop et probablement d'autres dont nous ignorons l'existence. Il est difficile de prévoir l'impact de ces systèmes sur les habitués du système Unix. Ces études nous semblent toutefois intéressantes parce qu'elles permettront peut-être de diminuer l'écart actuel qui existe entre la programmation sous Unix et sous Windows. L'environnement Kdevelop est à cet égard exemplaire. Il contient tous les outils qui sont intégrés dans les environnements Windows tels que la série des Visual XXX.

L'objectif de Bee est un peu plus ambitieux. Non seulement nous voulons parvenir à fournir un environnement de programmation intégré mais en plus nous voulons que cet environnement soit plus agréable et plus puissant que ceux existants dans les langages impératifs classiques. Nous comptons pour cela sur la simplicité cumulée de la syntaxe et de la sémantique de Scheme. Par exemple, la recherche des sites d'appel d'une fonction est simple parce la syntaxe de Scheme est très régulière et elle peut donc être réalisée au moyen d'une simple expression rationnelles. Par ailleurs, si nous parvenons à intégrer l'affichage des informations d'*aliasing* dans le *debugger* cela sera entièrement dû au fait que Scheme utilise un GC. Une bibliothèque d'exécution typée facilite également l'implantation du *debugger*. En plus elle permet des affichages qui ne sont pas possibles pour des langages comme C. Il est alors possible d'envisager des explorations de structures de données inaccessibles aux langages qui n'incluent pas d'informations de type dans les valeurs construites.

4.5.3 Les autres environnements fonctionnels

Le seul environnement de programmation pour Scheme que nous connaissons est développé par l'équipe PLT de l'Université de Rice [76, 73]. Elle a réalisé l'environnement DrScheme qui est principalement conçu pour l'enseignement. Ce n'est pas exactement un environnement de programmation intégré parce qu'il lui manque les outils qui permettent la finalisation de programmes réels. Par exemple, DrScheme ne dispose ni de *profiler* ni de *debugger* dynamique. Les aptitudes de DrScheme à la compilation semblent également très limitées.

DrScheme repose sur un environnement graphique pour éditer et évaluer les programmes Scheme. Il fournit aux utilisateurs une aide syntaxique et sémantique. Par exemple, l'éditeur de DrScheme est capable de relier graphiquement l'utilisation d'une variable à sa définition et vice versa. Le système possède également des « niveaux » de langage. Cela permet de ne dévoiler celui-ci que petit à petit, par exemple, au fur et à mesure que les séances d'un cours progressent. Par ailleurs, DrScheme dispose d'un *debugger* statique [80, 79] qui, au moyen d'une interprétation abstraite, calcule de l'inférence de types pour Scheme. Une fois l'analyse appliquée, il est alors possible de visualiser le cheminement qui a conduit DrScheme à attribuer tel ou tel type à une variable. Cette visualisation permet par exemple de comprendre les erreurs que détecte le vérificateur statique de types. Cette utilisation d'une interprétation abstraite nous semble particulièrement prometteuse. Peu de travaux ont jusqu'à présent été entrepris dans cette direction. En plus de ceux de C. Flanagan, les seuls dont nous ayons connaissances sont dûs à F. Bourdoncle [28] et F. Pessaux et X. Leroy [159].

4.5.4 Tendances passées et actuelles

L'optique dans laquelle Bee a été conçu est à l'opposée de celle des anciens environnements Lisp tels que décrits dans un article de Sandewall [191]. La vision de Bee est plus *statique* et moins *interactive*. Par exemple, les modules de Bigloo ne permettent pas de faire référence à des variables non déclarées car le compilateur vérifie les types et les arités lors de la compilation. De plus, Bee s'inscrit dans le courant dominant depuis environ vingt ans qui préconise la production de petits exécutables autonomes.

Les environnements Lisp étaient plus conçus pour une interaction incessante avec les programmeurs. Il était possible presque à tout moment de changer dynamiquement telle ou telle partie d'un programme. Les environnements mélangeaient habilement code interprété et code compilé ce qui permettait assez simplement la redéfinition de toutes les fonctions du système. Cette pratique n'a pas survécu à la recherche de l'amélioration des performances qui s'est opérée depuis les années 80. Il ne serait toutefois pas impensable que ces « vieux » environnements reviennent au goût du jour. Après tout, n'a-t-on pas besoin pour la programmation moderne d'environnements où le code peut être (re)chargé dynamiquement ? N'a-t-on pas besoin de plus d'interaction avec les utilisateurs ? Est-ce qu'un système comme Netscape avec ses *plugins* et ses *applets* n'est-il pas, conceptuellement, assez proche de ces anciens environnements interactifs ? Cette idée est à la base de l'architecture du système DrScheme [82]. Dans sa version minimale le système est très rudimentaire mais il peut être étendu par le biais d'outils additionnels. Ainsi, par exemple, le debugger statique *MrSpidey* ou encore le vérificateur de syntaxe *CheckSyntax* peuvent être ajoutés dynamiquement au système.

Chapitre 5

Conclusion

In computing, invariants are ephemeral.
– Alan Perlis

EN 1991, R. Gabriel affirmait qu'il manquait à Lisp deux points essentiels qui menaçaient son existence même [86] : *1)* une connexion avec C et *2)* la capacité de produire de petits exécutables autonomes. Bee allie ces deux traits. Nous ne pensons toutefois pas que cela soit suffisant pour garantir la pérennité de Scheme. Scheme a des arguments à faire valoir que nous avons essayé de mettre en avant tout au long de ce mémoire. Pour des applications de taille moyenne, c'est-à-dire des applications pouvant être réalisées par une seule personne ou par une petite équipe, nous sommes persuadé que son expressivité, sa concision et son haut niveau d'abstraction permettent des temps de développement extrêmement brefs. Ces belles propriétés sont toutefois inutiles sans un environnement de programmation permettant de les mettre en application dans des contextes « réels ». C'est l'ambition du travail présenté dans ce mémoire.

Nous avons réalisé Bee, un environnement de programmation intégré pour Scheme, qui offre les caractéristiques et les fonctionnalités suivantes :

- Un compilateur optimisant (Bigloo) qui produit de petits exécutables autonomes aux performances raisonnables (dans les cas les plus favorables, elles sont comparables à celles d'applications C).
- Un langage de module qui permet la structuration des applications, la compilation séparée et la création de librairies précompilées.
- Une couche objet qui se fond dans le modèle de la programmation fonctionnelle et dans celui imposé par les modules de Bee. En particulier, il n'y a pas d'ambiguïté sur le rôle des classes dans Bee. Elles ne servent pas à structurer les programmes car ce rôle est déjà assuré par les modules. À la place, elles servent à représenter les structures de données et elles permettent d'écrire des programmes plus facilement extensibles.
- Une librairie pour la construction d'interfaces graphiques (Biglook). Plus que toutes les autres auxquelles nous l'avons comparée (exception faite de LabTk [85]), elle permet d'écrire des applications graphiques en très peu de lignes de code. Biglook propose une *API* originale qui exploite tous les traits de haut niveau de Scheme et de Bee. Ainsi, il est possible d'adopter un style entièrement déclaratif et compact pour les interfaces.
- Un environnement de programmation intégré qui, à ce jour, contient un navigateur de code source (Kbrowse), une interface d'accès à la documentation en ligne, un *profiler* (Kprof) et un *debugger* (Kbdb). Bientôt de nouveaux composants seront ajoutés à l'environnement. En particulier, un constructeur d'interfaces (Buildoo) sera prochainement disponible.

Pour construire cet environnement complet nous avons dû étendre le langage Scheme en plusieurs points. Certaines de ces extensions sont simplement de nouvelles définitions de fonctions (par exemple, la bibliothèque d'exécution de Bigloo dispose d'une série de fonctions permettant l'accès aux fichiers de la plateforme d'accueil) mais parfois, ce sont de nouvelles constructions linguistiques qui n'ont pas forcément d'équivalents connus dans d'autres systèmes. Dans ce mémoire nous en avons montré deux : *1)* les classes larges et *2)* les champs virtuels. Bigloo en contient d'autres comme :

- Les *lecteurs rationnels* décrits dans l'article [200] qui peuvent être utilisés en collaboration avec les *lecteurs algébriques* implantés par D. Boucher pour lire des textes structurés.
- Les filtres non linéaires de C. Queinnec [169] implantés par J-M Geffroy [172].
- L'interface système évoluée qui permet, par exemple, la manipulation de processus, *sockets* et *pipes*. Bigloo facilite la programmation de ces objets système en proposant une interface de haut niveau qui tente de masquer tous les détails inutiles pour la majorité des applications simples. Ces constructions sont directement inspirées, tant pour leur interface de programmation Scheme que pour leur implantation, du système STk de E. Gallesio [87].

Enfin, la caractéristique principale de Bee est probablement la connexion proposée entre Scheme et C. Pour un coût très faible, il est possible depuis du code Scheme : *1)* d'appeler du code C, *2)* d'utiliser des variables et macros C et même, *3)* d'utiliser directement toute structure de données C (valeurs immédiates, tableaux et valeurs agrégées). Le code Scheme peut également être appelé depuis C. Une application peut donc faire des va-et-vient incessants entre Scheme et C sans être pénalisée par de mauvaises performances. Cette caractéristique essentielle a de fortes conséquences sur la bibliothèque d'exécution et la compilation mais nous avons montré qu'un tel système peut avoir de bonnes performances [212, 204, 213, 211].

5.1 Bilan

Bee est en cours de réalisation. Même si sa composante essentielle, le compilateur Bigloo, est distribuée depuis environ 7 ans, l'environnement dans son ensemble est encore jeune car sa distribution n'a pas plus d'un an. Dans les premiers paragraphes de cette conclusion nous avons rappelé les principaux résultats obtenus que nous avons présentés dans ce mémoire. Ces résultats sont tous purement techniques. Il nous faut aussi faire un bilan plus global de la portée de nos travaux.

5.1.1 La popularité de Scheme

Le sort de Bee est intimement lié à celui de Scheme. Évaluer la popularité de Scheme est donc important parce que cela permet d'estimer l'impact que Bee peut avoir sur la communauté des langages fonctionnels. De toute évidence Scheme n'a plus le succès qu'il a eu lors de son apparition à la fin des années 70 et au début des années 80 et qu'il a gardé jusqu'au début des années 90. Qu'en est-il néanmoins du langage ? La dernière version du document normatif [120] de Scheme n'a apporté que relativement peu de modifications par rapport à la précédente version [49]. Visiblement le langage a atteint une sorte de maturité suffisante et probablement ses évolutions ne seront plus le fait d'un comité des « sages » prenant des décisions collégiales. Son évolution ne se fera que sur la base d'avancées pragmatiques. Le processus des *SRFIs* que nous avons mentionné dans le chapitre 2 peut être un de ces moyens mais c'est plutôt par le biais de quelques implantations dominantes que les innovations seront apportées.

Scheme reste un langage assez largement utilisé dans l'enseignement. Sa position est relativement bien assise dans les universités où visiblement il est employé avec satisfaction. À ce titre, le projet DrScheme, spécialement conçu pour l'enseignement doit être salué car il permet de disposer d'un environnement complet fonctionnant sur toutes les plateformes systèmes majeures actuellement disponibles.

L'enseignement n'est pas tout. Paradoxalement même, alors que Scheme n'est pas spécialement dans l'air du temps il n'y a jamais eu autant d'applications qui s'en servent. Cela est probablement dû au militantisme assez inattendu, mais avec lequel, bien sûr, nous sympathisons, de la *Free Software Foundation*. Scheme est devenu le langage de scripts « officiel » du projet Gnu et il est donc employé dans un certain nombre de projets. C'est souvent le système Guile qui est utilisé mais c'est aussi parfois des versions *ad hoc* spécialement développées. Parmi les projets qui emploient Scheme ou Lisp on peut citer : Emacs (l'éditeur), Gimp (le programme de manipulation d'images) et Sawfish (un gestionnaire de fenêtres pour X). Il est donc possible d'utiliser, aujourd'hui, un environnement où l'éditeur se programme en Lisp et où la configuration de l'environnement graphique et la manipulation d'images se font en Scheme ; si en plus, on se sert du *shell* Scheme Scsh de O. Shivers alors on dispose d'un environnement presque digne d'une Lisp Machine ! Bee y a clairement sa place puisqu'il permet notamment l'écriture de petites applications système en Scheme.

5.1.2 L'impact de Bee

La conclusion de la précédente section ne doit pas être prise trop au sérieux même s'il est réellement possible de constituer un environnement où les principaux outils se programment en Lisp. Quelles sont néanmoins les chances de succès de Bee dans les environnements modernes ? Pour répondre à cette question il nous faut, au préalable, décider à partir de quand on peut parler de réussite. Pour nous, elles passent par deux points :

- Si Bee (et donc Bigloo) ont un nombre d'utilisateurs suffisants.
- Si Bee reste un vecteur de recherche probant et stimulant.

Pour l'instant ces deux conditions sont vérifiées. Bigloo dispose de nombreux utilisateurs, plus souvent industriels qu'académiques. Nous espérons que nos efforts en terme de programmation graphique permettront d'en augmenter encore le nombre. De plus, Bee est toujours un élément de très forte motivation. C'est le temps qui nous manque. Bien souvent les sept jours de la semaine ne suffisent pas à la réalisation de nos nouveaux projets. Immanquablement le dilemme entre choisir de rédiger des articles sur des travaux terminés ou d'entreprendre de nouveaux se pose à nous.

5.2 Les futurs travaux

Comme nous l'avons mentionné en conclusion du chapitre 4, nous allons prioritairement poursuivre nos recherches sur la mise au point des programmes Bigloo et en particulier, explorer de nouveaux modes dynamiques de visualisation du tas pour mieux comprendre les allocations des programmes. Ces recherches dépassent le cadre de Scheme car elles peuvent avoir un impact sur tous les langages qui utilisent des GCs (ML et Java par exemple). Il s'agit là d'une recherche à moyen terme qui, nous l'espérons, pourra faire l'objet d'une thèse. Plusieurs voies sont à tracer. En utilisant les informations actuellement disponibles dans la bibliothèque d'exécution, il faut trouver de meilleures représentations. Nous pensons également qu'il est possible de construire de nouveaux outils de développement qui s'appuient fortement sur le GC. C'est là un terrain entièrement vierge où les outils seront à définir, à implanter et à intégrer dans Bee.

5.2.1 De la neutralité...

Avec la composante Bigloo(k) de Bee nous sommes parvenus à concevoir un système satisfaisant sur le plan technique (même si de nombreuses améliorations doivent être apportées au système). Bigloo(k) allie la rapidité d'exécution des programmes compilés et un grand confort de programmation. Nous voulons mettre en avant ces arguments pour le faire adopter le plus largement possible. Néanmoins, actuellement, Bigloo(k) souffre d'une lacune : il n'a pas encore trouvé d'application maîtresse (de *killer application* en anglais). C'est une chose que nous nous efforcerons de faire dans un avenir proche. Pour cela, nous voulons proposer des extensions de Bigloo(k) qui lui permettront d'être la solution technique la plus confortable pour la résolution de problèmes spécifiques. Par exemple, nous allons étudier les possibilités de connecter fortement Bigloo(k) (ou des extensions) à un système de fenêtrage. Cela permettrait de développer simplement des applications séduisantes.

Très rapidement, nous pensons abandonner la position de neutralité vis-à-vis des systèmes d'exploitation qui a été la nôtre depuis le début de nos travaux. Compiler vers C plutôt que vers des langages d'assemblage relevait par exemple de ce principe. Ce positionnement n'était possible que tant que Scheme était suffisamment populaire et que le simple fait de proposer une implantation efficace du langage assurait à ce système un succès certain. Scheme n'en est plus là. Comme nous l'avons dit précédemment, Scheme (ou des variantes de Scheme) sont utilisées dans divers projets qui peuvent être importants mais le langage en tant que tel n'est plus à la mode. Le seul moyen pour l'imposer ainsi que notre environnement est de démontrer sa qualité par des applications modernes et séduisantes. Ancrer Bee dans un système d'exploitation est le moyen le plus sûr d'y parvenir.

5.2.2 Modèle d'exécution

Lorsque nous avons entrepris en 1991 le compilateur Bigloo, C détenait une position dominante. À cette date, nous pensions que C, constituait le moyen pour disposer d'un système portable et évolutif. Avec presque dix années de recule, ce choix nous semble toujours être valable. C ne permet peut-être pas d'obtenir le système le plus rapide (en particulier à cause du GC) mais il permet d'obtenir un « bon »système dont l'architecture ne sera pas remise en cause par l'apparition de nouvelles architectures. Par exemple, l'apparition de l'IA-64 d'Intel et Hewlett-Packard, n'a introduit aucune modification dans Bigloo. En compilant vers C, on hérite des qualités de ce langage : il est bien implanté et sa pérennité est encore assurée pour de nombreuses années.

Toutefois, C(/C++) a perdu de sa domination au profit de Java. Java n'est pas efficace mais il présente plusieurs atouts :

- Il n'est pas plus portable que C, mais il permet l'interopérabilité que C ne permet pas (le fichier `.class` peut être exécuté sur des plateformes différentes).
- Java dispose de très nombreuses bibliothèques.
- Java s'intègre plus harmonieusement que C, dans les perspectives contemporaines de l'informatique. Java permet plus simplement que C, le chargement dynamique de programmes. Il est plus facile de réaliser des systèmes embarquant des moteurs Java que des moteurs C.
- Java présuppose l'existence d'un environnement d'exécution standard : la JVM. C'est cette caractéristique qui lui donne nombre de ses avantages mais aussi de ses inconvénients (en particulier sa lenteur). Disposer d'une machine virtuelle permet d'envisager simplement la réalisation outils de programmation évolués. Par exemple, il est plus simple de réaliser un outil de mise au point permettant l'exécution pas à pas pour une machine virtuelle que pour du code natif.

Pour ces quatre raisons, il nous semble pertinent d'écrire un nouveau générateur de code pour Bigloo. En plus de produire du C, nous envisageons d'ajouter un générateur de code JVM pour Bigloo. Cela nous permettra de bénéficier des efforts consentis par la communauté Java pour réaliser un environnement de programmation confortable. Nous pourrons alors détourner les outils Java comme nous l'avons fait pour les outils C. Nous pourrons également simplement créer nos propres outils parce que les versions modernes de la JVM permettent l'introspection de l'exécution. Notre but n'est pas ici d'abandonner l'exécution basée sur C, nous pensons parvenir à maintenir les deux générateurs de code. Les utilisateurs de Bigloo pourront alors choisir entre la production d'exécutables efficaces, en passant par C, ou alors la production d'exécutable portable et pouvant être embarqués, en passant par Java.

Chapitre 6

Bigloo : a portable and optimizing compiler for strict functional languages

Cet article a été publié dans [213]. Il a été écrit en collaboration avec Pierre Weis.

Bigloo : un compilateur portable et optimisant pour les langages fonctionnels stricts

Nous présentons Bigloo, un compilateur très optimisant. Bigloo est le premier compilateur pour langages fonctionnels qui peut compiler plusieurs langages sources : il est le premier compilateur mixte Scheme *et* ML et, pour ces deux langages, il est l'un des compilateur les plus efficace actuellement existant (Bigloo est disponible par ftp à l'adresse `ftp.inria.fr` [192.93.2.54]).

Le haut niveau de performance de Bigloo est atteint par l'emploi de nombreuses optimisations. Certaines sont des optimisations classiques que nous avons adaptées aux langages fonctionnels (comme par exemple l'intégration fonctionnelle), d'autres sont spécifiques et n'existent que dans Bigloo (étude de fermetures sophistiquée, une représentation impérative des variables, des analyses du flot de contrôle d'ordre supérieur). Toutes ces optimisations contribuent au même objectif : réduire le nombre d'allocation dans le tas.

Bigloo : a portable and optimizing compiler for strict functional languages

We present Bigloo, a highly portable and optimizing compiler. Bigloo is the first compiler for strict functional languages that can efficiently compile *several languages* : Bigloo is the first compiler for full Scheme *and* full ML, and for these two languages, Bigloo is one of the most efficient compiler now available (Bigloo is available by anonymous ftp on `ftp.inria.fr` [192.93.2.54]).

This high level of performance is achieved by numerous high-level optimizations. Some of those are classical optimizations adapted to higher-order functional languages (e.g. inlining), other optimization schemes are specific to Bigloo (e.g. a new refined closure analysis, an original optimization of imperative variables, and intensive use of higher-order control flow analysis). All these optimizations share the same design guideline : the reduction of heap allocation.

6.1 Introduction

Strict functional programming languages have many different variations, but they all belong to the same family, the so-called "λ-languages" family. The Bigloo compiler is devoted to the compilation of this class of languages. It was not designed to be the compiler of a specific programming language. It is carefully crafted to be a good compiler for the untyped λ-calculus with n-ary functions, and features many analyzes and optimizations to efficiently deal with functions.

To turn this optimizing functional core language compiler into a compiler for a full language, Bigloo provides room for the addition of preprocessors before the beginning of the compilation process. This is mandatory since Bigloo has almost no hard-wired hypotheses about its high-level source language : for instance Bigloo does not assume the source language to be type checked, neither statically or dynamically. If necessary, and before the compilation by Bigloo, a preprocessing pass must explicitly indicate these runtime type-tests in the source text. This preprocessing mechanism is powerful enough to change the syntax of the source language : it suffices to write a preprocessor that generates λ-terms from whatever the source level syntax could be.

Having to compile a large spectrum of languages, the compiler must also be independent of the set of primitives of the language : Bigloo's library has been designed to be easily modified when changing the source language.

This technique has already been used to turn Bigloo into a compiler for various functional languages : ML [255, 256], Scheme [114], Meroon [170] ; a preprocessor for the Dylan language [12] is on the way. The compilers obtained by adding these preprocessors to Bigloo are efficient, and in each case, they compare favorably to the best compilers specialized to the source language.

Bigloo is also highly portable : it virtually exists on every Unix platform. This quality is due to its target language : Bigloo generates C code.

In this paper we present an overview of analyses and optimization used in Bigloo (more detailed descriptions of the algorithms can be found in [204]). In section 6.2 we explain why we use C as target language, and in section 6.3 why Bigloo generates "natural" C code. In section 6.4 we describe Λ^n, the Bigloo source language. In section 6.5 we present general optimizations. Section 6.6 is devoted to the presentation of the Scheme and ML front-ends. Before concluding, we present benchmarks in section 6.7.

6.2 Portability

The C programming language plays a crucial role in today's computers equipment. With their new computers, most vendors offer an optimizing C compiler which is able to exploit the processor's new capabilities. C is so widely used that it presumably drives the design of new computer architectures. Even for such low-level tasks as system programming, computers are now designed to be programmed in C, and no more directly in assembly code. As practical evidence of this, we can cite the lack of documentation for some assembly languages and the strange behavior of some proprietary C compilers that correct the defects of the hardware by avoiding the generation of instruction sequences that will fall into a processor bug.

Because C runs on so many machines, a C program is highly portable : having portability in mind, one can reduces machine dependent parts to a minimum. Moreover, C compilers produce efficient code. For these reasons, we choose C as the target language of our functional language compiler.

6.3 Which kind of C code to generate ?

Using C, we get almost for free a portable compiler. Unfortunately this advantage has some drawbacks : as desired, C lets us ignore the machine specific features, but this abstraction may slow down the code produced by our compiler, or complicate the generation of C code. In effect, the C compiler's way to implement some constructs may not fit the semantics of our input languages, or may result in an inefficient implementation of the corresponding input language construct. The solutions to overcome this problems are highly dependent of the kind of C code generated by the compiler. They are many variations, but two main directions emerge :

1. the generation of C code that mimics a virtual machine, in this case the C compiler is considered as a virtual assembler.
2. the generation of C code that resembles handwritten C code.

Each method has its own advantages and its own drawbacks, but we claim that an optimizing compiler must generate "natural" C code (direction 2).

6.3.1 C as a virtual assembly language

If we use C as an assembly language, we abandon the usage of C control structures. In the C code generated by the compiler there are no C functions, the C stack is hardly ever used, and C variables just serve to maintain the registers of some abstract machine. In this case, the loss of control over the underlying hardware is minimized, since the compiler generates code for an abstract machine which is completely under its control. That way, source language features that need some knowledge about the runtime behavior of programs are easier to compile (e.g. garbage collection or the Scheme `call/cc` function). Unfortunately, these advantages are paid a high price : since the C code generated by the compiler bears no resemblance with the one written by C programmers, it is likely that C compilers will fail to optimize this code properly. The numerous optimizations performed by C compilers will probably not apply and the functional languages compiler will not benefit from the high quality of C compilation.

6.3.2 "Handwritten-like" C code

The alternate C code generation method is to mimic handwritten C code. In contrast with the preceding approach, the source language features that need to know the runtime behavior of programs are difficult to implement (and even more difficult to implement efficiently). The `call/cc` function is now hard to implement, and the memory management is constrained by the presence of C values in the runtime space of the program (for instance C values stored into locations in the stack) : garbage collection must deal with ambiguous roots. On the other hand, we gain a good compilation of our generated C code, resulting in good and homogeneous performances not bound to some particular machine architecture. As an added benefit of this "standard C code" generation, we can run the C programming environment tools on the generated C code : symbolic debuggers, code analyzers (such as `purify`) or profilers.

In this method of C code generation, every source language construct is compiled in an equivalent C construct. More precisely, if an equivalent C construct exists, then the source language construct is compiled in that C construct. For instance, source language functions are compiled into C functions (or even in C loops when the source language functions are tail-call loops), source language variables are compiled into C variables, and so on. This mapping is an instance of what we call the "natural projection" from one language to the other. Bigloo uses this technology and projects λ-terms to handwritten-like C code. To manage C values in the stack, Bigloo uses the Boehm garbage collector [25].

6.4 The Λ^{n} language

The input language of Bigloo is a non-standard version of the λ-calculus : in this Λ^{n}-calculus the λ-abstractions are not restricted to abstract one variable at a time. Similarly the application is not binary, as usual in λ-calculus, but n-ary. This n-ary λ-calculus clearly contains regular λ-terms. For instance the λ-term $\lambda x.\lambda y.x$ can be encoded in Λ^{n} as a term T_1 that abstracts one variable twice : $T_1 = \lambda^1 x.\lambda^1 y.x$. Alternatively, it can be encoded as a single abstraction of two variables : $T_2 = \lambda^2 xy.x$. Terms T_1 and T_2 have different meaning and must not be confused : when applying Λ^{n} terms there is a static arity consistency check that forbids inconsistent partial application ; thus T_2 cannot be applied to only one argument, while T_1 cannot be applied to two arguments : T_1 must be applied twice to only one argument at a time. For instance we can write $\texttt{@}^1(\texttt{@}^1(T_1, e_1), e_2)$ reflecting the fact that T_1 is partially applied to e_1, leading to a function result, which is then applied to e_2. On the contrary $\texttt{@}^2(T_2, e_1, e_2)$ expresses the fact that T_2 is directly applied to two arguments and does not require any partial application.

Λ^{n} calculus is call-by-value, and its semantics can be expressed using the classical weak β-reduction (with multiple substitutions in parallel to deal with n-ary abstractions and applications).

To get the complete input language of the compiler, the basic Λ^{n} calculus is extended with several other constructs : `let` bindings, conditionals, `case` constructs, and constants. Note that `let` bindings introduce variables which can be assigned to (to deal with the imperative features of the source language). Note also that definitions introduced by `let` can be recursive. For the sake of simplicity we only consider integer constants.

Λ^{n}	::=	
|	i	*integer constants*
|	id	*variable*
|	$\lambda^m\ id_1 \ldots id_m . \Lambda^{\mathrm{n}}$	*n-ary function*
|	$\texttt{@}^m(\Lambda^{\mathrm{n}}{}_1, \ldots, \Lambda^{\mathrm{n}}{}_m)$	*n-ary application*
|	`let` $\{id = \Lambda^{\mathrm{n}}\}^+$ `in` Λ^{n}	*let binding*
|	`if` Λ^{n} `then` Λ^{n} `else` Λ^{n}	*conditional*
|	`case` $\{$`[` i `:` Λ^{n}`]`$\}^+$ `[[else :` Λ^{n}`]]`	*integer switch*

6.5 Optimizing Λ^{n} code

The compilation scheme we use, (called "natural projection" to C), requires two kinds of high level optimizations : *(i)* optimizations to improve the natural projection (careful selection of target C constructs) *(ii)* source to source transformations on Λ^{n} code to perform optimizations that the C compiler cannot do because these optimizations require semantics knowledge about the high level source language (e.g. source language module informations).

6.5.1 Optimizing the "natural projection"

The main design effort for Bigloo has been to compile functions as well as possible : in our mind this implies the mapping of Λ^{n} functions to C functions (or even to C loops) and to hard work to avoid heap-allocation of closures.

Closure analysis

We name *closure analysis* the methodology and algorithms used in Bigloo to compile functions efficiently. It is described in the paper [206]. It involves an abstract interpretation which is based on Shiver's 0cfa analysis [218], and uses an algorithm described by Séniak in [199]. We do not present the closure analysis again, but give its results for some examples to give an idea of the way Bigloo compiles functions.

For simplicity, examples are not given using the compiler source language (Λ^{n} code) but in high level source languages (Scheme or ML).

Compiling functions to C loops Closure analysis aims at compiling Λ^{n} functions into C loops. This scheme applies when Λ^{n} functions do not escape (that is, when functions are not used as first-class values) and when these functions are always tail-recursively invoked. Here is an example of two mutually recursive functions that map to C loops :

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

is compiled as :

```
obj n, m;
n = 10;
_odd:
if( n == 0 ) return FALSE;
else { m = SUB( n, 1 );
_even:
   if( m == 0 ) return TRUE; else { n = SUB( m, 1 ); goto _odd }
}
```

Compiling functions to C functions When the previous scheme does not apply, non escaping n-ary functions map to C n-ary functions.

Consider the ML `map_succ` definition :

```
let map_succ l =
  let rec map = function [] -> []
                       \| x :: l -> 1 + x :: map l in
  map l;;
```

It is compiled as :

```
obj map_succ( obj l ) { return map( l ); }

static obj map( obj l )
{
   if( NULLP( l ) ) return NIL;
   else return MAKE_PAIR( ADD( 1, CAR( l ) ), map( CDR( l ) ) );
}
```

Heap-allocated closures When functions escape, these efficient mappings do not apply : the compiler must allocate heap space for function environments. Bigloo uses flat closures : the environment part is a heap block containing exactly the free variables of the function. Despite the larger heap-allocation needed for flat closures, we do not use linked environments since they lead to memory leaks which cannot be circumvented by the user.

Thus, only escaping functions are allocated in the heap. This is the case for the λ-expression returned as value by the functional composition of two functions :

```
1 :    let o f g =
2 :        function x -> f (g x)
```

Hence Bigloo creates a heap-allocated closure for the expression of line *2* :

```
obj o( obj f, obj g )
{
   obj clo;
   clo = make_closure( lambda_1, f, g );
   return clo;
}

obj lambda_1( obj clo, obj x )
{
   obj f, g;
   f = PROCEDURE_REF( clo, 0 );
   g = PROCEDURE_REF( clo, 1 );
   return PROCEDURE_ENTRY( f )( f, PROCEDURE_ENTRY( g )( g, x ) );
}
```

As mentioned above, Bigloo's closures are arrays. Each closure contains the free variables of the function, pointers to C functions, and an integer to record the function arity. This arity slot is mandatory for dynamically typed languages to ensure soundness of functional application. Amazingly enough it is not useless for statically typed languages : it is used to optimize n-ary computed calls (applications where the functions called are not known at compile-time). Thanks to the arity slot, partial application can be checked dynamically, and if the computed call is total, then the "uncurried" entry point of the function is called (see section 6.5.1). This prevents closure allocations for n-ary computed calls.

Even functions used as first class values can be compiled without heap-allocation using the 0cfa analysis. In the paper [206] we have shown that, in general, with our various optimizations, less than 20 % of functions require to be heap allocated.

The $\mathcal{C}$-transformation

Λ^{n} terms faithfully reflect functions arity and closures creation. In particular, the formalism expresses that n-ary functions can be applied to n arguments without any intermediate closure creation. The $\mathcal{C}$-transformation (uncurrying) attempts to turn a spine of unary abstractions into a single n-ary abstraction. Conversely, spines of unary applications are replaced by single n-ary applications. In any case, this transformation speeds up function applications and reduces heap allocation. For some source languages which do not feature n-ary functions, this optimization is essential. For instance, applied to Caml, the $\mathcal{C}$-transformation speeds up compiled programs by a factor of two. In effect, in Caml, n-ary functions are encoded by the programmer as n-level nested unary functions. We use as benchmark the bootstrap of the Caml compiler (12.000 lines of code).

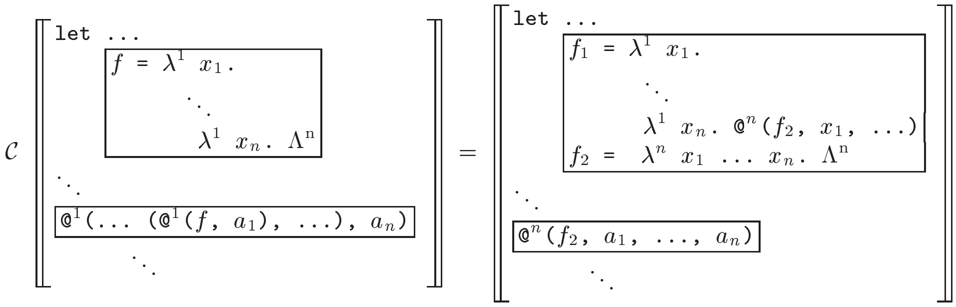


FIG. 6.1 – The $\mathcal{C}$-transformation

The $\mathcal{C}$-transformation (figure 6.1) produces for every curried function f two function definitions, f_1 and f_2. The function f_1 is used for partial applications of f whereas f_2 serves for total applications. The function f_2 is n-ary, hence total applications of f become n-ary applications.

Functions f_1 and f_2 are related to f by a naming convention so that the $\mathcal{C}$-transformation also applies globally and even across module barriers. Moreover f_1 and f_2 are dynamically linked, so that n-ary computed calls to f can benefit from the transformation.

The $\mathcal{C}$-transformation can be illustrated by the following Scheme code program transformation :

```scheme
(define add
   (lambda (x)
      (lambda (y)
         (+ x y))))

((add 0) 1)
```

```scheme
(define add1
   (lambda (x)
      (lambda (y)
         (add2 x y))))
(define add2
   (lambda (x y) (+ x y)))

(add2 0 1)
```

η-transformation

The η-transformation optimization belongs to the same class as the $\mathcal{C}$-transformation : it avoids intermediate closures creation and partial evaluations. The η-transformation (figure 6.2) encapsulates *function definitions* into extra functional abstractions.

The following source to source rewriting illustrates the transformation :

```
let do_list f =
  let rec loop = function
      [] -> ()
    \| a::l -> f a; loop l
  in loop;;

do_list print_int [1; 2];;
```

```
let do_list f new =
  let rec loop = function
      [] -> ()
    \| a::l -> f a; loop l
  in loop new;;

do_list print_int [1; 2];;
```

The functional `do_list` is in fact binary but the original program hides this fact, since the partial evaluation of `do_list` to a function explicitly returns a closure. In the rewritten program, the definition of `loop` is abstracted into an extra λ-abstraction `function new -> let rec loop ....` Hence `do_list` becomes binary and the $\mathcal{C}$-transformation now applies.

This transformation has not to be confused with the η-rule of λ-calculus : in the presence of side-effects the η-rule of λ-calculus is not valid in general for arbitrary expressions. Fortunately, our transformation uses the η-rule for syntactic *functions*, for which the η-rule is always valid.

$$\eta\left[\!\!\left[\begin{array}{l}\boxed{\begin{array}{l}\texttt{let } f\ \texttt{=}\\ \quad \lambda^1\ x_1\,.\\ \qquad\ddots\\ \qquad\quad \lambda^1\ x_n\,.\ \Lambda^{\mathrm{n}}\end{array}}\\ \quad\texttt{in}\\ \quad\boxed{\texttt{@}^1(\ldots\ (\texttt{@}^1(f,\ a_1),\ \ldots),\ a_m)}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\lambda^1\ \texttt{new}_n\,.\\ \quad\ddots\\ \quad\lambda^1\ \texttt{new}_{m+1}\,.\\ \qquad\boxed{\begin{array}{l}\texttt{let } f\ \texttt{=}\\ \quad \lambda^1\ x_1\,.\\ \qquad\ddots\\ \qquad\quad \lambda^1\ x_n\,.\ \Lambda^{\mathrm{n}}\end{array}}\\ \quad\texttt{in}\\ \quad\boxed{\begin{array}{l}\texttt{@}^1(\ldots\\ \ (\texttt{@}^1(\ldots\ (\texttt{@}^1(f,\ a_1),\ \ldots),\\ \ \ a_m),\ \ldots),\ \texttt{new}_n)\end{array}}\end{array}\right]\!\!\right]$$

FIG. 6.2 – The η transformation ($n > m$).

The relevance of this optimization depends on the programming style. In practice we have noticed that Caml programs are sensitive to η, especially library routines (the `do_list` example is extracted from the Caml standard library). When η applies, the impact is impressive : thanks to η, the **kb** program (section 6.7.2) and the Coq system [67] run two times faster (the Coq system is 20 000 lines of Caml code long, and the benchmark runs during more than 30 minutes on a dec alpha station, allocating 12 gigabytes).

The $\mathcal{I}$-transformation

The $\mathcal{I}$-transformation substitutes cascades of nested tests by n-way branches. This is just rewriting some nested conditionals in the Λ^{n} code into a `case` construct (figure 6.3).

The scope of $\mathcal{I}$ is limited to cascades of tests that verify :

- Every test is an expression ${\Lambda^{\mathrm{n}}}_c = i$, for some integer i and some fixed expression ${\Lambda^{\mathrm{n}}}_c$.
- The expression ${\Lambda^{\mathrm{n}}}_c$ is side effect free (since after $\mathcal{I}$ rewriting, ${\Lambda^{\mathrm{n}}}_c$ is evaluated only once).

$$\mathcal{I}_{trans}\left[\!\!\left[\begin{array}{l}\texttt{if } \Lambda^{\mathrm{n}}{}_c = i_0 \\ \quad \texttt{then } \Lambda^{\mathrm{n}}{}_0 \\ \quad \texttt{else} \\ \quad \ddots \\ \qquad \texttt{if } \Lambda^{\mathrm{n}}{}_c = i_m \\ \qquad\quad \texttt{then } \Lambda^{\mathrm{n}}{}_m \\ \qquad\quad \texttt{else } \Lambda^{\mathrm{n}}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{case } \Lambda^{\mathrm{n}}{}_c \\ \quad \texttt{[}i_0 \texttt{ : } \Lambda^{\mathrm{n}}{}_0\texttt{]} \\ \qquad \dots \\ \quad \texttt{[}i_m \texttt{ : } \Lambda^{\mathrm{n}}{}_m\texttt{]} \\ \quad \texttt{[else : } \Lambda^{\mathrm{n}}\texttt{]}\end{array}\right]\!\!\right]$$

FIG. 6.3 – The $\mathcal{I}$-transformation

The $\mathcal{I}$-transformation simplifies pattern matching expansion : specific pattern matching compilers (those for ML or extended Scheme) have not to take care of this optimization, which is performed by Bigloo when compiling Λ^{n} expressions.

6.5.2 Optimizing Λ^{n} source code

Inlining

Open-coding (or inlining) suppresses some function calls, replacing these calls by the body of the function. Functions are so widely used and they are so many function calls in functional programs that inlining has a large impact on efficiency. This impact is mostly due to the functional programmer habit to define many small functions to improve the readability of their code. With a efficient inlining strategy, this good style of programming does not compromise efficiency.

Inlining improves compiled code in several aspects : *(i)* the function invocation cost disappears (that includes the cost of parameter passing instructions, context switch, register moves, and the jump to the function code that breaks the control flow) *(ii)* local optimizations of the compiler are more relevant since they apply to larger code blocks *(iii)* recursive functions inlining reduces dynamic closure allocations.

One may think that Bigloo could delegate inlining to its C compiler back-end. C compilers may have some inlining strategy, but this strategy is never aggressive enough to efficiently compile functional programs. In addition, the inlining pass of Bigloo extends the scope of other optimizations (for instance the data-flow optimizations).

Which functions to inline ? Inlining is very efficient but has to be "handled with care" : inconsiderate inlining may imply an unbearable growth of the compiled code size. To prevent code explosion, Bigloo inlines only functions that verifies some pragmatic properties :

- either the function is called only once in the entire program (in this case there is no code expansion) ;
- or the body of the function is small enough : the size of its Λ^{n} abstract syntax tree is less than a value $\mathcal{S}$; $\mathcal{S}$ depends on the numbers of parameters of the function and on the optimization level of the compiler.

In addition, inlining cannot occur in some contexts :

- No inlining of f can occur when inlining the body of f. This prevents infinite inlining even in case of mutually recursive functions.
- Nested inlining depth is limited by the compiler since unlimited nested inlining may produce code explosion (when inlining a small function that calls another small function that calls ..., the previous criteria do not apply, even if we do want to stop inlining since the code could be growing too much).

The inlining process As mentioned above, Bigloo prevents recursive inlining of the same function, but it still accepts to inline *recursive definitions*. Bigloo features an original and efficient scheme to perform inlining of recursive functions.

In case of inlining of non recursive functions the $\mathcal{L}_{let}$ transformation (figure 6.5.2) applies. Inlining of recursive functions is a bit more delicate. Rather than unrolling recursive calls to a certain depth, Bigloo creates a local recursive definition for the open-coded function. When inlining recursive function f, $\mathcal{L}_{rec}$ (figure 6.5.2) creates the local definition and then replaces calls to f by calls to this local function.

$$\mathcal{L}_{let}\left\llbracket\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ x_m.\ \Lambda^{\mathrm{n}}\\ \ddots\\ \quad\boxed{\texttt{@}^m\texttt{(}f\texttt{,}\ a_1\texttt{,}\ \ldots\texttt{,}\ a_m\texttt{)}}\\ \qquad\ddots\end{array}\right\rrbracket = \left\llbracket\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ x_m.\ \Lambda^{\mathrm{n}}\\ \ddots\\ \quad\boxed{\begin{array}{l}\texttt{let}\ x_1\ \texttt{=}\ a_1\\ \qquad\vdots\\ \quad\texttt{in}\\ \quad\Lambda^{\mathrm{n}}\end{array}}\\ \qquad\qquad\ddots\end{array}\right\rrbracket$$

$$\mathcal{L}_{rec}\left\llbracket\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ .\ \Lambda^{\mathrm{n}}\\ \ddots\\ \quad\boxed{\texttt{@}^m\texttt{(}f\texttt{,}\ a_1\texttt{,}\ \ldots\ \texttt{)}}\\ \qquad\ddots\end{array}\right\rrbracket = \left\llbracket\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ .\ \Lambda^{\mathrm{n}}\\ \ddots\\ \quad\boxed{\begin{array}{l}\texttt{let}\ f'\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ .\ \Lambda^{\mathrm{n}}{}_{[f/f']}\\ \quad\texttt{in}\\ \quad\texttt{@}^m\texttt{(}f'\texttt{,}\ a_1\texttt{,}\ \ldots\ \texttt{)}\end{array}}\\ \qquad\qquad\ddots\end{array}\right\rrbracket$$

FIG. 6.4 – inlining

An inlining example To illustrate the optimization, let us define `map_succ` using the `map` functional :

```
let rec map f = function [] -> []
                       \| x :: l -> f x :: map f l;;
let succ x = x + 1;;
let map_succ l = map succ l;;
```

When inlining `map` into `map_succ` the compiler discovers that `map` is self-recursive, so it inlines it using a local definition :

```
let map_succ l =
  let rec map f = function [] -> []
                         \| x :: l -> f x :: map f l in
  map succ l;
```

A further pass of the compiler states that the formal parameter `f` is a loop invariant, so `f` is substituted by its actual value (compile-time $\beta-$reduction). We get :

```
let map_succ l =
  let rec map = function [] -> []
                       \| x :: l -> succ x :: map l in
  map l;;
```

Then, `succ` is open-coded, and we finally get a very efficient equivalent piece of code :

```
let map_succ l =
  let rec map = function [] -> []
                       \| x :: l -> x + 1 :: map l in
  map l;;
```

Impact of Inlining In [8] A. Appel demonstrates that inlining is the most efficient optimization of the Sml/NJ compiler. In order to measure the impact of inlining, we applied it to several Scheme and ML programs (including the full bootstrap of the Bigloo compiler which is 30.000 lines of Scheme code long). The effect of inlining seems to be architecture dependent : we obtained a speedup of 20 % on Sparc and 30 % on Mips. Our inlining decision algorithm is accurate, since on both architectures code growth is limited to 5 %.

Data-flow analysis

Cse Common subexpression elimination (CSE) is a classical optimization which avoids multiple computations of pure expressions [3]. It is included in many compilers (gcc, Sml/NJ, SELF [38], …).

This optimization is generally performed when the program has been translated into a low level language not far from assembly language. In Bigloo it is performed at a higher level, since it applies to Λ^n terms. The benefit we gain is that more complex expressions can be factored by the CSE, in particular function calls become subject to elimination.

Bigloo features two different schemes to perform CSE : $\mathcal{CSE}_{prune}$ and $\mathcal{CSE}_{cond}$. Both of them require a purity analysis passe (**effect** pass). This pass is a simple annotation on expressions giving a rough approximation of the side effects that occur during the evaluation of the given expression. These annotations are exact for primitives and properly propagated to functions. This analysis is not higher-order and is local to modules (imported functions are considered impure).

Then, $\mathcal{CSE}_{prune}$ (which corresponds to usual CSE) is performed by the two extra passes :

- Let binding pass : this rewriting of the abstract syntax tree makes the control flow explicit (in the spirit of cps). It binds subexpressions to variables using `let`.
- Abstract syntax tree pruning pass (figure 6.5) : it is a recursive descent into the abstract syntax tree, which removes pure expressions already computed, replacing the expressions by the variable they are bound to. This is achieved using a stack of previously encountered pure expressions.

$$\mathcal{CSE}_{prune}\left[\!\!\left[\begin{array}{l}\texttt{let}\ \ldots\\ \qquad id\ \texttt{=}\ \Lambda^n\\ \ddots\\ \quad\boxed{\Lambda^n}\\ \qquad\quad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{let}\ \ldots\\ \qquad id\ \texttt{=}\ \Lambda^n\\ \ddots\\ \quad\boxed{id}\\ \qquad\quad\ddots\end{array}\right]\!\!\right]$$

FIG. 6.5 – The $\mathcal{CSE}_{prune}$ transformations

$\mathcal{CSE}_{cond}$ is a common subexpression elimination specialized to conditionals : like the usual CSE, it removes duplicated tests, but it also removes useless conditional expressions (figure 6.6). After a conditional test, and when analyzing the branches of the conditional, the compiler remembers the value of the test, so that nested conditionals with the same test can be reduced to one of their alternative branches according to the test value.

Note that the purity analysis is mandatory since expressions with side effects cannot be reduced by CSE.

$$\mathcal{CSE}_{cond}\left[\!\!\left[\begin{array}{l} \texttt{if}\ \textit{if-exp} \\ \quad\texttt{then} \\ \quad\ddots \\ \qquad\boxed{\begin{array}{l}\texttt{if}\ \textit{if-exp}\\ \quad\texttt{then}\ \Lambda^{\mathrm{n}}{}_{then_1}\\ \quad\texttt{else}\ \Lambda^{\mathrm{n}}{}_{else_1}\end{array}} \\ \qquad\qquad\qquad\ddots \\ \quad\texttt{else} \\ \quad\ddots \\ \qquad\boxed{\begin{array}{l}\texttt{if}\ \textit{if-exp}\\ \quad\texttt{then}\ \Lambda^{\mathrm{n}}{}_{then_2}\\ \quad\texttt{else}\ \Lambda^{\mathrm{n}}{}_{else_2}\end{array}} \\ \qquad\qquad\qquad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{if}\ \textit{if-exp}\\ \quad\texttt{then}\\ \quad\ddots\\ \qquad\boxed{\Lambda^{\mathrm{n}}{}_{then_1}}\\ \qquad\qquad\ddots\\ \quad\texttt{else}\\ \quad\ddots\\ \qquad\boxed{\Lambda^{\mathrm{n}}{}_{else_2}}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

FIG. 6.6 – The $\mathcal{CSE}_{cond}$ transformations

For instance, on the following program neither `(car x)` nor `(print 3)` can be factorized.

```
(let ((head (car x))
      (void (print 3)))
   (begin (set-car! x 4) (print 3) (car x)))
```

The purity analysis makes our CSE more powerful than the common subexpression elimination that could be performed by the back-end C compiler : the C compiler cannot optimize the second occurrence of `(car x)` since it fails to detect that `print` does not perform side-effects on `x`. Once more, this is due to extra semantic informations provided by the module interfaces of the high level source languages (in this case the Scheme library).

β-reduction The β-reduction pass suppresses useless bindings, and is widely use in optimizing compilation. β-reduction operates two transformations :

– β_{bind} suppresses aliases of never mutated variables.

$$\beta_{bind}\left[\!\!\left[\begin{array}{l}\texttt{let}\ \ldots\\ \qquad id_x\ \texttt{=}\ id_y\\ \quad\ddots\\ \qquad\boxed{id_x}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{let}\ \ldots\\ \quad\ddots\\ \qquad\boxed{id_y}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

– β_{occ} removes the binding of a non mutated variable to a pure Λ^{n} expression when the bound variable is used only once.

$$\beta_{occ}\left[\!\!\left[\begin{array}{l}\texttt{let}\ \ldots\\ \qquad id\ \texttt{=}\ \ \Lambda^{\mathrm{n}}\\ \quad\ddots\\ \qquad\boxed{id}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{let}\ \ldots\\ \quad\ddots\\ \qquad\boxed{\Lambda^{\mathrm{n}}}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

Impact of Data-flow analysis Data-flow analysis have a little impact on runtime performances (about 5 %) but significantly reduces the size of object files (about 20 %).

6.6 Specific front-ends

To obtain a compiler for a full language we must add two modules to the core Λ^{n} compiler : a runtime library and a front-end translator from the source language to Λ^{n}.

To define a new set of primitives for a new language, the Λ^{n} compiler provides a powerful interface with its target language : the "extern" interface. This tool gives the programmer full access to the C world : C functions, C macros, and C data structures are transparently available.

A specific front-end has to deal with particular features of its source language and to provide a careful natural mapping to Λ^{n}. In the following we do not detail the whole Scheme and Caml front-ends : we just present some fine points of each front-end.

6.6.1 The Scheme front-end

Dynamic type tests The mapping from Scheme to Λ^{n} is easy : the front-end is mainly in charge to make dynamic type tests explicit. Data-flow analysis optimization of the Λ^{n} compiler will automatically remove many of them.

call/cc The Λ^{n} compiler ignores the `call/cc` function which has the same status as any other primitive. This function has been implemented in C, in the Scheme specific library.

6.6.2 The Caml front-end

The Caml front-end is in charge of type reconstruction and pattern matching expansion. In addition, the front-end optimizes references. This front-end is described in details in the paper [212].

Optimizing References A naive implementation of ML references would be to use Λ^{n} arrays. A more efficient scheme is to map references to Λ^{n} variables. This is admissible when the reference is not used as a first-class value (that is passed to a function or stored into data structures). The Caml front-end includes an analysis devoted to this mapping. Let us take as example a simple while loop, using a reference to control the loop :

```
let x = ref 10 in
  while !x > 0 do print_int !x; x := !x + 1 done;;
```

The Caml front-end maps the variable `x` into an imperative Λ^{n} variable ; then, Bigloo maps the `while` construct into a C loop, and the `x` variable to a C variable :

```
{
   obj x;
   x = 10;
loop:
   if( GT( x, 0 ) )
   { print_int__io( x ); x = ADD( x, 1 ); goto loop; }
   else BNIL;
}
```

This clever mapping of references is highly optimizing for imperative ML programs : the **sort** program of section 6.7.2 runs 50% faster.

The `try/raise` construct As for the Scheme `call/cc` function, these constructs are implemented as library functions.

6.7 Benchmarks

This section presents the obligatory benchmark figures obtained by our compiler, compared with other ML and Lisp compilers. Time figures present the minimum of three consecutive runs ; times are expressed in seconds and represent user+system times. For compilers which produce C code (Bigloo, Camlot [54], scheme-to-c [22], sml2c [239]), C files are compiled with `gcc` using the -O2 option.

Bigloo is available on many Unix platform (Sparc, HP-PA, Mips, Alpha, Intel, Next, RS6000, MC68k, ...). We measure the execution times on a Sun 4 (Sparc 2 architecture, running SunOs 4.1.2, 64 Mo of memory) : we report the best user+system time of 3 consecutive runs.

For all programs, compilers were used with the maximum optimization levels and suppression of range checking when available.

6.7.1 Lisp benchmarking

We compare three Scheme compilers (Bigloo 1.7, S2c 15mar93, and Orbit t 3.1), the LeLisp [37] 15.24 compiler (complice), and the Cmu-cl Common lisp compiler Python 1.0 (17e)) [142].

	Compiler				
Benchmark	**Bigloo**	**S2c**	**Orbit**	**Complice**	**Cmucl**
Dderiv	*0.57*	0.80		1.23	1.80
Deriv	*0.44*	0.57	0.82	1.00	0.59
Destru	*0.15*	0.27	0.30	0.38	0.32
Div2-iter	*0.19*	0.27	0.30	0.50	0.48
Div2-rec	0.97	1.10	0.56	*0.48*	0.66
Puzzle	*0.54*	0.55	0.59	1.46	9.56
Tak	*0.02*	0.03	0.06	0.08	0.05
Fread	*0.02*	0.96	0.10	0.04	0.06
Boyer	3.22	4.70	5.64	*2.53*	4.40

FIG. 6.7 – Gabriel's benchmarks on Sparc architecture

We use the Gabriel benchmarks suite : this is not completely satisfactory for Scheme since the benchmark do not feature higher order functions. However, they allow the comparison of Scheme and Lisp compilers.

Bigloo regularly obtains the best results on these benchmarks (figure 6.7). **Div2-rec** on the Sparc architecture is a noticeable counterexample : this is due to Sparc's register windows. Compilers having C as target language use this feature of the Sparc processor and consistently obtain poor results. Register windows' usage is expensive in case of deep recursion (as for **Div2-rec**).

6.7.2 ML benchmarking

To test ML compilers, the situation is not so easy : there is no well established ML benchmark suite. Moreover our ML compilers do not share the same input syntax (Sml or Caml), nor the same libraries. In consequence there is a non trivial rewriting to perform when porting a program from a compiler to another. This presents the use of very large benchmarks such as compiler or theorem provers (for instance, Coq only runs with Bigloo and Camlc). We thus choose small but accurate benchmarks : our programs are specially designed to test specific features of compiler (see for instance **taku** and **takc** to test the compilation of n-ary functions).

We used various programs for a total amount of 2500 lines of code. We test loops (**sort**, **soli**), array manipulations (**sort**), references (**sort**, **soli**), lists (**queens**, **life**), strings (**life**), function calls (**fib**, **takc**, **taku**, **ffib**) and exceptions (**boyer**, **kb**). Benchmark programs manipulating arrays are safe : all bound tests are explicit in the source code. So, we can safely use the compiler option that

	Compiler				
Benchmark	**Bigloo**	**Camlc**	**Camlot**	**Sml/NJ**	**Sml2c**
sort	*9.0* s	260.0 s	15.8 s	27.9 s	47.7 s
life	3.2 s	44.4 s	4.5 s	*2.8* s	5.4 s
takc	*1.4* s	30.8 s	*1.4* s	11.6 s	33.4 s
taku	7.1 s	36.2 s	6.7 s	*4.2* s	8.6 s
boyer	*4.9* s	12.6 s	8.9 s	12.0 s	8.6 s
soli	*1.2* s	28.6 s	1.3 s	4.4 s	6.8 s
kb	27.6 s	73.6 s	80.6 s	*22.1* s	37.9 s
queens	*13.3* s	95.0 s	14.6 s	31.6 s	45.3 s
ffib	*2.2* s	39.2 s	*2.2* s	5.2 s	9.4 s
fft	*22.8* s	163.0 s	28.4 s	226.0 s	_

FIG. 6.8 – ML benchmarking on Sparc architecture

omit bound tests when this option is available. The programs **takc** and **taku** are two equivalent versions of the Takeuchi function, that only defer by the encoding of n-ary functions : **takc** encodes them with currification and **taku** with tuples defined as :

```
let rec tak (x, y, z) =
  if x > y then tak(tak (x-1, y, z), tak (y-1, z, x), tak (z-1, x, y))
         else z;;
```

Benchmarks concern the following ML compilers : Bigloo 1.7, Camlot (0.64), Camlc (0.6), sml/nj (1.03f) and sml2c (based on sml/nj 0.75).

Benchmark results show (figure 6.8) that Bigloo is very good at compiling function calls, in particular curried functions (see **takc** and **ffib**). On the other hand, the optimization of n-ary *uncurried* functions is badly missing (compare **takc** and **taku**).

Bigloo has good performances for **sort** due to its natural mapping of references. Bigloo compiles loops efficiently, either imperative loops (**sort**, **soli**) or functional ones (**queens**, **life**). Exceptions are efficiently implemented by Bigloo (as well as Camlot) since the **boyer** benchmark feature good performances in spite of its intensive use of exceptions.

6.8 Future work

The time figures obtained for benchmarks demonstrate that a clever compilation of functions and function calls is crucial. For instance the `taku` and `takc` lines in figures 6.8 reveal the internal strategy used by the compilers to optimize n-ary functions : Sml/NJ and sml2c perform better on **taku** than **takc**, which reveals the optimization of n-ary functions encoded as unary functions with tuple arguments ; on the other hand Camlc, Bigloo, and Camlot optimize n-ary functions encoded as curried unary functions, hence the better results for **takc** compared to **taku**. In any case, it appears that minimization of heap allocation for control flow is a corner stone of efficient functional language compilation. More generally, we claim that an optimizing compiler must work hard to minimize *any* heap allocation, either for control or for data. Bigloo already has an optimizing scheme to turn data heap allocations to stack allocation, which is very promising, since for some programs it improves runtime figures by up to a factor of 3. This allocation optimization scheme has to be studied and generalized, in particular to use “flat” allocations in the spirit of X. Leroy’s “wrap/unwrap” [134].

Conclusion

Benchmark figures for Scheme and ML show that the sharing of the Λ^n intermediate language leads to an easy and efficient compilation scheme for different languages. The generation of “Handwritten-like” C code ensures good performances, whatever the machine architecture could be. The Bigloo compiler demonstrates that the key stone to get high performances is to avoid heap allocations, in particular for control and for functions representation. Multiple high level analyses and optimizations combined with the low level optimizations of the C compiler make Bigloo one of the best existing compilers for strict functional programming languages.

Chapitre 7

Storage Use Analysis and its Applications

Cet article a été publié dans [211]. Il a été écrit en collaboration avec Marc Feeley.

Analyse des structures de données et ses applications

Cet article présente une nouvelle analyse de programmes nommée *Storage Use Analysis*. Elle consiste en l'étude des utilisations des structures dans les programmes. L'objectif étant la réduction des allocations mémoire. Cette analyse peut être appliquée à des langages typées statiquement tel que ML ou à des langages typés dynamiquement tel que Scheme. Elle autorise les effets de bords, les fonctions d'ordre supérieur, la compilation séparée et ne nécessite pas de préalable transformation CPS. Nous montrons son application à deux importantes optimisations : l'allocation en pile et la représentation plate (*unboxing*). La première optimisation déplace des allocations du tas vers la pile pour les données utilisateur et les données système (par exemple, les listes, vecteurs, procédures). La seconde optimisation permet d'utiliser des références directes aux objets plutôt que d'utiliser une indirection. Cette analyse et les optimisations qui en découlent ont été implantées dans le système Bigloo, un compilateur mixte Scheme/ML. Des mesures expérimentales démontrent que pour de nombreux programmes allouant beaucoup de données le gain en temps est important. En particulier, pour les applications numériques un facteur 20 a même été atteint parce que grâce aux optimisations présentées les nombres flottants ne sont plus alloués dans le tas.

Storage Use Analysis and its Applications

In this paper we present a new program analysis method which we call *Storage Use Analysis*. This analysis deduces how objects are used by the program and allows the optimization of their allocation. This analysis can be applied to both statically typed languages (e.g. ML) and latently typed languages (e.g. Scheme). It handles side-effects, higher order functions, separate compilation and does not require CPS transformation. We show the application of our analysis to two important optimizations : stack allocation and unboxing. The first optimization replaces some heap allocations by stack allocations for user and system data storage (e.g. lists, vectors, procedures). The second optimization avoids boxing some objects. This analysis and associated optimizations have been implemented in the Bigloo Scheme/ML compiler. Experimental results show that for many allocation intensive programs we get a significant speedup. In particular, numerically intensive programs are almost 20 times faster because floating point numbers are unboxed and no longer heap allocated.

7.1 Introduction

Modern strict functional languages such as Scheme and ML are still often much less efficient than traditional imperative languages such as Fortran and C. Few compilers for functional languages are able to produce executable programs whose efficiency is close to that of imperative ones [106]. To a large extent, this inefficiency is due to poor use of memory. Because read and write operations are much more expensive than arithmetic operations and control operations on modern computers, memory access is a major performance issue. Consequently, an efficient system must allocate as few objects as possible and must choose very carefully the location where the objects are allocated. Let's discuss these two points further.

- High allocation rate :
  For languages like Scheme and ML polymorphism is difficult to implement efficiently. With these languages, functions which accept several kinds of arguments are legal, such as an "identity" function which accepts characters, fixnums and flonums. This feature is hard to handle efficiently (fixnums and flonums are not generally of the same size and cannot be stored in the same kind of hardware registers). The traditional solution is to "box" values, i.e. use pointers to values in memory rather that direct values. This *uniform representation* [134] is inefficient because it requires memory allocation for all objects. In this paper, we present an algorithm which allows *mixed representation*. With this framework, values can be directly represented without requiring indirections. There are two benefits : memory allocation is less frequent and values are accessed more efficiently.
- Location of allocation :
  Some programs make heavy use of objects with dynamic extent (nested lifetime). Stack based language implementations[1] which do not exploit this characteristic will pay a higher cost for allocation than is required. Instructions are already present in the program to allocate and deallocate activation frames. Objects with dynamic extent could be allocated (and deallocated) at no cost in the frames. In order to automatically find when objects can be allocated in the stack we have designed a conservative analysis which determines if objects have dynamic extent.

The two optimizations presented in this paper (mixed representation and stack allocation) use the same analysis but in different ways. Section 3.1 first develops this analysis for a small source language and then the language is extended to obtain a language with data storage, side effects and modules. The stack allocation decision algorithm is presented in Section 7.3 and the mixed representation in Section 7.4. We have implemented these two optimizations in the Bigloo Scheme/ML compiler, and have measured the gain in performance on benchmark programs. The experimental results are presented in Section 7.5.

7.2 Storage Use Analysis (SUA)

In this section, we will present Storage Use Analysis (SUA) by first describing the analysis for a simple first-order language with only fixnum and flonum values and then we will extend it by adding several data types (higher-order functions, lists and vectors).

7.2.1 The input language Λ

The input language for the first version of our analysis is a simple language resembling Lisp (functions are second class citizens and closures do not exist), with only immediate values (fixnum and flonum), and without any data storage. Λ's grammar is shown below :

Syntactic categories

[1] Some systems like Sml/NJ [8] allocate activation frames in the heap.

v	$\in$	VarId	(Variables identifier)
f	$\in$	FunId	(Functions identifier)
Λ	$\in$	Exp	(Expressions)
k	$\in$	Cnst	(Constant values)
Π	$\in$	Prgm	(Program)
Γ	$\in$	Def	(Definition)

Concrete syntax

$$
\begin{array}{lll}
\Pi & ::= & \Gamma \ldots \Gamma \\
\Gamma & ::= & (\texttt{define}\ (f\ v\ \ldots\ v)\ \Lambda) \\
\Lambda & ::= & k \\
 & | & v \\
 & | & (\texttt{labels}\ ((f\ (v\ \ldots\ v)\ \Lambda)\ \ldots)\ \Lambda) \\
 & | & (\texttt{if}\ \Lambda\ \Lambda\ \Lambda\ ) \\
 & | & (\texttt{set!}\ v\ \Lambda) \\
 & | & (f\ \Lambda\ \ldots\ \Lambda) \\
 & | & (\texttt{+}\ \Lambda\ \Lambda)
\end{array}
$$

Note that since functions are first-order Λ is *not* a functional language. A program is composed of several global function definitions ; local functions are introduced by the `labels` special form. The language includes side effects on variables (the `set!` form).

7.2.2 The first-order SUA

For the sake of simplicity, we will consider the last function definition as the entry point of the program (equivalent to the C `main` function). So running a Λ program means calling the last function defined, with no arguments.

The SUA algorithm shown in algorithm 7.1 computes type information about variables and function results (which will both be called variables). The result of the analysis is an "approximation set" for each variable, which indicates the type of values that can be bound to the variable. Since the only data types are fixnum and flonum, an approximation set is a subset of the set $\mathcal{A}^0 = \{\texttt{fixnum}, \texttt{flonum}\}$. Note that because the analysis is conservative an approximation set is a superset of the true set of types that can be bound to the variable.

Note that the SUA algorithm requires α-converted programs. It is written in an intuitive pseudo language which uses the following notation :

$\mathcal{T}(k)$	The type of a constant (`fixnum` or `flonum`).
$\Pi\downarrow_{main}$	The program entry point.
$\mathcal{A}_{\text{var}}(v)$	The approximation set of variable v.
$f\downarrow_{body}$	The body of function f.
$f\downarrow_{arg_i}$	The i^{th} formal parameter of function f.
$f\downarrow_{res}$	The artificial variable representing the result of function f.

The algorithm performs a fix point iteration. Each iteration is a depth first traversal of the entire call graph initiated by the program's entry point. The fix point iteration stops when an iteration does not add any new information. This process is guaranteed to stop because there is a finite number of variables, a finite number of possible approximation sets, and no element is ever removed from a variable's approximation set. Let's study SUA's behavior with the program :

```
(define (id x) x)
(define (plus a b) (+ a b))
(define (foo) (plus (id 4) (id 5.0)))
```

The analysis collects approximation sets for `x`, `a` and `b` and the result of `id`, `plus` and `foo`. The traversal of the call graph starts with the body of `foo` which leads to a call of the function `id` with the value `4`. We are collecting type information so this call to `id` assigns the approximation set

$\mathcal{S}ua^0$(Π)=
repeat
$\mathcal{S}ua^0_{app}$($\Pi\downarrow_{main}$)
until no approximation set changed in this iteration

$\mathcal{S}ua^0_{app}$(f, a_1, …, a_n)=
$\forall i \in [1 \,..\, n]$
let $x = \mathcal{S}ua^0_{ast}$(a_i)
$\mathcal{A}_{\text{var}}$($f\downarrow_{arg_i}$) $\leftarrow$ $\mathcal{A}_{\text{var}}$($f\downarrow_{arg_i}$)$\bigcup x$,
if f not yet processed in this iteration
then let $x = \mathcal{S}ua^0_{ast}$($f\downarrow_{body}$)
$\mathcal{A}_{\text{var}}$($f\downarrow_{res}$) $\leftarrow$ $\mathcal{A}_{\text{var}}$($f\downarrow_{res}$)$\bigcup x$,
$\mathcal{A}_{\text{var}}$($f\downarrow_{res}$)

$\mathcal{S}ua^0_{ast}$($atree$)=
case $atree$
$[\![$ k $]\!]$:
$\{\mathcal{T}$(k)$\}$
$[\![$ var $]\!]$:
$\mathcal{A}_{\text{var}}$(var)
$[\![$ (`if` $atree$ $atree_t$ $atree_f$) $]\!]$:
$\mathcal{S}ua^0_{ast}$($atree$),
$\mathcal{S}ua^0_{ast}$($atree_t$)$\bigcup \mathcal{S}ua^0_{ast}$($atree_f$)
$[\![$ (`set!` var val) $]\!]$:
let $x = \mathcal{S}ua^0_{ast}$(val)
$\mathcal{A}_{\text{var}}$(var) $\leftarrow$ $\mathcal{A}_{\text{var}}$(var)$\bigcup x$,
$\emptyset$
$[\![$ (`labels` ((f_1 (v_1 … v_n) $atree_1$) …) $atree$) $]\!]$:
$\mathcal{S}ua^0_{ast}$($atree$)
$[\![$ (`+` a_1 a_2) $]\!]$:
{`fixnum`, `flonum`}
$[\![$ (f a_1 … a_n) $]\!]$:
$\mathcal{S}ua^0_{app}$(f, a_1, …, a_n)
end

FIG. 7.1 – A first-order analysis

{`fixnum`} to `x`. Since `id` returns `x`, the approximation set {`fixnum`} is also assigned to the result of `id`. After processing this first call to `id`, the analysis examines the second call `(id 5.0)`. This time, the approximation set for `id`'s argument is {`flonum`} so the analysis assigns the approximation set {`fixnum`,`flonum`} to `x`. In a given iteration, the depth first traversal of the call graph will not visit a function's body more than once, so `id`'s body is not processed again and the approximation set of `id`'s result is not changed (this is done in the next iteration). After the second call to `id`, the call to `plus` is processed. Since at this point the approximation set of `id`'s result is {`fixnum`}, the analysis assigns the approximation set {`fixnum`} to `a` and `b`.

During the second iteration, when the analysis processes the body of `id`, `flonum` is added to the approximation set of `id`'s result. This approximation set is propagated to `a` and `b` and the fix point is reached in 3 iterations. SUA concludes that all variables and function return values of this program can be a `fixnum` or a `flonum`.

Implementation note : *The set of variables and function results is finite and known at compile time. This property is important because it allows efficient implementation of $\mathcal{A}_{var}$ table using efficient set representations (e.g. using bit-vectors).*

$$
\begin{array}{l}
\mathcal{S}ua^1(\ \Pi\ )= \\
\quad \forall f {\in} \Pi{\downarrow_{export}} \\
\qquad \mathcal{S}ua^1_{export}(\ f\ ) \\
\\
\mathcal{S}ua^1_{export}(\ f\ )= \\
\quad \textbf{let}\ n{=}f\text{'s arity} \\
\qquad \forall i{\in}[1\ ..\ n] \\
\qquad\quad \mathcal{A}_{\text{var}}(\ f{\downarrow_{arg_i}}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f{\downarrow_{arg_i}}\ ) \bigcup \{\top\}, \\
\qquad \textbf{if}\ f\ \text{not yet processed in this iteration} \\
\qquad\quad \textbf{then let}\ x{=}\mathcal{S}ua^1_{ast}(\ f{\downarrow_{body}}\ ) \\
\qquad\qquad\qquad \mathcal{A}_{\text{var}}(\ f{\downarrow_{res}}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f{\downarrow_{res}}\ ) \bigcup x, \\
\qquad \mathcal{A}_{\text{spread-}\top}(\ \mathcal{A}_{\text{var}}(\ f{\downarrow_{res}}\ )\ ) \\
\\
\mathcal{S}ua^1_{app}(\ f,\ a_1,\ \ldots,\ a_n\ )= \\
\quad \textbf{if}\ f\ \text{is imported} \\
\qquad \textbf{then}\ \{\top\} \\
\qquad \textbf{else}\ \forall i{\in}[1\ ..\ n] \\
\qquad\qquad\quad \textbf{let}\ x{=}\mathcal{S}ua^1_{ast}(\ a_i\ ) \\
\qquad\qquad\qquad \mathcal{A}_{\text{var}}(\ f{\downarrow_{arg_i}}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f{\downarrow_{arg_i}}\ ) \bigcup x, \\
\qquad\qquad \textbf{if}\ f\ \text{not yet processed in this iteration} \\
\qquad\qquad\quad \textbf{then let}\ x{=}\mathcal{S}ua^1_{ast}(\ f{\downarrow_{body}}\ ) \\
\qquad\qquad\qquad\quad \mathcal{A}_{\text{var}}(\ f{\downarrow_{res}}\ ) \leftarrow \mathcal{A}_{\text{var}}(\ f{\downarrow_{res}}\ ) \bigcup x, \\
\qquad\qquad \mathcal{A}_{\text{var}}(\ f{\downarrow_{res}}\ ) \\
\\
\mathcal{A}_{\text{spread-}\top}(\ a\ )=\ a
\end{array}
$$

FIG. 7.2 – A first-order analysis with modules

7.2.3 The first-order SUA with modules

We now extend Λ to support modules. Rather than add new constructions to the language we will assume that all global functions are *exported* (i.e. that they are visible in other modules).

From the compiler's point of view, the fact that a function is exported means that its actual parameters may be unknown because it can be invoked outside the current module. To handle this we have to introduce a representative for unknown values. As customary [52] this is noted "top" ($\top$). Approximation sets are now subsets of $\mathcal{A}^1 = \{\top, \texttt{fixnum}, \texttt{flonum}\}$. When a variable's approximation set contains $\top$, it means that any value may be bound to the variable. Algorithm 7.2 contains the updated SUA algorithm[2]. It makes use of the new notation :

$\Pi\downarrow_{export}$ The set of Π's exported functions.

In this new version of the analysis, the traversal of the call graph is initiated by all exported functions. The formal parameters of all exported functions are initially approximated by $\{\top\}$. The function $\mathcal{A}_{\text{spread-}\top}$ will be useful later on to spread $\top$ into data storage approximations.

7.2.4 The higher-order SUA with modules

We now extend the analysis to accept as input a higher-order functional language. Three new constructions are added to Λ : `make-closure` (to create closures), `closure-ref` (to access a closure's free variables) and `closure-call` (to invoke a closure). Here are the modifications to Λ's grammar :

$$
\begin{array}{lll}
\Lambda & ::= & \ldots \\
& | & (\texttt{make-closure}\ f\ v\ \ldots v) \\
& | & (\texttt{closure-ref}\ v\ k) \\
& | & (\texttt{closure-call}\ \Lambda\ v\ \ldots v)
\end{array}
$$

[2]function $\mathcal{S}ua^1_{ast}$ is not defined here because it has the same definition as $\mathcal{S}ua^0_{ast}$ (with references to $\mathcal{S}ua^0_{ast}$ replaced with $\mathcal{S}ua^1_{ast}$). This kind of misuse will be used in the remainder of the paper to avoid redundancy.

$\mathcal{S}ua^2_{ast}$($atree$)=
  case $atree$
      ⋮
    ⟦ (closure-call e a_1 … a_n) ⟧ :

$$\bigcup_{f \in \mathcal{S}ua^2_{ast}(\ e\ )} \mathcal{S}ua^2_{clocall}(\ f,\ a_1,\ \ldots,\ a_n\ )$$

    ⟦ (make-closure$_i$ f v_1 …) ⟧ :
      let $a_1 = \mathcal{S}ua^2_{ast}(\ v_1\ ), \ldots$
        $\mathcal{A}_{\mathrm{clo}}(\ i\ ) \leftarrow \widehat{\texttt{make-closure}}(\ f,\ a_1,\ \ldots\ )$,
        $\{\texttt{clo}_i\}$
    ⟦ (closure-ref f k) ⟧ :

$$\bigcup_{\texttt{clo}_i \in \mathcal{S}ua^2_{ast}(\ f\ )} \widehat{\texttt{closure-ref}}(\ \mathcal{A}_{\mathrm{clo}}(\ i\ ),\ k\ )$$

  end

$\mathcal{S}ua^2_{clocall}$(e, a_1, …)=
  case e
    $\texttt{clo}_i$:
      $\mathcal{S}ua^2_{app}(\ \widehat{\texttt{closure-function}}(\ \mathcal{A}_{\mathrm{clo}}(\ i\ )\ ),\ a_1,\ \ldots\ )$
    else :
      $\mathcal{S}ua^2_{failure}()$
  end

$\mathcal{S}ua^2_{app}$(f, a_1, …, a_n)=
  if $n = f$'s arity
    then $\mathcal{S}ua^1_{app}(\ f,\ a_1,\ \ldots,\ a_n\ )$
    else $\mathcal{S}ua^2_{failure}()$

$\mathcal{S}ua^2_{failure}() = \emptyset$

$\mathcal{A}_{\text{spread-}\top}$(a)=
  $\forall \texttt{clo}_i \in a$
    $\mathcal{S}ua^2_{export}(\ \widehat{\texttt{closure-function}}(\ \mathcal{A}_{\mathrm{clo}}(\ i\ )\ )\ )$,
  a

FIG. 7.3 – A higher-order analysis with modules

The usefulness of these constructs rests in the ability to easily translate Scheme programs into Λ. The translation of the following program :

```
(define (curry-plus x) (lambda (y) (+ x y)))
(define (add a b) ((curry-plus a) b))
(define (main) (add 3.4 5.6))
```

is :

```
(define (curry-plus x) (make-closure1 f x))
(define (f p y) (+ (closure-ref p 0) y))
(define (add a b) (closure-call (curry-plus a) b))
(define (main) (add 3.4 5.6))
```

The translation required to map Scheme, ML or other higher-order functional languages to Λ is the so-called λ-lifting transformation [118].

Closures introduced by `make-closure` are for now the only data structures of our language. SUA is modified to compute information about types *and* data storage by adding "closure approximations". A closure approximation is a tuple containing a closure function and a closure environment (a list of approximation sets, one for each free variable). Closure approximations are created by the function $\widehat{\texttt{make-closure}}$. The function associated with closure approximation a is obtained with $\widehat{\texttt{closure-function}}(a)$ and $\widehat{\texttt{closure-ref}}(a, i)$ returns the approximation set associated with the i^{th} free variable of closure approximation a.

Closure approximations are stored in a closure approximation table named $\mathcal{A}_{\text{clo}}$. There is a one-to-one correspondence between entries in this table and `make-closures` in the program. To add a closure approximation to an approximation set, $\texttt{clo}_i$ is added to the set, where i is the entry's index in $\mathcal{A}_{\text{clo}}$. In this version of our algorithm, approximation sets are subsets of $\mathcal{A}^2 = \{\top, \texttt{fixnum}, \texttt{flonum}, \texttt{clo}_1, \ldots, \texttt{clo}_k\}$,
where k is the number of `make-closures` in the program.

A new problem arises with functional values. In latently typed languages `closure-call` can lead to two possible errors : the object given to `closure-call` is not a closure or the number of arguments is incompatible with the function called. We have to deal with these possible errors in the SUA. For the sake of simplicity we suppose that `closure-ref` is always correct. This is reasonable since these constructions are inserted by the program which is in charge of the λ-lifting and not by the user. The treatment of errors is straightforward : errors just produce empty approximation sets. This means that an error leaves all the approximation sets as they are. This is sound because at run time, if an error occurs, the program is interrupted, so errors *do not* return values.

Our handling of closures has been guided by their special nature : they are immutable data (since we are using flat closures, mutable free variables are stored in cells) and they are always accessed via the `closure-ref` procedure which requires a constant index as second argument. It is thus possible to distinguish the free variables.

Algorithm 7.3 presents the extensions to SUA needed to accept the higher-order version of Λ (we assume function $\mathcal{S}ua^1_{app}$ in the algorithm uses the new version of the graph traversal function, i.e. $\mathcal{S}ua^2_{ast}$). The $\mathcal{A}_{\text{spread-}\top}$ function also requires a slight modification : if a closure can be returned by an exported function, this closure is also exported as it can be invoked with unknown actual parameters. The new $\mathcal{A}_{\text{spread-}\top}$ handles this.

Let's study SUA on the example of the curried addition `curry-plus`. Assuming no exported functions, the iteration process starts by traversing `main`. The call to `add` assigns the approximation set $\{\texttt{flonum}\}$ to `a` and `b`. The call `curry-plus` in turn assigns the approximation set $\{\texttt{flonum}\}$ to `x`, and the function's result is assigned the approximation set $\{\texttt{clo}_1\}$ after storing a closure approximation over one flonum (i.e. $\widehat{\texttt{make-closure}}(\texttt{f}, \{\texttt{flonum}\})$) in $\mathcal{A}_{\text{clo}}(1)$. The first argument of the `closure-call` has the approximation set $\{\texttt{clo}_1\}$, so SUA continues by analyzing `f`'s body with the approximation set $\{\texttt{clo}_1\}$ for `p`. The `closure-ref` thus returns the approximation set $\{\texttt{flonum}\}$.

7.2.5 The higher-order SUA with modules and lists

SUA can be easily extended to accept other data types. In this section, we present how lists are added to the analysis. Lists (in Lisp and Scheme) differ from closures because they are mutable data. Lists are built out of pairs. The two fields of a pair can be distinguished in our approximation scheme (just like all the free variables of a closure are distinguished).

$$
\begin{array}{lll}
\Lambda & ::= & \ldots \\
& | & \texttt{(cons } \Lambda\ \Lambda\texttt{)} \\
& | & \texttt{(car } \Lambda\texttt{)} \\
& | & \texttt{(cdr } \Lambda\texttt{)} \\
& | & \texttt{(set-car! } \Lambda\ \Lambda\texttt{)} \\
& | & \texttt{(set-cdr! } \Lambda\ \Lambda\texttt{)}
\end{array}
$$

The handling of pairs in the SUA is very similar to closures. The SUA extended for pairs is shown in Algorithm 7.4 (the cases for `cdr` and `set-cdr!` are left out because of their obvious symmetry with `car` and `set-car!`). $\mathcal{A}_{\text{cons}}$ is a table similar to $\mathcal{A}_{\text{clo}}$ but for pair approximations. A pair approximation is a tuple of two approximation sets (one for each field of the

$\mathcal{S}ua^3_{ast}$(*atree*)=
  case *atree*
    $\vdots$
   $[\![$ (cons$_i$ *a d*) $]\!]$:
    $\mathcal{A}_{\text{cons}}(i) \leftarrow \widehat{\texttt{cons}}(\mathcal{S}ua^3_{ast}(a), \mathcal{S}ua^3_{ast}(d)),$
    $\{\texttt{cons}_i\}$
   $[\![$ (car *p*) $]\!]$:
    $\bigcup_{\texttt{cons}_i \in \mathcal{S}ua^3_{ast}(p)} \widehat{\texttt{car}}(\mathcal{A}_{\text{cons}}(i))$
   $[\![$ (set-car! *p x*) $]\!]$:
    let $x' = \mathcal{S}ua^3_{ast}(x)$
     $\forall \texttt{cons}_i \in \mathcal{S}ua^3_{ast}(p)$
      $\mathcal{A}_{\text{cons}}(i) \leftarrow \widehat{\texttt{cons}}(\widehat{\texttt{car}}(\mathcal{A}_{\text{cons}}(i)) \bigcup x',$
                $\widehat{\texttt{cdr}}(\mathcal{A}_{\text{cons}}(i))),$
     $\emptyset$
  end

$\mathcal{A}_{\text{spread-}\top}(a)$=
  $\forall \texttt{cons}_i \in a$
   if cons$_i$ not already spread in this iteration
    then let $a' = \widehat{\texttt{car}}(\mathcal{A}_{\text{cons}}(i)),$
      let $d' = \widehat{\texttt{cdr}}(\mathcal{A}_{\text{cons}}(i))$
       $\mathcal{A}_{\text{spread-}\top}(a'),$
       $\mathcal{A}_{\text{spread-}\top}(d'),$
       $\mathcal{A}_{\text{cons}}(i) \leftarrow \widehat{\texttt{cons}}(a' \bigcup \{\top\}, d' \bigcup \{\top\}),$
  ...,
  a

FIG. 7.4 – A higher-order analysis with modules and lists

pair) and is created with the function $\widehat{\texttt{cons}}$. $\widehat{\texttt{car}}(a)$ and $\widehat{\texttt{cdr}}(a)$ respectively return the approximation set associated with the `car` and `cdr` field of pair approximation a to which is added the special approximation *obj* (as explained in section 7.4.5, *obj* denotes the generic Scheme object type and is needed to prevent unboxed pairs). Approximation sets are now subsets of $\mathcal{A}^3 = \{\top, \texttt{fixnum}, \texttt{flonum}, \texttt{clo}_1, \ldots, \texttt{clo}_k, \texttt{cons}_1, \ldots, \texttt{cons}_c, obj\}$,
where c is the number of calls to `cons` in the program.

The main change is in the $\mathcal{A}_{\text{spread-}\top}$ function. Even when closures are exported, the values they hold cannot be changed because closures are immutable data storage. Because pairs are mutable they may be altered when exported (i.e. the fields of the pair can be changed using the `set-car!` and `set-cdr!` functions). The new $\mathcal{A}_{\text{spread-}\top}$ function handles this. When pairs are exported, $\top$ is added to the approximation set of each field. Note that a pair is spread at most once per iteration by $\mathcal{A}_{\text{spread-}\top}$. This is necessary to handle cyclic approximations.

Let's study SUA on the following Scheme program (the program is presented in Scheme rather than in Λ so that it is easier to read) :

```
(define lst (let ((p1 (cons₁ 1 0)))
              (let ((p2 (cons₂ 2 p1)))
                p2)))
(define (length l)
  (if (pair? l) (+ 1 (length (cdr l))) 0))
(length lst)
```

Types used by this program are : `fixnum`, pairs (i.e. $\texttt{cons}_1$ and $\texttt{cons}_2$) and the special *obj* type (we omit `boolean`, needed for the `pair?` predicate, because it does not appear in a variable's approximation set). Here is the state of the tables at the end of the analysis :

$\mathcal{A}_{\text{var}}(\texttt{p1}) = \{\texttt{cons}_1\}$
$\mathcal{A}_{\text{var}}(\texttt{p2}) = \{\texttt{cons}_2\}$
$\mathcal{A}_{\text{var}}(\texttt{lst}) = \{\texttt{cons}_2\}$
$\mathcal{A}_{\text{var}}(\texttt{l}) = \{\texttt{fixnum}, \texttt{cons}_1, \texttt{cons}_2, obj\}$
$\mathcal{A}_{\text{cons}}(1) = \widehat{\texttt{cons}}(\{\texttt{fixnum}\}, \{\texttt{fixnum}\})$
$\mathcal{A}_{\text{cons}}(2) = \widehat{\texttt{cons}}(\{\texttt{fixnum}\}, \{\texttt{cons}_1\})$

The invocation of `length` at line *6* has added $\texttt{cons}_2$ to `l`'s approximation set. Because of the call to `cdr`, the recursive call at line *5* has added $\widehat{\texttt{cdr}}(\mathcal{A}_{\text{cons}}(2))$, that is $\{\texttt{cons}_1, obj\}$, in one iteration and $\{\texttt{fixnum}, obj\}$ in the next iteration.

7.2.6 The higher-order SUA with modules, lists and vectors

We conclude this section by adding vectors.

$$
\begin{array}{lll}
\Lambda & ::= & \ldots \\
 & | & (\texttt{make-vect}\ \Lambda\ \Lambda) \\
 & | & (\texttt{vref}\ \Lambda\ \Lambda) \\
 & | & (\texttt{vset!}\ \Lambda\ \Lambda\ \Lambda)
\end{array}
$$

$\mathcal{S}ua^4_{ast}$(*atree*)=
  case *atree*
      ⋮
    ⟦ ($\texttt{make-vect}_i$ *len filler*) ⟧ :
      $\mathcal{S}ua^4_{ast}$(*len*),
      $\mathcal{A}_{\text{vect}}$(i) ← $\widehat{\texttt{make-vect}}$($\mathcal{S}ua^4_{ast}$(*filler*)),
      $\{\texttt{vect}_i\}$
    ⟦ (vref v o) ⟧ :
      $\mathcal{S}ua^4_{ast}$(o),
      $\bigcup_{\texttt{vect}_i \in \mathcal{S}ua^4_{ast}(v)} \widehat{\texttt{vref}}(\mathcal{A}_{\text{vect}}(i))$
    ⟦ (vset! v o x) ⟧ :
      $\mathcal{S}ua^4_{ast}$(o),
      let x'=$\mathcal{S}ua^4_{ast}$(x)
        $\forall \texttt{vect}_i \in \mathcal{S}ua^4_{ast}(v)$
          $\mathcal{A}_{\text{vect}}$(i) ←
              $\widehat{\texttt{make-vect}}$($\widehat{\texttt{vref}}$($\mathcal{A}_{\text{vect}}$(i))$\bigcup x'$),
        $\emptyset$
  end

$\mathcal{A}_{\text{spread-}\top}$(a)=
  $\forall \texttt{vect}_i \in a$
    if $\texttt{vect}_i$ not already spread in this iteration
      then let r'=$\widehat{\texttt{vref}}$($\mathcal{A}_{\text{vect}}$(i))
          $\mathcal{A}_{\text{spread-}\top}$(r'),
          $\mathcal{A}_{\text{vect}}$(i) ← $\widehat{\texttt{make-vect}}$($r' \bigcup \{\top\}$),
  ...,
  a

FIG. 7.5 – A higher-order analysis with modules, lists and vectors

Vectors differ from closures and lists in that they are mutable and because it is not possible, *a priori*, to know, at compile time, which part of a vector is addressed when using vector accessors. SUA computes information about types and data storage but it does not discover the exact value of a fixnum. So for vectors the SUA merges all possible values contained in a vector into a single approximation set (e.g. if a vector is composed of a character and a fixnum, SUA will indicate that each entry is a "character or a fixnum"). Algorithm 7.5 presents the modification to our previous analysis to support vectors.

$\mathcal{A}_{\text{vect}}$ is a table similar to $\mathcal{A}_{\text{cons}}$ but for vector approximations. Vector approximations are created by the function $\widehat{\texttt{make-vect}}$ and $\widehat{\texttt{vref}}(a)$ return the approximation set associated with the vector approximation a. Approximation sets are now subsets of $\mathcal{A}^4 = \{\top, \texttt{fixnum}, \texttt{flonum}, \texttt{clo}_1, \ldots, \texttt{clo}_k, \texttt{cons}_1, \ldots, \texttt{cons}_c, \texttt{vect}_1, \ldots, \texttt{vect}_v, obj\}$, where v is the number of calls to `make-vector` in the program.

Just like for pairs, vector exportations have to be handled carefully. This kind of object is mutable so when exported vectors can hold any value.

7.2.7 Related work

The SUA is an extension of Shivers' 0cfa (0th order control flow analysis) [219, 218]. We have generalized his analysis to different data storage. Shivers' analysis only handles closures, our analysis also handles lists and vectors.

In a previous paper [206] we have presented an algorithm which is close to the present $\mathcal{S}ua^1$. The goal of that work was to study the impact of control flow analysis on function compilation. The analyses presented here are more general because closures are only considered as *one* special data storage. Efficient closure compilation is not the focus here. We study the problems of unboxed representation and stack allocation.

Ayers has studied similar improvements to Shivers' 0cfa. In his PhD thesis, he presents extensions for lists, vectors, etc. Our work has been realized concurrently with his [201, 15]. The large difference between our analyses comes from the formalism. Ayers uses Galois connection while we chose a more algorithmic approach.

Jagannathan and Wright describe in [116] a control-flow analysis and an application which removes type checks. Their analysis gives more precise type information than ours because it does not merge types for polymorphic programs. More precise type information is valuable to remove type checks but is not more valuable for a *mixed representation*. As explained in section 7.4, we use unboxed representation for monomorphic program parts which are efficiently detected by our analysis.

7.2.8 Extensions

Our input language Λ, is still much simpler than a full programming language such as Scheme or ML. Some important constructions are missing. We present here how to add them to SUA.

- Variable arity functions : this construction can be added to SUA by splitting functional application in two separate cases. Each time a function is applied (in a direct call or in a `closure-call` construction), the analysis handles the last parameter of variable arity functions specially. In Scheme, this parameter is bound to the list of the optional actual parameters. In SUA, this means that the approximation set of the last formal parameter is the approximation set of the list of the optional actual parameters.
- The Scheme special form `apply` : this construction is an alternative way to apply functions. Rather than apply a function to its n actual parameters, it is applied to a list of length n which holds the actual values. Since our analysis is able to distinguish individual elements of a list, the `apply` form can be efficiently handled : each formal parameter is assigned the corresponding approximation set from the list's approximation set.
- `call/cc` : the analysis does not treat `call/cc` specially. This library function takes closures as arguments. These closures therefore escape because `call/cc` is managed as any imported function. `call/cc`'s result is simply approximated by the set $\{\top\}$.
- Multiple values can be easily added by the addition of a treatment similar to the one for vectors.
- Scheme and ML global variables : global variables can be managed using a global environment. A subtle problem with global variables can arise when the source language allows

references to global variables before their declaration. For example, the Bigloo Scheme compiler considers this program as legal :

```
(module foo (static x))

(define (foo) (print x))
(foo)
(define x 8)
```

Before its declaration `x` holds a special value (*uninitialized*, which stands for the lack of initialization) which has to be stored in `x`'s approximation set. This implies that no optimization can be applied to `x` because it holds at least two types : the type of the *uninitialized* value and the first value used on the declaration site.
In order to give a unique type to global variables, before the SUA analysis, we perform a simple conservative analysis to determine the set of variables which are always defined (then initialized) before being referenced. This analysis is straightforward, because it consists of a simple abstract tree traversal. These variables do not hold the special *uninitialized* value in their approximation set.

7.3 Stack allocation

The storage allocation optimizations discussed here assume an area of memory managed by a garbage collector and an area of memory managed as a stack. The stack is scanned by the collector to find the root pointers. Activation records are allocated on the stack when entering a procedure and they are removed from the stack upon procedure exit. Within a procedure activation the allocation of additional storage from the stack is permitted ; this storage is freed when the procedure exits.

We present a conservative optimization based on SUA which automatically replaces heap allocations by stack allocations when it is legal to do so. This optimization is valuable if stack allocations and deallocations are fast with respect to heap allocations. For the sake of simplicity we add a `let` form to Λ :

$$\Lambda \ ::= \ \ldots \;\mid\; \texttt{(let ((}v\ \Lambda\texttt{))}\ \Lambda\texttt{)}$$

This form has no impact on the SUA algorithm (it can be seen as a macro over function application). For our stack allocation optimization, we assume that in Λ source programs all allocations (the result of `make-closure`, `cons` and `make-vect`) are bound to local variables using `let` forms. Consequently, each allocation has a unique name.

Here are three examples that present interesting situations for our optimization.

```
(define (foo)
   (let ((x (cons₁ 1 2)))
      (car (id x))))

(define (id z) z)
```

In this first example, the pair bound to `x` can be stack allocated since it is never used outside of `x`'s `let` extent.

```
1 : (define (bar)
2 :    (let ((x (cons₁ 1 2)))
3 :       (let ((y (cons₂ 3 x)))
4 :          (cdr y))))
```

In this second example, the pair allocated at line *2* is live when `bar` exits (since it is the result of `bar`) while the one at line *3* is dead outside `x`'s scope. Only the second pair can be stack allocated.

```
(define (hux)
   (let ((p0 (cons1 1 2)))
      (let ((p1 (cons2 3 4)))
         (let ((p2 (gee p0 p1)))
            p0))))

(define (gee a b) (set-cdr! a b))
```

Finally, in this example, no pairs can be stack allocated because they are all live when `hux` returns.

7.3.1 When is it legal to stack allocate ?

We present in this section the condition an allocation must satisfy to be done on the stack rather than in the heap. For now we are not concerned with preserving the tail-recursive property of the program (at the end of this section we discuss modifications of our optimization to make it suitable for languages like Scheme which have to implement tail-recursive calls without consuming stack space).

An allocation can be done on the stack if the data storage allocated is not live at the end of the procedure that allocates it. Data storage is live at the end of a function if it appears (directly or indirectly) in the result of the function that allocated it or if it appears (directly or indirectly) in a global variable. Compile time computation of the liveness property requires information about data storage which is provided by SUA. This information is : the set of allocations a variable may be bound to, the set of allocations possibly contained in an allocation, and the set of allocations possibly returned by a function.

7.3.2 Stack allocation decision algorithm

Each allocation is marked with a stamp. The "current stamp" is incremented each time a `let` form is encountered. When a function definition for f is reached, the current stamp is saved in h and then f's body is processed. Each allocation in the approximation set of f's result that is stamped with a more recent value than h escapes from f and so cannot be stack allocated. In addition all allocations which are accessible from a global variable cannot be stack allocated.

The first part of the algorithm (algorithm 7.6) dispatches between two function types : exported functions and static functions. These two kinds of function differ. For the first one, no returned (or pointed by) value can be stack allocated (since the function is exported the result value usage is not known to the compiler). For the second one, only data storage allocated by the function cannot be stack allocated.

Our stack algorithm uses a $\text{spread}_{unstackable}$ function. This function is similar to $\mathcal{A}_{\text{spread-}\top}$. It follows a data storage chain to mark as "unstackable" all allocations which are younger (marked as younger) than the value of the second argument.

The algorithm's main part is the function $\mathcal{S}tack$. It dispatches on the abstract syntax tree. Before calling $\mathcal{S}tack_{prog}$ all allocations which have been passed as argument to $\mathcal{A}_{\text{spread-}\top}$ have to be marked as *unstackable*. These allocations escape from the current module. The compiler is not able to discover the exact usage of these allocations and thus, it cannot make any assumption about their lifetime. Once the algorithm has completed, allocations which have not been marked as *unstackable* can be stack allocated (we will introduce new constraints to safely allocate data storage in stack in the next section).

Let's study the algorithm's behavior on our previous `bar` function example. The function allocates two pairs $\texttt{cons}_1$ and $\texttt{cons}_2$. SUA proves that `y` points to $\texttt{cons}_2$ (which points to $\texttt{cons}_1$) and `x` points to $\texttt{cons}_1$. The $\texttt{cons}_1$ pair is pointed to by the result of `bar`. So, the algorithm concludes that this pair cannot be stack allocated.

```
*H* : 0    (0 stands for an initial stamp value)

Stack_prog( Π )=
   ∀f∈Π
      if f∈Π↓export
         then Stack_export( f )
         else Stack_static( f )

Stack_static( f )=
   let h=*H*
      Stack( f↓body ),
      spread_unstackable( A_out( f ), h, *H* )

Stack_export( f )=
   Stack( f↓body ),
   spread_unstackable( A_out( f ), -1, -1 )

Stack( atree )=
   case atree
        ⋮
      ⟦ (closure-call e a_1 ... a_n) ⟧ :
         Stack( e ),
         ∀i∈[1 .. n]
            Stack( a_i )
      ⟦ (set! var val) ⟧ :
         Stack( val )
      ⟦ (labels ((f_1 ...) ... (f_n ...)) atree) ⟧ :
         ∀i∈[1 .. n]
            Stack_static( f_i ),
         Stack( atree )
      ⟦ (let ((var val)) atree) ⟧ :
         *H* ← *H*+1,
         Stack( val ),
         Stack( atree )
      ⟦ (f a_1 ... a_n) ⟧ :
         if f is an allocator
            then mark!( atree, *H* ),
         ∀i∈[1 .. n]
            Stack( a_i )
   end
```

FIG. 7.6 – The “stackability” algorithm

```
spread_unstackable( a, min, max )=
  if a not yet processed for the values min and max
   then case a
        pair_i :
          if mark( a ) >min and mark( a ) ≤max
            then mark-unstackable!( a ),
          ∀a'∈car^( A_pair( i ) )
            spread_unstackable( a', min, max ),
          ∀a'∈cdr^( A_pair( i ) )
            spread_unstackable( a', min, max )
        vect_i :
          ...
        clo_i :
          ...
      end
```

FIG. 7.7 – Spreading "unstackability"

7.3.3 Extension for proper tail-recursion implementations and safety considerations

Some languages like Scheme require that executions of an iterative computation take constant space. Let's consider the following two functions :

```
(define (foo1 x y)
  (if (= y 0)
      (display x)
      (foo1 (cons 1 2)
            (- y 1))))
```

```
(define (foo2 x y)
  (if (= y 0)
      (display x)
      (foo2 (cons x x)
            (- y 1))))
```

The two functions differ only in their recursive call. In `foo1`, only one allocated pair of the recursive call is live at a time ; in `foo2`, allocated pairs are linked together and they are all live at any given point. The common intuitive idea of the tail-recursive property imposes an implementation to require only one free pair to run `foo1`. Our algorithm presented in algorithm 7.6 provides rough data storage lifetime. It is not able to distinguish that pairs allocated in `foo1` cannot be stack allocated while pairs allocated in `foo2` can be.

The problem is more general than tail-recursion. As revealed by Chase in [43], there is a general safety problem for stack allocation optimizers. Sometime, allocating an object in the stack rather than in the heap extends its lifetime. For instance, in `foo1`, a garbage collector is free to reclaim previous allocated pairs but if these pairs are stack allocated they will be all freed at the same time and required space to run this program is no longer constant. Stack allocation can convert a running program into one that fails. In his paper, Chase, presents "safety conditions for stack allocation" in order to decide the replacement of heap allocations by stack allocations in presence of loops or recursions. His method and our work are complementary.

7.3.4 Related work

Kranz presents in [128] the strategy used by Orbit to realize stack allocations. His method is less precise than ours. In Orbit only closures can be stack allocated and only if they are passed as an argument or applied. These conditions are very restrictive.

In [92], B. Goldberg and G. Park present a method for optimizing the allocation of closures in memory. Their method is based on what they call an *escape analysis*, an application of abstract interpretation to higher-order functional languages. Escape analysis determines, at compile time, if any arguments to a function have a greater lifetime than the function call itself. The language studied does not contain side effects and the only data storage used are closures and lists. List management is very rough because their analysis is not able to distinguish the elements of a list. Separate compilation is not studied in that paper.

Ruggieri and Murtagh present in [184] a data storage allocation framework called *sub-heap allocation*. This framework consists of partitioning the heap into sub-heaps, one associated with each active procedure. The contents of the sub-heap associated with a procedure is exactly the objects whose lifetime are guaranteed to be contained by the lifetime of the procedure but not by any younger procedure. The paper presents an algorithm to compute lifetime analysis in order to divide the heap with an input source language which contains no higher-order functions nor side effects.

Ayers also presents sub-heap optimization in [15]. Our lifetime analysis is similar to his but we do not use it for the same goal. We decided to stack allocate rather than sub-heap allocate for two reasons :

- Safety considerations presented by Chase [43] are very difficult to satisfy with the sub-heap allocation framework because it tends to enlarge object lifetime. Optimized objects are not freed when leaving the function they have been created but when leaving the function which is the upper bound of their lifetime.
- Sub-heap allocation is difficult to implement efficiently. This framework needs allocated memory to hold several objects which share a lifetime upper bound. If all objects are freed at the same time, they are allocated at different moments. This has two negative incidences :
  - Sub-heap size is difficult to estimate. Sub-heaps will probably need linking machinery to be extended which slows down the allocation process.
  - Sub-heaps must be allocated empty (i.e. a sub-heap cannot be filled up at the moment of its creation). Included in a runtime with automatic memory management, uninitialized blocks of memory are annoying.

Tofte and Talpin present in [242] a way of implementing λ calculus based languages using regions for memory management. At runtime the store consists of a stack of regions. All values are put into regions with the intended goal to avoid garbage collection in the runtime system. The allocation of new regions and the bindings of values to regions rely on a typing system and so this technique cannot be applied to dynamically typed languages.

In [19], Banerjee and Schmidt present a static criterion to detect stackability of environments for a call-by-value λ-calculus. The presented analysis does not include higher-order nor imperative features. Thus their approach and ours are hard to compare.

Other approaches to stack allocation have been proposed by Hudak in [111].

7.4 Data representation

Approximations computed by our SUA can be used to remove some runtime type checks. A type check can be removed when SUA proves that all the values possibly contained by the argument of the test is (or is not) of the tested type. This idea has been presented by Shivers [219, chapter 9] in his *type-recovery*. The goal is to speed up program execution of latently typed languages. In the same way, Henglein in [108] and Ayers in [15, chapter 6] have presented frameworks to remove useless tagging/untagging operations. Heinglein uses type inference while Ayers uses an extended control flow analysis close to our SUA. The intended goal is more than compile-time type check reductions. Appel claims “the use of tag bits leads to inefficiency” [7], Steenkiste and Henessy evaluate, in [229], at 25% the cost of type checking and tagging operations for “classical” Lisp applications. We think this time figure is an upper bound of the real cost. Classical data flow optimization (such as *copy propagation*) removes most type checks and, for smart runtime design, tagging and untagging operations could require only a logical mask in the most frequent cases. On modern computers, applying a mask costs one cycle and these operations are much cheaper than memory fetches. They have a very small impact on global performance (for more details see [204]).

We think a much more important source of inefficiency for language like Scheme or ML come from uniform data representation. Tag handling is cheap but uniform representation is very expensive.

7.4.1 Uniform representation

Using uniform data representation, all objects have exactly the same size (usually one word, i.e. pointer size). Objects that do not fit naturally in one word, such as long floating-point numbers, have to be boxed (allocated in the heap and handled through a pointer). This scheme makes it possible to assume a default size, common to all objects, and default calling conventions, common to all functions.

Polymorphism leads to the use of the uniform representation because an object can belong to several different types at the same time and the actual type cannot be known at compile-time. Polymorphic functions (e.g. the identity function) can be applied to arguments of any type. Therefore, when compiling these functions, the compiler knows neither the size of the argument nor the correct calling convention.

7.4.2 The uniform representation is inefficient

We claim uniform representation results in a serious loss of efficiency and we present two arguments for this assertion.

Objects that do not fit in one word have to be boxed. Long floating-point numbers dramatically illustrate this. For floating-point intensive programs, boxing numbers can slow down applications by a large factor. This problem has such an important impact that many ML and Lisp implementations use *ad hoc* methods to reduce the creation of number handles (descriptions of these methods for modern implementations can be found in [142, 106]). Mainly, they consist of local optimizations to avoid boxing numbers for intermediate results.

Another negative impact of floating-point boxing is register allocation. When flonums are allocated in memory, every floating-point operation, requires a memory fetch for each operant. These operations are expensive and much less efficient than a solution where the numbers are held in registers.

Tagging optimization is not boxing optimization in the sense that it removes tagging/untagging operations but such optimized programs must still satisfy the polymorphism constraint. They are still obliged to box numbers (even if no tag is written on the handle or stored in the allocated memory).

Small objects (i.e. characters) are also inefficiently managed by uniform representations as memory is wasted.

7.4.3 Mixed representation

Mixed representation is a representation where all objects are not required to be of the same size. It mixes boxed objects and unboxed objects. Initial efforts using mixed representation are Leroy [132, 134] and Peyton-Jones and Launchbury [161], and more recently Shao and Appel [216]. Their works are complementary because Peyton-Jones and Launchbury introduce a (non-strict) language where boxing operations are explicit and introduce several source-to-source optimizations for this language while Leroy and Shao and Appel present a translation of ML to this mixed language. In this paper, we will focus on the translation of source languages (like ML or Scheme) to mixed languages, comparing our work to Leroy's.

Leroy's translation only uses type information. It mixes specialized representations when the static types are monomorphic and uniform representation when the static types are polymorphic. Coercions between the two representation styles are performed when a polymorphic object is used with a more specific type. As Leroy presents, "in the case of a polymorphic function, for instance, coercions take place just before the function call and just after the function result". This solution is very elegant because the translation's quality does not suffer from separate compilation (type informations are propagated across ML modules) but is has some disadvantages. Every time a polymorphic function is used, objects have to be boxed. Some data accessors *are* polymorphic. For instance, vector accessors are polymorphic where the same function is used to access a vector of fixnums or a vector of flonums. Vectors are a too important kind of data storage to be out of the scope of this transformation. In other words, Leroy's translation requires some *ad hoc* treatments for some special functions. The second restriction to Leroy's work is about the input languages of its translation. Programs have to be statically type checked, and as a consequence Leroy's optimization is not applicable to the Lisp family.

Untagging vs. unboxing

Optimizing tagging/untagging operations as in [108, 15] does not require the same analysis as mixed representation. Constraints about untagged representations are weaker than for unboxed representations. An object can be untagged as soon as its type is never required at runtime, without any polymorphism consideration. For instance, with untagged representation, a vector can hold in its first slot an untagged floating point number and in its second slot a tagged one. This is not possible with unboxed representation because these two kind of objects do not have the same size. Untagging optimization consists of type analysis (possibly using type system as Henglein, control flow analysis as Ayers or any other data flow analysis) while unboxing optimization requires type *and* polymorphism analyses.

Other polymorphism implementation improvements

In [97], Goubault presents an optimization for latently typed languages. Like our efforts, its source language is not required to be statically type checked. Goubault uses data flow equations to choose unboxed representation. However it is difficult to compare his work with our because the method employed is very different than ours and the paper contains neither measurements nor examples.

Harper and Morrisett present in [105] a new scheme to implement polymorphism. The key idea is to separate values and types for polymorphic functions and to defer the selection of the code to execute until types are known (e.g. at run-time). Unfortunatelly this work addresses statically typed languages and cannot be applied to languages such as Scheme.

7.4.4 SUA and mixed representation

In this section, we use the SUA approximation to introduce unboxed representations. This is done in two stages.

Type election

A first stage after the SUA analysis is the *type election* which gives types to all variables and function results. This pass obviously uses SUA approximations. It does not perform any data flow analysis to choose better type. Let us study type election on the following Scheme program (`-fx` and `=fx` are the fixnum subtraction and equality test procedures) :

```
(define (bcopy! dst src size)
   (let loop ((i (-fx size 1)))
      (if (=fx i -1)
          0
          (let ((c (string-ref src i)))
             (string-set! dst i c)
             (loop (-fx i 1))))))

(define (copy-string src)
   (let* ((len (string-length src))
          (new (make-string len)))
      (bcopy! new src len)
      new))

(copy-string "foo")
```

SUA shows that variables new, dst and src are strings, variables len and i are only fixnums and variable c is a character. SUA is able to compute these type approximations because the types of the library functions (string-length, make-string, string-ref and string-set!) are known by the compiler. Each one of the variables contains one type in its approximation set. Hence type approximations also are the results of type election. If a variable contains more than one type in its approximation set, then it is given the special *obj* type.

SUA merges all possible values in single sets. Hence, if a variable contains one unique type in its approximation, this variable can only take place in a monomorphic program. For instance, if SUA shows that the formal parameter of the identity function can only be a fixnum it means that this function has only been given fixnums as argument, no matter the polymorphism of this function. This is the main advantage of our method compared to Leroy's one. SUA isolates monomorphic parts of polymorphic programs, thus our method allows us to use unboxed representation when Leroy's fails.

Type conversion

The second stage is called *type conversion*. It introduces conversions between boxed representation and unboxed representation in the abstract syntax tree. Objects can be boxed or unboxed. One type exists for these two states. The boxed state is denoted by the *obj* type. Conversion introduction is straightforward because the abstract syntax tree is fully annotated. Here is an example that illustrates type conversion :

```
(define (id x) x)
(define (foo y)
   (id (+fl 1.0 (id y))))
```

Let's assume that foo is exported, hence, y and foo's result have type *obj*. Identity id is invoked with a flonum (result of +fl invocation) and an *obj*, so formal x and the function result are typed as *obj*. Conversions are then inserted.

```
(define (id x) x)
(define (foo y)
   (id (float-box (+fl 1.0 (float-unbox (id y))))))
```

Algorithm 7.8 presents a fragment of the complete type conversion algorithm. Function $\mathcal{T}$ is a function that returns the type of an expression. Function *convert*! takes three arguments : an abstract syntax tree, a *from* type, and a *to* type. It introduces boxing operations required by the translation. Function *convert*! is source language dependent. For latently typed languages with boxing operations, it introduces runtime type checks to ensure the soundness of the translation. For instance, when introducing conversions from *obj* to character, the Scheme *convert*! version also introduces a type check using the char? predicate. No type checks are introduced for statically type checked languages.

7.4.5 Unboxed data storage

In this section we discuss the unboxed representation of the three Λ data types presented in section 3.1.

Unboxed pairs

Pairs have a special status : they are widely used (many library functions exist to manage them) and in Scheme they are heterogeneous data structures (i.e. elements of a list can be of different types). For these reasons, we have decided to make pairs hold boxed values. If pairs were allowed to hold unboxed values, they would not have a fixed size and library functions which have to be applicable to all pairs would be inefficient and difficult to write. To prevent pairs from holding unboxed objects, we simply force $\widehat{\texttt{car}}$ and $\widehat{\texttt{cdr}}$ to have the *obj* type in their approximation sets.

```
𝒞( Π )=
   ∀f∈Π
     𝒞ast( f↓body, 𝒯( f ) )

𝒞ast( atree, τ )=
   case atree
     ⟦ k ⟧ :
       convert!( k, 𝒯( k ), τ )
     ⟦ v ⟧ :
       convert!( v, 𝒯( v ), τ )
     ⟦ (if atree atree_t atree_f) ⟧ :
       let atree=𝒞ast( atree, boolean ),
       let atree_t=𝒞ast( atree_t, τ ),
       let atree_f=𝒞ast( atree_f, τ )
         (if atree atree_t atree_f)
         ⋮
     ⟦ (f a1 ...) ⟧ :
       let atree=⟦ (f 𝒞ast( a1, 𝒯( f↓arg1 ) ) ...) ⟧
         convert!( atree, 𝒯( f ), τ )
   end
```

FIG. 7.8 – Type conversion introduction

Unboxed vectors

Vectors are widely used in all programming languages. We think vectors are not used in the same way as pairs. Even if Scheme vectors are heterogeneous (each vector slot can be of a different type), we think they are mostly used as homogeneous data storage and thus have allowed unboxed values in vectors. Mixed vectors (vector holding boxed *and* unboxed objects) are forbidden because this would prevent the efficient implementation of vector indexing functions. If a vector only contains elements of a given type, it will be transformed into an unboxed vector. If a vector contains at least two elements of different types, it will be a boxed vector. As shown in section 7.2.6, SUA merges all possible values held by a vector in a single approximation so it is easy to check if all its elements are of the same type.

Unboxed procedures

Because closure creation and access to the free variables are handled by the compiler, each closure creation can be treated independently. Closures can hold boxed *and* unboxed values (because SUA distinguishes approximated values in the closures free variables) but unboxed closures are not allowed.

7.5 Experimental results

SUA has been implemented in the new release of the Bigloo Scheme/ML compiler. Both stack optimization and unboxed representation are implemented. Hence, we have been able to make experimental analyses and performance measurements.

Experimental results obtained by running some Scheme benchmarks on a DEC Alpha (DEC 3000/300 (150 MHz), running OSF/1 v3.0, with 160 MBytes of memory) are given in Figure 7.9. The times given are user+system time, including garbage collection time. Bigloo1.7 is the current distributed version of the system, Bigloo1.8 is the new version including the unboxed representation and the stack allocation optimization. Both versions of Bigloo use Boehm's garbage collector release 4.7 [25]. This collector allows ambiguous pointers and uses a traditional mark & sweep algorithm. Gsc is the Gambit-C compiler version 2.3a, S2c is Bartlett's Scheme-to-C compiler version 15mar93jfb [22] and Gcc is the popular Gnu C compiler version 2.6.3, used at optimized level 2. Here is a short description of the Scheme test programs we used :

Nucleic (3496 lines) :	Floating-point arithmetic.
Fft (127 lines) :	Floating-point arithmetic, loops.
Bcopy (43 lines) :	Strings, chars, fixnum, loops.
Ttak (20 lines) :	Function calls (with tuples).
Beval (548 lines) :	Functionals, conditional.
Boyer (606 lines) :	Term processing, functionals.
Maze (800 lines) :	Arrays, fixnum, iterations.
Slatex (2821 lines) :	IO, strings, lists.
Mbrot (46 lines) :	Floating-point arithmetic, loops.

	Compiler				
Test	Bigloo1.8	Bigloo1.7	Gsc	S2c	Gcc
Nucleic	9.0 s	47.2 s	10.3 s	◦	*4.1* s
Fft	1.3 s	22.7 s	5.6 s	39.7 s	*1.1* s
Bcopy	*9.9* s	12.2 s	14.5 s	12.2 s	*9.9* s
Ttak	2.9 s	12.0 s	4.8 s	57.2 s	*1.9* s
Beval	6.7 s	*6.5* s	6.9 s	14.8 s	•
Boyer	*3.4* s	*3.4* s	3.8 s	4.1 s	•
Maze	*6.2* s	7.7 s	8.0 s	18.7 s	•
Slatex	*7.8* s	*7.8* s	23.9 s	22.9 s	•
Mbrot	*1.0* s	20.1 s	9.2 s	35.6 s	*1.0* s

FIG. 7.9 – Runtime statistics on DEC Alpha

Significant speed up occurs with the numerical benchmarks **Nucleic**, **Fft** and **Mbrot**. On **Mbrot** and **Fft** (which is a translation of a C routine from [167], not the Lisp version from the Gabriel suite) Bigloo's performance is very close to Gcc. **Fft** and **Mbrot** are efficiently compiled by Bigloo ; no floating point values get boxed. **Fft** makes use of vectors of floats which are optimized as described in Section 7.4.5.

Nucleic computes 7 million floating point values. Our unboxing optimization allows Bigloo to only allocate 13608 flonums in the heap. The difference in performance between the Bigloo and Gcc versions is mainly due to the use of structures to hold 3 D points. In the C version, all the structures are explicitly allocated on the stack. The Scheme version does not allow our stack optimization to be frequently applied. Hence, profiling the Bigloo executable shows that even though many heap allocations are avoided, 30% of the execution time is still spent in the garbage collector.

Ttak is written in a ML style using tuples to pass arguments. Our stack optimization avoids heap allocation entirely and the speedup is thus important. The impact of stack allocation depends on the program tested. The most important speedup is observed for **Ttak** (75% of memory is allocated on the stack, which leads to a speed up factor of 5).

Figure 7.10 presents dynamic statistics on the amount of memory allocated by the programs. For each program tested and for each compiler, the amount of heap memory allocated is given. The total amount of memory allocated on the stack for Bigloo1.8 is also given.

In accordance with the execution time speedup, the main reduction of heap allocation is observed on numerical programs (**Nucleic**, **Fft** and **Mbrot**). Except for the **Ttak** program, stack allocations are not widely applied. This poor result may come from the style of our programs. The natural Scheme style is to write "allocating" functions which return fresh allocations as in :

```
(define (foo x) (car (gee x)))
(define (gee x) (cons x x))
```

The pair built in `gee` cannot be stack allocated by our method.

The worst case complexity of the SUA algorithm is high. The maximum number of iterations to reach the fix point is the product of the maximum size of approximation sets and the maximum

	Memory allocated		
Test	Bigloo1.7	Bigloo1.8 (heap)	Bigloo1.8 (stack)
Nucleic	747589 k	127523 k	5655 k
Fft	425432 k	174 k	0 k
Bcopy	140 k	98 k	0 k
Ttak	301148 k	0 k	301148 k
Beval	32947 k	32903 k	0 k
Boyer	14369 k	14318 k	0 k
Maze	17563 k	15902 k	1 k
Slatex	67607 k	67431 k	2 k
Mbrot	265589 k	27 k	0 k

FIG. 7.10 – Allocation statistics on DEC Alpha

number of approximation sets (i.e. n^2 for a program of size n). Each iteration has a $\mathcal{O}(n^2)$ complexity (the call graph traversal is $\mathcal{O}(n)$ and operations performed on the tree's nodes are $\mathcal{O}(n)$). The overall complexity is thus $\mathcal{O}(n^4)$. In spite of this complexity, our analysis is relatively fast in practice. Figure 7.11 presents statistics on compilation time. For each program tested, we have measured the time required by SUA, the compilation time until the C code production and the global compilation time including the C compilation. In the worst case (**Slatex**), the time required by SUA is only 20% of the overall compilation.

	Compilation time				
Test	SUA	Bigloo1.8	δ	Bigloo1.8+cc	δ
Nucleic	7.1 s	22.9 s	*0.31*	538 s	*0.01*
Fft	0.1 s	0.8 s	*0.13*	2.9 s	*0.03*
Bcopy	0.1 s	0.5 s	*0.20*	1.5 s	*0.07*
Ttak	0.1 s	0.7 s	*0.14*	2.6 s	*0.04*
Beval	1.8 s	3.9 s	*0.46*	17.8 s	*0.11*
Boyer	0.1 s	0.9 s	*0.11*	3.5 s	*0.03*
Maze	0.4 s	2.0 s	*0.20*	6.9 s	*0.06*
Slatex	15.7 s	22.6 s	*0.69*	79.7 s	*0.20*
Mbrot	0.0 s	0.5 s	*0*	1.8 s	*0*

FIG. 7.11 – Compilation statistics on DEC Alpha

7.6 Conclusion

We have presented in this paper a new static analysis method called *Storage Use Analysis* (SUA) which extends Shivers' 0cfa to modules and general data storage. This analysis allows two important optimizations : unboxed representation and stack allocation. None of these optimizations require type information, so both statically typed languages like ML and latently typed languages like Scheme can use them. Experimental results demonstrate important speedups for numerical applications where a speedup factor of 20 has been measured for some programs.

Acknowledgments

Many thanks to Pierre Weis and Alain Deutsch for early discussions and to Xavier Leroy and Joel F. Bartlett for their helpful feedbacks on this work.

Chapitre 8

Inline expansion : *when* and *how* ?

Cet article a été publié dans [207].

Intégration fonctionnelle : *quand* et *comment* ?

L'intégration fonctionnelle (*inlining*) est une optimisation qui améliore les performances des programmes compilés en supprimant certaines des séquences d'appel de fonctions et en permettant aux autres optimisations de s'appliquer plus largement. En contrepartie, l'*inlining* a aussi l'inconvénient d'augmenter la taille des programmes. Cela peut conduire à une dégradation des performances. Afin de se débarrasser de ce phénomène fâcheux nous présentons une technique d'*inlining* qui est facile à implanter et qui limite l'augmentation du code produit en combinant une analyse statique permettant de décider *quand* procéder à l'intégration et plusieurs schémas d'expansions indiquant comment les intégrations doivent être effectuées. Nous présentons les résultats expérimentaux qui ont conduit à la conception de ce nouveau procédé d'intégration. Nous concluons cet article par des mesures de performances qui montrent que notre optimisation parvient à diminuer les temps d'exécution des programmes compilés tout en les maintenant de petites tailles.

Mots clés : Compilation, Optimisation, Intégration fonctionnelle, Programmation fonctionnelle.

Inline expansion : *when* and *how* ?

Inline function expansion is an optimization that may improve program performance by removing calling sequences and enlarging the scope of other optimizations. Unfortunately it also has the drawback of enlarging programs. This might impair executable programs performance. In order to get rid of this annoying effect, we present, an easy to implement, inlining optimization that minimizes code size growth by combining a compile-time algorithm deciding *when* expansion should occur with different expansion frameworks describing *how* they should be performed. We present the experimental measures that have driven the design of inline function expansion. We conclude with measurements showing that our optimization succeeds in producing faster codes while avoiding code size increase.

Keywords : Compilation, Optimization, Inlining, Functional languages.

8.1 Introduction

Inline function expansion (henceforth "inlining") replaces a function invocation with a modified copy of the function body. Studies of compilation of functional or object-oriented languages show that inlining is one of the most valuable optimizations [8, 38]. Inlining can reduce execution time by removing calling sequences and by increasing the effectiveness of further optimizations :

- Function call sequences are expensive because they require many operations (context save/restore (i.e. memory fetches) and jumps). For small functions, the call sequence can be more expensive than the function body.
- Inlining can improve the effectiveness of the other compiler optimizations because inlined function bodies are modified copies. The formal function parameters are replaced with (or bound to) the actual parameters. Known properties of the actual parameters can then be used to optimize the duplicated function body. For instance, inlining helps the compilation of polymorphic functions because modified versions may become monomorphic. Finally, inlining helps the optimization of both the called and the calling function. Information about actual parameters can be used to improve the compilation of an inlined function body ; information about the result of an inlined function can be used to improve the calling function.

As inlining duplicates function bodies, it has an obvious drawback : it increases the size of the program to be compiled and possibly the size of the produced object file. In this case, compilation becomes more time consuming (because after inlining the compiler must compile larger abstract syntax trees) and, in the worst case, execution can become slower. On modern architectures, best performance is obtained when a program fits into both the instruction and data caches. Clearly, smaller executables are more likely to fit into the instruction cache.

To prevent code explosion, inlining expansion cannot be applied to all function calls. The most difficult part of the inlining optimization is the design of a realistic inlining decision algorithm. This paper focuses on this issue. It presents a compile-time algorithm which allows fine control of code growth and does not require profiling information or user annotations.

Functional languages are characterized by extensive use of functions and particularly recursive functions. Both types of functions can be inlined within a unique framework but a refined expansion can be achieved for recursive functions. This paper presents an efficient framework for inlining recursive functions. Experimental measurements show this framework to be one of the keys to controlling code size growth. This is done by a combination of an inlining decision algorithm (deciding *when* to perform the expansions) and *ad-hoc* expansion frameworks (describing *how* to perform the expansions).

Inlining may impair code production because of los of high level informations. For instance, Cooper *et al.* report in [51] that inlining discards aliasing informations in Fortran programs, so that compilers are unable to avoid interlock during executions. Davison and Holler show in [60] that inlining for C may increase register save and restore operations because of C compilers artifacts. On our part, we have never noticed decreases of performance when activating inlining optimization.

This paper is organized as follows : section 8.2 presents a study of previous algorithms and schemes. Section 8.3 presents our new algorithm. Section 8.4 presents the different inlining frameworks. Section 8.5 reports on the impact of the inlining expansion optimization in Bigloo, our Scheme compiler.

8.2 Inline expansion : *when*

8.2.1 Previous approaches

This section contains a presentation of the main inlining decision rules and algorithms previously published. The remarks made in this section are the basis for the design of the algorithm presented in section 8.3.

User inlining indications

Some systems (e.g. gcc) or language definitions (e.g. C++) allow programs to contain inlining annotations.

Rule 1 (inlining annotation) *Let d be the definition of a function f; a call to f is inlined if d has an "inline" annotation.*

Inlining annotations are useful to implement some parts of the source language (for instance, in Bigloo [202], our *Scheme & ML* compiler, inlining annotations are used intensively in the libraries for accessing and mutating primitives like `car`, `cdr`, `vector-ref` or `vector-set!`) but cannot replace automatic inlining decisions. By contrast an automatic inlining decision algorithm can choose to inline a function at a call site, but also not to do it at another call site. User inlining annotations are attached to function definitions and, as a consequence, are not call site dependent. As a result, misplaced inlining annotations can lead to code size explosion. Furthermore, inlining annotations require the programmer to have a fair understanding of the compiler strategy in order to be able to place annotations judiciously.

Size-based criteria

Simpler inlining decision algorithms [225, 38] are based on the body size of called functions :

Rule 2 (body-size) *Let $\mathcal{K}$ be a constant threshold, f a function, s the body size of f; a call to f is inlined if s is less than $\mathcal{K}$.*

Rule 2 succeeds in limiting the code size growth without recursive function definitions because any function call of a program cannot be replaced by an expression bigger than $\mathcal{K}$. Hence, inlining expansion of a program containing c calls can increase the code size by at most $c \times \mathcal{K}$. In order to prevent infinite loops when inlining recursive functions, rule 2 has to be extended :

Rule 3 (body-size & nested-call) *Let f be a function, k a call to f; k is inlined if f satisfies rule 2 and if k is not included in the body of f.*

This rule helps preventing code explosion and is fairly easy to implement, but it is very restrictive. It may prevent inlining of functions which could have been inlined without code growth. For example, let's study the following Scheme definition :

```
(define (plus x1 x2 x3 ... xn)
   (+ x1 x2 x3 ... xn))
```

Let n be larger than the constant threshold $\mathcal{K}$. Therefore, this `plus` function cannot be inlined because it is said to be too large (its body size is greater than $\mathcal{K}$). However, inlining calls to `plus` does not increase the compiled code size because the body size of `plus` is not greater than any call to `plus`. From this remark, Appel presents in [8] an improvement of Rule 3 :

Rule 4 (body-size vs call-size) *If the body of a function f is smaller than the overhead required for a function call, then f may be inlined without increasing the program size.*

Savings estimates

The inlining of a call to a function f replaces the formal parameters by the actual parameters and some of them may have special properties (constants, for instance). Other optimizations such as *constant folding* may be able to shrink the modified version of the body of f. Since Rule 4 neglects this, we define :

Rule 5 (saving estimations) *If the body of f, after inlining, will shrink by further optimizations to become smaller than the overhead required for a function call, then the call to f may be inlined.*

Two implementations of this approach have been described. The first one, due to Appel [8], estimates the savings of the other optimizations without applying them. The second, due to Dean and Chambers [65], uses an *inline-trial* scheme : rather than just estimating the savings of other optimizations, these are applied and the savings are measured by inspecting the result of the compilation. To limit the number of required compilations, each inlining decision is stored in a persistent database. Before launching a full compilation of a call site function, the inlining decision algorithm scans its database to find a similar call (a call to the same function with the same kind of parameters). Rule 5 has, however, two important drawbacks :

- The two techniques estimate the impact of the other optimizations on the body of a called function f in the context of a specific call site included in a function g. Neither computes the savings of applying the other compiler optimizations on the body of g, due to the added expense. However, after inlining, additional information may be known about a function result. For instance, in the following Scheme program :

```
1 :  (define (inc i x)
2 :     (if (fixnum? i)
3 :         (fixnum+ i x)
4 :         (flonum+ i x)))
5 :  (define (foo x)
6 :     ...
7 :        (let ((y (inc 1 x)))
8 :           (if (fixnum? y)
9 :               ...)))
```

  Inlining the call to `inc` in `foo` (line *7*) allows better compilation of the body of `inc` because, since `i` is bound to the fixnum constant `1`, the test `(fixnum? i)` (line *2*) can be removed. After inlining and test reduction, it appears that `y` can only be bound to a fixnum. This information can be used to improve the compilation of `foo` (by removing, for instance, the line *8* test).
- The saving estimations can only be computed for local optimizations. Global optimizations (such as *inter-procedural register allocations* or *control flow analysis*) require compilation of the whole program and their results are mostly unpredictable. Because these optimizations are often slow, it is, in practice, impossible to apply them each time a function could be inlined.

Because saving estimations are computed on an overly restricted set of optimizations, we think rule 5 is not highly efficient. It fails to measure the real impact of inlining in further optimizations.

Profile-based decision

Some inlining decision algorithms use profile information. Programs are run with various sets of input data and statistics are gathered. Inlining decisions are taken based on these statistics. Two papers present such works [195, 113]. They are based on the same rule that can be merged with rule 2 to prevent excessive code growth :

Rule 6 (profiling statistics) *When profiling statistics show that the execution time of an invocation of a function f is longer than the execution time of the evaluation of the body of f, f could be inlined.*

We do not think profile-based decision algorithms are practical because they require too much help from the programmer. A judicious set of executions must be designed and many compilations are needed.

8.3 The inlining decision algorithm

From the remarks of Section 8.2.1, we have designed our own inlining decision algorithm.

8.3.1 The input language

The input language of our algorithm is very simple. It can be seen as a small Scheme [114] language with no higher order functions. It is described in the grammar below :

Syntactic categories

v	$\in$	VarId	(Variables identifier)
f	$\in$	FunId	(Functions identifier)
Λ	$\in$	Exp	(Expressions)
k	$\in$	Cnst	(Constant values)
Π	$\in$	Prgm	(Program)
Γ	$\in$	Def	(Definition)

Concrete syntax

Π	::=	$\Gamma \ldots \Gamma\ \Lambda$
Γ	::=	`(define` ($f\ v \ldots v$) Λ)

Λ	::=	k
	|	v
	|	`(let` (($v\ \Lambda$) ...) Λ)
	|	`(set!` $v\ \Lambda$)
	|	`(labels` ((f ($v \ldots v$) Λ) ...) Λ)
	|	`(if` $\Lambda\ \Lambda\ \Lambda$)
	|	`(begin` $\Lambda \ldots \Lambda$)
	|	($f\ \Lambda \ldots \Lambda$)
	|	`(+` $\Lambda\ \Lambda$)

A program is composed of several global function definitions and of one expression used to initiate computations. Local recursive functions are introduced by the `labels` special form. Other constructions are regular Scheme constructions.

8.3.2 Principle of the algorithm

Our algorithm uses static information. The decision to expand a call site depends on the size of the called function, the size of the call (i.e. the number of actual parameters) and the place where the call is located. Our decision algorithm does not require user annotation or profiling statistics. Inspired by [8, page 92] the idea of the algorithm is to allow code growth by a certain factor for each call site of the program. When a call is inlined, the algorithm is recursively invoked on the body result of the substitution. The deeper the recursion becomes, the smaller the factor is.

We illustrate the algorithm's behavior on the following example :

```
1 :  (define (inc-fx x) (+ x 1))
2 :  (define (inc-fl x)
3 :    (inc-fx (inexact->exact x)))
4 :  (define (inc x)
5 :    (if (fixnum? x)
6 :        (inc-fx x)
7 :        (inc-fl x)))
8 :  (define (foo x) (inc x))
9 :  (foo 4)
```

Suppose that at recursion depth zero we permit call sites to become 4 times larger and we make each recursive call to the inlining algorithm divide this multiplicative factor by 2 (later on, we will study the impact of the choice of the regression function). The line *8* call to `inc` has a size of 2 (1 for the called function plus 1 for the formal argument). The body size of `inc` is 7 (1 for the conditional, 2 for the test and 2 for each branch of the conditional). Hence, the call is expanded. Before expanding the body of `inc` in line *8*, the inlining process is launched on the body of the function with a new multiplicative factor of 2 (half the initial factor of 4). The inlining process reaches line *6*, the call to `inc-fx`. The size of the body of this function is 3 (1 for the `+` operator call and 1 for each actual argument), the call size is 2 hence this call is expanded. No further inlining can be performed on the body of `inc-fx` because `+` is a primitive operator. The inlining process then reaches the call of line *7*. The call to `inc-fl` is inlined, the multiplicative factor is set to 1 and the inner call to `inc-fx` (line *3*) is reached. This call cannot be expanded because the amount of code growth is less than the body of `inc-fx`. After the inlining process completes, the resulting code is :

```
𝒦 :  an external user parameter

ℐ( Π )=
   ∀f∈Π↓definitions
     f↓body ← ℐast( 𝒦, ∅, f↓body )

ℐast( k, 𝒮, Λ )=
   case Λ
     ⟦ k ⟧ :
        Λ
     ⟦ v ⟧ :
        Λ
     ⟦ (let ((v0 Λ0) ...) Λ) ⟧ :
        let Λ'0=ℐast( k, 𝒮, Λ0 ), ...
          ⟦ (let ((v0 Λ'0) ...) ℐast( k, 𝒮, Λ )) ⟧
     ⟦ (set! v Λ) ⟧ :
        ⟦ (set! v ℐast( k, 𝒮, Λ )) ⟧
          ⋮
     ⟦ (f a0 ...) ⟧ :
        let a'0=ℐast( k, 𝒮, a0 ), ...
          ℐapp( k, 𝒮, f, a'0, ... )
   end
```

FIG. 8.1 – The abstract syntax tree walk

```
(define (inc-fx x) (+ x 1))
(define (foo x)
  (if (fixnum? x)
    (+ x 1)
    (inc-fx (inexact->exact x))))
(foo 4)
```

Dead functions have been removed (`inc` and `inc-fl`) and, as one can notice, the total size of the program is now smaller *after* inline expansion. Experimental results (see section 8.5) show that this phenomenon is frequent : in many situations, our inline expansion reduces the resulting code size.

8.3.3 The algorithm

The main part of the algorithm is a graph traversal of the abstract syntax tree. All function definitions are scanned in a random order (the scanning order has no impact on the result of the optimization). The inlining process, $\mathcal{I}_{ast}$ (algorithm 8.1), takes three arguments : a multiplicative factor (k), a set of functions ($\mathcal{S}$) and an abstract syntax tree (Λ). It returns new abstract syntax trees.

Function calls satisfying the $\mathcal{I}_{app?}$ predicate are inlined using the $\mathcal{I}_{app}$ function (algorithm 8.2). As one can notice, the inlining decision is context dependent. A given function can be inlined on one call site and left unexpanded on another site. A function call is inlined if its code size growth factor is strictly smaller than the value of k. This criteria is strong enough to avoid infinite recursions. This current version of $\mathcal{I}_{app?}$ does not make use of the $\mathcal{S}$ argument. Later versions (Section 8.4.2) will. The expansion of a call to a function f is computed by $\mathcal{I}_{let}$ (algorithm 8.2). It replaces the call by a new version of the body of f. This new version is a copy of the original body, α-converted and recursively inlined. To recursively enter the inlining process, a new factor is computed. Section 8.5.1 will study the impact of the $\mathcal{D}ec$ function on the inlining process.

8.3.4 Inlining in presence of higher-order functions

Inlining a function call requires knowing which function is called in order to access its body. In the presence of higher-order functions, the compiler sometimes does not know which function is invoked. The algorithm presented above can be extended to accept higher-order functions by

$$
\begin{array}{l}
\mathcal{I}_{app}(\ k,\ \mathcal{S},\ f,\ \Lambda_0,\ \ldots,\ \Lambda_n\ )\texttt{=}\\
\quad \textbf{if}\ \mathcal{I}_{app?}(\ k,\ \mathcal{S},\ f,\ n{+}1\ )\\
\quad\quad \textbf{then}\ \mathcal{I}_{let}(\ k,\ \mathcal{S},\ f,\ \Lambda_0,\ \ldots,\ \Lambda_n\ )\\
\quad\quad \textbf{else}\ [\![\ (f\ \Lambda_0\ \ldots\ \Lambda_n)\ ]\!]\\
\\
\mathcal{I}_{app?}(\ k,\ \mathcal{S},\ f,\ csize\ )\texttt{=}\\
\quad function\text{-}size(\ f{\downarrow}_{body}\ ) < k{*}csize\\
\\
\mathcal{I}_{let}(\ k,\ \mathcal{S},\ f,\ \Lambda_0,\ \ldots\ )\texttt{=}\\
\quad \textbf{let}\ x_0{=}f{\downarrow}_{formals_0},\ \cdots\\
\quad\quad [\![\ \texttt{(let ((}x_0\ \Lambda_0\texttt{) ...)}\ \mathcal{I}_{ast}(\ \mathcal{D}ec(\ k\ ),\ \{f\}\bigcup\mathcal{S},\ f{\downarrow}_{body}\ ))\ ]\!]
\end{array}
$$

FIG. 8.2 – The let-inline expansion

adding the straightforward rule that calls to unknown functions are not candidates to expansion. In order to enlarge the set of the possibly inlined function calls of higher order languages, Jagannathan and Wright have proposed in [117] to apply a control flow analysis before computing the inline expansion. For each call site of a program, the control flow analysis is used to determine the set of possibly invoked functions. When this set is reduced to one element the called function can be inlined. This work is complementary to the work presented here. Jagannathan and Wright enlarge the set of inlining candidates, while we propose an algorithm to select which calls to inline from this set.

8.4 Inline expansion : *how*

Section 8.2 described the algorithm to decide *when* a function call should be inlined. In this section we show *how* a functional call should be inlined. Functions are divided into two classes : *non-recursive* and *recursive* ones.

8.4.1 Inlining of non-recursive functions (let-inlining)

The inlining of a call to a non-recursive function has been shown in algorithm 8.2. Non-recursive function inlining is a simple β-reduction : it binds formal parameters to actual parameters and copies the body of the called function.

8.4.2 Inlining of recursive functions (labels-inlining)

Self-recursive functions can be inlined using the transformation $\mathcal{I}_{let}$ but a more valuable transformation can be applied : rather than unfolding recursive calls to a certain depth, local recursive definitions are created for the inlined function (following the scheme presented in [212]). When inlining a recursive function f, $\mathcal{I}_{labels}$ (algorithm 8.3) creates a local definition and replaces the original target of the call with a call to the newly created one. It is more valuable to introduce local functions than unrolling some function calls because the constant propagation and other local optimizations are no longer limited to the depth of the unrolling; they are applied to the whole body of the inlined function. The previous definitions of functions $\mathcal{I}_{app}$ and $\mathcal{I}_{app?}$ have to be modified. Recursive calls should not be further unfolded. This is avoided by making use of the $\mathcal{S}$ argument in the $\mathcal{I}_{app?}$ function.
We show the benefit of the $\mathcal{I}_{labels}$ on the following Scheme example :

```
(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))
```

When inlining `map` into `map-succ`, the compiler detects that `map` is self-recursive, so it inlines it using a local definition :

$\mathcal{I}_{app}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)=
  if $\mathcal{I}_{app?}$(k, $\mathcal{S}$, f, $n+1$)
    then if f is a self recursive function ?
        then $\mathcal{I}_{labels}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)
        else $\mathcal{I}_{let}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)
    else $[\![$ (f Λ_0 ... Λ_n) $]\!]$

$\mathcal{I}_{app?}$(k, $\mathcal{S}$, f, $csize$)=
  if $f \in \mathcal{S}$
    then false
    else $function\text{-}size$($f\downarrow_{body}$) $< k * csize$

$\mathcal{I}_{labels}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)=
  let $\lambda' = \mathcal{I}_{ast}$($\mathcal{D}ec$(k), $\{f\} \bigcup \mathcal{S}$, $f\downarrow_{body}$),
  let $x_0 = f\downarrow_{formals_0}$, ...
    $[\![$ (labels ((f (x_0 ...) λ')) (f Λ_0 ... Λ_n)) $]\!]$

FIG. 8.3 – The labels-inline expansion

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

A further pass of the compiler states that the formal parameter `f` is a loop invariant, so `f` is replaced with its actual value `succ`. Then, `succ` is open-coded and we finally get the equivalent definition :

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

Thanks to the labels-inline expansion and to constant propagation, closure allocations are avoided and computed calls are turned into direct calls. The whole transformation speeds up the resulting code and it may reduce the resulting code size because the code for allocating closures is no longer needed.

8.4.3 Inlining as loop unrolling (unroll-inlining)

We have experimented an *ad-hoc* inlining scheme for loops. Here we consider a loop to be any recursive function with one single inner recursive call. When a loop is to be inlined, a local recursive function is created (according to the $\mathcal{I}_{labels}$ transformation) followed by a traditional unrolling. This new expansion scheme requires the slight modifications to $\mathcal{I}_{app}$ and $\mathcal{I}_{app?}$ given in algorithm 8.4. The unrolling is actually a simple mix between the previous inlining frameworks. The transformation $\mathcal{I}_{labels}$ is applied once, followed by as many $\mathcal{I}_{let}$ transformations as the multiplicative factor k allows. We illustrate the unroll-inline transformation on the preceding `map-succ` example :

```
(define (map-succ l)
 (labels ((map (l1)
   (if (null? l1) '()
    (cons (+ 1 (car l1))
     (let ((l2 (cdr l1)))
      (if (null? l2) '()
       (cons (+ 1 (car l2))
        (map (cdr l2))
       )))))))
  (map l)))
```

$\mathcal{I}_{app}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)=
  if $\mathcal{I}_{app?}$(k, $\mathcal{S}$, f, n+1)
    then if f is a self recursive function ? **and** $f \notin \mathcal{S}$
        then $\mathcal{I}_{labels}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)
        else $\mathcal{I}_{let}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)
    else $[\![$ (f Λ_0 ... Λ_n) $]\!]$

$\mathcal{I}_{app?}$(k, $\mathcal{S}$, f, *csize*)=
  function-size($f{\downarrow}_{body}$) <k*$csize$

FIG. 8.4 – The unroll-inline expansion

8.4.4 Related work

Little attention has been formerly given to *how* the expansion should be performed. Previous works have considered it as a straightforward β-reduction. Inlining of recursive functions has been mainly addressed into three previous papers :

- In the paper [17] H. Baker focuses on giving a semantics to the inlining of recursive functions. Inlining of recursive functions is thought as a loop unrolling by unfolding calls until a user determined depth level. The paper neither studies the impact of this inlining framework on run time performance nor attempts to present optimized transformations. We even think that the proposed transformations would probably slow down executions (because they introduce higher order functions which are difficult to implement efficiently).
- We have made a previous presentation of the labels-inline transformation in [212]. It does not focus on the inlining optimization and it merely presents the transformation without studying its impact.
- The labels-inline transformation has been used by A. Appel in [9]. His approach differs a little bit from our because Appel introduces header around every loop independently of the inlining optimization. We think this has two drawbacks. First, un-inlined loops have to be cleaned up (that is, headers have to be removed) otherwise there is an extra call overhead. More importantly, introducing header make functions abstract syntax tree bigger. Since the inlining algorithm uses functions size to decide to inline a call, loops with header introduced are less likely to be inlined.

8.5 Experimental results

For experimental purposes, we used ten different Scheme programs, written by different authors, using various programming styles. Experiments have been conducted on a DEC ALPHA 3000/400, running DEC OSF/1 V4.0 with 64 Megabytes of memory. In order to be as architecture-independent as possible, we have measured duration in both user plus system cpu time and number of cpu cycles (using the `pixie` tool). For all our measures, even when the multiplicative factor is set to 0, primitive operators (such as `+` or `car`) are still inlined.

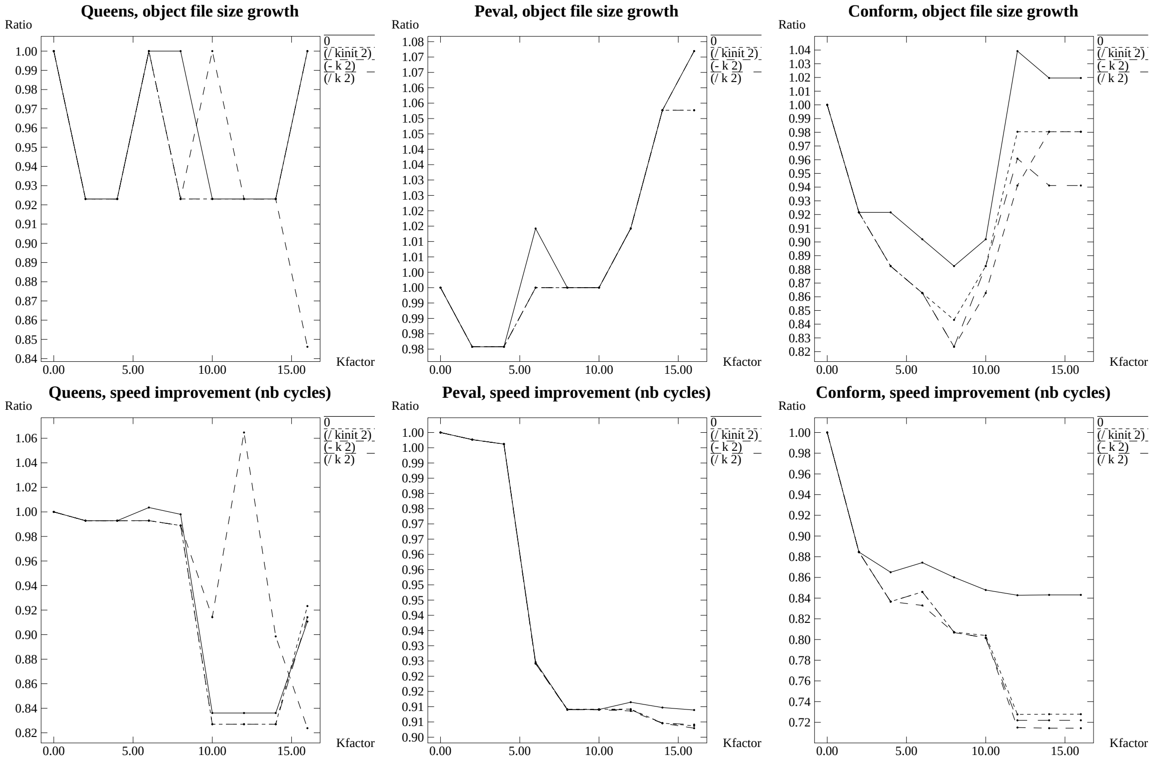
programs	*nb lines*	*author*	*description*
Bague	104	P. Weis	Baguenodier game.
Queens	132	L. Augustsson	Queens resolution.
Confo	596	M. Feeley	Lattice management.
Boyer	640	R. Gabriel	Boyer completion.
Peval	643	M. Feeley	Partial evaluator.
Earley	672	M. Feeley	Earley parser.
Matrix	753	_	Matrix computation.
Pp	757	M. Serrano	Lisp pretty-printer.
Maze	879	O. Shivers	Maze game escape.
Nucleic	3547	M. Feeley	Molecular placement.

8.5.1 Selecting the $\mathcal{D}ec$ regression function

We have studied the impact of the $\mathcal{D}ec$ function, first used in algorithm 8.2. We have experimented with three kind of regression functions : decrementations : $\lambda_{-\mathcal{N}}$=`(λ (k) (- k` $\mathcal{N}$`))`, divisions : $\lambda_{/\mathcal{N}}$=`(λ (k) (/ k` $\mathcal{N}$`))` and two *step* functions : $\lambda_{=0}$=`(λ (k) 0)` and $\lambda_{=k_{init}/\mathcal{N}}$=`(λ (k) (if (= k k`$_{init}$`) (/ k` $\mathcal{N}$`) 0))`.

For each of these functions, we have measured the code size growth and the speed improvement. Since measurement showed that varying $\mathcal{N}$ has a small impact, we just present measurements where $\mathcal{N}$ has been set to two.

We present results for only three programs, **Queens**, **Peval** and **Conform** because they are representative of the whole 10 programs. The X axis represents the initial value of the k multiplicative factor. The Y axis of the upper graphics represents the normalized object file size. The Y axis of the lower graphics represents the normalized durations (cpu cycles).

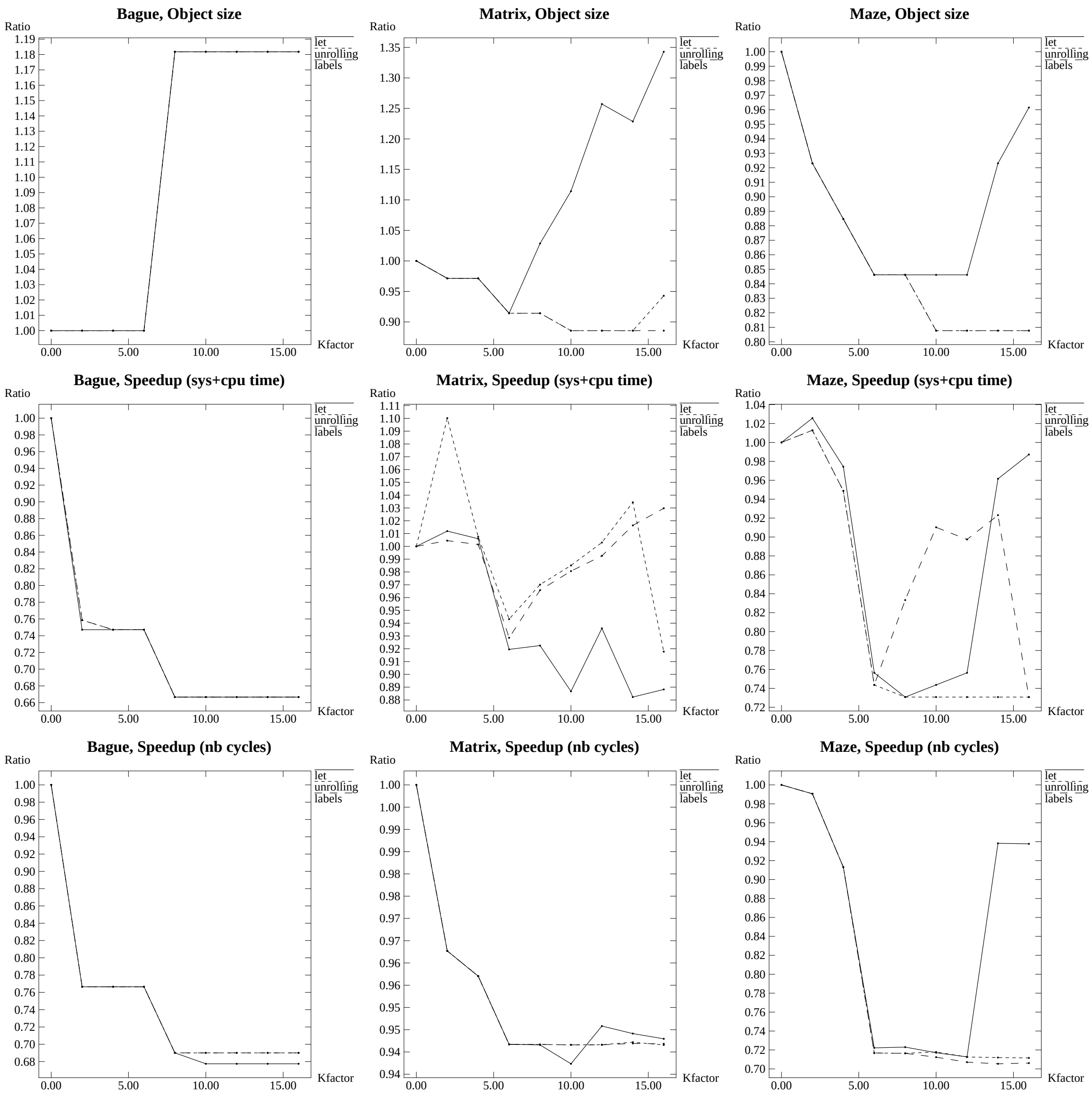


Two remarks can be made on these measurements :

- The regression function $\lambda_{=0}$ produces less efficient executables (see for instance **Conform**) than others regression functions. This proves that a recursive inlining algorithm (an algorithm trying to inline the result of an inlined call) gives better results that a non recursive one.
- $\lambda_{=k_{init}/\mathcal{N}}$, $\lambda_{-\mathcal{N}}$ and $\lambda_{/\mathcal{N}}$ lead to about the same results. This demonstrates that only the very first recursive steps of the inlining are important. Functions like $\lambda_{-\mathcal{N}}$ or $\lambda_{/\mathcal{N}}$ have the drawback to authorize large code size expansion (about $2^{\frac{1}{\mathcal{N}} * \log_2 k^2}$ in the worst case for $\lambda_{/\mathcal{N}}$). *Step* functions like $\lambda_{=k_{init}/\mathcal{N}}$ are much more restrictive ($k^2/\mathcal{N}$ in the worst case). Choosing *step* functions leads to small and efficient compiled programs.

8.5.2 The impact of the labels-inlining and unroll-inlining

The second step of our experiment is to study the impact of the labels-inlining and unroll-inlining expansions. Object file size growth ratio and speedup ratio have been compared on 3 programs, enabling or disabling the labels-inlining, enabling or disabling unroll-inlining (unroll-inlining requires labels-inlining). The k multiplicative factor is represented by the X axis while the normalized object file sizes and durations are represented by the Y axis.



8.5.3 The general measurements

The second step of our experiment has been to study the impact of the let-inline, labels-inline and unroll-inline expansions. The results summarized in Figure 8.5 show the speedups and the code size increases for the 10 programs in each of the 3 frameworks. For each of them we have computed a speedup, measured as the ratio of the speed of the inlined divided by the speed of the same program with inlining disabled, and a size, measured as the ratio of the size of the inlined program divided by the size of the original program. The experiment was performed for all values of $\mathcal{N}$ (see Section 8.5.1 for the definition of $\mathcal{N}$) between 1 and 15. The Figure shows the results of the best speedup for each program and framework. Furthermore, we computed speedups and size increases with respect to number of cycles and actual time elapsed. Thus, in Figure 8.5, *cycle*/speed refer to the ratio of cycles and *cycle*/size refers to the ratio of sizes, while *sys+cpu*/speed refers to the ratio of times and *sys+cpu*/size refers to the ratio of size. Note that for a given program and framework, the value of $\mathcal{N}$ giving the best speedup is sometimes different if we measure it in cycles or elapsed time, thus the size increases may also differ.

Programs	Best speedup for $\mathcal{N} \in$[1..15]											
	let				*labels*				*unroll*			
	cycle		*sys+cpu*		*cycle*		*sys+cpu*		*cycle*		*sys+cpu*	
	speed	size	speed	size	speed	size	speed	size	speed	size	speed	size
bague	0.67	1.18	0.66	1.18	0.69	1.18	0.66	1.18	0.69	1.18	0.66	1.18
queens	0.86	1.53	0.89	1.53	0.82	0.92	0.85	0.92	0.81	0.92	0.89	0.92
conform	0.72	1.21	0.83	1.17	0.72	0.96	0.76	0.96	0.72	0.98	0.76	0.98
boyer	0.89	1.0	1.0	1.0	0.89	0.90	0.89	0.90	0.89	0.90	0.89	0.90
peval	0.98	1.05	1.0	1.0	0.90	1.05	0.90	1.0	0.96	1.13	0.94	1.03
earley	0.95	1.0	0.97	1.0	0.95	1.0	0.97	1.0	0.95	1.0	0.97	1.0
matrix	0.93	1.11	0.88	1.11	0.94	0.88	0.92	0.91	0.94	0.94	0.90	0.94
pp	0.94	1.22	0.90	1.11	0.95	1.10	0.92	1.07	0.95	1.14	0.92	1.09
maze	0.71	0.84	0.73	0.84	0.70	0.80	0.73	0.84	0.71	0.80	0.73	0.80
nucleic	0.99	1.15	0.95	1.15	0.98	1.15	0.98	1.1	0.98	1.15	1.0	1.0
Average	*0.86*	*1.13*	*0.88*	*1.11*	*0.85*	*0.99*	*0.86*	*0.99*	*0.86*	*1.01*	*0.87*	*0.98*

FIG. 8.5 – Best speedups

Labels-inlining

Labels-inlining is an important issue in order to minimize object file size. In the worst case, labels-inlining behaves as let-inlining, enlarging object file size, but in general, it succeeds in limiting or even avoiding expansions. This good behavior does not impact on speedup ratios.

Except for a very few programs, let-inlining does not succeed in producing faster (less cpu cycle consuming) executables than labels-inlining but it enlarges too much the object file size. For instance, the let-inlining aborts on **Earley** and **Matrix** for very high initial values of the multiplicative factor k because the abstract syntax trees of these programs become excessively large.

Unroll-inlining

Few improvements come from unroll-inlining. Only **Boyer** and **Maze** are improved when this framework is used. One explanation for this result is that unroll-inlining is barely applied because most loops have only one argument (after removal of invariant parameters) and thus they can then be inlined only if their body is very small.

This is a slightly disappointing result but even the benefits of classical loop unrolling optimizations are not well established. For instance, experience with gcc in [225] is that "[general loop unrolling] usually makes programs run more slowly". On modern architectures performing dynamic instruction scheduling, loop unrolling that increases the number of tests can severely slow down execution. This was partially shown by the measures of the labels-inlining impact that showed that it is more valuable to specialize a loop rather than to unfold outer loop calls.

8.5.4 Related work

The inlining impact depends on the cost of function calls and thus is highly architecture-dependent. Hence, comparison with previous measures reported on inlining speedup, made on different architectures, is difficult. Furthermore, we think that it does not make sense to compare the inlining impact for very different languages. For instance, since a typical C program does not make extensive use of small functions such as a Scheme and ML program, an inlining optimizer for C should probably not adopt a framework tuned for a functional language. We limit the present comparison to a few publications.

– C. Chambers shows that inlining reduces the object file size produced by the Self compiler [38] by a very large factor (in the best case, inlining reduces the object file size from a factor of 4) while making programs run much faster (between 4 to 55 times faster). The explanation is found in [38, section B.3.1] : "In SELF, the compiler uses inlining mostly for optimizing user-defined control structures and variable accesses, where the resulting inlined control flow graph is usually much smaller than the original un-inlined graph. These sorts of inlined constructs are already 'inlined' in the traditional language environment. Inlining of larger 'user-level' methods or procedures does usually increase compile time and compiled code space as has been observed in traditional environments...".

– We share with Jagannathan and Wright [117] three test programs : **Maze**, **Boyer** and **Matrix**. For all of them, we have measured the same speed improvement but our techniques do not increase the object file size while that of Jagannathan and Wright enlarges it by 20%.
– In [9] Appel reports on an average speedup of 5% for the labels-inlining. Our measures do not allow us to conclude in a same way. We have found that the main effect of the labels-inlining is to reduce the size of the inlined programs. As suggested by Appel his improvement may come from the reduction of the closure allocations. Less closures are allocated because in a CPS style, labels-inlining and its hoisting of loop invariant arguments may avoid the construction of some continuations.

Conclusion

We have shown in this paper that the combination of a decision algorithm with different expansion frameworks makes the inline expansion optimization more valuable. It improves run time performances (about 15% on average) while avoiding its traditional drawback, the object code size growth. The decision-making algorithm we have presented is based on static compile time informations and does not require user annotations or profiling data. The expansion framework allows inlining of recursive functions. Both are easy to implement. In Bigloo, our Scheme compiler, the decision-making algorithm and the different expansion strategies constitute less than 4% of the whole source code.

Acknowledgments

Many thanks to Christian Queinnec, Xavier Leroy, Jeremy Dion, Marc Feeley, Jan Vitek and Laurent Dami for their helpful feedbacks on this work.

Chapitre 9

Wide classes

Cet article a été publié dans [208].

Les classes larges

Dans cet article nous présentons les concepts de classes larges (*wide classes*) et d'élargissement (*widening*) qui étendent le modèle objet des langages à base de classes comme Java ou Smalltalk. L'élargissement permet à un objet d'être temporairement étendu, c'est-à-dire transformé en un instance d'une sous-classe, une classe large, et, ultérieurement rétrécit, c'est-à-dire, replacé dans son état initial. Les classes larges partagent les principales caractéristiques des classes standards : elles ont un nom, une super-classe, elles peuvent être instanciées, elles sont associées à un prédicat et à un type et, enfin, elles peuvent être utilisées pour surcharger des définitions de fonctions.

L'élargissement est utile pour implanter des structures de donnée éphémères. En particulier, l'élargissement permet de diminuer la *rétention de données* dans les programmes informatiques. Ce phénomène ce produit quand l'implantation d'une structure de donnée ne permet pas de tenir compte des propriétés de durée de vie des valeurs manipulées. Les programmes subissant ce phénomène consomment trop de mémoire durant leurs exécutions.

Les classes larges peuvent être implantées à faible coût dans tous langages à classes utilisant du typage dynamique. Très peu de modifications au système sont nécessaires. Dans cet article, nous décrivons une implantation raisonnablement simple et efficace des classes larges pour le système Bigloo.

Mots clés : Implantation des langages, héritage dynamique, typage dynamique, modification de type.

Wide classes

This paper introduces the concepts of *wide classes* and *widening* as extensions to the object model of class-based languages such as Java and Smalltalk. Widening allows an object to be temporarily *widened*, that is transformed into an instance of a subclass, a *wide class*, and, later on, to be *shrunk*, that is reshaped to its original class. Wide classes share the main properties of plain classes : they have a name, a superclass, they may be instantiated, they have an associated class predicate and an associated type that may be used to override function definitions.

Widening is also useful to implement transient data storage for long-lasting computations. In particular, it helps reducing *software data retention*. This phenomenon arises when the actual data structures used in a program fail to reflect time-dependent properties of values and can cause excessive memory consumption during the execution.

Wide classes may be implemented for any dynamically-typed class-based programming language with very few modifications to the existing runtime system. We describe the simple and efficient implementation strategy used in the Bigloo runtime system.

Keywords : language implementation, dynamic inheritance, dynamic type checking, instance modification.

9.1 Introduction

Few object-oriented programming languages permit objects to change their nature during execution. If an object `obj` is an instance of a class $\mathcal{C}$ at some point of the execution, then `obj` remains an instance of $\mathcal{C}$ for the whole computation. In other words, the *type* of an object may not evolve over time. Most Object-oriented languages only provide a mechanism to associate behaviors to types by the means of *methods*. In consequence, object-oriented programming languages do not allow the behavior to evolve over time. For instance, consider window objects in an object-oriented window manager. We may imagine several kind of windows such as windows with a menu bar and windows without menu bar. This can be easily implemented in a class-based object-oriented language. We may build a bare window class and a subclass for windows with menu bar. This is easy because if a window has a menu bar, it will keep it for its entire life time. The trouble comes when one wants to express that a window is either visible or iconified. A window may be visible at some point and iconified at another point. The property of being visible or iconified is dynamic ; it changes over time. In consequence, the behavior associated to the window should also change over time. For instance, refreshing a visible window or refreshing an iconified window requires different operations. In general, object-oriented languages do not provide mechanisms that allow a straightforward representation of this phenomenon.

Wide classes extend the standard object-oriented programming model with a disciplined type conversion operation called *widening*. This allows an object to be temporarily *widened* (that is transformed into an object of a subclass, a *wide class*) and then to be *shrunk* (that is reshaped to its original class). Widening is a runtime operation. During the execution, a program explicitly widens and shrinks objects. Widening and shrinking are tools of choice to implement transient states. In our window manager example, we may widen a visible window to an instance of the iconified window class when the program requests iconification and we may shrink the iconified window to a plain visible window when the program requests de-iconification. Widening and shrinking change the type of the window from visible to iconified, so we may write methods that will apply to visible windows and methods that will apply to iconified windows. Furthermore widening may add data attributes needed by the new behavior while shrinking sheds data that is not needed anymore.

Both Smalltalk and Clos propose a construction that allows unrestricted changes to the runtime type of objects. These type-change operations may turn any object into an instance of any class. Widening is more disciplined as type-changes are restricted to (wide) subclasses. In other words, widening preserves subtyping relationships (after widening the new runtime type of an object is a subtype of its former runtime type). This limitation is important for program correctness and essential for obtaining an efficient implementation of wide classes. We show in this paper that wide classes have a simple and fast implementation with two important properties :

– Wide classes have no impact on the implementation of the rest of the runtime system.

– A program that does not make use of widening does not pay any extra cost for that feature.

Neither of these properties hold for the equivalent mechanisms offered in Smalltalk and Clos.

From a programmer's point of view, widening is very useful to implement transient data storage and thus helps facing the problem of *software data retention*.

9.1.1 Software data retention

During the execution of a program, it might happen that some data items have a limited life time but that the actual data structure in which they are contained is too rigid and will waste memory because it will retain space longer than necessary. Softwares that use a *pipelined* architecture, that is softwares that cascade several distinct stages, must frequently face these retention problems.

As an example, let's suppose that in a compiler one would like to introduce a stage that removes useless variables. For the purpose of that stage, the number of occurrences of each variable is needed and we suppose that this information is never needed elsewhere. The problem is "where to store the number of occurrences in the abstract syntax tree". A data structure allocating an extra field holding the number of occurrences for each node that implements variables wastes memory as soon as the stage is completed. (Of course, many stages of a compiler use more than one temporary information.) We say that a program making use of such data structure faces *data retention*.

9.1.2 Fighting software data retention

Current programming languages do not propose mechanisms to reduce software data retention and thus programmers are used to one of the following *ad-hoc* strategies.

1. *Global data* : this strategy consists in allocating a global data structure that provides room for all the data of all the stages of the execution. This solution is inefficient in space because the structure is much larger than the actual data that it contains. Many fields of the data structure will be empty at any given time. Moreover with a programming language that enables automatic memory management, the program must take provisions to explicitly free the unused fields ; otherwise the algorithm to reclaim memory will fail at re-using the memory that has been allocated for a completed stage. In other words, the program must explicitly free the dead data structures ; this greatly minimizes the pros of automatic memory management.

   Further, the *global data* approach fails to enforce strict barriers between stages, and since the information from the previous stages is still available there is a temptation of using it. This is in contradiction to the principles of clear separation between stages of the pipelined architecture.
2. *Copy on entrance* : when entering a new stage the global data structure is copied into a new one that provides rooms for the local data that belongs to that stage. This solution is safe to avoid confusion between distinct stages. It is efficient in space because the size of the largest structure live at a time is the size of the largest structure allocated by each stage. Unfortunately, this framework is inefficient in execution time as it implies a large amount of memory allocation/deallocation. Thus, it is likely that the time spent in the memory manager will be significant. Moreover the time required to copy the structure is important because sharing properties must be preserved from one copy to another.
3. *User field :* with this strategy, a so-called *user field* is added to each structure of the global data type and then each stage hooks its own local structures using this extra field. This solution is frequently used : for example, the C structure `fieldnode` of the `form` library declares a field `usrptr` that can be used by the clients of that library. If this technique has the advantage of being space efficient (only one word is wasted by structure to hold the extra field) it has two severe disadvantages :
   – It forfeits static type checking. The type of the user field has to be the type of the most generic user type (e.g., `void *` in C). As a consequence, the *user field* framework leads to a very common error. The stage S_i reads the user field that is still holding a local data of the stage S_{i-1}. Then an error (or an exception for languages with higher level runtime support) is raised and the execution stops.
   – More pragmatically, it is simply inconvenient to use, as each access to the user slot requires an extra indirection (and possibly an extra coercion for statically type checked languages). This all makes the code dealing with user fields unnecessarily verbose.

Wide classes and widening represent a solution to software data retention. Objects may be widened to hold transient data and shrunk when that data appears to be useless. Wide classes use inheritance. That is, widening affects the dynamic binding of methods. Widening is used extensively in the Bigloo compiler for the Scheme programming language. Bigloo compilation passes are implemented using the *widening/shrinking* mechanism. Upon entrance to a pass, objects are widened with pass-specific fields and, on pass exit, they are shrunk back in order to forget the information related to that pass. Our experience has been that using widening has greatly simplified the organization of the code of the compiler ; from a software engineering point of view this has been beneficial, and from an efficiency point of view the current release of the compiler uses memory more sparingly than its predecessors.

9.1.3 Organization

Section 9.2 presents a general overview of the Bigloo object model. Readers familiar with the Clos-like style may skip that section. Section 9.3 introduces the design of wide classes. The new language constructions are presented here. Examples of small programs using widening are given. This section concludes with a discussion of the limitations of wide classes. Section 9.4 presents widening used in the context of a large program, our Scheme compiler, Bigloo. Section 9.5 discusses the implementation of wide classes and widening operations. Section 9.6 compares wide classes with other object oriented features. Section 9.7 presents some possible extensions to widening and wide classes.

9.2 Bigloo

Bigloo is an open implementation of the Scheme language. From the beginning, the aim was to propose a realistic and pragmatic alternative to the strict Scheme implementation defined by [114]. Bigloo does not implement "all" of Scheme : for example, the execution of tail-recursions may allocate memory and very few arithmetics are implemented (Bigloo only supports integers and reals). However, Bigloo implements numerous extensions : support for lexical and syntactic analysis, pattern matching, an exception mechanism, a foreign interface and a module language. The recent Bigloo versions offers a new extension with an object layer, to a great extent inspired by Meroon [170] of C. Queinnec.

The term "object-oriented" is a rather loose fitting description that has been prefixed to a number of very different languages and systems over the years. We will restrict our attention to two object models : the Smalltalk model [91] where methods are associated to classes and the Clos model [24] where methods are associated to generic functions. Each of those models has many incarnations : with static or dynamic type checking, with single or multiple inheritance, and single or multiple dispatch. In this paper we assume a Clos-like object model with single inheritance and single dispatch.

To introduce the Clos object model, let us present a small example. Consider a class *point* implementing 2 dimensional points and a class *point-3d* implementing 3-d points and suppose that we want to implement a function that computes the norm of these points. Instead of placing methods in the declaration of *point* and *point-3d* that implement this functionality as we would do in Smalltalk or Java, we define a *generic function* that will apply to the *point* and *point-3d* classes :

```
(define-generic (norm::real p::point))
```

In a first step a generic function may be considered as a declaration. No user code is associated to it. A generic function is a placeholder for method definitions. The above declaration of `norm` specifies that methods called `norm` can be defined, taking exactly one argument which is a subtype of *point*. Methods are said to *override* the generic function.

```
(define-method (norm::real p::point)
   (sqrt (sqr (point-x p)) (sqr (point-y p))))

(define-method (norm::real p::point-3d)
   (sqrt (sqr (point-3d-x p)) (sqr (point-3d-y p)) (sqr (point-3d-z p))))
```

The functions `point-x`, `point-y`, `point-3d-x`, ... are accessors, automatically generated by the compiler. Methods can be used as :

```
(let ((p   (instantiate::point (x 3.3) (y 4.1)))
      (p3d (instantiate::point (x 8.9) (y 4) (z 2))))
   (print (+ (norm p) (norm p3d))))
```

The Clos model fits well with the traditional functional programming style. A function can be thought as a generic function overridden with exactly one method. Another reason to choose the

Clos model is the nature and the architecture of the Bigloo compiler that we wanted to rewrite with objects.

Former versions of the Bigloo compiler were written in plain Scheme. The compiler is a program of 40,000 code lines. It reads a program to be compiled and builds the abstract syntax tree used to represent the program. This tree contains a structure of 23 different node types. There is a node for constants, variable assignments, conditionals, function calls, etc. The compiler is made up of stages, each of which can be seen as a process that modifies the abstract syntax tree. The driver is a Scheme function that looks like the following one :

```
(define (compiler src)
   (let ((ast (build-ast src)))
      (macro-expand! ast)         ;; 1st stage
      (function-inline! ast)      ;; 2nd stage
      ...
      (code-generate! ast)))      ;; 20th stage
```

Each stage is implemented in a single file, except for the most complex ones that can be split over various files. For the new releases making use of object-oriented programming, we wanted to keep this decomposition as we felt it was the most natural way to write pipelined systems.

This encouraged us to adopt a generic function object model rather than the traditional Smalltalk model as this last model would have forced us to give up the stage-driven structure in favor of a structure driven by the abstract syntax tree. Each compiler stage would have been split in all classes implementing the tree nodes. A file, implementing the `function-inline!` stage for example, would not exist anymore and its code would have been split among classes.

Adding new nodes to the syntax tree, however, would have been easier for our compiler with the Smalltalk model. Using this model, adding a stage requires to modify and to recompile all the existing code (because we need to add the methods defining the new stage for each class). But, adding a node does not require changing the existing code. Adding a node means that we are adding a feature to the target language (which is uncommon), while adding a stage usually means that we are adding a compiler optimization (very common for compiler writers). Our experience has shown that it is important to ease the task of writing new stages as these have a local impact, while language additions are difficult anyway because they may impact many features.

9.2.1 Class declarations

Classes can be declared static or exported. It is then possible to make a declaration accessible from another module or to limit its scope to one module. The abbreviated syntax of a class declaration is :

```
(class class-id::super-class-id <field>*)

<field> ::= field-id ::type-id
  | (field-id ::type-id <option>*)
  | (* field-id ::type-id <option>*)

<option> ::= read-only
  | (default value)
```

A class can inherit from a single super class. Classes with no specified super class inherit from the `object` class. The type associated with a subclass is a subtype of the type of the super class.

A field is typed (with the annotation `::type-id`), might be immutable (`read-only`), and may have a default value (`default`). Fields with clauses that begin with `*` are said to be indexed : those fields do not contain a single value but a sequence of values. Indexed fields (directly inspired by Meroon) can be used, for example, to implement strings of characters. Here are possible declarations for the traditionals `point` and `point-3d` :

```
(module small-points
   (export (class point
              (x::double read-only (default 0.0))
              (y::double read-only (default 0.0)))
           (class point-3d::point
              (z::double read-only (default 0.0)))))
```

9.2.2 Generic functions

Generic function declarations are function declarations annotated by the `generic` keyword. They can be exported : this means that they can be used from within other modules and that methods can be added to those functions from other modules. They can be static, that is not accessible from within other modules. In this case, no method can be added to the ones introduced by the module that defines the generic function. The syntax to define a generic function is very similar to an usual function definition :

```
(define-generic (fun-id::type-id arg-id::class-id ...) optional-body)
```

Generic functions should have at least one argument as it is the first argument that is used to solve the *dynamic dispatch* of methods. This argument is of type T that must be associated to a class. It is impossible to override generic functions with methods whose first argument is not of a subtype of T. Generic functions can have a body. This body will be evaluated if, when calling a generic function, no method is defined for the type of the first argument. If this situation occurs and the generic function has no body, an error is raised. Here is an example of a generic function definition whose distinguishing argument must be a subtype of `point` :

```
(define-generic (display-point::obj p::point)
   (print "<???>"))
```

9.2.3 Methods

Methods are declared by the following syntactic form :

```
(define-method (fun-id::type-id arg-id::class-id ...) body)
```

Defining a method if there is no defined generic function with a compatible prototype (arguments and result types in a sub-typing relation with those of the generic function) is an error. A method can be written as follows :

```
(define-method (display-point::obj p::point)
   (with-access::point p (x y)
      (print "<point:" x ", " y ">")))
```

Methods override generic function definitions. When executing the following code :

```
(let ((p (instantiate::point)))
   (display-point p))
```

because there is a method defined over the class *point* the code of that method will be executed instead of the default code of the generic function.

9.2.4 Instances

When declaring a class *cla*, Bigloo automatically generates the predicate `cla?`, an allocator `instantiate::cla`, a cloner `duplicate::cla`, accessors (e.g., `cla-x` for a field `x`), modifiers (e.g., `cla-x-set!` for a field `x`), and, an abbreviated access form, `with-access::cla`, to allow accessing and writing fields by referencing them by their name. Here is how to allocate and access an instance :

```
(let ((p (instantiate::point-3d (y -3.4) (x 1.0))))
   ;; The initialization value of a field can be omitted from the arguments list if the
   ;; field has a default value; this is the case for the field z of class point-3d.
   (with-access::point-3d p (x y z)
      (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

9.3 Wide classes

The key point to *wide classes* is that objects may become (may be widened to) instances of wide classes and thus that the type of an object may change dynamically over the execution.

9.3.1 Wide class declarations

The declarations of *wide classes* are similar to the declarations of plain classes :

```
(wide-class class-id::super-class-id <field>*)
```

The super class of a wide class may be a plain class or a wide class. By contrast the super class of a plain class cannot be a wide class. This point will be discussed in Section 9.3.6. As an example, here are the declarations of "named points" and "named 3-d points".

```
(module named-point
   (export (wide-class named-point::point
              name::string)
           (wide-class named-point-3d::named-point
              (z::double (default 0.0)))))
```

9.3.2 Wide instances

Wide objects (i.e. objects that are instances of wide classes) may be used in the exact same way as plain objects. Wide objects are instantiated from wide classes. Wide objects are used to solve generic function run time dispatches and fields of wide objects are fetched from the instances. The types of the wide objects are checked by automatically generated class predicates. Thus, we may write :

```
(define-method (display-point::obj p::named-point)
   (with-access::named-point p (x y name)
      (print "<point(" name "): x:" x ", y:" y ">")))

(define (display-instance x)
   (if (point? x)
       (display-point x)
       (display x)))

(let ((p (instantiate::named-point (name "org") (x 0) (y 0))))
   (display-instance x))
```

9.3.3 Widening

The new feature introduced by wide classes is that during its life-time, an instance of a given class may be *widened* to an instance of one of its wide subclasses. Later on, the same instance may be *shrunk*, that is reshaped to its original class. The mandatory point here is that the widening and shrinking operations do not introduce copies. A widened instance stays the same, being merely extended with new slots. In addition, if an object `obj` satisfies a class predicate $\mathcal{P}$, then after widening, `obj` *still satisfies* $\mathcal{P}$.

The syntax of the widening operator resembles the syntax of the instance allocator :

```
(widen!::class-id instance (field-id0 val0) (field-id1 val1) ...)
```

Class-id is the name of a wide class, the name of the wide class the *instance* is widened to. It is an error if *class-id* is the class identifier of a plain class. The *field-id$_i$* identifiers are the name of the fields of the wide class *class-id*. The values *val$_i$* are initial expressions for the fields of the wide class. As for object instantiation, fields declared with default values may be omitted within a `widen!` expression. Here is an example of widening from the class *point* to the class *named-point*.

```
(let ((pt (instantiate::point (x 0.0) (y 5.3))))
   (widen!::named-point pt (name "dummy"))
   (display-point pt))
```

Instantiation :

Instantiating a wide object is semantically equivalent to creating a plain object and then applying successive widening until the desired wide class is obtained. That is

```
(instantiate::named-point-3d (x 0) (y 0) (z 0) (name "gee"))
```

is a shorten for :

```
(let ((p (instantiate::point (x 0) (y 0))))
   (widen!::named-point p (name "gee"))
   (widen!::named-point-3d p (z 0))
   p)
```

Identity :

The semantics of widening preserves object identity, in Scheme terminology this means that a widened instance is still `eq?` to what it was before widening. Thus,

```
(let* ((p     (instantiate::point))
       (new-p (widen!::named-point p (name "new"))))
   (eq? new-p p))
```

evaluates to true. This feature is very important because it allows some parts of a whole data structure to be widened while preserving sharing relationships. We have tried hard to design widening so as to limit the runtime cost. Section 9.5 presents an implementation strategy which allows us to get widening at no extra implementation cost.

9.3.4 Shrinking

Shrinking is simpler. Instances get shrunk one class at a time by the means of the form :

```
(shrink! instance)
```

It shrinks an *instance* by one level from its widest level. Thus,

```
(let ((p (intantiate::point)))
   (widen!::named-point p (name "dummy2"))
   (widen!::named-point-3d p (z 3.0))
   (display-point p)
   (shrink! p)
   (display-point p)
   (shrink! p)
   (display-point p))
```

when executed, prints out :

```
<point(dummy2): x: 0.0 y: 0.0 z: 3.0>
<point(dummy2): x: 0.0 y: 0.0>
<point: x: 0.0 y: 0.0>
```

It is an error to shrink an object that is not a wide object. Wide objects may be tested with the predicate :

```
(wide-object? obj)
```

As an example, here is the code of a polymorphic function that takes any object and returns the same object with all wide layers removed (if there were any) :

```
(define (shrink-all! obj)
   (if (wide-object? obj)
       (begin (shrink! obj) (shrink-all! obj))
       obj))
```

In presence of multiple widening, it could be useful to specify a destination class when shrinking. The `shrink!` operator accepts an optional argument that denotes that class. The syntax of this form is :

```
(shrink!::class-id instance)
```

Shrinking with a destination class enables several shrink operations to be automatically performed. Its implementation is close to `shrink-all!`. A restriction exists for ***class-id***. It must denote a super class of the actual class of *instance* that is a wide class or the first plain class that *instance* belongs to. The following example is a legal shrink :

```
(let ((p (intantiate::named-point-3d (name "dummy2") (z 3.0))))
   (shrink!::point p)
   (display-point p))
```

9.3.5 Example

Widening may be used to implement transient object properties. For instance, suppose we would like to model marital status. The class implementing persons would be :

```
(class person
   name::string
   fname::string
   (sex::symbol read-only))
```

As we can see, people are allowed to change their name but not their sex. The identity of a person can be printed as follows :

```
(define-method (object-display p::person)
   (with-access::person p (name fname sex)
      (print "firstname : " fname)
      (print "name      : " name)
      (print "sex       : " sex)
      p))
```

Married men and women are implemented by the means of two wide classes :

```
(wide-class married-man::person
   mate::person)
(wide-class married-woman::person
   maiden-name::string
   mate::person)
```

A married woman's identity is printed by the following method (we suppose an equivalent method definition for married-man) :

```
(define-method (object-display p::married-woman)
  (with-access::married-woman p (name fname sex mate)
     (call-next-method)
     (print "married to: " (person-fname mate) " " (person-name mate))
     p))
```

The form `(call-next-method)` invokes the method that would have been used if no method for *married-woman* was overriding the generic function. Here, the `(call-next-method)` expression will act as a call to the method `object-display` defined over the *person* class. A wedding is celebrated using the `get-married!` function :

```
(define (get-married! woman::person man::person)
   (if (not (and (eq? (person-sex woman) 'female)
                 (eq? (person-sex man) 'male)))
       (error "get-married" "Illegal wedding" (cons woman man))
       (let* ((mname (person-name woman))
              (wife  (widen!::married-woman woman
                        (maiden-name mname)
                        (mate man))))
          (person-name-set! wife (person-name man))
          (widen!::married-man man (mate woman)))))
```

We can check if two persons are married by :

```
(define (married? woman::person man::person)
   (and (married-woman? woman)
        (married-man? man)
        (eq? (married-woman-mate woman) man)
        (eq? (married-man-mate man) woman)))
```

We may now study what happens during a wedding :

```
(define *junior* (instantiate::person
                    (name "Jones")
                    (fname "Junior")
                    (sex 'male)))
(define *pamela* (instantiate::person
                    (name "Smith")
                    (fname "Pamela")
                    (sex 'female)))

(define *old-boy-junior* *junior*)
(define *old-girl-pamela* *pamela*)
(get-married! *pamela* *junior*)
```

We may check the wedding :

```
(married? *pamela* *junior*)
   ⇒ #t
(display-object *pamela*)
   ⇒ name       : Jones
      firstname  : Pamela
      sex        : FEMALE
      married to : Junior Jones
```

Note that both `*pamela*` and `*junior*` are still the same persons :

```
(eq? *old-boy-junior* *junior*)  ⇒ #t
(eq? *old-girl-pamela* *pamela*) ⇒ #t
```

Finally, divorce is implemented as :

```
(define (divorce! woman::person man::person)
   (if (not (couple? woman man))
       (error "divorce!" "Illegal divorce" (cons woman man))
       (let ((mname (married-woman-maiden-name woman)))
          (begin
             (shrink! man)
             (shrink! woman)
             (person-name-set! woman mname))
```

And thus :

```
(divorce! *pamela* *junior*)
(display-object *pamela*)
   → name       : Smith
      firstname  : Pamela
      sex        : FEMALE
(eq? *old-boy-junior* *junior*)  ⇒ #t
(eq? *old-girl-pamela* *pamela*) ⇒ #t
```

9.3.6 Restrictions

We present here several rules that restrict widening and class declarations. We conclude with a description of the features a language must provide to be able to host wide classes.

Widening and plain classes :

An object cannot be widened to a plain class. This is not possible because as described in Section 9.5 accessing fields of wide classes requires different operations than accessing fields of plain classes. We enforce this separation between plain and wide classes for efficiency reasons : we want to be able to generate efficient code for programs which do not use wide classes ; this would not be the case if arbitrary widening along the inheritance tree would be allowed.

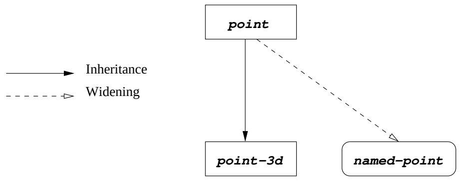
In consequence, wide classes may inherit from plain classes or from wide classes. But plain classes cannot inherit from wide classes. There is no technical reason for that. The only reason is that it is useless to declare a class that inherits from a wide class because no object may be widened to a plain class !

Single subtyping and widening :

As we mentioned in Section 9.2, the object model of the Bigloo's source language is based on single subtyping. That is a class has one single superclass. To be consistent with this choice we have chosen to restrict widening accordingly. Thus, an object can only be widened to a wide class that directly inherits from the current class of the object. In other terms, it is illegal to widen an instance whose actual class is $\mathcal{C}_1$ into an instance of class $\mathcal{C}_2$ if $\mathcal{C}_2$ does not inherit from $\mathcal{C}_1$. Thus

```
(module named-point-3d
   (export
    (class point ...)
    (class point-3d::point ...)
    (wide-class named-point::point ...)))

(let ((p (intantiate::point-3d)))
   (widen!::named-point p ...))
```

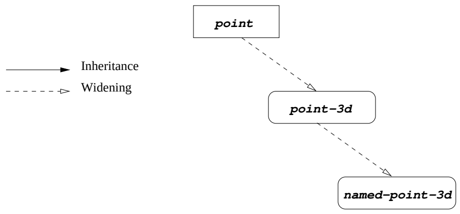
is illegal because *named-point* does not inherit from *point-3d* (the actual class of p). If simultaneous widening were permitted for different branches of inheritance, then we could build a tree like the following one :



This obviously breaks the single subtyping rule. An object is simultaneously a *point-3d* and a *named-point* and neither the rule *point-3d* < *named-point* nor *point-3d* > *named-point* stands. On the other hand :

```
(module named-point-3d
   (export
    (class point ...)
    (wide-class point-3d::point ...)
    (wide-class named-point-3d::point-3d ...)))

(let ((p (intantiate::point)))
   (widen!::point-3d p ...)
   (widen!::named-point-3d p ...))
```

is legal because *point-3d* inherits from *point* (the actual class of p) and because at the time of the widening to *named-point-3d*, p is an instance of *point-3d* and *named-point-3d* inherits from *point-3d*. The restriction that applies to widening is justified to avoid multiple simultaneous widening. A plain object may be widened only once at a time. It builds the following inheritance tree :



which preserves single subtyping property.

Type checking :

Widening may be implemented for any language that supports dynamic type checking. Languages belonging to the Lisp family such as Scheme or Smalltalk fulfill that criteria. For statically typed languages wide classes cannot be used as such. The problem comes from the nature of shrinking. Shrinking, like widening, preserves the sharing structure of programs (that is, it respects `eq?`). So for example, consider the case where `obj` is shrunk from class `B` to `A`. The semantics of wide classes and shrinking guarantees that `B` is a subtype of `A`. The problem is that if any other object in the system keeps a reference to `obj` as an instance of `B`, then there is a risk of accessing fields that have been erased. That situation is very frequent for languages using "`self`" pseudo-variables. Widening is able to change the type of "`self`" ! The only solution is to enforce a dynamic type test when accessing *any* wide object. This solution could be implemented within Eiffel or Java runtime system because they provide enough run-time type informations. Obviously this solution is equivalent to (partially) giving up on static type checking.

Concurrent programming :

Because they operate physical updates, it is mandatory that in presence of multi-threading, widening and shrinking be atomic operations. Otherwise, it might be possible that a thread accesses the field of one wide object before that field is created. Such a situation may arise when two threads share an object and that one of the threads accesses the object while the other thread is widening it.

9.3.7 Possible extensions

It may be useful to propose a `shrink-then-widen!` operator. This would first shrink a wide object to the lowest common ancestor class of the current and desired wide class, then widen. This new form would allow wide classes to be defined as subclasses of an abstract class $\mathcal{C}$ because the `shrink-then-widen!` operator eliminates the need for explicit constructions of instances of the class $\mathcal{C}$.

9.4 Widening in a real context

As we mentioned in section 9.2 the compiler for the Scheme programming language Bigloo has been re-implemented with wide classes. Bigloo tries hard at optimizing the source code and thus contains many different analyses and optimization phases. For instance, Bigloo *inlines* user functions, reduces the number of instructions executed with data flow analyses such as *common sub-expression elimination* and *copy propagation*, minimizes the amount of memory needed to execute the program (*closure analysis* that improves the compilation of first class closures, *control flow analysis* that isolates polymorphic part of the programs that can be compiled has monomorphic programs), etc. All these analyses and optimizations are implemented in separate stages ; in total there are twenty stages, that comes one after the other.

The whole compilation uses one unique global abstract syntax tree (AST henceforth). This data structure contains the minimal common information required by every stage. For instance, during the whole compilation nodes that implement variables are required to have a field holding the name of the variable. The AST changes from one stage to the next. Some nodes are removed while new nodes are introduced (for instance, *dead code elimination* optimization obviously prunes the AST and *loop unrolling* introduces code duplications). Nevertheless, the global structure of the AST barely changes during the compilation. Each compilation stage accepts as input the same data structure.

The only communication channel between stages is the AST. This strict policy makes each stage acts as a stand alone program. This property is very useful for a compiler because it helps the debugging process, it makes possible to change the order in which the stages are applied, and multiple applications of one stage (for instance, *copy propagation* enlarges the set of candidates to *common sub-expression elimination* and *common sub-expression elimination* enlarges the number of candidates to *copy propagation* : thus it is a convenient solution to apply the *copy propagation*, the *common sub-expression elimination* and then to re-apply the *copy propagation*).

Despite the global sharing of the AST some stages need local informations. For instance, the stage that makes the closures allocation (the stage that turns Scheme closures into C functions) needs to access the closure free variables. Once closures are allocated there are no more free variables. Freeness property is thus strictly local to that stage. Another example is the control flow analysis stage. This compilation pass computes static approximations to actual dynamic values. Once these approximations are computed the optimizer searches for polymorphic definitions that are used with only one single type of argument. These definitions may be compiled using an efficient monomorphic framework. Once this optimization is completed there is no further need for approximations. Approximations are then local to the control flow analysis. The common framework for a stage is :

1. Widen the nodes of the abstract syntax tree that are of some use for the stage.
2. Process the computation of the stage on the wide tree.
3. Shrink the nodes of the abstract syntax tree.

This framework reduces memory consumption for the AST because only the informations relevant to one pass are live at a time. Moreover this framework helps the development of the compiler because it strictly enforces the separation between stages. If a piece of information is needed by more than one pass, it has to be held by the global tree making apparent that the information is global.

Former Bigloo versions (prior to our implementation of wide classes) relied on the *user field* strategy. With this strategy, the AST used to be extended with one "free" field acting as a hook for each stage that needed local informations. The new version of Bigloo has eliminated all the *user fields* in favor of wide instances. Widening helps at the design and the implementation of programs that conform to the compiler's framework. The improvement is fivefold :

– The source code is easier to write because we benefit from the object library and object facilities. For instance, with traditional languages access to a value held by a user field looks like :

```
(define (add-approx! funcall ap)
   (let* ((uf (funcall-user-field funcall))
          (approx (cfa-approx uf)))
      (cfa-approx-set! uf (cons ap approx))))
```

  Using widening and the `with-access` construction the same code is turned into :

```
(define (add-approx! funcall ap)
   (with-access::wide-funcall funcall (approx)
      (set! approx (cons ap approx))))
```

  The second version is shorter to write and we think easier to read. It is still possible to write sets of macros or functions that simulate the look and feel of the wide instance but it is in charge of the programmer. Wide classes mechanisms help the programmer with convenient ways to access objects and we think that it is the role of a programming language to provide with such help.

– In addition to accesses, wide instances benefit from the full features of the object-oriented model. Wide classes use inheritance and subtyping effectively : methods may override generic function definition for wide classes.

– The code of the compiler can be easily extended. The extensions may be implemented by the means of supplementary widening. Adding local informations to the AST or adding new compiler stages do not require a change in the whole program because the common AST can be left unchanged.

– Wide classes are safer because they benefit from the type checking of the compiler and the runtime system. As we discussed in the introduction, *user fields* may not be of another type than the most generic type (`void *` for C). In the context of a programming language that supports full polymorphism as Scheme does, this gives no opportunities for the compiler to produce good code by eliminating dynamic type tests and to help the programmer by signaling, at compile time, type errors. Wide classes is of great help in this context. For instance, in t

  The consequence of that extra safety is that within the context of our compiler, wide classes help enforcing strict separation between compiler passes. Pass separation is implemented as type tests and these tests are *automatically* introduced by the compiler.

– Wide instances implementation is more efficient than *user fields*. This is discussed in the following Section 9.5.

9.5 Wide classes implementation

This section describes an implementation strategy for wide classes. It is very close to the implementation of the current Bigloo release.

9.5.1 Instances

Bigloo compiles Scheme modules into C programs. The natural mapping for a Bigloo object is a C data structure. In addition to the fields found in the class declaration, two extra fields are added to the C structure. The first one, `class_num`, holds the runtime type information used for type predicates and generic functions. The second one is used as a `mark` for object marshaling. As an example, the class :

```
(class a-class x::long y::char)
```

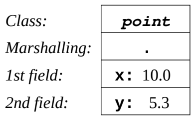
is compiled into :

```
typedef struct a_class_125 {
   class_t  class;  // The class pointer.
   void    *mark;   // The object marshaling mark.
   long     x;      // The 1st field
   char     y;      // The 2nd field
} *a_class_125_t;
```

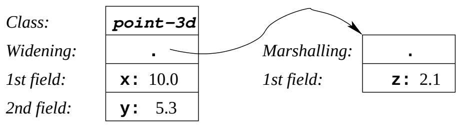
Accessing a field of an instance is thus implemented as accessing a C structure slot.

9.5.2 Widening and shrinking

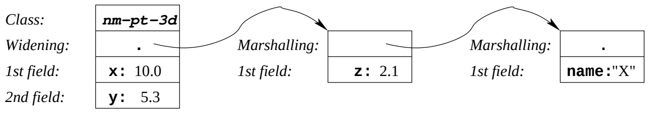
Widening implementation is straightforward : widening means allocating rooms for the new fields of the wide class ; establishing a link between the instance and that new memory chunk (henceforth *wide chunk*) ; changing the header field of the object to reflect the operation. The wide chunk contains one extra field to implement the instance marshaling. The hooking between the instance and wide chunk uses the slot previously devoted to marshaling. Let us consider the memory implementation of an instance of class *point*.



After widening to the *point-3d* class, the instance becomes



Multiple widening repeat that framework and thus the memory layout for the widening of the instance of *point-3d* to the *named-point-3d* wide class is :



With our implementation framework, shrinking is pruning the last *widening* pointer and restoring class pointers. There is no need to store in the wide chunk the former class of the instance because, when shrinking, that class is statically known.

9.5.3 Accessing wide instances

Accessing the field of a plain instance is implemented by accessing the slot of a C structure. Accessing the field of a wide class requires several accesses to several C structures. For instance, the C sequence for fetching the `z` field of an instance `p` of the class *point-3d* is :

```
(point_3d_t)(p->mark)->z;
```

Fetching the `name` field of an instance of class *named-point-3d* is :

```
(named_point_3d_t)((point_3d_t)(p->mark)->mark)->name;
```

The wider the object is, the more expensive is fetching a slot of its wider chunk. Fetching a wide instance slot has the same performances as the *user field* framework. On the other hand, because of the dynamic type checking nature of Scheme, even a smart compiler is not always able to prove at compile time that the access implied in a fetch is of the correct type. In consequence, dynamic type checks have to be executed at runtime. In the case of wide instances, this requires exactly one check : is the object an instance of a given class ? This operation is constant in execution time. With the *user field* approach, this does not apply anymore. The object has to be checked for its class and each accessed user field has to be checked itself. Accessing the `name` field of an instance of the wide class *named-point-3d* costs exactly one test. Implemented without wide classes the same operation would have required 3 tests.

9.6 Related work

9.6.1 Smalltalk `become :` and Clos `change-class`

The `become:` mechanism of Smalltalk-80™ allows the values of two objects to be swapped. The Clos library function `change-class` allows an object of a class $\mathcal{C}_1$ to be reshaped into an instance of any other class $\mathcal{C}_2$. Both Smalltalk's `become:` and Clos' `change-class` are some kind of unrestricted widening+shrinking operations. Using the Smalltalk construction two objects of unrelated classes may be swapped. Using the Clos generic function an object may be reshaped into an instance of any other class. This is not possible with widening. Widening is restricted to subclasses. When an object is widened, its actual type is turned into a subtype of its previous actual type. Such a property does not hold for `become:` or `change-class`.

We believe that preserving subtyping relationships is very important. It helps the system at statically detecting type errors and it makes the semantics of widening more intuitive and simpler.

The other important advantage of widening is that it is easy to implement. Widening may be efficiently implemented because it does not change the memory layout of an object (see Section 9.5). This is not true for `become:` [245] nor `change-class`. The CommonLisp `change-class`, as the widening operator, preserves `eq` equivalence. If this has no consequence for widening it has a consequence for `change-class` where a kind of double indirection is required to access object slots. If an object stays `eq` to what it used to be before a `change-class` and if `change-class` may have totally changed the shape of that object, it means that the object slots are not encoded inside the object but outside the object. This is why a double indirection is needed to access one object field.

Another important property of widening is that a program that does not use widening does not pay any extra cost for that construction. This is not true for `change-class` nor `become:` as they impact the whole run-time design.

9.6.2 Class predicates and instance classification

The programming language Cecil [39] extends traditional object-oriented languages with *Predicate classes* [40]. Wide classes may look like predicate classes because a predicate class is also designed to reify transient states or behavior modes of objects. A predicate class is a class extended with one predicate. An object belongs to that class if the predicate applied to that object is satisfied. The promotion from a class to a predicate class is automatic as soon as the predicate is satisfied. This is the main difference with wide classes. An object is explicitly widen and shrunk in the program.

Traditional languages cannot dispatch on general object property such as "is a window iconified ?". Predicate classes are designed to deal with such situations. Wide classes could be used too because the iconification action is explicitly required and then a widening operation may take place at that time. Wide classes cannot be used when there is no location in the source file associated to the object state modification. It is not easy to model changes in object states on some global property such as "more than 10000 instances".

Predicate classes are not designed to help at reducing software retention which is the main goal of wide classes. "Cecil predicate objects (instances of predicate classes) reverse space for any fields that might be inherited from a predicate object, i.e., those fields inherited by an object assuming all predicate expressions evaluate to true. The value stored in a field of an object persists even when the controlling predicate evaluates to false and the field is inaccessible".

Moreover, it is likely that predicate classes do not enable efficient implementation. Method lookup is augmented with additional evaluations of predicate expressions for methods attached to predicate objects. Type checking might be expensive because it requires user code evaluation and the time complexity of the type checking depends on the number of predicate classes.

The Kea programming language is a functional object-oriented language [100] that proposes *classifiers*. These are similar to Cecil's predicate classes. Both support property-based classification. Being a strict functional language Kea is not designed to help at implementing object which types vary over the execution time. Objects in Kea can't be modified.

9.6.3 Dynamic slots

The paper [196] presents dynamic slots that enable runtime extensions of objects. A dynamic slot is an attribute that exists in all objects, an for which storage is only allocated if the dynamic slot is used, i.e., if a value is assigned to it. Dynamic slots are statically declared in the source code, hence statically typed. The main difference between dynamic slots and wide classes is that dynamic slots are available on all objects, i.e., they are orthogonal to the type system. When an instance is widen, its type changes and thus, its behavior may change. Because dynamic slots are not related to the type system, they cannot be used to change the behavior of an object. Contrarily to widening, defining dynamic slots for an object does not allow that object to react to new methods.

Dynamic slots are statically declared but the way objects can be dynamically extended is left undeclared. Thus, dynamic slots are implemented by the means of linked lists. Fetching the value of a dynamic slot requires a lookup in an association table. It is linear in time. Has it has been presented in Section 9.5, because widening restricts the way objects may evolve over the execution, accessing the slot of a wide object is constant in execution time, i.e., the offsets associated to the wide object slots are statically computed.

Conceptually dynamic slots exist for all object. Thus, it is unclear if dynamic slots can be erased from an object. In consequence, we do not think that dynamic slots can help at solving the software data retention problem.

The *adoption* presented in [177] is close to dynamic slots. In that model inspired from Smalltalk's `become:`, an instance may be *adopted* by another class. The adoption may enlarge the number of fields implemented by an object. Fields are thus held by lists as for dynamic slots. In opposition to dynamic slots, an instance may be adopted several times. Thus it is possible to simulate widening and shrinking with class adoptions.

9.6.4 Object-oriented programming with modes

The paper [238] presents an extension to the class-based object-oriented model in order to support explicit definition of logical states, named *modes*. Modes can be added to class by allowing the definition of multiple sets of operations within class definitions and modifying the late binding algorithm to automatically select between these sets of operations on the basis of the current mode of the current receiver object. In addition, a notion of transition is introduced to enable mode determination to be performed without explicit mode changing statements.

Modes helps at implementing specific behaviors with respect to some object states. For instance, if we consider the example of the window of the introduction : a window has two states, open or iconified. The behavior of a window differs if it is open or iconified. For instance, clicking on an icon opens a window but clicking on an open window simply selects it. Modes proposes an elegant way to implement such situations. Two different implementations for the `click` message will be given in the window `class` : one for each states. The transition function for windows simply records if the window is open or iconified.

The primary goal for widening is not to express states for objects. It is to help the implementation of transient data storage. Modes does not address this problem. Modes can be simulated with wide classes. To change the behavior of an instance `o` of a class $\mathcal{C}$, it is sufficient to add a wide class $\mathcal{C}'$ with no additional slot. Widening `o` from $\mathcal{C}$ to $\mathcal{C}'$ or shrinking from $\mathcal{C}'$ to $\mathcal{C}$ may change the behavior of `o` because methods can be added in $\mathcal{C}'$. However, because widening uses inheritance, it is restricted by subtyping relationships. Modes implement type unions rather that subtypes. Type unions are not easily mapped to inheritance trees, specially with our system that makes use of single inheritance.

9.7 Extensions

9.7.1 Multiple widening

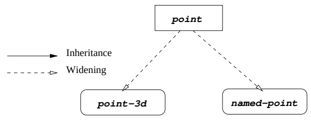
A possible extension to wide classes could be to allow multiple widening of a same object. This could be useful when unrelated extra informations have to be added to an object. To get back to the point example, it could be a useful feature to allow a plain `point` to be widened into a `point-3d` but also to be widened next into a `named-point`. The example below illustrates that possibility :

```
1 :  (module points
2 :     (export (class point ...)
3 :             (wide-class point-3d::point ...)
4 :             (wide-class named-point::point ...)))
```

The same component of an instance may be widened more than once :

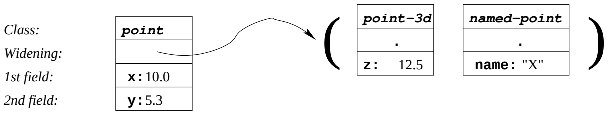
```
1 :  (let ((p (instantiate::point)))
2 :     (widen!::point-3d p)
3 :     (widen!::named-point p))
```

The inheritance tree is :



This is forbidden with the model previously presented because at line *3* `named-point` is not a subtype of the actual type of `p` (i.e. `point-3d`). The multiple widening would require to change the `shrink!` operation. It is now mandatory to decide which component of a wide object must be shrunk.

We have decided not to implement the multiple widening because we have found no way to implement it as efficiently as simple widening. In the presence of multiple widening an object could contain several wide chunks (see Section 9.5). The new memory layout with multiple widening may look like :

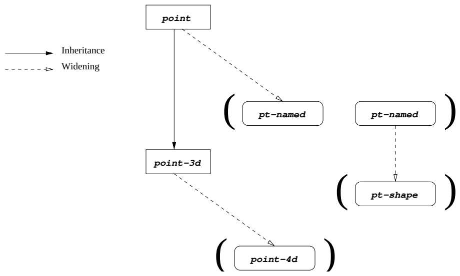


Testing for a typing property (e.g. is the object p an instance of the class $\mathcal{C}$?) is harder with multiple widening because an object may have several simultaneous actual types that are not in a subtyping relationship. It is likely that there is no way to perform type checking without checking in sequence all actual types of a wide object.

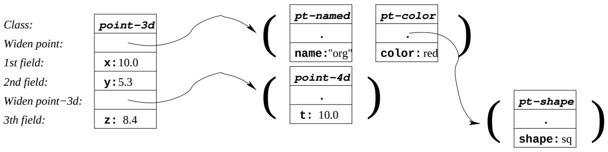
Accessing the slots of a wide object in presence of multiple widening is a more complex operation than with single widening because it is first necessary to fetch the correct wide chunk. The complexity of the worst case of that operation is $\mathcal{O}(c)$ where c is the number of classes.

9.7.2 Multiple widening and multiple inheritance

The second step to widening is to meet the multiple inheritance paradigm. With that programming model it would be coherent to allow a plain object to be widened more than once for each of the class it inherits from. Thus, it would be also possible to widen several times the same component of an instance. That is, it would be possible to first widen the `point` part into, for instance, a `point-named` and then to widen one more time the `point` part into a `point-colored`, etc. An instance must be provided with as many slots for widening as the number of classes it inherits from. For our example, a `point-3d` instance must be provisioned with 2 slots for widening, one for the `point` component, one for the `point-3d` component. This is not enough : a slot for widening must contain the list of wide chunks instead of pointer to a single wide chunk. The inheritance tree would look like :



The data layout for such an object would look like :



With that naive framework accessing the slot of a wide chunk and checking for a wide type is expensive. Most likely the clever representations that are used within implementation for languages that support multiple inheritance would applied here.

Conclusion

Wide classes extend traditional object-oriented programming languages with a construction that models transient properties. Such a construction does not exist in regular languages. Thanks to wide classes and wide inheritance the shape of an object may change during its life-time. An object can be, at a certain point of the execution, *widen* to another class, that is, an object may be extended with additional fields. Later on, this same object can be *shrunk* to its original shape.

We think these features may help the programming task because we think programs using data structures that vary over execution time are frequent.

Wide classes are efficient at solving software retention. This problem arises when an actual data structure is not able to reflect time properties in order to minimize the memory consumption of an execution. We have shown how wide classes apply to the construction of our Scheme compiler and how they succeed in improving the programming style.

Wide classes may also be used to model object states. If an object goes from one state to another this transformation can be implemented by the means of wide classes. Wide classes provide a mechanism to change the type of an object and thus to change the way generic functions will react when applied to it.

Wide classes could probably be easily implemented in other object-oriented programming languages although dynamic type checking or, at least runtime type informations, are likely to be required. We don't think languages that make use of type inference like O'Caml [176] can benefit from wide classes. W [58] based type inference algorithms are not able to detect errors in programs like the one shown below where the last `point-3d-z` access is illegal :

```
(let ((p (instantiate::point)))
   (widen!::point-3d p (z 45))
   (+ (point-3d-z p) (begin (shrink! p) (point-3d-z p))))
```

The reason is that current type inference algorithms of that trend cannot associate several types to a same object. An object has to be either a *point* or a *point-3d* but it can't be at some point in time a *point*, becomes a widen *point-3d* and returns back to a plain *point*. On the other hand, languages that propose coercion operations like Java could probably be extended with wide classes although at the cost of extra dynamic type checks.

Acknowledgments

Many thanks to Christian Queinnec, Jan Vitek and to Céline for their helpful feedbacks on this work.

Chapitre 10

Bee : an Integrated Development Environment for the Scheme Programming Language

Cet article a été publié dans [209].

Bee : un environnement de programmation intégré pour le langage Scheme

Bee est un environnement de programmation intégré pour le langage Scheme. Il fourni à l'utilisateur une connexion entre les langages Scheme et C, un outil de mise au point, un outil d'évaluation de performances, un interprète, un compilateur optimisant qui produit des petits exécutables autonomes, un arpenteur de code source, un gestionnaire de projets, un constructeur de bibliothèques et une documentation en ligne. Cet article détaille les services de Bee, son interface utilisateur ainsi qu'une présentation de l'implantation de ses principales composantes.

Bee : an Integrated Development Environment for the Scheme Programming Language

The Bee is an integrated development environment for the Scheme programming language. It provides the user with a connection between Scheme and the C programming language, a symbolic debugger, a profiler, an interpreter, an optimizing compiler that delivers stand alone executables, a source file browser, a project manager, user libraries and online documentation. This article details the facilities of the Bee, its user interface and presents an overview of the implementation of its main components.

10.1 Introduction

In a recent provocative paper, P. Wadler [250] states that functional languages (henceforth FLs) are more than endangered, they are like dead because nobody uses them anymore! In studying the reasons for this failure he reports that FLs implementations which support the needs of programmers with real problems barely exist. Real programming requires real libraries and real environments that are today lacking for FLs. We strongly agree with these arguments. In the past years implementors of FL have mostly concentrated on efficiency. These studies have been successful because nowadays, compilers for FLs deliver programs that are reasonably fast in comparison to C compiled programs [106, 211, 136]. Unfortunately the good performance of FLs have not been sufficient to enlarge their diffusion. To some extent, programmers don't care about efficiency. This is one of the main lessons we can draw from the success of Java, since until very recently, Java was still slow but yet much more used than FLs. We think that programmers are more concerned with programming facility than with performance. In addition to the programming language itself, we think there are three main needs when programming :

The need for libraries Computing is becoming more and more complex because computers are more and more powerful, allowing them to process more and more tasks. A programming language must enable user interface programming, network programming, operating system programming, etc. These activities require libraries. The key point is that most of these libraries exist but they are not available from FLs. For instance, Motif is a popular graphics library available for the Unix operating system but Motif requires C to be used. Most of the libraries have a C API or, more recently, a Java API. These APIs are not available from FLs. Thus, there are many programs that are easier to write in C or Java. For instance, writing a program that fetches and displays documents from the network is easier to write in Java than in most FLs.

The need for packaging Following the tradition of Lisp, many FL are implemented in closed environments relying exclusively on a *read-eval-print* loop. This kind of implementation has two drawbacks : *(i)* It leads people to think that FLs cannot be compiled, or at least only with difficulty and as thus it leads people to think that FLs are slow. *(ii)* A monolithic *read-eval-print* loop based environment does not comply with the main modern operating systems philosophy. We may classify operating systems into two categories : the Gui class that contains operating systems like MacOS or Microsoft Windows, and the command line interface class that contains systems like Unix. The *read-eval-print* loop approach fits neither of these categories because it acts as another operating system inside an operating system. Useful programs interact with the host operating system (at least for opening and closing files), but the *read-eval-print* loop makes it almost impossible to write programs that meet the spirit of the host operating system. For instance, Unix programs frequently communicate by the means of pipes. This requires small stand alone programs that may be easily and quickly spawned from command line interfaces.

The need for development environments Programming is a complex task made of many small subtasks each requiring a specific tool. Let us suppose a programmer starting a new project under the Unix environment. His first task will be to write a `Makefile` used to compile his program. In order to ease source files browsing it may be convenient to generate a `tag` file (for instance using `etags`) that can be used by a text editor. In order to get a record of the various versions of the program, a revision control system such as `cvs` must be used. When debugging the program he needs special compilation flags and a tool such as `gdb`. If he provides documentation, other tools such as `man`, `texinfo` or a `html` browsers are required. When the program is fixed and it is to be tuned for efficiency, a profiler such as `prof` must be used. This requires other special compilation flags. If the developed software makes use of libraries then library builders such as `ar`+`ranlib` or `ld` must be used. In short, the development of real software requires the use of at least 10 different tools. Each of these uses its own syntax and its own methodology. This heterogeneity makes these tools difficult for non-expert programmers to use. The goal of an integrated environment is to make all the development tools available and easy to use. For instance, writing a `Makefile` is a typical target for a development environment. From a list of source files a `Makefile` may be automatically generated. This `Makefile` will handle re-compilation, `tags` file generation, profiling compilation, etc. Another example is an online documentation system. If the documentation is available from the source editor and if it simply requires a mouse click to be popped up then it saves time for the programmer. To summarize, an integrated environment helps the programmer to manage most of the tedious parts of the programming activity.

For a period of fifteen years very little research has been concerned with development environments for FLs. Very few implementations for FLs have provided development environments. The logical consequence is that programming in Lisp at the beginning of the 80's was far easier than using FLs today ! Nowadays, we have extraordinarily powerful workstations but we use development environments that are much less attractive and much less powerful than the Lisp machines [148, 237] used to be ! Even worse, development environments for functional languages are weaker than development environments for C !

10.1.1 The Bee, an Integrated Development Environment for Scheme

Experienced programmers have a lot of habits that constitute the basis of their experience. They know the methodology to be used when facing a specific problem. If they don't like FLs, it could be because the tools they are used to are not available for FLs or because the methodology they are used to does not fit well with FLs. We think there is a need to design and implement a development environment that will feel familiar to programmers. This has been the goal of the design of the Bee, an integrated development environment for the Scheme programming language [120].

Designing and implementing a development environment is a long process requiring a development capacity that is beyond most academic development teams. The development of the Bee has been fast because our implementation strategy was based on reusing existing tools. In order to minimize the implementation effort, we have decided to concentrate on one operating system. The Bee has been designed to fit harmoniously the Unix style of programming which fosters separate compilation, enables user libraries and delivers small stand-alone applications. The development model follows the traditional *edit/compile/debug* cycle. Eventually, when programs are tuned for efficiency, a profiler reports on execution runtime statistics.

The Bee is an open environment. It is open to the C programming language. That is, it is easy to connect Scheme and C code inside Bee which tries hard to minimize the differences between the execution model of Scheme and the execution model of C. For instance, there is no extra cost associated with calling a C function from Scheme and vice-versa. We may also mention that the Bee's memory manager is compatible with the lack of memory management for C. As a consequence, Scheme objects may be used within C functions and C objects may be used within Scheme functions. With Bee, C APIs are available from Scheme code.

10.1.2 Overview of the article

The first Section is an overview of Bee user interface which focuses its presentation on Bee's look and feel.

Prior to creating the development environment we implemented a compiler for Scheme. This compiler, named Bigloo, compiles Scheme code to C code. Because we wanted the connection between Scheme and C as efficient as possible, we have designed Bee to produce C code that conforms C's standard programming [31]. The direct consequence is that compiled Scheme code is strictly compatible with C. Because of this strong compatibility, the regular tools for C may be applied to Scheme code once it has been compiled into C. This has been the major idea for our implementation of the Bee. Instead of developing a brand new environment from scratch, we have been working at adapting C tools to Scheme. Section 10.3 presents the resulting C code compiled from a Bigloo Scheme module. It shows how Scheme constructions are mapped into C. This section is a requisite to the Sections 10.4, 10.5 and 10.6. Section 10.4 presents how Scheme

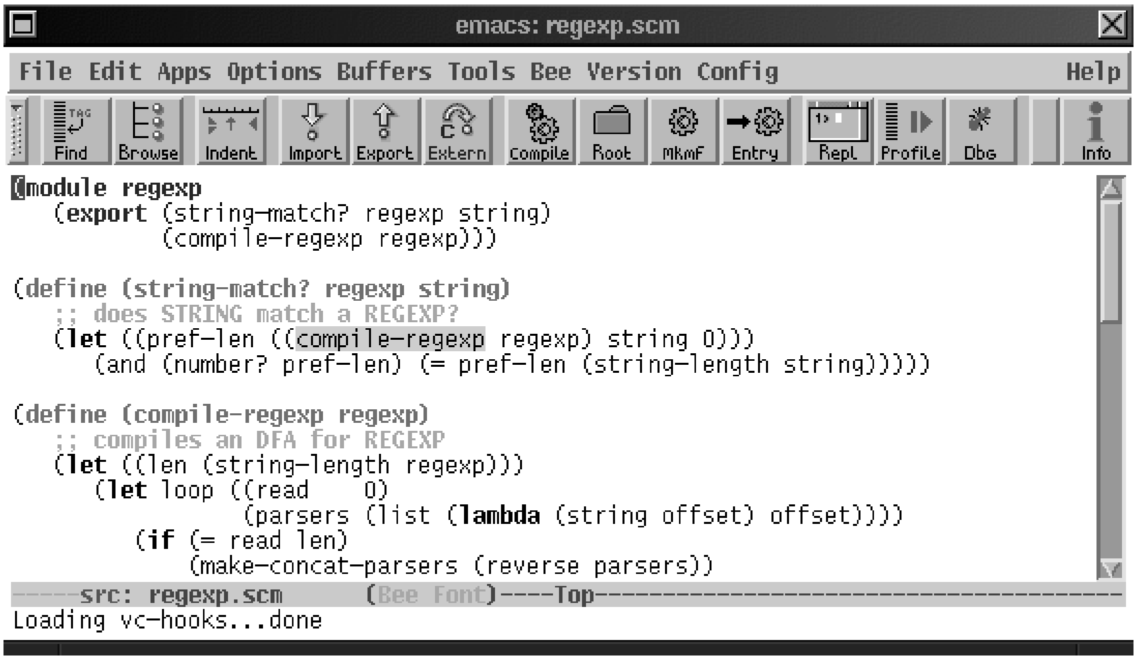


FIG. 10.1 – A plain IDE window

code and C code can be mixed. Sections 10.5 and 10.6 present how C profilers and C debuggers may be adapted to Scheme. They present how the Scheme compiler provides applications with services that are to be used by the Bee. Finally, Section 10.7 presents some related work.

10.2 The Bee user interface

The central tool of an integrated development environment (henceforth IDE) is the text editor. Rather than embedding the editor into the IDE we have chosen to embed the IDE inside the text editor. This avoids wasting time in implementing a text editor that mimics the features of an already existing editor. Because Unix is the initial target system of our IDE, the choice of Emacs as editor was obvious. We have chosen the Xemacs variant (`http ://www.xemacs.org`) because it provides more graphics facilities than other Emacs implementations. Figure 10.1 is a snapshot of a plain Bee editing session.

10.2.1 Bigloo Modules and the Bee Project Manager

Bigloo extends the Scheme programming language with many constructions. In particular Bigloo extends Scheme with modules which have an important impact to the programming style. Bigloo modules have two basic roles : to allow separate compilation and to increase the number of errors that can be detected by the compiler. For instance, because of modules, the compiler may complain about undeclared but referenced variables. Bigloo Modules are simple and easy to implement. They are represented by one or more files and have the following syntax :

(**module** *module-name*
   (**main** *main-ident*)$^?$
   (**import** *import*$^+$)*
   (**export** *export*$^+$)*
   (**static** *static*$^+$)*)

optional-body

The superscript $^?$ means zero or one occurrence, * means a possibly empty repetition and $^+$ means a non empty repetition.

Main clauses specify the name of the function that will be the starting point of the application.

Import clauses are used to import bindings into the module. In order to import, one just needs to state the identifier to be imported and its module name.

Export and *static* clauses play a dual role. They point out to the compiler that the module implements some bindings and distinguish those that can be used within other modules (they are exported) and those that cannot (they are static). These clauses do not contain simple identifiers, as in import clauses, but information about the exported bindings. It is then possible to export variables (mutable bindings) or functions (read-only bindings). Static clauses are optional (the bindings of a module which are not referenced in a clause are, by default, static).

Let us look at the example of the `fib` module that imports the functions `fib-fx` of module `fib-fixnum` and `fib-fl` of module `fib-flonum` and that exports the function `fib`. The `fixnum?` function is implicitly imported, as are all primitives.

```
(module fib
   (import (fib-fx fib-fixnum)
           (fib-fl fib-flonum))
   (export (fib x)))

(define (fib x)
   (if (fixnum? x) (fib-fx x) (fib-fl x)))
```

During compilation, the compiler performs a number of verifications :

- All referenced variables and functions must have been defined in the module, mentioned in an import clause or be primitive, i.e., there are no free variables.
- All the identifiers mentioned in static or export clauses must have been defined in the module with a value conforming to their prototype.
- Identifiers mentioned in static or export clauses whose prototypes are functions must not be `set!`'d.
- All function calls where the functional operator is known to be a function (i.e., the identifier in a functional position has a function prototype) must have an argument number that conforms to the prototype of the invoked function.

Expressions within the body of a module can be variable or function definitions (with the Scheme form `define`) or any Scheme expression. *Initializing* a module is an operation that binds variables to values and evaluates expressions that are not definitions. Definitions and expressions are evaluated in the order of their appearance in the module. Importation graphs of modules are not restricted to being trees or even acyclic since any module can import bindings from any other. The module initialization order is unspecified. For acyclic graphs, modules are initialized during a depth first traversal starting from the module containing the entry point of the application.

This design for the module system leads to a straightforward implementation. In particular, imported modules do not need to be compiled in order to compile the importing module. Bigloo only uses module clauses to know the exported bindings. As a consequence, Bigloo only needs that the source file of the module exists and contains at least its module declaration.

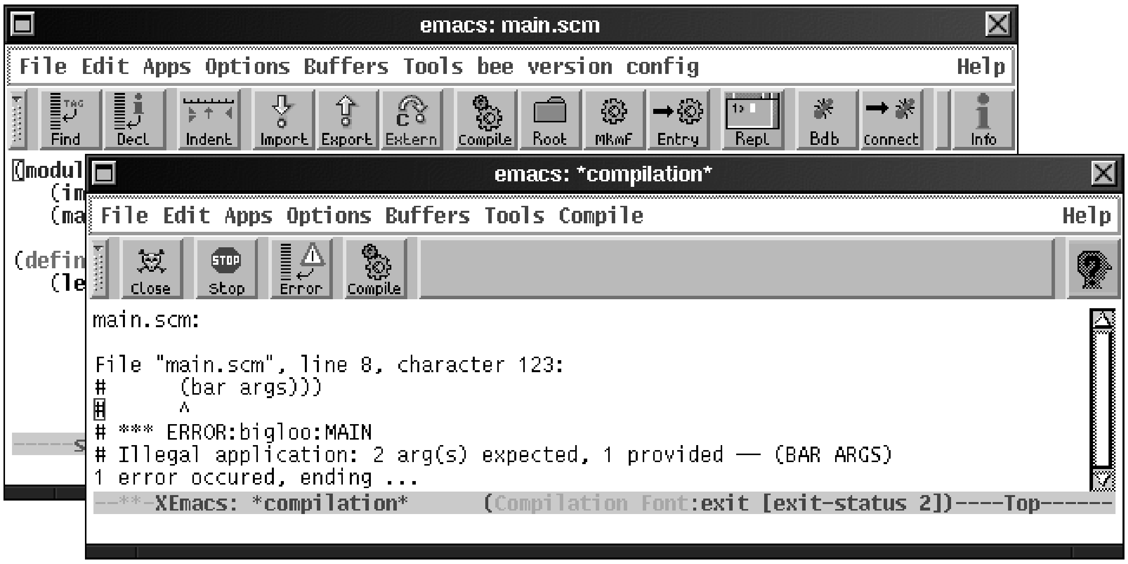
The goal of the Project Manager is to help with both compilation and browsing of modules. Prior to any other operation a project must be registered. Provided with Bigloo modules, clicking the $\boxed{\text{Mkmf}}$ icon (see Figure 10.1) registers a project by creating three extra files. It creates a file that associates Bigloo modules to Unix files. It creates a tag file that is used to retrieve Scheme definitions. Finally, it creates a `Makefile` file that suits the Unix `make` tool. The construction of these three files explores the file hierarchy in order to find out in which files Bigloo modules are implemented. Once a project is registered, all of Bee's features are enabled. Let us consider a very simple Bee session consisting of a Bigloo application made of two modules, `main-module` :

```
(module main-module
   (import bar-module)
   (main   main))

(define (main argv)
   (let ((name (car argv))
         (args (cdr argv)))
      (bar args)))
```

and `bar-module` :

```
(module bar-module
   (export (bar x y)))

(define (bar x y)
   '...)
```

The project is compiled by clicking the Compile icon. Because our example source code is erroneous, this will raise a new window displaying the compilation error status :



Scheme errors are reported in the source code. The compilation status window may be used to pop up a window displaying the culprit line of source code. This feature is common for C environments but unusual for Scheme environments. It requires a compiler that is able to find out source locations from its compilation abstract syntax tree. Section 10.6.3 presents an overview of the implementation of that feature.

Because the project manager automatically handles the association between module name and source file name, there is no need to explicitly specify in which file a module is implemented. For instance, `main-module` imports `bar-module` but the name of the file implementing `bar-module` is not specified within the `main-module` import clause.

Libraries

Bigloo compiles modules into object files and links these files together to build stand-alone executables. The Bigloo object file format conforms with the object file format advocated by the platform hosting Bigloo. The benefit of using regular object files is that it is easy to use the tools of the host operating system to build both static and shared libraries. With Bigloo, users may create their own libraries built of Scheme object files.

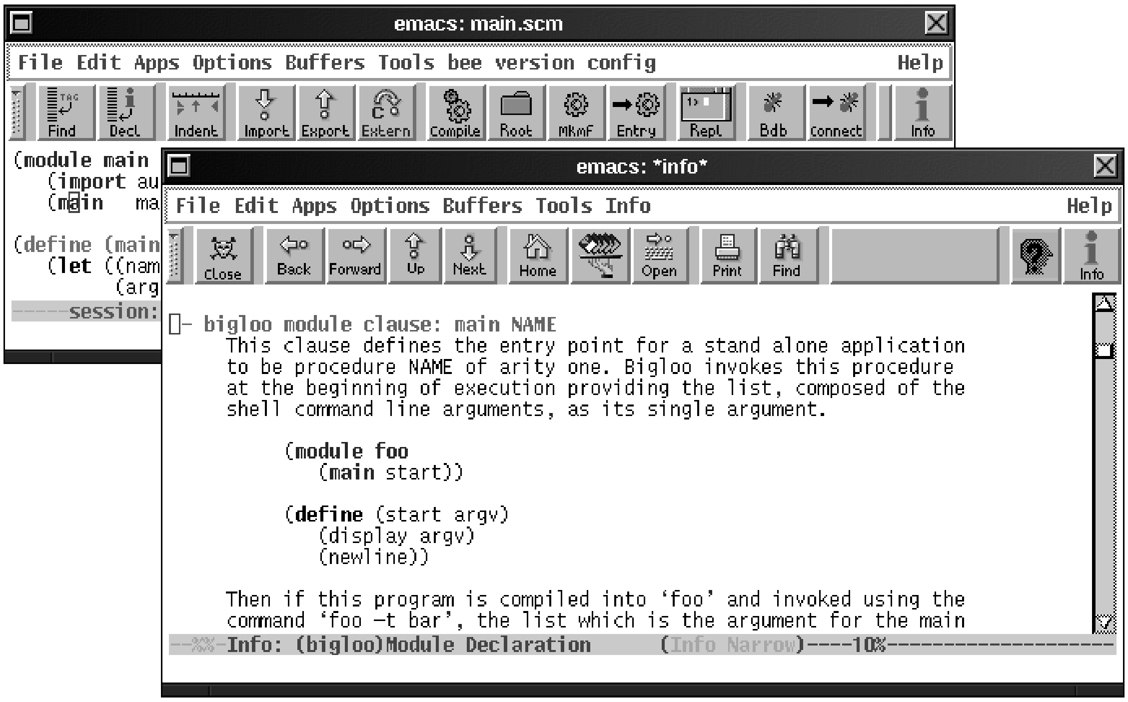


FIG. 10.2 – The online documentation

Separate compilation

Bee does not recompile an entire project when only a small part of the project changes. To see how this works, let us re-examine the example of Section 10.2.1 and suppose that `bar-module` has been fixed. Re-compiling the project will spawn the compilation of `bar-module` and, possibly, the compilation of `main-module`. If we suppose that only the body of the `bar-module` module has been modified, then re-building will just require the compilation of `bar-module`. On the other hand, if we suppose that the prototype of one of the functions exported by `bar-module` has been modified, the new compilation will process `bar-module` *and* `main-module`. That is : module coherence is maintained by the project manager.

The technical solution for that coherence is mostly implemented in the generated `Makefile`. A Bigloo module *depends* on the *module checksum object* files, one for each module it imports. The *module checksum object* is a file associated to a Bigloo module that contains a unique number computed from the module clauses. When compiling a module `mod`, Bigloo computes its *module number* $\mathcal{M}$. If $\mathcal{M}$ is different than the number contained in the `mod`'s *module checksum object* `mod.mco`, then a new `mod.mco` file is generated. All modules that import `mod` depend on `mod.mco`. They are recompiled when `mod.mco` changes and `mod.mco` changes only when the clauses of its associated module `mod` change. Even if unlikely two different sets of module clauses may have the same checksum number. This may cause troubles for compiled applications. If modifying the export clauses left the module checksum number invariant, the project manager will fail at ensuring the coherency between modules. An application made of this module could then fail at run time. Using the whole set of import and export clauses instead of a number to represent modules resolves this problem. However, we have decided to use checksum numbers for the sake of efficiency (it is faster to compare two numbers than two complex data structures) and because the probability that a modification to the module clauses left the checksum invariant is extremely low.

Source file browsing

Once we have a registered project, source file browsing is enabled. There are two ways of browsing : using modules or definitions. The modules browser lists all of the modules and provides access to the modules for editing. To use definitions browsing, the user simply puts the editor cursor on an identifier and clicks on the Find icon. The definition browsing is context sensitive. That is, if the cursor is located inside a module clause, then the definition source browser will look for a module definition. If the cursor is located on a type identifier (a Bigloo type identifier is a specific token that looks like `::`*ident*), the source definition browser will look for a type or class definition. In all other situations, the browser will look for a variable or function definition. The definition browsing facility is well known by developers using Emacs ; the key point here is that the generation of the tags file that enables that browsing has been automated. It does not require any user operation.

10.2.2 Online documentation

Because online documentation is much more effectively indexed, it is far superior to printed documentation. Just like source browsing, Bee's online documentation is context sensitive. When the cursor is located on a source identifier, clicking the Info icon pops up a new window displaying the documentation for the current identifier. Online documentation may be requested while the cursor is above other kinds of tokens. For instance, when the cursor is on a number and the user clicks on Info the general documentation for numbers will be displayed. Figure 10.2 contains an example of window displaying the *main* module clause documentation.

10.2.3 Foreign interface

Scheme is a small language. The most up to date version of its documentation is merely 50 pages long [120]. Being so concise is advantageous in some situations such as teaching and formal reasoning but it has the obvious drawback that it cannot describe much of libraries. Actually, Scheme has very few libraries. Bigloo extends Scheme in many respects. For instance, Bigloo embeds a lexical grammar compiler, an algebraic grammar compiler, a pattern matching compiler, an object layer, a library for Unix processes and Unix sockets, and so on. Advanced system programming within Bigloo is implemented by means of the foreign interface. Section 10.4 presents in great detail the Bigloo foreign interface. The current section focuses on how the user connect's this program across the foreign interface.

Bee's Cigloo tool extracts function prototypes, variable prototypes, compound type declarations, and, to some extent, prototypes of Cpp macros from C files. Cigloo has some limitations due to C source file coding. Some Cpp macros are difficult because they may break the C syntax (Cpp macros not necessarily complies the C syntax), making a parser based tool such as Cigloo inefficient. Also, Cpp macros are not typed. All C definitions are monomorphically typed but Cpp macros may be polymorphic ! Bigloo needs types for any foreign declaration and thus, it needs types for Cpp macros. Cigloo attempts to find out types for Cpp macros. When it fails, it assumes the macro arguments type to be C integers. Of course in many situations this assumption is erroneous and a Cigloo user will be forced to fix the Cigloo produced prototypes for Cpp macros. Cigloo can be invoked clicking the Extern icon. This will pop up a file browser that will let the user select the C file to be imported. For instance, let's examine the module `hash` :

```
(module hash
   (export (hash x)))

(define (hash x)
   (cond
      ((symbol? x)
       (hashstring (symbol->string x)))
      ((string? x)
       (hashstring x))
      (else
       ...)))
```

Let's suppose the `hashstring` function to be a C function implemented inside the file `hash.c` :

```
long hashstring( char *string ) {
   char c;
   unsigned long result = 0;

   while( c = *string++ )
      result += (result << 3) + (long)c;

   return result;
}
```

As it will presented Section 10.4 our Scheme compiler is able to handle C values (such as C `char *`) by automatically converting them into corresponding Scheme objects. Then, clicking the Extern icon and selecting the file `hash.c` will update `hash.scm` as follows :

```
(module hash
   (extern
   ;; beginning of hash.c
    (hashstring::long (::char*) "hashstring")
    (type char*->long "long $(char *)")
   ;; end of hash.c
    )
   (export (hash x)))

(define (hash x)
   (cond
      ((symbol? x)
       (hashstring (symbol->string x)))
      ((string? x)
       (hashstring x))
      (else
       ...)))
```

Automatically, the extern import clause adds the file into the entry of the `Makefile` so that an object file will be produced for `hash.c`. To illustrate the limitation of Cigloo let's study its processing of the C file `point.h` :

```
typedef struct pt {
   double x, y;
} *point;

#define PT_EGAL( pt1, pt2 ) \
   (((pt1)->x == (pt2)->x) && (((pt1)->y == (pt2)->y)))
```

`point.h` contains a type definition and a macro definition. Cigloo produces :

```
 (extern
   ;; beginning of src/session/point.h
File "point.h", line 5, character 48:
##define PT_EGAL( pt1, pt2 ) \
#^
# *** WARNING:bigloo:define:
Unknown type expression -- Using 'int' type
   (macro PT_EGAL::int (::int ::int) "PT_EGAL")
   (type s-pt (struct (x::double "x") (y::double "y")) "struct pt")
   (type point s-pt* "point")
  ;; end of src/session/point.h
   )
```

As one may see, the macro `PT_EGAL` has not been correctly handled by Cigloo because it got confused with the argument types. As a consequence, the return type for the `PT_EGAL` macro will have to be fixed manually. By contrast, Cigloo correctly handles the type declaration for `point`. Declaring a C structure will cause Bigloo to create some accessors, some mutators, a predicate and a creator for that type. Bigloo programs will thus get the possibility to even allocate `point` C objects. When manipulated from Scheme, a C structure is referenced to via a handle containing a runtime type information and a pointer to the plain C structure.

10.2.4 Interpreting

A Scheme implementation must provide an evaluation mechanism which can be implemented by means of an interpreter or by online compilation coupled with objects dynamic loading. This feature is mandatory because the language in which macros are written is Scheme itself! This creates the need for an evaluation stage at compile time (or more precisely at macro expansion time). For Bigloo, we have chosen the first solution : the Bee contains an integrated interpreter. That interpreter is spawned by clicking the Repl icon. Expressions should be sent from the editor to the interpreter in various ways. One should keep in mind that the Bee interpreter is just a tool aiming at testing functions or writing macros. Large development projects should not rely on the interpreter which is slow because it is not tuned for performance.

We have tried to make the semantic of interpreted programs as close as possible to the semantic of compiled programs. The interpreter is able to call compiled code and vice versa. This enables the standard compiled Scheme library to be shared between compiled and interpreted code. However we have failed to get rid of all the differences between the interpreter and the compiler. For instance, the evaluation order of the arguments of a function call is left unspecified in Bigloo. Thus, it is unlikely that the arguments are evaluated in the same order if the call is interpreted or compiled. Apart from these small differences the interpreter resembles the compiler.

10.2.5 Profiling

During the tuning for performance, Bigloo programs may be profiled. This requires very little input from the user. The profile popup menu (within the Bee menu) has two entries, *compile for profile* and *run for profile*. Selecting the *run for profile* entry first asks the user for command line parameters, then spawns an execution, analyses the run sample, and pops up a new window displaying the profiling statistics. For instance, let's show the profile for :

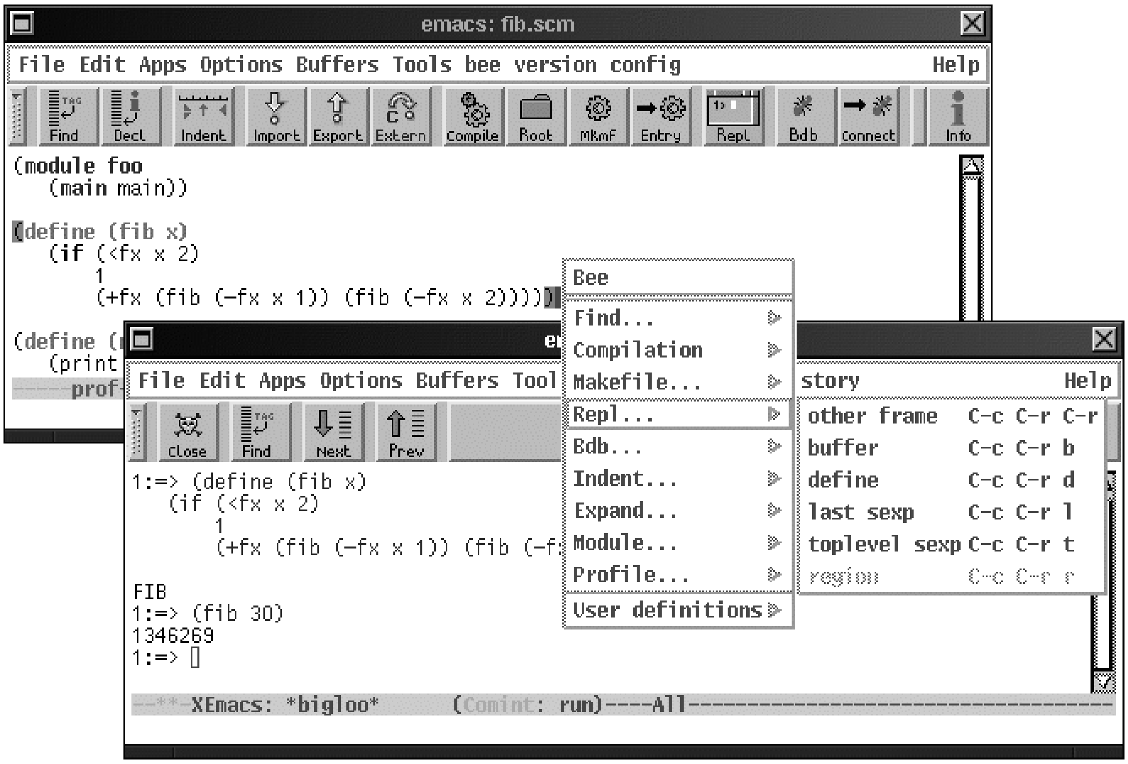


FIG. 10.3 – An interpreter session

```
(module foo
   (main main))

(define (fib x)
   (if (<fx x 2)
       1
       (+fx (fib (-fx x 1)) (fib (-fx x 2)))))

(define (main argv)
   (print (fib 30)))
```

Profiling this program pops up a window as shown in Figure 10.4. This tells us that `100.0%` of the execution time (`0.72` seconds) is spent in the `FIB` function and the functions its calls (actually `FIB` only calls itself). `FIB` has been called `1292536` times and amongst these calls only one has been operated from the `MAIN` function. Section 10.5 details how profiles are computed. Let's just mention that because Bigloo allows mixing of Scheme and C functions, the profiler displays both kinds of functions. They are presented in the profile results using different colors. One user option enables the Bee to hide every non Scheme function. That way only Scheme identifiers are presented to the user.

10.2.6 Debugging

The last tool we are to present is the Bigloo debugger, BDB which is invoked by clicking the Bdb icon. This pops up a debugging window. Various icons allow stepping, continuing, displaying local variables, function arguments, execution stack. In order to access BDB facilities a buffer must be connected to BDB. A buffer can be connected by clicking on its Connect icon. When the execution stops, the (possibly new) buffer displaying the source line is automatically connected.

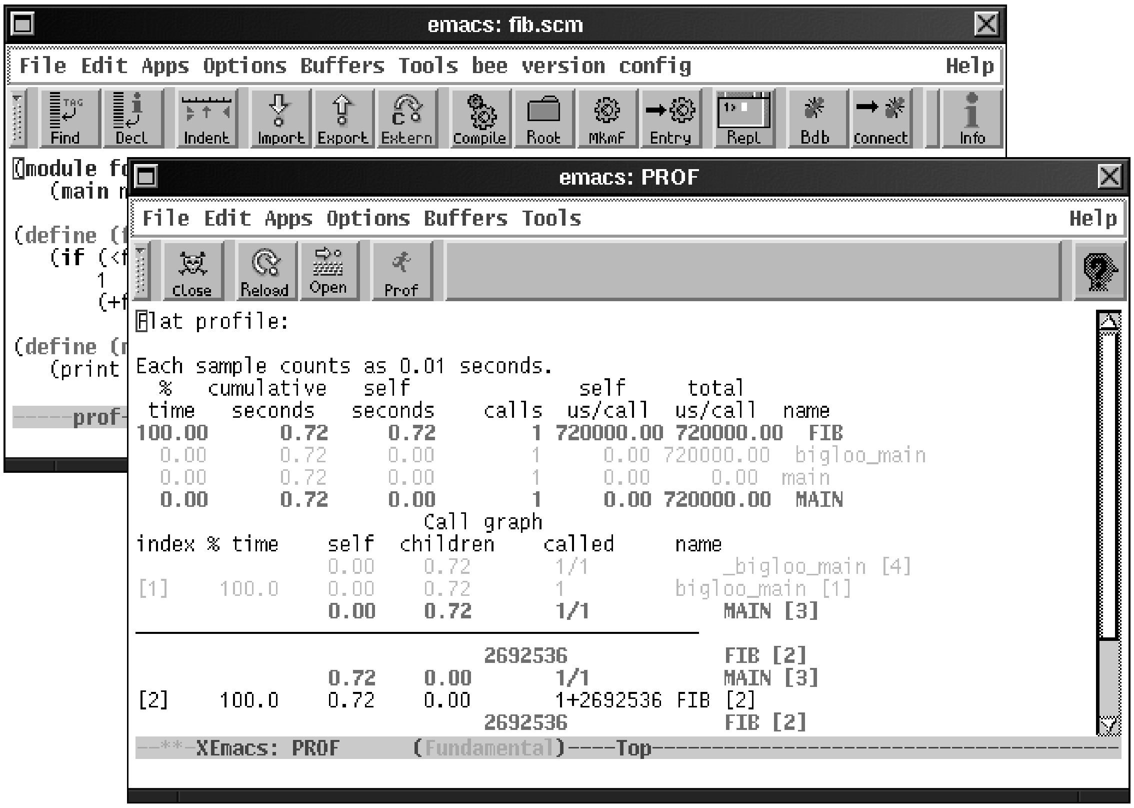


FIG. 10.4 – An excerpt of profiling

When a buffer is connected it displays a left margin. That margin is used to set, remove or change the state of breakpoints. Icons in the left margin show the locations of breakpoints and the execution line is highlighted in the source text by the means of an arrow in the left margin. Figure 10.5 contains a snapshot of a debugging session where two breakpoints have been set : one disabled in the `main` function, one enabled in `fib`. On the snapshot, the debugger window has been split into three. The first one is the command window. Users may submit command line operations from that window. The second one displays the arguments of the current function and the third window displays the execution stack. Clicking on the various frame lines opens new windows displaying the source line corresponding to the clicked activation record. More BDB commands will be presented in Section 10.6.

10.3 The C code production

The purpose of this paper is not to present the compilation techniques used to get efficient C code production. They have already been presented in previous work [213]. But, we must present the framework we use to compile Scheme programs into C programs because this has a tremendous impact on the design and implementation of all the tools that are included in the programming environment.

10.3.1 What kind of C code to generate ?

Because C is mostly portable, choosing to generate C code gives us a mostly portable Scheme compiler for free. There are many variations on the sort of C code which can be produced but two main directions have emerged :

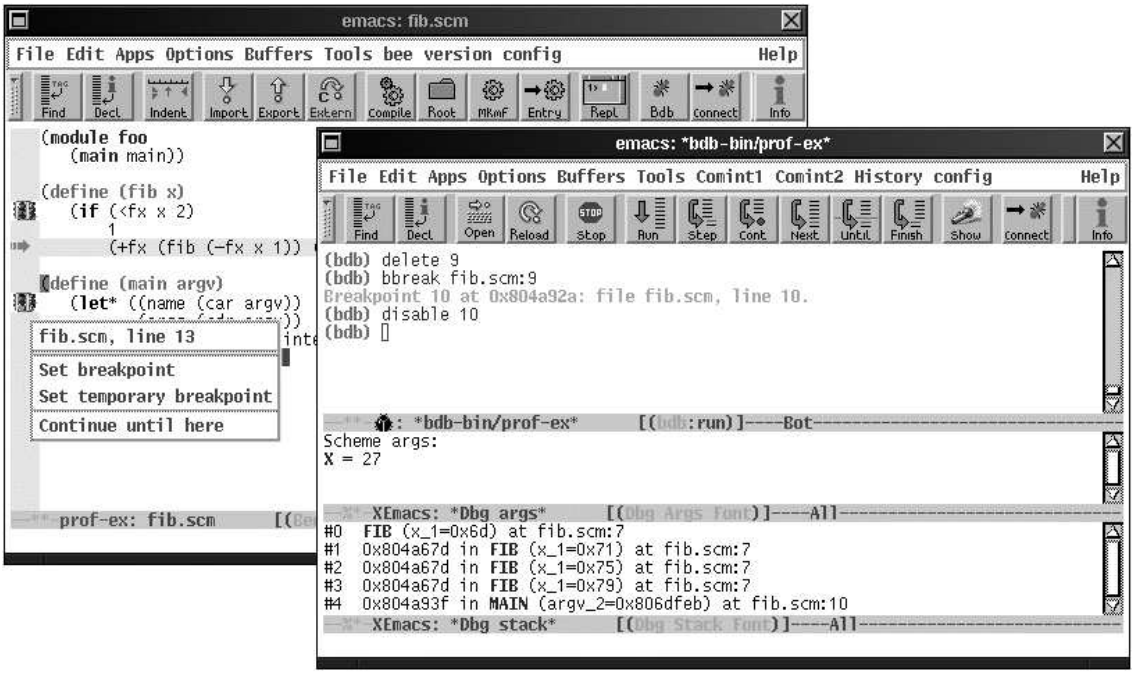


FIG. 10.5 – A debugging session

1. the generation of C code that mimics a virtual machine ; in this case, the C compiler is considered as a virtual assembly language [239].
2. the generation of C code that resembles handwritten C code.

Each method has its own advantages and its own drawbacks but we claim that an optimizing compiler must generate "natural" C code (direction 2).

C as a virtual assembly language

If we use C as an assembly language, we abandon the usage of C control structures. In this case, the C code generated by the compiler has no C functions, the C stack is hardly ever used, and C variables just serve to maintain the registers of some abstract machine. That way, source language features that need some knowledge about the runtime behavior of programs are easier to compile (e.g., garbage collection or the Scheme `call/cc` function). Unfortunately : since the C code generated by the compiler bears no resemblance to the one written by C programmers, it is likely that C compilers will fail to optimize this code properly. The numerous optimizations performed by C compilers will probably not apply and the functional language compiler will not benefit from a high quality C compiler.

"Handwritten-like" C code

The alternate C code generation method mimics handwritten C code. By contrast with the preceding approach, the source language features that need to know the runtime behavior of programs are difficult to implement (and even more difficult to implement efficiently). The `call/cc` function is now hard to implement, and memory management is constrained by the presence of C values in the runtime space of the program (for instance C values stored into the stack). Furthermore, garbage collection must deal with ambiguous roots. On the other hand, the compilation of our generated C code is better, resulting in good, homogeneous performance that is not tied to some particular machine architecture. A previous study [213] has reported that this technique, in general, delivers better performance than the virtual assembly one. As an added benefit of this "standard C code" generation, we can run the C programming environment tools on the generated C code : symbolic debuggers, code analyzers (such as `purify`) and profilers.

With this method of C code generation, every source language construct is compiled into an equivalent C construct if one exists. For instance, source language functions are compiled into C functions (or even into C loops when the source language functions are tail-call loops), source language variables are compiled into C variables, and so on. This mapping is an example of what we call the "natural mapping" from one language to the other. Bigloo uses this technology and projects Scheme terms to handwritten-like C code. To manage C values in the stack, Bigloo uses the Boehm's garbage collector [27, 26].

10.3.2 Compiling Scheme functions

The main design effort for Bigloo has been to compile functions as well as possible : in our mind this implies mapping Scheme functions to C functions (or even to C loops) and working hard to avoid heap-allocation of closures. The result is that global Scheme functions are compiled into global C functions and local functions that play the role of loops are compiled into C loops such as in :

```
(define (find-char str char)
   (let ((len (string-length str)))
      (let loop ((i 0))
         (cond
           ((= i len)
            -1)
           ((char=? (string-ref str i) char)
            i)
           (else
            (loop (+ i 1)))))))
```

The resulting C code is :

```
find_char_85_foo( obj str_1, obj char_2 ) {
  long len_10 = STRING_LENGTH( str_1 );
  long i_12 = 0;
  long aux_35;

loop_11:
  if ( i_12 == len_10 )
    aux_35 = ((long) -1);
  else {
    if ( STRING_REF( str_1, i_12 )==CCHAR( char_2 ) )
      aux_35 = i_12;
    else {
      i_12 = (i_12 + 1);
      goto loop_11;
    }
  }
  return BINT( aux_35 );
}
```

Some points have to be outlined :

- The scheme global function `find-char` is compiled into a C global function, `find_char_85_foo`.
- The scheme local function `loop` is compiled into a C loop by the means of a `goto`. Our Scheme compiler does not make any effort to implement the loop by the means of a C `while` construction. There is no need for this because, from optimizing C compilers point of view, `while` constructions or backward `goto` constructions are equivalent.
- Scheme arguments and Scheme local variables (e.g., `str` and `len`) are compiled into C arguments and C local variables.

– arithmetic is locally unboxed and untagged (the C variables `len_10` and `i_12` are of type `long`). That is, locally, numbers are implemented as immediate values that do not contain type information. In particular, no memory allocation is required to hold such numbers. The analysis that enables such code generation has been published in a previous paper [211].

Obviously, such compilation is enabled only when all recursive calls to the local function are tail recursive. If this condition is not met, a C global function is generated. That is, the Scheme code :

```
(define (copy-list-until l val)
   (let loop ((l l))
      (if (or (null? l) (eq? (car l) val))
          '()
          (cons (car l) (loop (cdr l))))))
```

Is compiled into :

```
obj copy_list_until( obj l_1 , obj val_2 ) {
   return copy_list_until_loop( val_2, l_1 );
}

obj copy_list_until_loop( obj val_22, obj l_3 ) {
   if( NULLP( l_3 ) || (CAR( l_3 ) == val_22) )
      return BNIL;
   else {
      obj arg1003_6, arg1004_7;
      arg1003_6 = CAR( l_3 );
      arg1004_7 = copy_list_until_loop( val_22, CDR( l_3 ) );
      return MAKE_PAIR( arg1003_6, arg1004_7 );
   }
}
```

When a function is used as a value, a closure must be allocated. For instance, the composition function `o` builds new closures :

```
(define (o f g)
   (let ((new-f (lambda (x) (f (g x)))))
      new-f))
```

The difficulties come from the fact that `new-f` is returned as a value (hence it cannot be allocated on the C stack) and that `new-f` closes over two free variables (`f` and `g`). Two functions are generated. One for the Scheme `o` global function :

```
obj o( obj f_1, obj g_2 ) {
  obj new_f_12;

  new_f_12 = make_closure( lambda_1, f_1, g_2 );
  return new_f_12;
}
```

And a second implementing the anonymous function :

```
obj lambda_1( obj clo_1, obj x_2 ) {
   obj f_12, g_23, aux_45;

   f_12 = PROCEDURE_REF( clo_1, 0 );
   g_23 = PROCEDURE_REF( clo_2, 1 );

   aux_45 = PROCEDURE_ENTRY( g_23 )( g_23, x_2 );

   return PROCEDURE_ENTRY( f_12 )( f_12, aux_45 );
}
```

Bigloo's closures are arrays. Each closure contains the free variables of the function, pointer to C functions, and an integer to record the function arity. This arity slot is mandatory for dynamically typed languages to ensure the soundness of function application.

10.3.3 Compiling Scheme variables

Scheme has one unique value space. That is, functions are values in the exact same way as integers are values. Scheme has one unique construction to bind global values. Binding a function to a variable or binding an integer to a variable is done using the same construction. However, in order to get efficient code, functions have to be distinguished from other values. Bigloo makes a distinction between global definitions which bind constants holding functions and those which bind values that may not be functions. Variables holding constant functions that are never changed (i.e., variables declared constant in the module clauses or non exported and never used as first argument of a `set!` form) are said to be functions and are compiled into C global functions. These functions require no runtime initialization and thus, forward references to these functions are safe. Other Scheme variables are compiled into C global variables. As previously seen, Bigloo succeeds in unboxing and untagging values for local variables. Unfortunately this is not possible for global variables. Because module import clauses may contain cycles (see Section 10.2.1) the value of an imported variable might be used before the module defining that variable is initialized. In such a situation the imported variable contains a special value denoting the *uninitialized* state. Any expression referring to that variable must then dynamically check for the type of that variable's value. Dynamic type checks only apply to boxed or tagged values. Thus global variables must contain boxed or tagged values. The consequence in the produced code is that Scheme variables are C global variables of a special type, called `obj`, that stands for all Scheme values.

10.3.4 Limits to the portability of the natural mapping

Scheme has its own customs and traditions (one may even think of them as local folklore) that make the natural mapping to C difficult to implement or even inappropriate for some Scheme constructions. Let us detail some of those.

Scheme tail recursion

As stated by the report defining the Scheme programming language [120] a compliant Scheme implementation is required to be *properly tail-recursive*. A formal definition of proper tail recursion can be found in a paper by Clinger [48]. Obviously this requirement cannot be fulfilled with the natural mapping because C does not provide any mechanism to implement tail recursion. One might consider that limitation too high a price to pay. Solutions to implement tail recursion in C without stack consumption exist but they cannot be used with the natural mapping [239, 18, 72]. We consider the gains of the natural mapping more important. In addition, since local functions are compiled into C loops, which do not allocate stack activation frames, many of tail recursive functions are correctly handled by our compiler. Of course, it is possible to write a Scheme program that cannot be executed when compiled using Bigloo but in practice, we have never encountered program limited by the Bigloo compilation model.

Scheme exceptions

Scheme surpasses most of the existing exception mechanism with its powerful primitives : `call-with-current-continuation` (or `call/cc` in short) and `dynamic-wind`. In short, `call/cc` enables exceptions to be raised backwards *and* forwards. That is, one may invoke an exception even when being out of the dynamic scope of the exception handler. Implementing `call/cc` within the natural mapping framework is painful. We have been a whisker away from creating a portable implementation for `call/cc`. Unfortunately `call/cc` requires the ability to resume a computation which, for C, means saving and restoring the execution stack. Obviously there is no way to restore the stack in a portable way because C is not even specified as using a stack [115]. A portable implementation for `call/cc` using the natural mapping is a lost cause. However we have been able to give an implementation of `call/cc` that compiles on every platform we have tested [204]. The tricks that are used in our implementation are totally inefficient. The Bigloo implementation for `call/cc` is likely to be the slowest implementation of `call/cc` available. It is possible to speed up many uses of `call/cc` however. Frequently, continuations are only used to escape from computations. That is, the continuations are invoked inside the dynamic extent of the `call/cc` expressions. To implement such continuations, C `setjmp`s (or even C `goto`s) are preferred. Bigloo does not implement that optimization. Instead, it proposes alternative escape constructions (`bind-exit` and `unwind-protect`) that are borrowed from the Dylan programming language [12, 214].

Garbage Collection

The very same problem exists for the Garbage Collector. There is just no portable way to find the collector roots for C. The collector roots are either on the C stack or in the C global variables. Finding the memory addresses for these two spaces is highly machine and operating system dependent. This require perennial efforts to make the garbage collector available for new architectures.

Other features

X. Leroy states several other features of C that make it an inadequate target language [136] : low level arithmetic with only primitive overflow detection, no multi-word arithmetic, no run time error catching, no traps for array bound checking. Some of these limitations can be worked around with specific implementations (e.g., when compiling for debug the Bigloo compiler inserts array bound checks in the C produced code). Some are unsolved (e.g., when the system raises a signal for runtime execution errors such as stack overflow or illegal arithmetic, Bigloo only traps the signal and suspends the execution).

10.3.5 Name mangling

Bigloo compilation entities are modules. A module encapsulates bindings and top level expressions. Amongst these bindings some are *exported* : that is, accessible from importing modules. The primary role of a module system is to prevent name collision. Two modules may declare two bindings that share a common name as long as these two modules do not import each other and are not both imported in a third module. This is in direct opposition to the C model. C does not have modules. A C binding of a C variable or a C function is either local to a file or global to the entire application. For C, it is illegal to use the same name to define several global variables. Every language owning more complex scoping rules than C must use some name encoding when compiled to C. This name encoding is called *name mangling*. The inverse operation, that is decoding, is called *name demangling*. The most common example of name mangling comes from C++ [140] which is used in this language to give unique names to class function members. The C++ name mangling cannot be directly used for our compilation model. As we have presented in Section 10.3.2, it may happen that a local nested function is compiled into a C global function. This C function must have a name that is guaranteed not to be in collision with any other global C name. C++ name mangling is not powerful enough thus we have decided to use our own name mangling. There is no need to present in detail our *ad hoc* technique because it is absolutely banal. The important point here is that common tools embedding C++ name demangling are of no help to us.

10.3.6 Efficiency of the natural mapping

We must conclude this section presenting the produced Bigloo C code with a short overview of the performance of the natural mapping. The full details are beyond the scope of this paper, so we just present an informal discussion. We think the natural mapping as a reasonable means to get close enough to the performance of pure C programs. Close enough means that Scheme programs written using an imperative style and their C counterparts perform similarly. This is made possible in 1993 by the garbage collector that delivers good performance. As described in an evaluation paper [263] Boehm's collector [27] had a performance comparable to that of Unix `malloc` implementations. Ever since, the Boehm's collector has continued to improve and it is likely that its performance is now even closer to that of `malloc`.

Two papers describing the Bigloo compiler [211, 207] contain some time figures showing that Bigloo performance is in the best case equivalent to the C performance and in the worst case two times slower.

The second aspect of efficiency is the time it takes the compiler to compile source files. Slow compilation used to be a problem because the C produced files are large but with our modern fast computers this is not a problem anymore. For instance, the full bootstrap of the compiler and its library takes about 10 minutes even though it requires the compilation of about 70000 lines of Scheme code.

10.4 The connection between Scheme and C

C is the most popular language for modern operating system implementations where *operating system* is used here in a very general sense. For instance, we consider that the window management system to be part of the operating system. With our modern, powerful computers it is a huge handicap for a programming language not to be opened to current hardware possibilities. For instance programming languages must provide a means to implement graphical tasks. For languages other than C, there are two main ways to meet that requirement :

1. To implement, by hand, wrapper libraries for any C desired library.
2. To enable a connection to C.

The first solution has the obvious drawback that any new library will require a new implementation effort. Small development teams cannot afford that effort. The second solution requires an initial development effort but once completed, all C APIs are available.

One of the main interests of the natural mapping is that it enables a full connection between Scheme and C. In this section, we detail the different levels of connection starting with the most common one and ending with the rarest.

10.4.1 Foreign function interface

Frequently, the connection to C is thought of as a Foreign Function Interface (henceforth FFI). Actually, the FFI is the first step of the connection. It is mandatory but it is not enough. As explained in Section 10.3.2, Scheme functions are compiled into C functions. Bigloo compiled Scheme functions are thus compliant to the calling protocol fostered by host systems. There is no difference between the activation frame for a Scheme function and the activation frame for a C function. This helps when calling a C function and enables a C function to call Scheme functions. The only difficulty comes from Scheme's name mangling. This problem is solved by a special Scheme construction that is equivalent to the C++ `extern "C"` declaration that disables name mangling.

The remaining difficulty is related to the type system. Scheme types embed runtime type information and are thus distinct from primitive C types. This is an obstacle to the FFI. However, we wanted raw C types to flow around in Scheme programs. We have chosen to address this problem inside the compiler. The internal representation of a type inside the compiler is a data structure containing two fields. A type identifier and a list of converters for that type. A converter is made of three components, a *destination type*, a *type check* and a *type coercion*. The *type check* is the code the compiler must emit for checking the validity of a type coercion. The *type coercion* is the expression that the compiler must emit when coercing values. C types and Scheme types are

TAB. 10.1 – Type conversions of integer types

		obj	bint	long
obj	(check)	#t	(integer? v)	(integer? v)
	(coercion)	v	v	(bint->long v)
bint	(check)	#t	#t	#t
	(coercion)	v	v	(bint->long v)
long	(check)	#t	#t	#t
	(coercion)	(long->bint v)	(long->bint v)	v

related by this mechanism of checks and coercions. For instance, let's consider the conversion from Scheme integers (denoted by the type `bint`) and C fix numbers (denoted by the type `long`). We recall here the existence of a general type for Scheme values, `obj`. Every Scheme type is a subtype of `obj`. That is, Scheme `bint` is a subtype of `obj` but C `long` is not. In order to convert the value of a variable v of type `bint` into a value of type `long`, a type conversion such as "`(bint->long` v`)`" is required. If the compiler does not know that v is of type `bint` but if v is known as being of type `obj` prior to the conversion, a type check must be inserted such as :

```
(if (integer? v)
    (bint->long v)
    (runtime-type-error v))
```

Table 10.1 summarizes the conversions of integer types. The left column contains the *from types*. For each type, the first line contains the type check that has to be applied, the second line contains the coercion function.

Converting a C integer into a Scheme integer is cheap because it is just a tag addition (such as "a C *shift* + a C *or*"). It may happen that the conversion from C to Scheme requires allocations. This arises for instance when converting a C `double` into a Scheme `real` or when converting a C `char *` into a Scheme `bstring`. Of course, for such objects, converting from C to Scheme is more expensive.

Thanks to our type conversion, a Scheme function may call any C function provided the compiler knows how to convert Scheme values into the C function formals type and how to convert the C value of the function return type into a Scheme value.

A Scheme function may also be called from C using the same technique. When declaring a Scheme function the program may contain annotations about the formal types and the function result. The types used in these annotations may be C types. Thus, from Scheme, it is possible to define a function that computes C values. In addition to the "no name mangling" construction, one may write a Scheme function that may be called from C, using ordinary C types.

Here is a small example of Scheme code using the FFI in both directions :

```
(module scheme<->c
   (extern (printf::int (::string ::long) "printf")
           (export (fib "scm_fib")))
   (export (fib::long ::long)))

(define (fib x)
   (if (< x 2)
       x
       (+ (fib (- x 1)) (fib (- x 2)))))

(printf "fib: %d" (fib 30))
```

That program implements a Scheme function `fib` that accepts a C `long` argument and return a C `long` value. That function is made available to C (i.e., it may be called from C code) under the name `scm_fib`. In addition, that program calls a C function `printf` from the Scheme code.

10.4.2 Extended foreign function interface

Because of the natural mapping, the structure of the compiled Scheme file and the structure of the source Scheme file are similar. The compiled file contains C statements and C expressions. One positive consequence is that it is easy to insert C macros into this compilation framework. Our foreign interface enables C macros. From the point of view of the Scheme compiler, there no difference between a C function and a C macro, except that a C macro may not be converted into a Scheme closure. Then, one may write code as follows :

```
(module scheme<->c.2
   (extern (macro c-islower?::bool (::char) "islower"))
   (export (islower? x)))

(define (islower? x)
   (if (char? x)
       (c-islower? x)
       #f))
```

even if `c-islower?` is implemented using a C macro.

The second step to the connection is to enable variables to be referenced from the foreign language. Once again, because of the natural mapping there is no difficulty for our Scheme compiler to introduce C global variable references. The general type coercion mechanism is used to ensure that C variable values are correctly boxed or wrapped before being used and that Scheme values are correctly unboxed or unwrapped before being set into C global variables. Scheme global variables may be exported to C. The name demangling construction presented for functions also applies to variables. To summarize, here is an example of Scheme code using the various forms of C import clause and C export clause.

```
 1 :  (module scheme<->c.3
 2 :     (extern (macro c-eof-object::int    "EOF")
 3 :             (macro getc::int (::file*) "getc")
 4 :             (nb-read-char::long        "nb_read_char")
 5 :             (export *stdout* "scheme_stdout"))
 6 :     (export *stdout*))
 7 :
 8 :  (define *stdout* (current-output-port))
 9 :  (define (getchar)
10 :     (let ((c (getc *stdout*)))
11 :        (set! nb-read-char (+ 1 nb-read-char))
12 :        (if (= c c-eof-object)
13 :            #f
14 :            (integer->char c))))
```

The Scheme global variable `*stdout*` is exported to C where it is known as `scheme_stdout`. That variable has the `obj` type. A C global variable (`nb_read_char`), a C macro (`EOF` and a C function (`getc`) are made available to Scheme by means of the `extern` module clause line *2*. As the example illustrates line *11*, C global variables can be set from Scheme code.

10.4.3 Exhaustive connection

In the previous sections we have presented how C functions and C variables may be mixed up in Scheme code and *vice versa*. The limitation of the connection comes from the C types the Scheme compiler is aware of. Until now, we assumed that the Scheme compiler has some knowledge of every primitive C type (e.g., `char`, `int`, `long`, `char *`). Moreover, the compiler knows how to convert most Scheme values into values fitting the C primitive types. In addition, Bigloo knows how to build C compound types and how to convert values using these new constructed types. Let us suppose, for instance, a C program declaring and using a structure :

```
struct point {
   int x, y;
};

struct point *print_point( struct point *p ) {
   printf( "<point: %d, %d>", p->x, p->y );
   return p;
}
```

C `struct point` values may be allocated, filled and modified from Scheme. A Scheme module that wants to use the C `struct point` type has simply to declare it as follows :

```
(module scheme<->c.4
   (extern (type s-point
                 (struct (x::int "x")
                         (y::int "y"))
                         "struct point")
           (pp::s-point (::s-point) "print_point")))

(let ((p (make-s-point)))
   (p-x-set! p 3)
   (print (point? (pp p))))
```

The compiler automatically creates getters, setters and a type predicate for each new C type. An interesting feature comes from the ambiguous garbage collector we are using [27]. Because the collector seeks objects even if they are only pointed to by the C stack or C variables, the `struct point` is allocated inside the Scheme heap.

10.5 The Profiler

Because each global Scheme function is translated into a global C function, assuming functions are not inlined, designing a profiler for Scheme is nearly as simple as designing a name demangler for our generated C code. One requirement of the implementation of our profiler, named BPROF, was not to modify the source code of any regular profiler, for the sake of simplicity and portability. As a consequence, BPROF works on any operating system which provides tools such as the Unix PROF, GPROF or similar commands.

Applications compiled for profiling, in addition to their regular computation, produce a file containing a name demangling table. With the Unix C model, a profiled execution computes a value and produces a call graph profile file (e.g., `gmon.out`). Our Scheme profiled executables produce a call graph profile file and a name mangling file. Then, BPROF simply invokes a regular C profiler, parses its output and demangles identifiers according to the demangling file. BPROF uses the demangling service provided by the application itself.

For instance, let us consider the following Scheme program :

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

(define (main argv)
   (print (fib 30)))
```

After compiling this program for profiling and running it, GPROF produces the following profile file :

```
% time    self  children    called      name
          0.00    0.58       1/1            main
 100.0    0.00    0.58       1          bigloo_main
          0.00    0.58       1/1            main_109_foo
-----------------------------------------
                       2692536              fib_1010_foo
          0.58    0.00       1/1            main_109_foo
 100.0    0.58    0.00       1+2692536 fib_1010_foo
                       2692536              fib_1010_foo
-----------------------------------------
          0.00    0.58       1/1            bigloo_main
 100.0    0.00    0.58       1          main_109_foo
          0.58    0.00       1/1            fib_1010_foo
```

Indentation is on purpose. It is intended to help the reading of large profile files. The name mangling table of `foo` is :

```
(MAIN                 "main_109_foo")
(FIB_1010             "fib_1010_foo")
```

Thus, invoking BPROF produces :

```
% time    self  children    called      name
          0.00    0.58       1/1            _bigloo_main
 100.0    0.00    0.58       1          bigloo_main
          0.00    0.58       1/1            MAIN
-----------------------------------------
                       2692536              FIB
          0.58    0.00       1/1            MAIN
 100.0    0.58    0.00       1+2692536 FIB
                       2692536              FIB
-----------------------------------------
          0.00    0.58       1/1            bigloo_main
 100.0    0.00    0.58       1          MAIN
          0.58    0.00       1/1            FIB
```

We have simplified a little bit these excerpts to fit the article format. Nevertheless what one should notice is that non-Bigloo functions (such as `bigloo_main` which is a library function implemented in C) are left in the profile output. We have made this choice because we have designed Bigloo to foster applications using both Scheme and C. In that context accessing C functions profile information is relevant. For instance, this choice makes the garbage collector execution visible in the profiler result.

In its current version, BPROF is implemented with about 164 lines of Scheme code. The specific profiling code inside the compiler is less than 100 lines of Scheme code. Thus, given a compiler which produces C code using the C calling conventions, writing a profiler for that compiler is not even a half day's job !

10.6 The Debugger

First, we present a short overview of the debugger commands followed by its global architecture. This will be followed by a focus on the main implementation difficulties where we show how source line stepping is implemented, how breakpoints are set and so on.

10.6.1 Overview of the debugger, BDB

We start presenting an overview of the debugger, which gives the reader a precise idea of what can be achieved using it.

A summary of the debugger commands

In the same spirit as GDB [226], BDB is a command line debugger. Its graphical interface is described in Section 10.2.6. In short, BDB proposes the exact same commands as GDB does, e.g., `break` to set a breakpoint, `step` to step on source line of code, `continue` to resume an execution, etc. The stack may be unwound by means of the `frame` command. For each stack frame, the value of the formal parameters and the local variables may be displayed. The dynamic type of a Scheme value may be displayed. When the program stops, Scheme expressions may be evaluated in the environment of the program in use at the breakpoint location. Instead of presenting a whole debugger manual, we present now a very short sample session. Consider this short Bigloo program :

```
 1 :  (module dbg-example
 2 :     (main main))
 3 :
 4 :  (define (fib x)
 5 :     (if (< x 2)
 6 :         1
 7 :         (+ (fib (- x 1)) (fib (- x 2)))))
 8 :
 9 :  (define (main argv)
10 :     (print (fib (string->integer (cadr argv)))))
```

Also consider this annotated excerpt from a debugging session (italicized sentences provide commenting, they are not input to or output from the debugger) :

```
$ bdb dbg-example
Reading symbols from dbg-example...
At this point we may set a breakpoint into the fib function
```

to be continued...

```
(bdb) break FIB
Breakpoint 1 at 0x804aa6e: file dbg.scm, line 4
We start an execution.
(bdb) run 30
Starting program: dbg-example 30
Breakpoint 1, FIB(...) at dbg-example.scm:line 4
4  (define (fib x)
The debugger prints the source line where the execution stopped.
Then the execution stack is printed.
(bdb :FIB) info stack
#0  FIB(...) at dbg-example.scm: 4
#1  MAIN(...) at dbg-example.scm: 9
#2  bigloo_main(...) at dbg-example.scm: 1
#3  main(...) at dbg-example.scm: 1
The value of the parameters of the current stack frame are printed.
(bdb :FIB) info args
X = (BINT) 30
Scheme expressions are evaluated.
(bdb :FIB) print (APPLY + (CONS X '(2)))
32
Execution is resumed.
(bdb :FIB) step
...
```

A dual mode debugger

The same BDB commands are used to inspect Scheme and C activation frames and to display Scheme and C expressions evaluation. However evaluating and displaying Scheme and C expressions require different operations. For instance, name mangling and demangling must take place for Scheme expressions. To let BDB discover when mangling or demangling is necessary, we have chosen the convention that Scheme identifiers are composed of mixed case, as long as there is at least one lowercase character. This explains why setting a breakpoint in the Scheme `fib` function is typeset `break FIB`. Using `break fib` would set a breakpoint in a C function called `fib`. On the other hand, when a stack frame is displayed BDB checks if the function of that frame is implemented in Scheme or in C. A function is implemented in Scheme if the function's identifier is found in the Scheme symbol table. When displaying Scheme stack frame, the function identifier and the formal parameters are demangled.

It is important that both Scheme and C inspection operations are available from BDB because the Bee advocates the integration of Scheme and C programs. Thus it is consistent to have a debugger that is either able to step into Scheme or C source text. This is reason why BDB does not hide activation frame that are associated to C functions. We'll make a small modification to the example of Section 10.6.1 by supposing that `fib-2` is a function defined in C :

```
 1 :  (module dbg-example
 2 :     (extern (fib-2::int (::int) "fib_2")
 3 :              (export fib "fib"))
 4 :     (export (fib::int ::int))
 5 :     (main main))
 6 :
 7 :  (define (fib x)
 8 :     (if (< x 2)
 9 :         1
10 :         (+ (fib-2 (- x 1)) (fib-2 (- x 2)))))
11 :
12 :  (define (main argv)
13 :     ...)
```

Let us spawn a new debugging session :

Let's set a breakpoint into the `fib_2` *C function*

```
(bdb) b fib_2
Breakpoint 3 at 0x804a294: file cdbg.c, line 4
```

Let's start an execution.

```
(bdb) run 30
Breakpoint 3, fib_2 (x=29) at cdbg.c:4
```

Execution stack is :

```
(bdb :fib_2) info stack
#0  fib_2(...) at cdbg.c:4
#1  FIB(...) at dbg.scm: 7
#2  MAIN(...) at dbg.scm: 12
#3  bigloo_main(...) at dbg.scm: 1
#4  main(...) at dbg.scm: 1
```

Because the execution stopped inside a C function, inspecting the current stack frame displays informations relative to the current C function.

```
(bdb :fib_2) info args
x = 29
```

10.6.2 The debugger architecture

In order to get portability and availability when designing the debugger, we chose to reuse existing C debuggers. That is, instead of writing a new debugger from scratch, we wrote a front-end for an existing debugger. We deployed the same strategy as DDD [261]. The reasons for this choice are twofold :

– Fully featured C debuggers exist. For instance, the GNU debugger GDB implements most of the common features of the C debuggers.

– Most of all we wanted to avoid wasting time in re-implementing C debugger machinery such as breaking, stepping and continuing.

When debugging a Bigloo program three processes are spawned. One process for Bdb, another for Gdb, and the last one corresponding to the debugged process (henceforth called the patient). An overview of the debugger architecture is presented in Figure 10.6.

Bdb connected to Gdb

Bdb spawns a Gdb process that will, in turn, spawn the patient process. Bdb takes over Gdb inputs and outputs. That is the command line is read by Bdb and all outputs are parsed by Bdb before being printed out. All information relative to the execution are held by Gdb. Thus, printing any information relative to the execution state requires a communication between Bdb and Gdb. Printing the execution stack frame is a good example. When the user invokes the Bdb "`info stack`" command, Bdb sends the command to Gdb, reads back the whole execution stack frames and filters that list according to the Scheme symbol table.

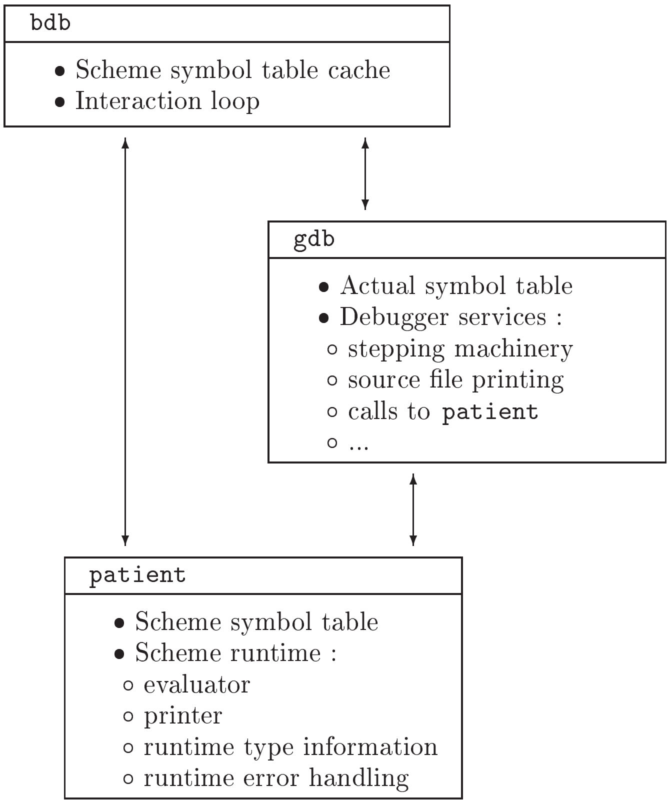


Fig. 10.6 – The debugger architecture

Bdb connected to the patient

In various situations Bdb must also request information from the patient process. A first example is the way Bdb accesses the global Scheme symbol table. Gdb is able to retrieve the symbol table of an executable. This symbol table does not correspond to the Scheme symbols that compose the program because of name mangling (see Section 10.3.5) and because many unique Scheme symbols are compiled into two C symbols (such as Scheme closures) leading to extra C identifiers. The actual Scheme symbol table is located inside the patient static area. Using a framework close to the one used for profiled compilations, the Scheme compiler inserts into the executable a piece of data that represents the Scheme symbol table. When BDB intercepts a symbol in a GDB output, it checks to see if that symbol has to be demangled. A symbol has to be demangled if it is associated to a Scheme binding. BDB maintains a local Scheme symbol table that acts as a cache. If a symbol is not found in its cache, BDB requests the global Scheme symbol table from the patient by the means of the GDB `call` command. The result of that operation is stored in the BDB Scheme symbol table cache.

Another situation where BDB must ask for services from the patient is when a Scheme expression must be evaluated. This evaluation is not performed by the debugger but instead by the patient. As with the symbol table, the request for evaluation is sent to the patient via the GDB `call` command. This technique prevents the debugger from embedding its own evaluator and, more importantly, it ensures that the evaluation environment will be that of the patient. As we have already seen for profiled executions, debugged applications embed services that are to be used by the Bee.

10.6.3 Source line stepping

As with traditional C debuggers, the granularity of stepping is the source line. That is, the command "`step`" executes until the execution thread reaches a new source line's code. The goal of the BDB implementation is to mimic as much as possible the GDB behavior when stepping. The difficulty to be faced is specific to the Scheme programming language. It consists of keeping track of the original source file position in the compiler abstract syntax tree.

Source file positioning in the compiler abstract syntax tree

One elegant aspect of Scheme is that the abstract syntax tree for the first compiler stage is a subset of Scheme's values. During macro expansion, a program is represented as an s-expression, which is either an atom or a pair of s-expressions. A macro is thus a function that takes an s-expression and returns a new s-expression as a result. The direct consequence is that there is no place in this s-expression to store source file information. Several solutions exist to solve this problem. We have implemented one within our compiler that is rather tricky. We describe it very briefly.

We have added one new primitive data type : the `extended pair` which is a subtype of the regular `pair` type. An extended pair is a pair with an additional slot, the `cer`. The reader builds s-expressions using `extended pairs` instead of normal pairs and the `cer` slot is filled with the source file position. Because `extended pair` is a subtype of `pair`, the s-expressions built by the reader are compatible with normal s-expressions and thus, may be used within macro expansion.

There are some limitations to this solution. For instance, we are not able to find out the position of atoms in the programs. Also, if all the expressions of a program are rewritten by user macro expansions it is likely that the compiler won't be able to find many source line locations. In practice, however, our solution seems sufficient. We have just taken care when implementing the compiler macros to re-use as many lists read from the source code as possible. Avoiding the allocation of new pairs allows the macro expansion not to discard the source file position contained in the `cer` slots of lists build by the reader. After macro expansion, the program is a mixture of plain pairs and extended pairs. The first stage of the compiler spreads into plain pairs the source file location of their nesting extended pairs. The result is that unannotated expressions are assigned the location of the last tracked location. Thus, locations for macro generated expressions are the locations of the calls to the macro.

Let us illustrate locations propagation over macro generated expressions with the simple example :

```
 1 :  (define-macro (when test . body)
 2 :     `(ifr ,test (begin ,@body)))
 3 :
 4 :  (define (display-point pt)
 5 :     (when (point? pt)
 6 :        (display "<point ")
 7 :        (display (point-x pt) pt)
 8 :        (display ", " pt)
 9 :        (display (point-y pt))
10 :        (display ">")))
```

There are two errors in this program. The `if` construction is miss-spelled `ifr` in line *2*. The `display` optional output port argument has the wrong type line *7*. When compiling Bigloo reports :

```
File "foo.scm", line 10, character 170:
#   (when (point? pt)
#   ^
# *** ERROR:bigloo:IFR
# Unbound variable -- IFR
```

That is the error inside the generated text is located at the macro call site. When this first error is fixed and the program is re-executed, Bee reports :

```
File "foo.scm", line 13, character 248:
#      (display ", " pt)
#      ^
# *** ERROR:bigloo:#{POINT 2 3}
# Type `OUTPUT-PORT' expected, `STRUCT' provided -- #{POINT 2 3}
```

The error is correctly located because the expression `(display ", " pt)` is not *generated* by macro expansion. It is only *inserted* by the expansion of the call to `when`. The expression holds its own source location which was propagated by the macro expansion stage.

Source file positioning in the executable file

Once source file locations are held by the compiler, it is very easy to propagate them into the executable file. As our compilation framework consists of producing C files, we simply use the Iso C [115] "`# line`" facility which allows source position annotation inside source programs. Given a Scheme source file, the resulting C program, annotated of many "`# line`" directives, is hardly human readable but this technique succeeds at propagating Scheme source file information into executables. Because one Scheme function may be compiled into two C functions, it might happen that two C functions are defined at the same virtual source line of code. GDB is not always able to correctly handle such a situation and we have been obliged to implement a workaround. This is presented in Section 10.6.4.

10.6.4 Breakpointing

When provided with fully C "`# line`" annotated programs, C compilers are able to deliver executables which embed debugging information that allow source line debuggers such as GDB to step, break and resume execution correctly most of the time. In theory, the technique presented in Section 10.6.3 is sufficient to enable GDB to break, step and resume inside Scheme programs. In practice, because of some GDB implementation limitations, it is not that simple ! When execution reaches a breakpoint that is set at the beginning of a function definition, it may happen that GDB stops the execution before the activation frame is completely initialized. This depends on the platform GDB is running on and how the breakpoint has been set up. Stopping execution before the activation frame construction is completed has one nasty effect : if inspected before stepping one assembly instruction from the breakpoint position, the actual arguments of the function contain invalid values. Although annoying, when debugging C programs this is not a real disaster because the actual arguments are fixed up with one single "`step`" issue. When debugging Scheme programs the problem is much more severe. In order to be displayed, Scheme objects have to be correctly tagged or boxed. When requesting the printing of an object located at a certain memory location that does not correspond to a legal object (as it might happen with a dangling pointer), the runtime system is likely to crash. Stopping execution at a position where arguments are not correctly set up is unacceptable for Scheme. Fortunately, two simple tricks are enough to fix the problem.

GDB is not able to correctly set breakpoints when they are inserted at the first line of a function definition or when they are inserted at a memory address that corresponds to a pointer to a function (such as the GDB command "`break *(&foo)`"). Fortunately, GDB is able to behave correctly when function identifiers are used instead of source lines. Let us assume a function `foo` defined in a file `foo.scm` at line *27*. Setting a breakpoint using the command "`b foo.scm:27`" may fail but "`b foo`" succeeds ! As a consequence the solution is easy for us : BDB maintains a hash table of all Scheme definitions where the user may set a breakpoint. When a breakpoint using the line position is requested, BDB checks in its hash table if a function definition is associated to that line position. If it does, the breakpoint to the source line number is turned into a breakpoint to the function identifier. The same solution applies when breakpoints are set to memory addresses.

The second alternative trick is even simpler. GDB gets confused when breakpoints are set at the very first lines that start function definitions. If breakpoints are set at the first line of the function bodies then GDB breaks correctly. The solution is to tailor the C code generation to make the function definition start at a fake line. For instance, if the Scheme code is :

```
27 :  (define (foo x)
28 :     (bar x))
```

The compiler must not generated such a code :

```
#27 "foo.scm"
obj foo_foo(obj x_1) {
#28 "foo.scm"
...
```

Because using the GDB command `b foo.scm:27` makes GDB confused. However if we change the C code for :

```
#99999 "foo.scm"
obj foo_foo(obj x_1) {
#27 "foo.scm"
{/* a dummy statement /*}
#28 "foo.scm"
...
```

Then, the command `b foo.scm:27` won't stop the execution at the function definition location but at the first line of the function body. That way GDB is able to correctly build the activation frame for `FOO`. Bee implements the last solution.

Stopping inside local definitions

A similar problem with variable initialization arises when the execution thread reaches a local variable definition such as in the following when execution reaches line 7 (we remind that Scheme local variables are compiled into C local variables) :

```
6 :  (define (foo x)
7 :     (let ((y (+ 1 x)))
8 :        (print y)))
```

Variable `y` will be available for printing only when fully initialized. A new solution is required for such a situation and, this time, the help of the compiler is required. The solution consists of adding dummy code labeled with the same line number as the source line of `y`'s definition that will make the execution stop before `y` is bound. Then, when using command such as "`info locals`", `y` won't even be defined before line *8*. The C code emitted contains a simple dummy variable definition :

```
/* foo */
#6 "foo.scm"
obj foo_foo(obj x_1) {
#7 "foo.scm"
   int bigloo_dummy_bdb; bigloo_dummy_bdb = 0; {
#7 "foo.scm"
      obj y_3 = plus_r4_numbers_6_5( BINT(((long) 1)), x_1);
...
```

Breakpointing into closures

The last difficulty with breakpointing comes from closures. Because Scheme has closures, it is mandatory for the debugger to have the ability to set breakpoints in closures. Here is the method used within BDB for this.

1. When the user requests a breakpoint in a closure, BDB asks the patient for the address of the C function associated with that closure (see Section 10.3.2 for a short description of closure compilation) and the breakpoint is set in that function.
2. Setting a plain breakpoint in the C function associated to the closure is not enough because the execution will stop each time it reaches this C function even when entered from another Scheme closure associated with that same C function. Thus, BDB does not set a plain breakpoint but a conditional breakpoint. Because the first argument of a C function associated to a closure is the closure itself, the condition for the breakpoint is a simple test that checks that the first argument of the closure is the closure that has been used to set the breakpoint.

10.6.5 Inspecting Scheme variables and evaluating Scheme expressions

When execution stops in a function implemented in Scheme, commands to inspect local variables and function parameters may be issued. These commands require the collaboration of BDB, GDB and the patient. To support this, BDB asks GDB for the current frame and the current local variables and arguments. From the current stack frame, BDB deduces the function where execution stopped. From that function and with its symbol table, BDB is able to retrieve the Scheme name associated with the function and the list of local variables. This list is an association list that associates C names to Scheme names. According to that list BDB is then able to build the current local Scheme environment. This local environment is used to evaluate Scheme expressions. Prints and evaluations are performed by the patient not BDB itself. BDB calls the patient via GDB and the `call` primitive. The called function is the patient evaluation function. The arguments to the `call` command are made of the local environment that is built from an association list that associates Scheme identifiers to memory addresses.

10.6.6 Pros and cons of the Scheme symbol table implementation

BDB maintains a partial copy of the Scheme symbol table that acts as a cache. That table stores information relative to name mangling. It is partial because it may lack some symbols. When BDB checks for an identifier, if it does not find the identifier in its cache, it requests the patient for a consultation of the complete global Scheme table. The result of that consultation is stored in the local cache. The cache is flushed each time a BDB `file` command is issued (the `file` command re-loads a binary executable file).

This framework presents some qualities :

– Unless all symbols are needed by user commands, the complete Scheme symbol table is not duplicated. It only resides in the static memory area of the patient.
– BDB loading of Scheme symbols is lazy and it only takes place when GDB stops an execution. That is, requests to the PATIENT are raised during interactions with the BDB user.

On the other hand, it has an important drawback :

– Requests are sent to the patient by the means of the GDB `call` command. This command can *only* be issued when a patient execution is running. In other words, before the execution is initiated, BDB cannot access the patient global symbol table. The direct consequence is that it is not possible, for instance, to set a breakpoint using the identifier of a Scheme function before the execution is started because at that point BDB is unable to process name mangling for that identifier. In short, the traditional debugging sequence `break main; run` is not accessible to BDB !

Previous version of BDB loaded the entire table whenever the patient was restored, but this was too slow so we implemented lighter weight strategy in the latest version o BDB, Bee automatically generates one file describing the global symbols of the source code. That file is used to allow fast browsing over the source files (see Section 10.2.1). BDB builds an early version of its Symbol table cache by simply loading that file each time a `file` command is issued ! That way, it has been possible to get rid of the complexity of spawning fake executions to download the global symbol table while enabling early breakpoints using Scheme identifiers.

10.6.7 Concluding remarks on the debugger

Although Bee's profiler is much smaller, the debugger is till relatively small, 5800 lines of Scheme code + 500 lines of C for the implementation of the debugger and 600 lines of Scheme code inside the compiler. Compared to the half a million lines of C code in GDB 7000 lines is quite small. Designing and implementing a debugger is a complex task. Thus, even the smartest design won't succeed in hiding all the complexity of the implementation. We have now acquired the conviction that our global debugger architecture was the only one allowing us to have a complete debugger for a couple of months' effort. We have shown in this section that our debugging technique has drawbacks that have required rather tricky solutions. On the other hand, it has two enormous advantages. First, all the machine specific part for the debugger has been re-used. Second, we benefit from the *savoir faire* of the whole GDB team for implementing breakpoints, stepping watchpoints, etc. Compared to the task of writing a debugger completely from scratch, the small shell we have written is worth the effort.

10.7 Related work

Wadler's statement that implementations for FL hardly ever provide full-featured development environments came as no surprise to us. We know of three exceptions : Ericsson's Erlang [71], Harlequin's ML Works [103] and Allegro Common Lisp Composer [84]. Implementing an IDE from scratch requires a huge and time-consuming effort. This probably explains why very few academic projects concentrate on programming environments. One of the most significant is PLT [76]. Before comparing development environments, we provide a comparison of connection techniques between Lisp-like languages and C.

10.7.1 Lisp and C

Three main papers describe a connection between Lisp and C :

– The Rose and Muller paper [180] describes the *Embeddable SHell* foreign interface. The ESH project aims at using Scheme for implementing a glue language. The foreign interface goal is to allow ESH to use all C types. The ESH runtime allows ESH objects to have a layout as close as possible to C objects. Unfortunately, the paper is missing many implementation details, so it is difficult to understand if their runtime decisions enable fast code.
– H. Davis, P. Parquier and N. Séniak have designed a rich interface between Ilog Talk and C++ [61]. The integration takes into account the object layer of the two languages. It seems that Ilog Talk offers the same facilities as the Bee, the difference coming from the compiler. The Bigloo compiler generates stub code for foreign objects. The Talk compiler does not embed a stub generator and connection between C++ and Talk requires extra C function

calls. The goal of Bigloo was to provide a very cheap connection between Scheme and C. The solution of Ilog Talk does not meet this goal.

– G. Attardi has presented an implementation of Common Lisp enabling connection between Lisp and C [14]. This implementation uses a compiler that produces C code that mostly conforms to the standard C programming rules [31]. The foreign connection relies on the sharing of a common runtime library. The Lisp compiler has no knowledge of any kind of C types, which is the main difference with our work.

10.7.2 DrScheme

The only integrated environment for Scheme we know of has been developed by the PLT team of Rice University [76, 73]. They have released an environment named DrScheme which is designed for teaching. It is not a true integrated environment yet because it does not contain project management, profilers, or a dynamic debugger.

DrScheme relies on a graphical user interface for editing and interactively evaluating Scheme programs. It helps programmers to understand their source code by providing syntactic help where, for instance, a program's syntax may be checked and, on success, DrScheme is able to draw transient arrows from an identifier introduction to its uses and vice versa. Programs need to be completely checked for DrScheme to be able to display the arrows. It should be possible to take advantage of Scheme syntax in order to make some kind of partial parsing and to start displaying information even when the whole program is not present. Nevertheless, we must acknowledge that DrScheme has nice features that are currently missing in the Bee. The most innovative feature of DrScheme relies on its embedded static debugging analyses [80, 79]. The key idea is to use a *set based analysis* (i.e., an abstract interpretation) for static debugging. This usage of abstract interpretation is rather new. The very few systems we know that use abstract interpretation do so for optimization purposes [183, 211]. These analyses have not been designed to help the understanding of the source code. This new usage of analysis for program understanding seems very promising and it will be welcome if new research work presents static analysis for program comprehension [159].

10.7.3 Java Development Environment

To some extent, Bee is inspired by the Java Development Environment (henceforth JDE) [124]. The JDE is an integrated environment for the Java language entirely implemented in Emacs Lisp. JDE and the Bee have lot of similarities. The JDE relies on the Sun Java Development Kit. The JDE enables source file browsing by the means of regular Emacs packages. For portability reasons JDE has not chosen between Emacs and Xemacs, that is, it runs on both. We think this is an error because it prevents the JDE from using any graphical interface. For instance, when debugging, it is preferable to set a breakpoint using a mouse rather than using a command line interface.

10.7.4 Allegro Common Lisp

Allegro Common Lisp (henceforth ACL) has two different integrated environments. The first one, called "ACL Composer", is not available within free distributions. As a consequence we have not been able to test it. Its second IDE is based on Emacs. Even if the choice of Emacs seems familiar, the approach used in ACL's IDE is different from the one used in the Bee. Many facilities are shared by the two environments (for instance, tags files which allow quick source file browsing are handled by both environments) but ACL's IDE does not attempt to match the habits of C programmers under the Unix environment. Common Lisp does not make use of a Makefile. The ACL environment is really unfamiliar to someone used to the C development cycle. Thus, using the ACL environment requires changes in the programmer habits. As we have pointed out in the introduction, we think that enforcing an unusual programming methodology discourages programmers.

In addition, the Emacs environment for ACL does not make use of any graphical interface. Even if command line interfaces are powerful, in some situations they are inferior to graphical interfaces. For instance, when debugging a program, it is much more convenient to set a breakpoint using a mouse click inside the source code than typing a line number to describe the position of the breakpoint.

10.7.5 Lisp machines and interactive environments

The programming methodology advocated in this paper is the opposite of the one in used in old Lisp environments [191]. We enforce static features such as modules, impossibility to refer to unbound variables, type and arity checks. Our environment is designed for the delivery of stand alone applications, which is the dominating trend for the last twenty years. Lisp environments were designed toward permanent interactions with the programmer. It is however possible that such environments get a revival interest because they seem to propose features now needed by network programming (such as the dynamic upgrades of some portion of a program).

Conclusion

More than eight years ago, R. Gabriel stated that Lisp was lacking some important features and that its development was endangered [86]. According to Gabriel the two main deficiencies were : a connection to C and the ability to produce stand alone executables. The Bee is an integrated development environment for the Scheme programming language that implements these features. Within the Bee, Scheme code and C code may be integrated harmoniously to build small stand-alone executables.

In the first section of this paper the graphical user interface was presented. The remaining sections have been devoted to the presentation of the portable implementation of two of the main environment tools : the profiler and the debugger. These implementations are driven by the very same idea : minimizing the implementation effort by relying as much as possible on existing C tools. For instance, our debugger is 6000 lines of Scheme code which is far fewer than the half million lines of C code for GDB ! Our debugger may be used on any computer and operating system provided the GNU debugger GDB is ported to that machine. Designing our debugger as a shell around GDB enabled us not to implement any architecture specific features. These are embedded in GDB.

In order for Scheme to be able to use regular C tools such as GDB, the runtime execution of Scheme must comply with the C execution model. For instance, Scheme function invocations must conform to the C call protocol of the host machine. The easiest way to reach this compliance is to compile Scheme to C using what we call the “natural mapping” framework. We can think of no reason why a natural mapping could not be used for other languages.

The class of functional languages will be more widely used if we bring the development facilities of classical languages to users. In the coming months we will have to maintain our implementation work in order to deliver more libraries. For instance, a connection to a graphical toolkit or a library for networking are likely to be the next targets of our implementation effort.

Acknowledgments

Many thanks to Barrie Stott, Bahman Rafatjoo, Céline and the anonymous referees of Journal of Functional Programming for their helpful feedbacks on this work.

Chapitre 11

Biglook : a Widget Library for the Scheme Programming Language

Cet article est actuellement en soumission. Il a été écrit en collaboration avec Erick Gallesio.

Biglook : une bibliothèque graphique pour le langage Scheme

Cet article présente Bigloo, une bibliothèque graphique pour une version étendue du langage de programmation Scheme. Cette bibliothèque est construite en utilisant une couche objet largement inspirée de CLOS. Les objets graphiques sont tous décrits par des classes. En utilisant à la fois le modèle objet et le modèle fonctionnel, l'interface de programmation de Biglook permet un style déclaratif pour la réalisation des interfaces et une stricte séparation entre les parties d'un programme dédiés à l'implantation de son interface et aux parties dédiées au calcul.

Afin de permettre plusieurs ports, l'implantation de Biglook dissocie clairement l'interface de programmation en Scheme et la partie native de gestion des objets graphiques. La version actuelle de Biglook utilise Tk mais une version alternative basée sur GTK+ est en actuellement en développement. Nous montrons dans cet article que le sur-coût associé à cette architecture logicielle en couche est faible.

Biglook : a Widget Library for the Scheme Programming Language

This paper presents Biglook, a widget library for an extended version of the Scheme programming language. It is built on top of a CLOS-like object layer where widgets are represented by means of classes. By combining functional and object oriented programming styles, Biglook's API advocates a strict separation between the implementation of the interfaces and the user associated commands.

In order to enable different ports, Biglook implementation separates the Scheme programming interface and the native back-end. The current version uses Tk, but an alternative version based on GTK+ is under development. We show in the paper that the overhead of this layered architecture is low.

We also present here some conclusions that we have drawn from our experience with GUI programming. We present the mandatory features a programming language must supply. In particular, we show, that functional and object-oriented programming harmoniously fit together, provided they offer keyword parameters, generic functions and a new construction called *virtual slots*.

Introduction

Functional languages (henceforth FLs) must provide facilities for constructing contemporary applications. For instance, FLs *must* provide facilities implementing database access, network roaming or graphical user interfaces (henceforth Gui) [86, 250]. We have studied the problem of constructing Gui in functional languages by designing a widget library for the Scheme programming language which is presented in that paper. By contrast to previous work, no attempt have been made to make that library familiar to programmers used to imperative or purely object oriented programming style. On the contrary, our library proposes an original application programming interface (Api) that benefits from the high level constructions of the extended Scheme implementation named Bigloo [202]. The main Bigloo component is an optimizing compiler that delivers small and efficient applications for the Unix™ operating system. Biglook's primary use is thus to implement graphical applications for the X Windows system (X clients such as `xload`, editors *à la* Emacs, browser *à la* Netscape or even programming tools such as `kprof`, our graphical Scheme profiler).

Bigloo implements an object layer inspired by Clos [24]. This is a class based model where methods override generic functions and are not declared inside classes as in Smalltalk [91], O'Caml [176] or Java [95]. Generic functions provide a greater flexibility and extensibility enabling users to customize libraries in many directions.

Biglook is implemented as a wrapping layer on top of native widget libraries (that we name henceforth the *back-end*). This software architecture saves the effort of implementing low level constructions (such as pixel switching, clipping, event handling and so on) while allowing focusing on the Scheme implementation of new features. For instance, amongst the fifty available Biglook widgets only half of them are directly mapped to primitive widgets. The other ones are compound widgets implemented in Scheme. These widgets are independent of any native back-ends. Currently Biglook uses Tk [155], but a GTK+ [158] back-end is in progress.

When designing the Biglook Api we have always had to decide which model to choose : the functional model or the object model. We think that these two models are not contradictory but complementary. For instance, if the widget hierarchy naturally fits a class hierarchy, user callbacks are naturally implemented by means of Scheme closures. The apparent opposition between the two models has driven us to study the features a language must provide in order to ease both the implementation of a widget library and client applications making use of that library.

In order to separate between the Biglook Api made of Bigloo classes and the native back-end, a new Bigloo construction has been needed : the *virtual slots*. They are presented in Section 11.1.6. To estimate the performance of this architecture, we have conducted experiments that we present in this paper. Virtual slots impose a low overhead, which enables Biglook compiled applications to be reasonably efficient both in execution time and in memory allocation in comparison to their native counterparts.

In the Section 11.1 we present the Bigloo system emphasizing its module system, its object layer and the new *virtual slot* construction. Then the underlying model of the Biglook library is described in Section 11.2. The way it can be used or extended is also shown in this section. In Section 11.3 we present the Biglook implementation. We describe its general architecture and we present the role of virtual slots in the Biglook implementation. In that section we also present a performance evaluation. In Section 11.4 we synthesize the mandatory features a language must be provided with in order to enable easy Gui programming. At last, in Section 11.5 and 11.6 we present the future directions and some possible extensions for Biglook as well as a comparison with related work.

11.1 Bigloo

Bigloo is an open implementation of the Scheme programming language. From the beginning, the aim was to propose a realistic and pragmatic alternative to the strict Scheme implementation defined by [120]. Bigloo does not implement "all" of Scheme : for example, the execution of tail-recursion may allocate memory. On the other hand, Bigloo implements numerous extensions : support for lexical and syntactic analysis, pattern matching, an exception mechanism, a foreign interface, an object layer and a module language. In this Section, we present the Bigloo's modules, its type system and its object model ; that is, class declarations, *virtual slots* and generic functions.

11.1.1 Modules

Bigloo modules have two basic functions : one is to allow separate compilation and the second is to increase the number of errors which can be detected by the compiler. Modules are simple and they have been designed with the concern of an easy implementation.

A module is a compilation unit for Bigloo. It is represented by one or more files and has the following syntax :

```
(module module-name
   (import import+)*
   (export export+)*
   (static static+)*
)
optional-body
```

Import clauses are used to import bindings in the module. In order to import, one just needs to state the identifier to be imported and its module.

Export and *static* clauses play a very close role. They point out to the compiler that the module implements some bindings and distinguish those that can be used within other modules (they are exported) and those that not (they are static). These clauses do not contain identifiers but prototypes. It is then possible to export variables (mutable bindings) or functions (read-only bindings). Static clauses are optional (the bindings of a module which are not referenced in a clause are, by default, static).

Let us look at the example of the `fib` module that imports the functions `fib-fx` of module `fib-fixnum` and `fib-fl` of module `fib-flonum` and that exports the function `fib`.

```
(module fib
   (import (fib-fx fib-fixnum)
           (fib-fl fib-flonum))
   (export (fib x)))

(define (fib x)
   (if (fixnum? x) (fib-fx x) (fib-fl x)))
```

During compilation, Bigloo performs a number of verifications :

- All referenced variables and functions must have been defined in the module or mentioned in an import clause (i.e. there is no free variable).
- All the identifiers mentioned in static or export clauses must have been defined in the module with a value conforming to their prototype.
- Identifiers mentioned in static or export clauses whose prototypes are functions must not been assigned.
- All function calls where the functional operator is known to be a function (i.e. the identifier in a functional position has a function prototype) must conform to the prototype of the invoked function.

Expressions that make the body of a module can be variable or function definitions (with the Scheme form `define`) or any Scheme expression. *Initializing* a module is an operation that binds variables to values and evaluates expressions that are not definitions. Definitions and expressions are evaluated in the order of their appearance in the module. Importation graphs of modules are not restricted to trees, that is, two modules or more can import themselves mutually. The module initialization order is then unspecified.

11.1.2 Type-checking

The Scheme language is strongly anchored to a dynamic type-checking tradition (i.e. type checking is done at execution time). People used to say that a ML program can be compiled if no false interpretation of the types exists whereas a Scheme program is compiled if it exists at least one correct interpretation. Bigloo positions itself in the Scheme tradition but unlike popular systems, it attempts to effectively show that there exists a correct interpretation of programs before compiling them. This can lead, sometimes, to compilation rejections because of typing errors.

Another Scheme tradition (shared with ML this time) is that type annotations do not appear in programs. There is no way to declare that a variable contains data of one given type. Bigloo does not conform to this tradition and fosters type annotations. They are optional but they allow to produce a better quality code and to detect more errors at compilation time.

A Bigloo type is either an atomic Scheme type (e.g. a Scheme integer, a Scheme string, a Scheme list, ...), an external type (e.g. a C integer, a C string, a C structure, ...), or a type associated with a class (classes are presented in Section 11.1.4).

Type annotations can be inserted in exportation clauses, in variable definitions or in lexical blocks. The syntactic construction is [<var-id>] `: :`<type-id>. As an example, here is a module implementing the Fibonacci function on integers.

```
(module fibo
   (export (fib::long ::long)))

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

The clause `(fib::long ::long)` just means that the module implements a function called `fib` with an argument of type `long` and that it returns a result of type `long`. Type annotations are not needed in definitions when they already appear in a module clause ; this is the reason why the annotation `::long`[1] is written only in the exportation clause of `fib`.

11.1.3 Object layer

The term "object-oriented" is a rather loose fitting description that has been prefixed to a number of very different languages and systems over the years. We will restrict our attention to two object models : the Smalltalk model [91] where methods are associated to classes and the Clos model [24] where methods are associated to generic functions. Each of those models has many incarnations : with static or dynamic type checking, with single or multiple inheritance, and single or multiple dispatch. In this paper we assume a Clos-like object model with single inheritance and single dispatch. The object layer implemented in Bigloo is a restricted version of Clos inspired, to a great extent, by Meroon [170] of C. Queinnec.

11.1.4 Class declarations

Classes can be declared static or exported. It is then possible to make a declaration accessible from another module or to limit its scope to one module. The abbreviated syntax of a class declaration is :

(**class** *class-id*::*super-class-id optional-init optional-fields*)

A class can inherit from a single super class. Classes with no specified super class inherit from the `object` class. The type associated with a subclass is a subtype of the type of the super class.

A class may be provided with an initialization function (*optional-init*) that is automatically called each time an instance of *class-id* is created. Initialization functions accept one argument, the created instance.

[1] We draw the reader's attention to the fact that the `long` type comes from the C integers and not the Scheme ones, to show that Bigloo handles indifferently C and Scheme data.

A field may be typed (with the annotation `::`*type-id*), may be immutable (if declared `read-only`), and may have a default value (`default` option). Here are some possible declarations for the traditional `point` and `point-3d` :

```
(module module-points
   (export (class point
               (point-init)
               (x::double (default 0.0))
               (y::double (default 0.0)))
            (class point-3d::point
               (z::double (default 0.0)))))

(define point-init
   (let ((count 0))
      (lambda (obj::point)
         (set! count (+ 1 count))
         (print "# of points: " count))))
```

11.1.5 Instances

When declaring a class *cla*, Bigloo automatically generates the predicate `cla?`, an allocator `instantiate::`*cla*, a cloner `duplicate::`*cla*, accessors (e.g., `cla-x` for a field `x`), modifiers (e.g., `cla-x-set!` for a field `x`), and, an abbreviated access special form, `with-access::`*cla*, to allow accessing and writing fields by referencing them by their name. Here is how to allocate and access an instance of *point-3d* :

```
(let ((p (instantiate::point-3d (y -3.4) (x 1.0))))
  ;; The initialization value of a field can be omitted from the arguments list if it has
  ;; a default value; this is the case for the field z of class point-3d.
   (with-access::point-3d p (x y z)
      (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

The `instantiate` and `with-access` special forms are implemented by the means of macros that statically resolve the keyword parameters. For instance, the above example is expanded into :

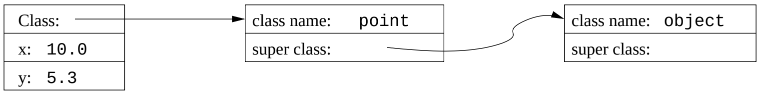
```
(let ((p (make-point-3d 1.0 -3.4)))
   (sqrt (+ (sqr (point-3d-x p))
            (sqr (point-3d-y p))
            (sqr (point-3d-z p)))))
```

As we can see, since macro are expanded at compile-time, there is no run-time penalty associated with keyword parameters.

11.1.6 Virtual slots

Bigloo proposes two kind of instance slots : regular slots that have already been described, and *virtual slots* which enable several views of a single data.

Using virtual slots gives the illusion of accessing the slots of class instances while instead, Scheme functions are called. Declaring a standard class defines the memory layout of its instances. Each class field is allocated a memory area. For instance, the memory layout of an instance of the `point` class is :



As we have seen in Section 11.1.5, the compiler automatically defines *getters* and *setters* that access the various values embedded in the instances regular slots. Accessing virtual slots is syntactically identical to accessing plain slots, but virtual slots differs of regular slots by :

– they are not allocated into memory.
– their getters and setters are not generated by the compiler but defined, by the user, in the class definition, using the new class field options : `get` and `set`.

For instance, let us consider a possible `rectangle` class implementation. An instance of `rectangle` is characterized by its origin (`x0`, `y0`) and either its upper right point (`x1`, `y1`) or its dimension (`width`, `height`). In the following `width` and `height` fields are virtual.

```
(class rectangle
   x0 y0 x1 y1
   (width (get (lambda (o)
                  (with-access::rectangle o (x0 x1)
                     (- x1 x0))))
          (set (lambda (o v)
                  (with-access::rectangle o (x0 x1)
                     (set! x1 (+ x0 v))))))
   (height (get (lambda (o)
                   (with-access::rectangle o (y0 y1)
                      (- y1 y0))))
           (set (lambda (o v)
                   (with-access::rectangle o (y0 y1)
                      (set! y1 (+ y0 v)))))))
```

Setting the `width` virtual slot (respec. the `height` slot) automatically adjusts the `x1` value and *vice versa*. No memory is allocated for `width` and `height`, as their *values* are computed each time they are accessed. The memory layout for `rectangle` instances is Figure 11.1.

Virtual Slots Implementation

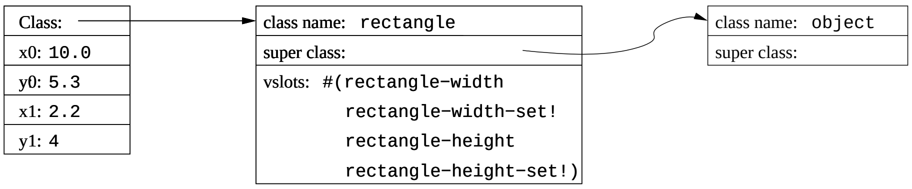


FIG. 11.1 – Rectangle memory layout

With a language provided with a Meta Object Protocol [123] (henceforth MOP) (such as CLOS or STk [88]) virtual slots implementation is straightforward. Meta object programming makes compilation harder and thus when designing Bigloo we have decided to not include it. So, Bigloo contains a builtin implementation for virtual slots that looks like a possible implementation for methods of a Smalltalk like language. Each class contains a vector of virtual slots (henceforth the *class virtual vector*). Accessing virtual slots fetches from the virtual vector the correct function (the offset is computed statically) and calls it. That is accessing a virtual slot of an instance `i` is :

```
(let ((vvec (class-virtual-vector (object-class i))))
   ((vector-ref vvec <a-static-offset>) i))
```

Obviously, accessing virtual slots is more expensive than accessing regular ones : the overhead is one additional memory access and one additional function call. However, the extra memory read can be removed if each instance is provided with an extra slot pointing to the virtual vector of its class. Bigloo does not use this framework in order to maintain class instances as small as possible.

11.1.7 Generic functions and methods

Generic function declarations are function declarations annotated by the `generic` keyword. They can be exported : this means that they can be used from within other modules and that methods can be added to those functions from other modules. They can also be static, that is not accessible from within other modules. In this case, no method can be added to the ones introduced by the module that defines the generic function. The Clos model fits harmoniously with traditional functional programming style because a function can be thought as a generic function overridden with exactly one method. The syntax to define a generic function is very similar to an ordinary function definition :

```
(define-generic (fun::type arg::class ...) optional-body)
```

Generic functions should have at least one argument as it is the first argument that is used to solve the *dynamic dispatch* of methods. This argument is of a type T that must be associated to a class and it is impossible to override generic functions with methods whose first argument is not of a subtype of T. Methods are declared by the following syntactic form :

```
(define-method (fun::type arg::class ...) body)
```

Defining a method if there is no defined generic function with a compatible prototype (arguments and result types in a sub-typing relation with those of the generic function) is an error. Methods override generic function definitions. When a generic function is called, the most specific applicable method (that is a method defined for the closest dynamic type of the instance) is dynamically selected. A method may explicitly invoke the following most specific method overriding the generic function definition for one of its super classes (a class has only one *direct* super class but several *indirect* super classes) by the means of the `(call-next-method)` form. It calls the method that should have been used if the current method had not been defined.

Here is an example of a generic function that illustrates the use of the Bigloo object layer. We are presenting a function that dumps the value of the slots of the `point` and `point-3d` instances. This generic function is named `show` :

```
(define-generic (show o::point))
```

Then, the generic function is overridden with a method for classes `point` and `point-3d`.

```
(define-method (show o::point)
   (with-access::point o (x y)
      (print "x=" x " y=" y)))

(define-method (show o::point-3d)
   (with-access::point-3d o (z)
      (call-next-method)
      (print "z=" z)))
```

Hereafter, is an example of a call to the `show` generic function.

```
(let ((p (instantiate::point-3d (x 10) (y 20) (z 465))))
   (show p)) ⟹ x=10 y=20 z=465
```

11.2 The Biglook library

Biglook is a graphical library which provides an extensive set of widgets. These widgets allow the creation of complex Guis with minimal effort. In general, when building such interfaces, one of the main difficulties lies in trying to clearly separate the code of the interface from the rest of the program. Making the Gui code independent from the rest of the application is important because :

– GUIs are often built on a *trial-fail* basis. It is hard to conceive an interface *ex-nihilo* and it is generally after using it for a while that the elements of the GUI find their place. Keeping the code independent from the rest of the application allows the development of prototypes of the interface without nasty consequences on the other parts of the program.
– A given program can have several interfaces according to the device on which it is ran (e.g. graphical screen, PDA, alphanumeric terminal). With an independent interface code, different interfaces can be "plugged" to the same program.
– The GUI of an application can be constructed interactively by an interface builder. In such a case, it is preferable to keep the mechanically generated code separated from hand-written code.

Biglook uses a declarative model for the construction of GUIs. This permits a clear separation between the code of the interface and the code of the application. The construction of an interface starts by declaring the various widgets which compose it. The graphical appearance of the widgets (color, font, size, ...) is generally set at this time. Once all the components of the GUI are created, they are placed on the screen by specifying a set of constraints expressing the relative layout of these components. Finally, the behavior of each widget is specified independently of its creation by associating an action (a Scheme closure) to a given event (key pressed, mouse click, mouse motion, ...).

In this section we first present how to create and place widgets, then we show the graphical event management. In Section 11.2.4, we present the Biglook classes. At last, in Sections 11.2.5 and 11.2.6, we present how simple and composite widgets can be implemented in Scheme.

11.2.1 Widget Creation

The graphical objects (i.e., *widgets*) defined by the Biglook library such as menus, labels or buttons are represented by Bigloo classes. Each class defines a set of slots that implement the configuration of the instances. Consequently, tuning the look of a widget consists in assigning correct values to its slots. The library offers *standard* default values for each widget, but theses values can of course be changed. Generally the customization is done at widget creation time, as shown in the following example :

```
(let ((lab (instantiate::label
             (text "A simple label")
             (background "grey")
             (foreground "red"))))
   ...)
```

Here, a new widget is created with its slots set to the standard default values defined by the `label` class, except the slots `text`, `background` and `foreground` for which a specific value is provided. A particular aspect of a widget can be changed by setting a new value to its corresponding slot. For instance, the expression

```
(with-access::label lab (background)
   (set! background (lighter-color background)))
```

changes the background of the previously defined `lab` widget to slightly lighter color than the one it used to contain.

Biglook library uses the object paradigm and heavily uses the inheritance mechanism. For instance, Biglook buttons are defined by the class `button` which directly inherits from the `label` class and provides them some additional slots such as the `command` slot which contains the action to be executed when the left mouse button is depressed over it.

11.2.2 Widget Placement

Widgets are not mapped on the screen at creation time. *Geometry managers* which ensure that user graphical constraints are always satisfied are in charge of this task. Biglook offers several geometry managers and an application can use them simultaneously. Since Biglook re-uses the geometry managers of its native back-end we cannot take credit for their high quality. In this paper we merely present an overview of only one of them, the *packer*. For instance, the following form :

```
(let ((fr   ...)
      (lab1 (instantiate::label
                (parent fr)
                (text "One")
                (height 4)
                (background "blue")))
      (lab2 (instantiate::label
                (parent fr)
                (background "red")
                (text "Two") )))
   (pack lab1 :side "left")
   (pack lab2 :side "right" :fill "y"))
```

specifies that the label `lab1` must be placed at the left of its parent (the `fr` container) and that the label `lab2` must be placed at the right. The packer's `:fill` option specifies that the `lab2` label will extend itself vertically if there is sufficient room to do so (this is the case here since `lab1` has requested a height of 4 lines).

Geometry management is declarative and the user generally does not need to compute the layout of the components of the interface. This is handled by the library which recomputes the graphical layout when the system is idle.

11.2.3 Event Management

Graphical events (mouse click, key pressed, ...) can be associated to actions by the means of the `event-case` form :

```
(event-case s-expr
   ((event-descriptor) s-expr)
   ((event-descriptor) s-expr)
   ...)
```

Library widgets have some predefined behavior in order to minimize the user's tasks. For instance, buttons launch the procedure associated to their `command` slot when there is a mouse click, or texts are scrolled when their associated scrollbar is used. However, these actions are not hard wired in the library and can be changed easily, if needed. The following `event-case` form shows how to make the widget `wdgt` reactive to the left and right mouse buttons (which are characterized, in Biglook, by the `"<Button-1>"` and `"<Button-3>"` strings) :

```
(event-case wdgt
  (("<Button-1>") (print "Button-1"))
  (("<Button-3>") (print "Button-3")))
```

The above `event-case` form notifies the Biglook back-end to associate the closures

```
(lambda () (print "Button-1"))
```

and

```
(lambda () (print "Button-3"))
```

to the event handler of the object `wdgt`.

The closures associated to event handler benefit from the expressiveness of Scheme procedures. They can capture their creation environment. So, to count the number of time `wdgt` is "clicked", one may write :

```
(let ((count 0))
   (event-case wdgt
     (("<Button-1>") (set! count (+ count 1))
                     (print "Pressed:" count))))
```

Some predefined functions retrieve various informations about the system state when the event occurred (mouse coordinates, values of the modifier keys, ...). For instance, the following code prints mouse coordinates each time it enters or leaves `wdgt` :

```
(event-case wdgt
   (("<Enter>") (print "Enter:" (the-event-x) ":" (the-event-y)))
   (("<Leave>") (print "Leave: " (the-event-x) ":" (the-event-y))))
```

Each Biglook widget can have some *tags* associated to it. Such a *tag* can also be shared between several widgets. The form `event-case` can be used on a tag instead of a widget, allowing to associate a behavior to a set of widgets rather than to a specific one. For instance :

```
(event-case "emacs-binding"
   (("<Left>" "<Control-b>") (backward-char (the-event-widget)))
   (("<Right>" "<Control-n>") (forward-char (the-event-widget)))
   ...)
```

associates a behavior to all the widgets which share the `"emacs-binding"` tag.

When an event is raised for a given widget, several expressions may be evaluated, since an expression can be added to this particular widget as well as to a tag it owns. By default, all the expressions associated to a widget will be evaluated when the event is raised. This list of expressions and the order in which they will be evaluated can be changed by the user with the library procedure `event-order-set!`. If one of these evaluations returns `#f`, the Scheme false value, the expressions next to it in the list are not evaluated.

The ordering mechanism on events allows the user to easily enhance the library components behavior by adding some control before or after the standard ones. Suppose, for instance, that we want to define an entry (a one line editor) which only accepts digits. The Biglook library defines a set of standard behaviors for entries, under the tag `"entry"`. In particular, all the non control characters are inserted at the cursor position and the content of the entry is scrolled if the cursor is out of scope after insertion. By default, the evaluation order for an entry e is a list formed of e, the tag `"entry"` and the tag `"all"` (which is shared by all the widgets and used to give them a standard default behavior). So, the e specific evaluation take precedence over the ones defined for the tags `"entry"` or `"all"`. The tag mechanism can be used to control the key pressed on our new *integer entry* in order to automatically reject non number inputs :

```
(event-case "num-entry"
   (("<KeyPress>") (char-numeric? (the-event-char))))

(let ((e (instantiate::entry)))
   (event-order-set! e (list e "num-entry" "entry" "all")))
```

The `event-order-set!` procedure is used here to add a new expression evaluation in front of the standard ones.

11.2.4 Biglook Classes

Each Biglook widget is defined by a specific class of the library. Some of theses classes correspond to Tk widgets, possibly enhanced, and some correspond to classes which are a composition of several basic widgets. A partial hierarchy tree of the library is presented in Figure 11.2. In this figure, exported classes are in rounded boxes and composite widgets are grayed. As we can see, all the classes share a common ancestor : the `bglk-object` class. This class defines the slots which are necessary to access a builtin object from Scheme. In particular, three important slots are defined in this class :

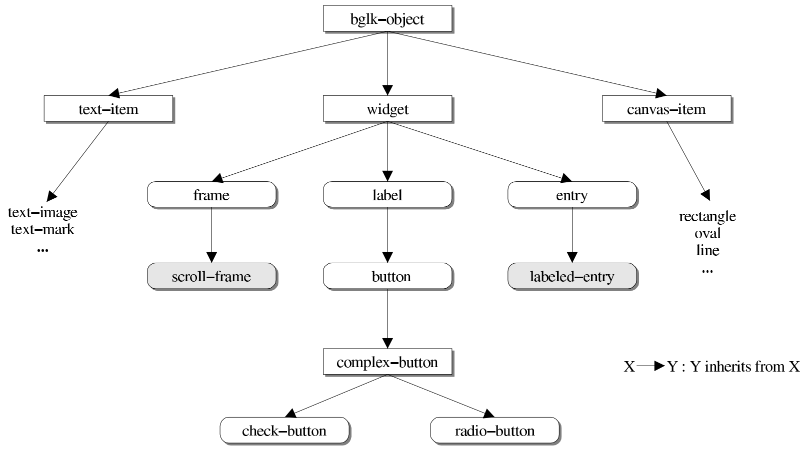


FIG. 11.2 – *A partial view of the Biglook hierarchy*

– The `builtin-widget` slot contains a reference to the *builtin-descriptor*. This reference is created when the instances are initialized.
– The `container` slot denotes the *builtin-descriptor* of the widget container.
– The `parent` slot contains a reference to the object which (graphically) includes the instantiated object.

Even if the back-end toolkit does not rely on an object oriented model, Biglook provides this vision to the end user.

The implementation of some widgets can use some additional material, or provide options which do not exist in their back-end counterparts. For instance, several Biglook widgets can have a help balloon which is displayed when the mouse stays inactive for a while on one of these widgets. Help balloons do not on some back-ends. In these cases, they are implemented by means of Biglook widgets, otherwise, Biglook balloons are mapped into native balloons.

11.2.5 Simple widgets

A simple widget is a widget which is directly mapped into a builtin widget. All simple widgets are implemented according to the same framework : they inherit from the `widget` classes and they define user customization options. These customizers are implemented by the means of virtual slots. We present here a possible implementation of the `button` class. For the sake of simplicity, we suppose that this class extends the `label` class with only one additional slot : the `command` slot which allows the user to specify the closure which must be applied when the button is clicked.

```
(class button::label
   (command
    (get (lambda (o::button)
            (with-access::button o (builtin-widget)
               (<builtin-button-command> builtin-widget))))
    (set (lambda (o::button v::procedure)
            (with-access::button o (builtin-widget)
               (<builtin-button-command-set!> builtin-widget v))))))
```

Implementing the `command` slot requires virtual slots. Its getter and setter functions directly interact with the back-end toolkit.

Now that the slots of a button widget are defined, we must define how such a widget has to be initialized. The class initialization specific code is given to the system via the generic function `realize-widget`. For each class of the library a method overrides the generic function and must call the back-end to create the graphical object associated to the class. For a button, the method we need to write is :

```
(define-method (realize-widget o::button)
   (with-access::button o (builtin-widget container)
      (set! builtin-widget (<builtin-make-button> container))))
```

This method creates a builtin button via the low level <*builtin-make-button*> function and store the result in the `builtin-widget` slot. Note that when `realize-widget` is called the `container` and `parent` slots defined above have already been initialized by the runtime system. On the contrary, the virtual slots of a button (i.e. those of the inherited `label` class such as the `font` of `text` slots as well as the `command` specific slot) are assigned after the widget is realized. Hence, the parameters passed by the user when the widget is instantiated can be applied to the newly created builtin widget.

11.2.6 Composite widgets

Most of the Biglook widgets are composite, that is, widgets made by composing simpler ones. A composite widget is completely written in Scheme and is generally built by inheriting from an existing widget. One of the simplest composite widget is *labeled entry*. Such a widget is a small line editor with a label on its left indicating the kind of value that must be entered in the editor.

These widgets can be easily implemented with two basic builtin widgets : a *label* and an *entry*. The `labeled-entry` class is defined in the Biglook library and we show here how to implement a simplified version.

The main difficulty is to determine the inheritance scheme to be used. Often, composite widgets are standard widgets augmented with additional graphical material (for instance a *text with a scrollbar* widget inherits its behavior from `text` and not from `scrollbar`). For the `labeled-entry` we consider it as a specialized version of an `entry` with a `label` as decoration. This observation yields to the structure of the `labeled-entry` class : it inherits from the `entry` class and provides a slot containing a reference to a `label`. The last design issue is to select which slots will be defined for `labeled-entry` instances. They can be characterized according to three categories :

- The ones that change the properties of the inherited class. This is a classical object problem which can be solved by a slot redefinition. In the case of a *labeled entry*, for instance, we could wish to redefine the `relief` property (which is a graphical attribute of widgets) to its container rather than the one of the entry, as implied by our inheritance scheme.
- The ones that give access to a property of a component of the composite which is not the main one (i.e. a decoration element). This can be easily achieved with a virtual slot which accesses a property of a real slot of the widget. For instance, the slot `title` of a labeled-entry can be implemented by a virtual slot giving access to the slot `text` of the embedded label.
- The ones that propagate a property to some components of the new class. This is very frequent in composite widgets, since we generally want that the decoration elements of the composite are affected in the same way as the main elements. For instance, changing the background color of a `labeled-entry` implies changing the color of the entry as well as its container and associated label. Slot propagation is generally implemented by the means of a virtual slot.

A possible implementation for the `labeled-entry` class could look like :

```
(class labeled-entry::entry
   (label (default #unspecified))
   ;; the container relief
   (relief
    (get (lambda (o)
            (container-relief (labeled-entry-container o))))
    (set (lambda (o v)
            (container-relief-set! (labeled-entry-container o) v))))
   ;; the text of the label
   (title
    (get (lambda (o)
            (label-text (labeled-entry-label o))))
    (set (lambda (o v)
            (label-text-set! (labeled-entry-label o) v))))
   ;; propagated on all components
   (background
    (get (lambda (o)
            (call-next-slot)))
    (set (lambda (o v)
            (call-next-slot)
            (container-background-set! (labeled-entry-container o) v)
            (label-background-set! (labeled-entry-label o) v)))))
```

As for methods, it is necessary to supply a way to call the virtual slot accessors of a super class of an instance. This is the role of the `(call-next-slot)` form used in the above example. This form can only take place inside a virtual and setter definitions. Because Bigloo requires the accessing of all the class declarations in order to compile a module, it can detect illegal `(call-next-slot)` usages. This form can only appear inside virtual slots getter and setter definitions. The compiler is then able to check if that virtual slot has already been defined for a super class (direct or indirect) of the current class. If not, a compile-time error is raised.

The `realize-widget` method of labeled entry must first call `call-next-method` in order to ensure a proper realization of the `entry` and then create the decoration `label` which reuses the container of the `entry` to be effectively *embedded* in the `labeled-entry`. The code for this method is :

```
(define-method (realize-widget o::labeled-entry)
  (call-next-method)
  (with-access::labeled-entry (label container)
    (set! label (instantiate::label (container container)))))
```

The way the components of composite widgets are graphically placed in their container still has to be determined. This is the role of the `set-widget-layout!` generic function which ensures the layout of all the components inside a container. For a `labeled-entry`, it could be overridden as follows :

```
(define-method (set-widget-layout! o::labeled-entry)
  (with-access::labeled-entry (label)
    ;; label on left and entry on right
    (pack label :expand #f :side 'left))
    (pack o     :expand #t :side 'right)))
```

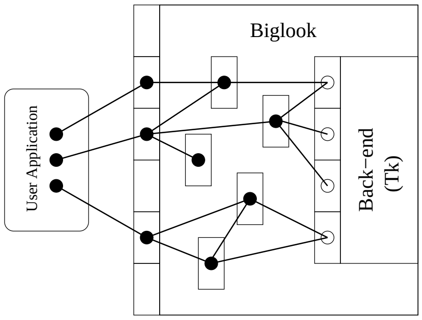
to ensure that the label is placed on the left of the entry component.

11.3 Implementing Biglook

In this section, we present the overall Biglook architecture. Then, we detail the role of *virtual slots*. In a third and fourth part we present the current Tk back-end and the forthcoming GTK+ back-end. We conclude this section with a performance evaluation of Biglook.

11.3.1 Library Architecture

The Biglook library is implemented on top of a native *back-end* toolkit (currently Tk). It takes advantage of the efficiency of the *back-end* low level operations. The Figure below illustrates the Biglook's software architecture.



Biglook extends the native back-end by supporting new original widgets. Actually, more than half of the Biglook widgets are completely implemented in Scheme and are, consequently, independent of any *back-end*. This is essential in order to keep the toolkit as independent as possible of the back-end toolkit and to envision the use of another back-end, if needed. We strongly feel that this independence of the back-end is one of the major strengths of our library. As a matter of fact, there are today a lot of efforts in designing new toolkits for modern desktop environments and it is important for an end-user to be sure that his applications can be adapted to a given environment with minimal effort. In particular, the two main competing desktop environments which are gathering momentum in the Open Source movement, KDE [57] and Gnome, are built on top of the toolkits Qt [56] and GTK+ [158] which rely on very different models.

Because Biglook makes few assumptions on the underlying toolkit, re-targeting its implementation to another back-end is generally *possible*[2]. To be a potential Biglook back-end, a toolkit must provide :

- a way to identify a particular component of the interface. Of course all the toolkits provide this, even if the representation used can be very different, such as an integer, a string, a function, and so on. In Section 11.2.4, we have called this identifier `builtin-widget`.
- the notion of *widget container* (`container` in Section 11.2.4). Here again, this notion can be different for each toolkit. Some toolkits provide a container for each widget ; some, as Tk, do not. In the latter case, we can consider that a widget is its own container or wrap each Biglook widget in a back-end container.
- an event manager which does not hide the events in its internals. Our event system is clearly based on the Tk one, but it can be easily simulated on any toolkit which exposes major events to the user (mouse or key events and window sizings) with a set of hash tables containing the closures of the handlers.

These points are generally available, or at least possible to simulate, with the major modern toolkits. We will see in Section 11.3.2 that using virtual slots allows the building of the rest of the library on this minimal basis. A prototypical port of the GTK+ library will also be presented in 11.3.4.

11.3.2 Virtual Slots and Biglook

Virtual slots are the heart of the Biglook implementation. Simple Biglook widgets are represented by a class containing three regular slots : a pointer to a builtin associated widget, a pointer to a builtin container for that widget and a pointer to its parent in the graphical hierarchy (the slots `builtin-widget`, `container` and `parent` presented Section 11.2.4). All additional slots are

[2] *Possible* does not mean that it is always *easy*.

implemented by the means of virtual slots. For instance the slot `background` is implemented by the means of a virtual slot for which the getter just calls the builtin `background` getter with the builtin object as argument. The `button` class declaration looks like :

```
(class bglk-object
   (builtin-widget read-only)
   (container read-only)
   (parent read-only))

(class widget::bglk-object
   ...)

(class label::widget
   (background
    (get (lambda (o)
            (<builtin-background-getter> (button-builtin-widget o))))
    (set (lambda (o v)
            (<builtin-background-setter> (button-builtin-widget o) v)))))
```

The memory layout for `button` instances is presented in Figure 11.3.

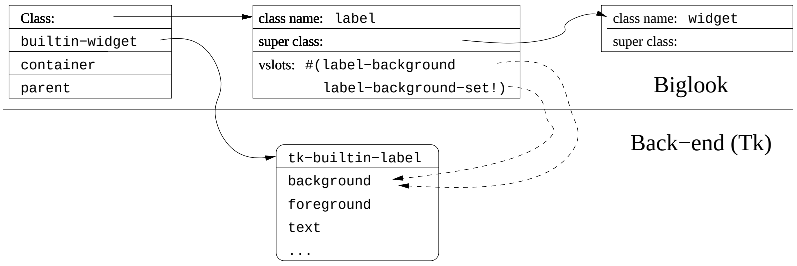


FIG. 11.3 – `Label` instances memory layout.

11.3.3 Tk back end

Tk assumes a command language, Tcl [154] and references to the underlying Tcl interpreter are spread all around its source code. As a consequence, most of the languages embedding Tk link it with the standard versions of the Tk and Tcl libraries. Biglook uses a different approach that was pioneered in STK [87], where the library has been modified to use Scheme as its support language instead of Tcl. This implementation strategy avoids linking applications against the large Tcl library and it enables a better integration with Scheme. In particular, our modified library uses Scheme types where needed (i.e. boolean operations return the values `#f` and `#t` rather than `0` and `1`, strings are always quoted and are conform to Scheme conventions, ...). Furthermore, the Tk library has been extended to accept Scheme closures as call-backs and to cooperate with the Scheme garbage collector.

When a new widget is created, Biglook creates an opaque Scheme object that we call a *builtin-descriptor*. This is a special kind of Scheme object which is used to uniquely identify the widget and to allow its customization via the `tk` library function. This function is the connection link between the Biglook library and the Tk toolkit. It permits a low level access to all the widgets properties which are exported by Tk. The following example shows how a button can be created and customized at the very internal low level of the Biglook library.

```
(let* ((f ...)
       (opaque (tk-builtin-button :parent f)))
   ...
   (tk opaque 'configure
              :text "Hello"
              :command (lambda () (print "f=" f)))
   ...)
```

The function `tk-builtin-button` is a (private) function of the library which makes a stock Tk button and returns a *builtin-descriptor*. It accepts the `:parent` keyword parameters that specifies the container which graphically embeds the newly created object. As we mentioned before, the modified version of the Tk library has been extended to take into account Scheme values and it will protect them against garbage collections during the `b` button life-time. This modification, for instance, enables Scheme `lambda` expression to be passed to the `tk` function.

11.3.4 GTK+ back-end

Two independent reasons have motivated this additional back-end for Biglook :

- The Tk execution model requires complex conversions for Scheme objects to suit the Tk format based on strings. These conversions have an impact on the Biglook applications : they consume more memory than strictly needed by the interfaces. Changing for a more neutral widgets library could get rid of these extra allocations.
- Tk does not evolve anymore. No new builtin widgets are added. Missing modern widgets is cumbersome because implementors of GUI definitely want their applications to be made of fancy modern visual effects. To some extent, we are able to maintain Biglook to the high standards of modern applications (for instance, Biglook contains sophisticated *toolbar* widgets which have no counterpart in Tk) but this requires large development times. Switching to a library already supplying these facilities will ease our development.

Two libraries have quickly emerged as possible replacement for Tk : Qt and GTK+.

- Qt is a modern widget toolkit that supplies fancy modern widgets and facilities. However, we have decided not to use Qt because it offers a C++ API. Interfacing Scheme code and C++, if possible, is much more difficult than interfacing Scheme and C.
- GTK+ is our choice. That library combines many advantages. It has a neutral C API, it is actively maintained and developed. To some extent, GTK+ advocates connections to high level languages (e.g., Eiffel and Caml).

We are currently testing the re-targeting of Biglook. The prototype already contains some builtin widgets (e.g., windows, buttons, entries, ...). This is sufficient to test some simple applications. Some old Biglook applications can be recompiled and linked against the new library. Being still under development, we cannot draw definitive conclusions yet. However, we can already state that :

- The GTK+ back-end has not been tuned for efficiency yet but we have already seen that applications linked against it allocate less memory and run faster than their Tk counterparts.
- Many widgets available in Tk or GTK+ are roughly equivalent. For instance buttons, entries, windows exist in both libraries and are as customizable. All these widgets are made available by both Biglook back-ends.
- Tk and GTK+ geometry managers are similar and Biglook can provide the very same API for widget placement with both back-ends.
- We have not implemented yet the `event-case` form for GTK+ but it seems that GTK+ enables fine control over X events. It is thus likely that `event-case` could be implemented with the same API for GTK+.

11.3.5 Biglook performance evaluation

The main purpose of this Section is to estimate the dynamic performance degradation due to the Biglook layered architecture. In other words, this section mainly compares Biglook performance to Tk performance. We also present some time figures for two other toolkits (GTK+ and Qt) in order to show that Biglook performance could be improved by changing its back-end. To position Biglook, we compare it with other systems supplying for GUI programming.

All widget toolkits we have tested are *sufficiently* fast. That is, on modern architectures, the GUIs implemented using all these toolkits are *fluid* : there is no delay associated to graphical commands. However, we have found that applications making use of GUIs could consume a lot of memory. This is cumbersome when several of these applications are simultaneously running on the same computer. For a multi-task and multi-user platform we think it is very important for applications to be as sparse as possible. For this performance evaluation we have thus decided to focus on memory consumption instead of execution time.

The benchmark

To evaluate the performance of several toolkits we have designed a simple *benchmark* interface (see Figure 11.4) which is simple enough to be implemented by the various toolkits (Tk, GTK+ and Qt). To avoid testing specific widgets implementation we have restricted it to very primitive graphical elements. That is, our benchmark only uses windows, containers and buttons. Actually it is made of one window, five containers, nine exclusive buttons, two toggle buttons and two plain buttons. We have measured the execution time and the memory allocated when creating 5000 times this interface. These graphical elements are positioned on the screen by the standard graphical constraint satisfyer (i.e. the respective Tk, GTK+ and Qt packers).

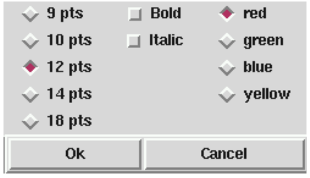


FIG. 11.4: The *benchmark* interface

Native toolkits

Programs have been executed on a Intel Pentium III 500Mhz (Katmai) running Linux 2.2.x.

	Toolkits			
Resources	Tcl/Tk	Qt	GTK+$_{min}$	GTK+$_{max}$
Allocations	75 MB	30.8 MB	23.2 MB	43.1 MB
Exec. time	104.6 s	24.5 s	8.7 s	12.7 s

GTK+ figures deserve some explanations. GTK+ applications can be configured at runtime using *themes*. A theme is a global user configuration for all the running GTK+ applications. That is some graphical configurations are not implemented in the applications themselves, but globally. It appears that the resources consumed by GTK+ applications are highly dependant on the theme in use. Execution reported as GTK+$_{min}$ uses the less resources demanding theme while execution reported as GTK+$_{max}$ uses the most demanding one. We have been unable to understand why themes increase resources demand. For instance, in spite of our expectations we have found that one of the most demanding theme hardly changes the graphical shape of the objects. It uses no picture and it only applies a green coloring instead of the default gray one!

Tk consumes more memory than any other toolkit. We think the reason only lies in that Tk is more configurable than any other toolkits. For instance, buttons are characterized by 44 attributes in Tk, 28 in GTK+ and 17 in Qt. The consequence is that on a 32 bits architecture the memory consumption of a button is about 600 bytes for Tk, 120 for GTK+ and 280 for Qt.

Tk vs Biglook

The main purpose of these performance measurements is to estimate the overhead introduced by the Biglook architecture. We have thus measured the benchmark application in Biglook too.

	Toolkits	
Resources	Tk	Biglook
Allocations	70 MB	85 MB (+20%)
Exec. time	–	60 s

The Tk column only reports allocations operated by Tk. That is, we have here stripped off the allocations required by Tcl. Biglook allocation overhead is of 20% for the benchmark application (but only 13% when compared to Tcl/Tk). We have instrumented Biglook application in order to understand where this overhead was coming from. The 15 additional megabytes allocated by the Biglook application are composed of : 9.0 MB of strings, 2.0 MB for the instances of the Biglook `button` classes and 1.0 MB for the instances of the `window` and `container` classes. The remaining 3.0 MB are due to bookkeeping.

Biglook allocates many strings because Tk is designed to be coupled with a shell-like language. That is, all Tk functions, even if implemented in C, requires string arguments. In other words, each time Biglook requests a service from Tk, it must convert the Scheme passed values into C strings. This is the main overhead of the Biglook library. We think that re-targeting Biglook to another native toolkit that is not using such a parameter passing protocol would save about 7.0 MB. We thus estimate that the Biglook overhead would be shrunk to 10%.

The execution time (60.0 s for Biglook to be compared with the 104.6 s reported for Tcl/Tk) only shows that Bigloo compiled code speed up eliminates the slowdown introduced by the additional allocations.

Because Biglook maintains memory allocations of graphical application relatively small and because Biglook applications are efficient, Biglook can be used to implement complex graphical applications. For instance, we use it on a daily basis to implement the graphical interfaces of Bee programming environment [209, 210]. Annex B presents some screen shots of complex graphical interface. Each of them uses several hundred of widgets.

11.3.6 Other Scheme widget libraries vs Biglook

We have implement our benchmark interface using three other main Scheme widget libraries :

- STk which relies on a Scheme interpreter extended with a widget library. The widgets are available by the means of a class hierarchy. STk uses Tk for its native back-end.
- MrEd [82], a part of the DrScheme project [73], which is a programming environment that contains an interpreter, a compiler and other various programming tools such as browser, debugger, etc. The execution time figures reported for DrScheme have been measured using a compiled version of our benchmark. The back-end toolkit used by Rice's MrEd is wx-Window [223] system a toolkit that is available under various platform (Unix, Windows, etc.).
- SWL which is a contribution to the Petite Chez Scheme system [68]. It relies on an interpreter. It uses Tk has back-end. Native Tk widgets are mapped to Chez Scheme classes.

	Toolkits			
Resources	STk	MrEd	SWL	Biglook
Allocations	140 MB	535 MB	155 MB	85 MB
Exec. time	62.5 s	138 s	94 s	60 s

- STk execution time is nearly the same as the Biglook one, even if it allocates more memory. The larger STk memory consumption comes from its object model which implies explicit construction of meta objects at runtime. Furthermore, this construction implies, in general, a rather large time overhead. In the particular case of this benchmark, this overhead is compensated by the way STk initializes widgets which is here more efficient than the Biglook one : using the MOP, STk creates and initializes the various slots of a widget in single call to the Tk toolkit whereas Biglook uses its standard slot initialization scheme which implies several calls to the Tk library.
- MrEd allocates more memory than the other systems. The main reason seems to be the implementation framework for classes instances : each instance contains a private copy of the class methods. Since classes used to implement widgets contains many methods, MrEd widgets tend to be larger than with other systems. These numerous memory allocations probably explain the relative slow performance of MrEd on the benchmark.

– SWL allocates nearly twice the memory allocated by Biglook (155 MB for SWL to be compared to the 85 MB of Biglook). These extra allocations could be responsible for its execution time penalty. We have examined the SWL's source code. Our conclusion is that SWL explicitly constructs Tcl expressions that are sent to the regular Tcl evaluator. By contrast, Biglook does not embed a Tcl evaluator. Tcl functions are replaced with Biglook compiled functions. This implementation design avoids, in some extent the construction of strings of characters that represent the graphical action to be executed.

11.4 Principles of Gui programming languages

Designing and implementing Biglook raised two questions concerning programming languages for graphical interface programming :

– what are the mandatory constructions to implement a toolkit library ?
– what are the mandatory constructions user applications may use ?

To answer these questions, we have compared functional programming and object-oriented programming applied to Guis. This comparison has been made possible because Bigloo is provided with both models. For each feature we were implementing we have been able to experiment both models in order to select the most efficient one (i.e., the most elegant or the most compact). This section presents the conclusion we have drawn from these experiments. We first present the role of n-ary functions and keyword parameters. Then, we show why closures are convenient to implement call-backs. In Section 11.4.3 we present why generic functions permits large extensibility. Then, we conclude this section with a comparison between virtual slots vs member functions and a discussion about introspection.

11.4.1 N-ary functions and keyword parameters

N-ary functions (functions accepting a variable number of arguments) and keyword parameters (parameters that can be passed in any order because their actual value is associated to a name) are *mandatory* features in order to enable declarative style for Gui applications. For instance, a Biglook plain button is characterized by 24 slots ! Some of them describe the graphical representation (colors, border sizes, ...), some others describe the internal state of the button (*parent*, associated value, associated text, ...). In general, these numerous slots have default values. When an instance is created only slots that have no default values must be provided. Slots are initialized with their default value unless a user value is specified. As a consequence, the form that operates widget construction (e.g., class instantiation) must accept a variable number of arguments : only some slot values must be provided, others are optional. In addition, because widget constructors accept a large number of parameters, it is convenient to *name* them and to pass them in any order. This is made possible in Biglook by the `instantiate::` form. Because they are implemented using macros that are expanded into calls to the class constructors where each declared slot is provided with a value there is no run-time overhead associated to `instantiate::` forms. Some examples of Biglook widget creations may be found in Section 11.2.2.

Lacking N-ary functions or keywords disables declarative programming style for Guis because widgets have to be created and, in a second step, specific attributes have to be provided. Even overloading and class constructors can't help. Let's suppose our button classes implemented in Java Awt [96] or Swing [253]. To enable a full declarative style, we should provide the button class definition with 2^{24} constructors (a constructor for each possible combination of provided slots). Even for much smaller classes, this is impractical because, in general, overloading only dispatches on types and slots can have the same type (for instance, the `width` and the `height` of a widget).

11.4.2 Closures for easy call-backs

Most modern widget toolkits (exception made of Qt) use a call-back framework. That is, user commands are associated to specific events (such as mouse click, mouse motion, keyboard inputs, ...). When an event is raised, the user command is invoked. We think that closure is the most simple and efficient means to implement call-backs even if some alternatives exist.

Languages provided with functions without environment

The C programming language is of such kind. ISO-C [115] enables global functions but no local functions. A C function is always top-level and may only access its parameters and the set of global variables. C functions have no definition environment. However, without an environment, a call-back is very restricted. In particular, a call-back is likely to access the widget that owns it. In GTK+ (a C toolkit) when a call-back is associated to an event, a optional value may be specified that will be passed to the call-back when the event is raised. This user value actually *is* the environment of the call-back. GTK+ mimics closures with its explicit parameters passing scheme. We may notice that the allocation and the management of the closure environment is in charge of the client application.

Languages provided with classes

For languages provided with classes such as Java, another strategy can be used. Call-backs may be implemented using class member functions. Member functions may access the object and the object's attributes for which they are invoked. Member functions look like closures. However, member functions are not closures because they are associated to classes. In other words, all the instances of a same class share the implementation of all their member functions. That is, different call-back implementations require different class declarations. For instance, if one wants to implement a button with a call-back printing a plain message and another one emitting a sound, two classes have to be defined ! These class declarations turn out to be an hindrance to simplicity and readability. In addition, if several events must be handled by one widget, this technique turns out to be impractical because it is not possible to define a new class for each kind of events the widget must react to (mouse-1, mouse-2, mouse-3, shift-mouse-1, ctrl-mouse-1, shift-ctrl-mouse-1, ...).

To avoid these extra class definitions, Java has introduced inner classes. An inner class is a class defined inside another class ; it may be anonymous. Because in GUI programming inner classes are used to implement call-backs and as they are numerous, Java proposes a new syntax that enables within a single expression, to declare and to instantiate an inner classe. For instance :

```
button.addActionListener(new ActionListener () {
   public void actionPerformed(ActionEvent e) {
      a user code that may reference current lexical bindings
   }
})
```

The expression `new ActionListener...` deserves some explanation : *1)* it declares an anonymous inner class that *2)* implements the interface `ActionListener` and *3)* it creates one unique instance of that new class that is sent to the method `addActionListener` of the `Button` instance. That is, anonymous inner classes are the exact Java implementation of closures ! It is worth pointing out that the Scheme syntax for closures is obviously more compact than the Java one.

Conclusion about closures

Closures are central to GUI programming because they are the most natural way to implement call-backs. As we have seen, all call-back based toolkits offer a mechanism similar to closures. It can be member functions of anonymous Java classes or extra parameter passed to C functions. However, we have found that all non functional programming languages solutions are not as convenient as Scheme `lambda` expression because either the user is responsible of the construction of the object representing the closure or extra syntax is introduced.

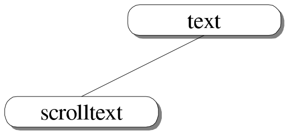
11.4.3 Generic functions

A generic function is a bundle of methods. Object behaviors are associated to generic functions instead of being associated to class definitions. Behavior specification is thus orthogonal to class definition. Generic functions enable behaviors to be associated *afterwards* to instances. When methods are defined inside the class definition, the behaviors of the instances of that class are specified once for all. Generic functions enable new behaviors to be specified anytime, anyplace. Generic functions permit user customizations that are hardly possible if methods are defined within classes.

Extending public widgets

In this Section we present by the means of an example how user applications may specify new behaviors for standard widgets. We present three scenarios : *1)* one based on generic functions *2)* another one using multiple inheritance and subtyping and *3)* another one using single inheritance and multiple subtyping and *4)* a last one using parametric types.

Biglook proposes several kinds of text widgets. Let us concentrate on two simple ones : plain text (`text`) and text provided with scrollbars (`scrolltext`). These widgets are represented by classes related by subtyping *and* inheritance relationships :



Each text widget has a cursor whose location is defined by two numbers : the line number and the column number. A user application may wish to represent the cursor position as Emacs does, that is, with one unique number which is a character number. Switching from an $X \times Y$ positioning to a linear positioning and *vice versa* is algorithmically easy.

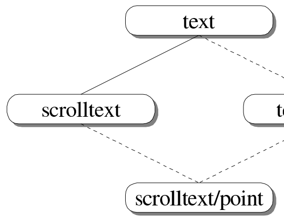
1. In order to implement linear positioning, provided with generic functions, a user application could simply define :

```
(define-generic (point::long t::text)
  (with-access::text t (cursor-x cursor-y line-width)
     ;; this is an over simplified implementation, because in
     ;; practice not all lines are of same width
      (+ cursor-x (* line-width cursor-y))))
```

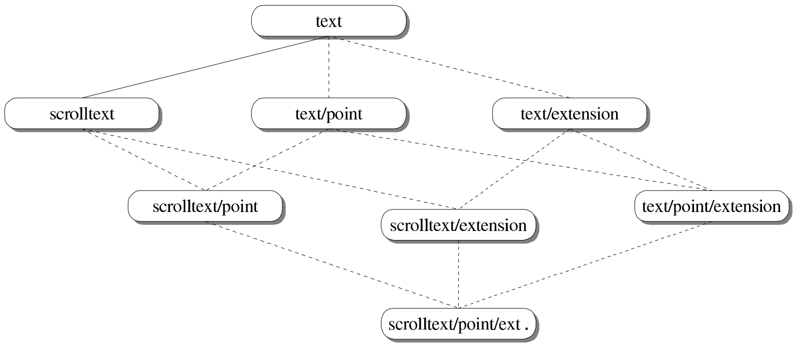
   This definition is valid for `text` and all its subclasses. It is thus valid for `scrolltext`. Without generic functions two solutions can be deployed.

2. If the language supplies multiple inheritance such as C++ or O'Caml [176][3], a user code may subclass `text` into a `text/point` class that will be provided with an extra `point` member function and will derive a new `scrolltext/point` inheriting from `text/point` and `scrolltext` :

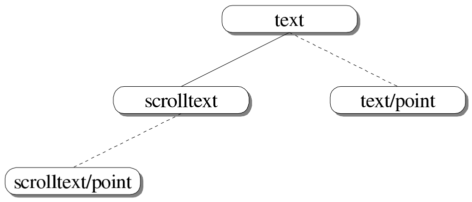
[3]One should note that because O'Caml supports parametric polymorphism, the `point` facility could be implemented using a plain function. This solution would avoid building new classes but it is not as powerful as generic functions because a plain function cannot be overridden.



If class declarations can be used, this solution is more complex than generic functions because *two* new classes have to be introduced for each new facility. If we want to add a second extension, we might have to introduce two additional classes and so on. This conducts rather rapidly to impractical inheritance schemes as shown below :



3. If the language does not provide multiple inheritance or offers only limited multiple inheritance such as Java, then adding our `point` facility, in user applications, is hardly possible. In Java, because `text` and `scrolltext` must be plain classes (by contrast to *interfaces*), it is impossible to define a class inheriting from both `text/point` and `scrolltext`. Thus a user application must use the inheritance tree :

   `Text/point` and `scrolltext/point` are no more in a subtyping relationship which means that an instance of `scrolltext/point` cannot be used when an instance of `text/point` is expected. Obviously, this is not as convenient and as powerful than the generic function based solution.

4. Parametric types such as C++'s templates [233] or MzScheme mixins [83] are a means to delay the concrete realization of classes. Parametric types help the program extensibility because actual classes are created in two steps : *i)* definition of the parametric class *ii)* effective realization of the concrete class. During step *ii* one may specialize concrete classes and inheritance relationships. Because parametric types is a means to generate classes, in many situations they can be used in place of multiple inheritance. However, extending the previous text example using parametric types would lead to a solution close to the one based on multiple inheritance. The restrictions of the multiple inheritance also applies to parametric types. In addition, as stated in the paper [77], parametric types require careful design in order enable extensibility. Failing to conform strict design rules compromises further extensibility.

Extending embedded widgets

When methods are defined in the class, adding a new service requires at least the declaration of a new extending class. However, this framework does not apply for already existing instances. This situation is frequent in object oriented libraries. For instance, the Biglook library proposes the `emacs` class which is defined as :

```
(class emacs::frame
   title::string
   window::frame
   minibuffer::entry
   ...)
```

Because most of the time, `emacs` frames are used to implement editors, the Biglook library offers the convenient `make-emacs-editor` function that instantiates an `emacs` frame and fills it with an editor and a set of default values for the other slots.

```
(define (make-emacs-editor title::string)
    (instantiate::emacs
       (title title)
       (window (instantiate::text))
       ...))
```

In this case, `text` instances used to fill the `window` slots are instantiated by the `make-emacs-editor` and the user has no means to control over their creation. So, it is impossible to add facilities to the embedded `text` windows, using previously described subclasses framework as in Section 11.4.3.

Because generic function are unrelated to class definitions, they can be used to extend already created instances. In our `emacs` example, a generic function could extend the facilities provided by the embedded emacs `window`. Complementary arguments in favor of generic functions may be found in papers by C. Chambers [41, 147]).

11.4.4 Virtual Slots vs Member Functions

Virtual slots *are not* Smalltalk, C++ or Java member functions. For languages advocating encapsulation, member functions use the classical way to access instance slots (called *data members* in C++) from outside the definition of the methods. Because member functions are user defined and because they make use of late binding (provided they are declared `virtual` in C++) they look like virtual slots. However, with languages provided with introspection such as Java, there is no way to distinguish amongst the set of methods which of them implement getters and setters and which implement user services. It appears that this distinction is mandatory in order to implement some graphical applications, such as interface builders that must be able to inspect the slots of each widget.

11.4.5 Introspection

It is tedious to implement a Gui because it requires a lot of graphical tuning that requires successive endeavors. Helping Gui programming is the task of *interface builders*. These tools rely themselves on Gui. They enable the available widgets of the toolkits to be graphically configured and inserted in the user interfaces. Interface builders must be aware of the whole available widgets and the whole configuration options of these widgets. If the toolkit implementation language is provided with introspection, builders can rely on it to present configuration panel for widgets. The existing configuration options are *automatically* available to builders by means of introspection. Lacking introspection forces the implementors of the interface builders to implement description files for widgets. Obviously this is error prone because the coherence between these files and the actual widgets implementation must be enforced by the interface builder programmer.

11.5 Extensions

This section presents the short term extensions that we envision, in addition to new back-ends, for the Biglook library and how we think that they can be implemented.

Themes A theme describes how the components of an application must be configured. It specifies a set of default values for the options of the underlying toolkit widgets. The Biglook library does not supply themes yet, but it is an extension we are considering to add. Using the Bigloo introspection mechanism, this extension should be easy to implement.

Drag and Drop This facility is generally available with modern desktop environments and toolkits. In particular, GTK+ and Qt natively offer DnD. Unfortunately, Tk does not define a standard mechanism for DnD and, consequently, we don't have yet a way to express it. However, the Tk event manager is sufficiently powerful to implement objects moving between Biglook applications.

Additional Operating Systems Currently, Biglook runs only on Unix systems. No other port has been envisioned yet. However, because of Biglook's neutral architecture, there is no technical difficulty to re-target it to another operating system : Bigloo generates ISO-C programs and uses portable graphical back-end (Tk or GTk have been ported to various systems such as Unix, BeOS, MacOS or Microsoft Windows).

11.6 Related work

Many functional languages are connected to widget libraries especially to the Tk toolkit. Few of them uses object-oriented programming. The only ones we are aware of are : *i)* Chez Scheme that proposes a connection to Tk where builtin widgets are mapped into Chez Scheme classes and *ii)* MrEd that maps wxWindow's widgets into DrScheme classes. We have already compared these systems to Biglook in Sections 11.3.5 and 11.4.3.

Other functional languages provide the graphical primitives using regular functions. The main contribution for strict functional programming languages has been developed for the Caml programming language [255].

The first attempt, CamlTk, is quickly surveyed in [181]. The design of CamlTk is different from the one of Biglook. CamlTk *binds* Tk functions in Caml while Biglook provides an original API made of classes. In addition, the implementation strategy also differs. CamlTk directly uses the Tcl `eval` facility while Biglook bypasses it. The cited paper does not present performance evaluation so it is not possible to conclude which approach performs the best.

Recently a new widget library based on GTK+ has been proposed for Caml. No article describes that connection. However, some work has been described to add keyword parameters to Caml in order to help the connection with widget libraries [85]. The philosophy of that work differs from our because that new library makes the GTK+ API available from Caml. No attempts is made to present a neutral API as we have done for Biglook.

Programming graphical user interfaces with lazy languages is far more challenging than with strict functional languages. The problem is to tame the imperative aspects of graphical I/O in such languages [152, 249]. Several solutions have been proposed : Fudget by Carlsson and Hallgren [32], Haggis by Finne and Peyton Jones [78], TkGofer by Vullings and Claessen [46] and, more recently the extension of the former TclHaskell library : FranTk by Sage [188].

Conclusion

In this paper we have presented Biglook, a widget library for the Bigloo system, an implementation of an extended version of the Scheme programming language. The software architecture of that library consists in a mapping of primitive builtin widgets into a set of high level Scheme classes. This enables compact and declarative implementations for GUIs that are independent of the builtin widgets. Currently Biglook uses Tk for implementing the builtin widgets but a GTK+

back-end is under development. Simple Biglook source code can be indifferently linked against the two libraries.

The paper presents time figures that show that the performance degradation imposed by the Biglook architecture is low (20% of additional allocations for the Tk back-end and 10% estimated for the GTK+ back-end). The difference in execution time is marginal. As a consequence, Biglook can be used to implement applications using complex graphical interfaces such as the one presented Annex B.

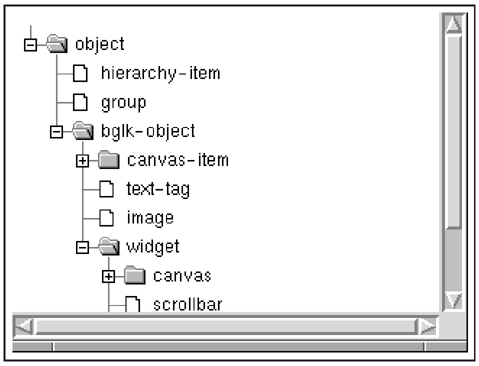
Acknowledgments

Many thanks to Jacques Garrigue, Didier Remy, Peter Sander, Matthias Felleisen, Simon Peyton Jones and to Céline for their helpful feedbacks on this work.

Annex A : Source code examplars

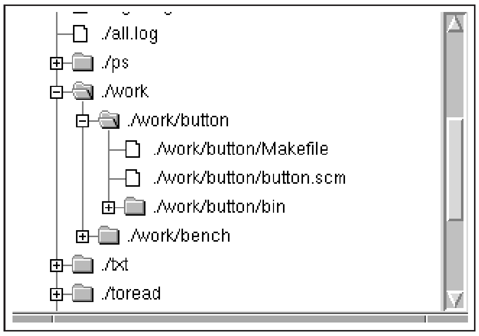
To illustrate how compact Biglook applications are, we present here three short demonstration programs. For each of them, we present a screen-shot of the running application followed by the ***complete*** source code of the application. The also present the actual of the benchmark presented Section 11.3.5.

11.6.1 Class browser




```
(module hierarchy-class
   (library biglook))

(define (get-class-children data)
   (map (lambda (x)
           (if (null? (class-subclasses x)) x (list x)))
        (class-subclasses data)))

(define t
   (instantiate::hierarchy-tree
      (get-node-children get-class-children)
      (root object)
      (get-node-label class-name)
      (node-hook (lambda (data text icon)
                    (event-case text
                       (("<Button-3>")
                        (print (class-subclasses data)))
                       (("<Button-2>")
                        (print (class-name data)))))))

(pack t :expand #t :fill 'both)
```

11.6.2 File browser




```
(module hierarchy-file
   (library biglook))

(define (get-file-children data)
   (map (lambda (x)
           (let ((f (string-append data "/" x)))
              (if (directory? f) (list f) f)))
        (directory->list data)))

(define t
   (instantiate::hierarchy-tree
      (get-node-children get-file-children)
      (root ".")
      (node-hook (lambda (data text icon)
                    (event-case text
                       (("<Button-3>")
                        (system "xedit" text "&")))))))

(pack t :expand #t :fill "both")
```

11.6.3 Animation skills

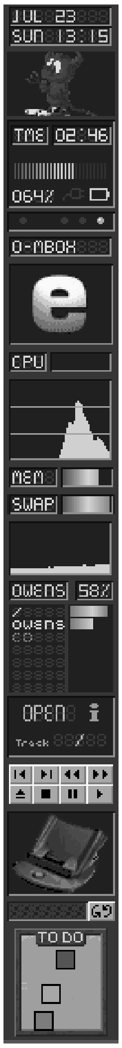

```
(module biglook-demo-runner
   (library biglook)
   (static  (class runner::image-item
               (number (default (gensym 'runner)))
               (images (default *runner-images*))
               (speed  (default (random)) read-only))))

(define *runner-images*
   (let ((l (list (instantiate::image (file "runner1.xpm"))
                  (instantiate::image (file "runner2.xpm"))
                  (instantiate::image (file "runner3.xpm"))
                  (instantiate::image (file "runner4.xpm")))))
      (set-cdr! (last-pair l) l)
      l))

(define *ground*
   (instantiate::canvas
      (height 30)
      (width  150)
      (background "white")))

(instantiate::line
   (parent *ground*)
   (fill "red")
   (coords (list 10 20 (- (canvas-width *ground*) 10) 20)))

(define *start*
   (instantiate::button
      (text    "Start new runner")
      (command (lambda ()
                  (let ((new (instantiate::runner
                                (init-coords '(10 15))
                                (parent *ground*))))
                     ((make-runner-run new)))))))

(define (make-runner-run r)
   (define (run)
      (with-access::runner r (coords image images speed)
         (match-case coords
            ((?x ?y)
             (if (< x (- (canvas-width *ground*) 13))
                 (with-access::runner r (image images speed)
                    (set! coords (list (+ 1 x) y))
                    (set! image (car images))
                    (set! images (cdr images))
                    (after speed run))
                 (destroy-canvas-item! r)))
            (else
             (error "run" "runner lost" coords)))))
   run)

(pack *ground*)
(pack *start* :expand #t :fill 'x)
```

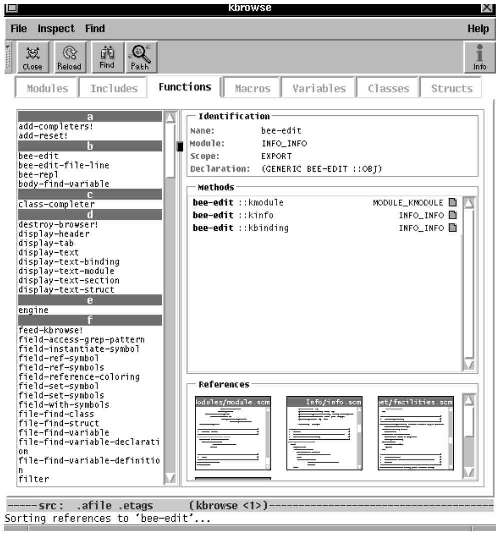
11.6.4 Benchmark

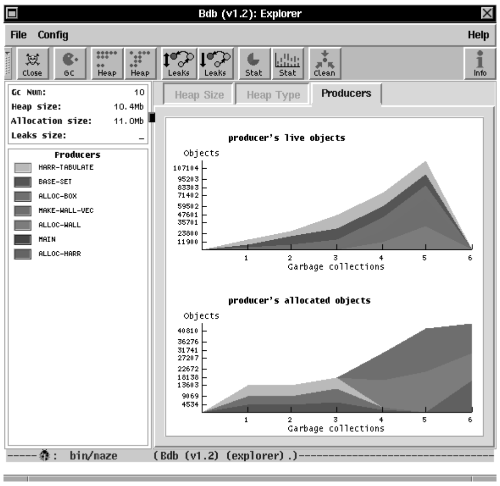
```
(define (make-interface)
   (let* ((t (instantiate::top-level ))
          (left (instantiate::radio
                   (parent t)
                   (texts '("9" "10" "12" "14" "18"))))
          (middle (instantiate::frame
                     (parent t)))
          (right (instantiate::radio
                    (parent t)
                    (texts '("red" "gree" "blue" "yellow"))))
          (bot (instantiate::frame
                  (parent t)
                  (border-width 2)
                  (relief "raised"))))
      (pack bot :side "bottom")
      ;; check buttons
      (let ((button1 (instantiate::check-button
                        (parent middle)
                        (key "bold")
                        (text "Bold")))
            (button2 (instantiate::check-button
                        (parent middle)
                        (key "italic")
                        (text "Italic"))))
         (pack button1 button2))
      ;; the three top frames
      (pack left middle right)
      ;; the buttons (ok and cancel)
      (let ((ok (instantiate::button
                   (parent bot)
                   (text "Ok")
                   (command (lambda ()
                               (biglook-exit 0)))))
            (cancel (instantiate::button
                       (parent bot)
                       (text "Cancel")
                       (command (lambda ()
                                   (biglook-exit 0))))))
         (pack ok cancel :expand #t :fill "x"))))
```

Annex B : Screen shots

Here are three screens shots of Biglook application. The first one is an application bar. It relies on the efficiency of Biglook application because the gauge used to represent information about the CPU and memory load are updated several times per second. The screen shots on the right side of the page have been grabbed while running Bee tools : the Bigloo source code browser and the Bigloo heap profiler. Each of these graphical interfaces uses several complex widgets. For instance, the source code browsers uses texts, text properties, panels, notepads, toolbars, etc. The heap profilers mainly uses canvas.







Chapitre 12

Understanding Memory Allocation of Scheme Programs

Cet article a été publié dans [210]. Il a été écrit en collaboration avec Hans-J Boehm.

Allocations mémoires dans les programmes Scheme

La gestion de la mémoire (allocation, accès et écriture) est le goulot d'étranglement des architectures modernes. Réduire au maximum la consommation de mémoire permet alors d'obtenir des applications rapides et qui n'encombrent pas inutilement tout le système. Malheureusement, il est difficile d'évaluer la quantité de mémoire utilisée pour les langages fonctionnels parce que ces derniers ont souvent recours à des encodages complexes des constructions de haut niveau qui nécessitent des allocations mémoires.

Pour faciliter la compréhension des ressources mémoires consommées par des programmes Scheme, nous avons conçus deux outils complémentaires. Le premier rapporte des informations sur la fréquence des allocations et les diverses configurations du tas au cours des exécutions. Le second permet de débusquer les fuites mémoires. Appliquer ces deux outils à notre compilateur Scheme, le plus gros programme en cours de développement nous a rapidement permis de diminuer la mémoire qu'il allouait pour son auto-génération.

Nous avons consacré beaucoup de notre attention à l'intégration harmonieuse de nos deux nouveaux outils dans notre environnement de programmation Scheme. En effet, nous sommes persuadé que si ces outils ne sont pas facilement accessibles et aisés à mettre en œuvre, ils seront immanquablement sous utilisés.

Understanding Memory Allocation of Scheme Programs

Memory is the performance bottleneck of modern architectures. Keeping memory consumption as low as possible enables fast and unobtrusive applications. But it is not easy to estimate the memory use of programs implemented in functional languages, due to both the complex translations of some high level constructs, and the use of automatic memory managers.

To help understand memory allocation behavior of Scheme programs, we have designed two complementary tools. The first one reports on frequency of allocation, heap configurations and on memory reclamation. The second tracks down memory leaks [1]. We have applied these tools to our Scheme compiler, the largest Scheme program we have been developing. This has allowed us to drastically reduce the amount of memory consumed during its bootstrap process, without requiring much development time.

Development tools will be neglected unless they are both conveniently accessible and easy to use. In order to avoid this pitfall, we have carefully designed the user interface of these two tools. Their integration into a real programming environment for Scheme is detailed in the paper.

[1] Part of this work has been accomplished before the second author joined Hewlett-Packard. This research has been initially conducted at Xerox Parc and at SGI.

12.1 Introduction

Since CPU speeds continue to increase faster than memory speeds, memory is and will increasingly be the factor that limits performance [157]. Excessive memory use has two drawbacks : The program itself makes less effective use of the higher layers in the memory hierarchy, and it may interfere with other processes running on the same machine. Functional languages have a reputation for memory consumption. In order to deliver fast applications implemented in these languages, we need tools that can be used to diagnose problems with unexpected memory use. We have designed two such tools that aid in reduction of memory consumption of Scheme programs.

12.1.1 Garbage collected languages

Garbage collectors (GCs henceforth) have very desirable properties : by automatically reclaiming useless memory cells, they make programs easier to write, safer, and easier to maintain.

Unfortunately, GCs often hide the complexities of memory management so well that programmers lose track of its cost. It is extremely difficult to determine when a GC will deallocate a data structure. It is a misunderstanding to think that because GCs automatically reclaim useless cells, they keep the memory occupation minimal. It has already been noticed that, in some situations, GCs enlarge the size of the programs working sets [263]. Consequently, very often, programs allocate and consume more memory than needed.

12.1.2 Scheme specificities

Precise evaluation of memory allocation size is more difficult in higher order programming languages than in conventional languages due to the distance between the high level constructs and the instructions executed on the hardware. Compilers have to generate sequences of operations for which the complexity is not always apparent at the source code level. It may happen that the compilers introduce run time heap allocations where none were obvious in the source.

Such "hidden" allocations are frequent in Scheme programs. For instance, let's take the following definition :

```
(define (show-value value)
  (print "value is: " value))
```

The function `print` accepts optional arguments. The Scheme semantics specify that such optional arguments must be placed in a freshly allocated list that is passed as the actual argument. That is, if `print` is implemented as :

```
(define (print . l)
  (for-each display l))
```

each time `show-value` is called three pairs are allocated. These allocations have no location in the source code. They are not apparent !

Scheme library functions may allocate substantial amounts of memory. One may write a function like :

```
(define (append-rev l1 l2)
  (append (reverse l1) (reverse l2)))
```

If `len1` is the number of pairs of `l1` and `len2` is the number of pairs of `l2` then `append-rev` allocates `2*len1+len2` pairs because both `append` and `reverse` allocate. Changing `append-rev` to :

```
(define (append-rev! l1 l2)
  (append! (reverse! l1) (reverse! l2)))
```

eliminates all allocations from `append-rev` because it reverses the two lists in place, and then appends them using one single pointer assignment. Obviously `append-rev!` and `append-rev` are not equivalent because `append-rev!` *changes* its arguments. But `append-rev` can sometimes be replaced with `append-rev!`.

When studying a program it may be difficult to detect that a function such as `append-rev` is responsible for many memory allocations. This is the role of an allocation profiler. It reports the frequency with which allocation sites are used.

12.1.3 Our Tools

We have implemented two distinct tools for analyzing memory allocation. They are part of the Bee [209], an integrated development environment for the Scheme programming language [120] that relies on a Scheme to C Compiler.

Kprof is an allocation profiler embedded in our regular Scheme time profiler. For each source code function, it reports on the number and kind of allocations. Kprof presents estimates of the exact allocation numbers, using a technique similar to gprof [98]. In addition, it provides information about the operation of the GC. All of this can be accomplished with low overhead, and no per object space overhead.

Allocation profiling reports on heap growth, but it cannot report on memory leaks. Kbdb is a heap inspection tool. At first, it acts as a debugger. Programs are run interactively. They can be stopped, the variables, the stack and the heap can be inspected, and execution can be resumed. But in addition, the heap can be displayed, with each pixel representing one cell of the heap.

Individual cells can be inspected by simply pointing at their corresponding pixel in the image. When a cell is inspected, the Scheme type of its value, its allocation site, and its approximate age are displayed. In addition, KBDB displays *root chain links*. That is, KBDB can be used to understand why a specific cell is considered live by the garbage collector. This facility can be used to track down most kind of memory leaks presented in Section 12.2.

The bitmap representation can be cheaply generated from the heap. However, unlike KPROF cell inspection requires additional per object memory overhead.

12.1.4 Contributions

We characterize the types of "memory leaks" we have encountered in garbage collected environments, and discuss in detail our experience with memory leaks using one particular Scheme program. We are not aware of other general discussions of the issue, especially in the context of strict languages.

We present a complete, easily usable, set of tools for examining memory allocation in garbage collected languages. We demonstrate how they can be used to identify and track "memory leaks".

Our KPROF allocation and time profiler is based on standard time profiling techniques. We explore and measure its utility as an allocation profiling tool.

Our KBDB tool allows exploration of heap reference patterns in a style similar to Jinsight [62]. However, it uses a different mechanism for displaying the results, and a different data gathering strategy. The latter simplifies use, allows easier scaling to large applications, and allows problems involving the garbage collector itself to be isolated.

We believe that the techniques presented here apply to any language environment with garbage collection and run-time type information (*e.g.,* Smalltalk, CLOS or Java). Even a language like ML could benefit from the presented techniques because traditional run-time systems for this language support limited dynamic type information (such a the byte size of each allocated objects).

12.1.5 Organization

Section 12.2 discusses memory leaks in garbage collected environments. Then, Section 12.3 presents the facilities provided by KPROF and how it fits into our integrated environment named BEE. It also discusses how allocations are estimated by KPROF. Section 12.4 presents KBDB and its integration in the BEE. Section 12.5 demonstrates how KPROF and KBDB can be used in the context of a real application and shows the run time overhead of instrumented programs. Section 12.6 compares KPROF and KBDB to existing tools. Section 9.7 presents some possible extensions to KPROF and KBDB.

12.2 Memory leaks of garbage collected languages

In an environment using C-like explicit memory deallocation, the term "memory leak" usually refers to memory that is no longer accessible via any chain of pointer dereferences, but has not been deallocated. Since it is no longer accessible, it cannot be deallocated in the future.

It is the job of a garbage collector to eliminate such leaks. Thus such leaks cannot occur in garbage collected environments. When we talk about a "memory leak" in a garbage collected language, we are referring to memory that still appears accessible to the garbage collector but is no longer needed by the program. There are a number of reasons why such a chunk of memory may exist :

- It may be referenced through an *algorithmically dead* variable or an easily identifiable slot in a data structure. This is the most common problem. Once identified, it can usually be repaired easily by resetting the reference. Such leaks are usually bounded, but see [25] for a case in which an extra reference introduces an unbounded leak. In most cases, the reference is eventually overwritten, and the leak is thus temporary. But even temporary leaks can appreciably increase the heap size required by a process.
- It may be referenced through an *algorithmically dead* slot in a data structure, but

the slots are interspersed in the data structure and expensive to identify. This is rare, especially in programs written for garbage collection. But we know of one case (a compiler) in which it prevented easy replacement of manual deallocation with garbage collection. This is similar to the preceding case, except that it may require much more substantial algorithmic changes to repair.

– It may be referenced through a pointer that is itself still live (*e.g.*, for use in a Scheme `eq?` comparison), but is never dereferenced. This can happen, for example, if the later stages of a compiler refer to identifiers exclusively with symbol table pointers, so that the actual strings representing the identifier could be discarded. This again appears to be rare.
– It may be genuinely referenced from a data structure that grows much larger than intended. The canonical example of this is a cache that was intended to remain bounded, but in fact grows without bound over time. Although the extra data may be accessed, overall performance of the program would increase if some of it were discarded.
– It may appear to be referenced to the collector, even though it is not truly accessible by following pointer chains from a live variable. This may happen because the collector has imperfect information about liveness of pointer variables, because the collector is conservative and has misidentified a non-pointer as a pointer [25], or because of an unfortunate promotion in a generational collector [178]. In most, though not all cases, such leaks are again temporary and bounded.

Usually, though not always, the hardest task in removing such a “memory leak” is to identify its source. We describe tools that can be used to do so. The tools rely on the garbage collector itself, and hence can be used to trace problems caused by idiosyncrasies of the garbage collection algorithm itself, or by interactions between the garbage collector and the client, in addition to those caused purely by the client program.

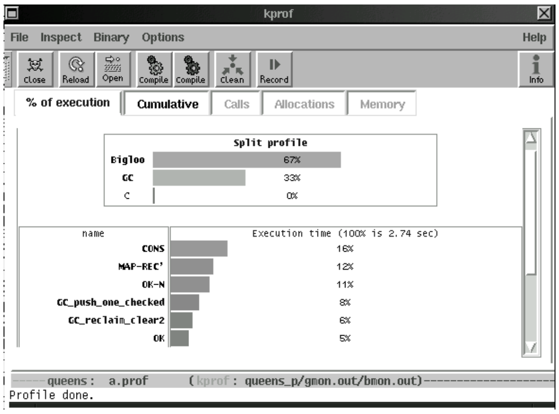


FIG. 12.1 – A plain profiling window

12.3 Kprof : an allocation profiler for Scheme programs

The first of our two profiling tools is named KPROF. It reports the number of times allocators are called from each function. In addition, KPROF reports on the evolution of the heap during the execution of the program. Each time a collection is triggered, the heap size, the number of live objects, and the number of allocations since the previous GC are recorded. Figure 12.1 displays the time profile for the Queens program, a small Scheme program that computes the number of solutions to the N-queens problem. The Queens program builds all possible configurations of the queens on a chessboard. Each configuration is implemented as a list. Thus the benchmark is allocation intensive.

12.3.1 Plain profiling

KPROF distinguishes between the execution time spent in Scheme functions (labeled `Bigloo` as the name of our Scheme compiler), the garbage collector, and other C functions. User programs may mix C functions with Scheme functions. In such an environment we have found it best to display information by implementation language. The top lines of the profile window teach us that about 67% of the execution time is spent inside Bigloo functions and 33% inside the garbage collector. The bottom lines display the time spent in each function of the program. In particular, we read that 16% of the total execution time is spent in the Scheme `CONS` function.

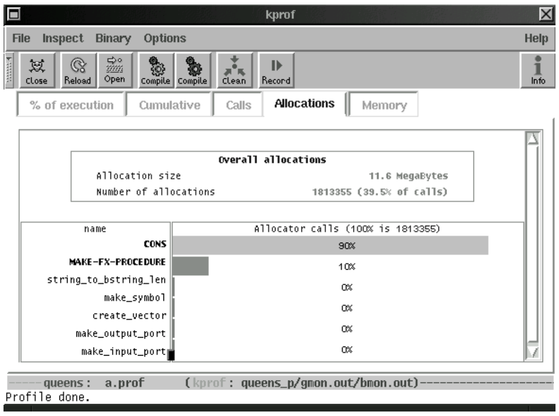


FIG. 12.2 – Queens's allocations

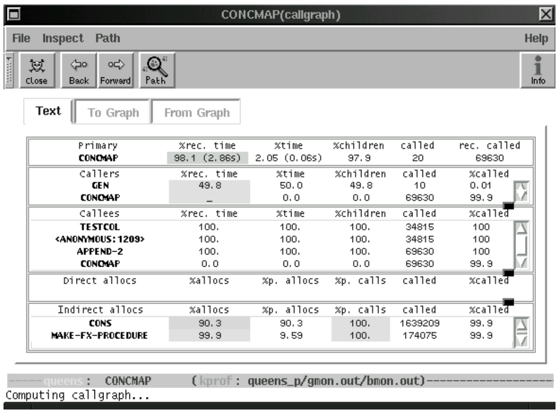


FIG. 12.3 – CONCMAP profiling

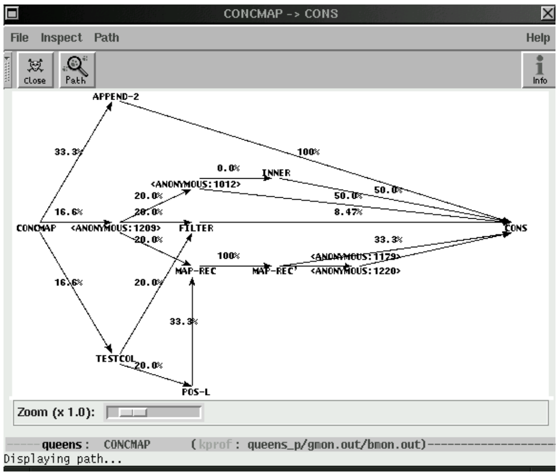


FIG. 12.4 – CONCMAP calls to CONS

KPROF may also display the overall amount of memory allocation during an execution (see Figure 12.2). We learn that 11.6MB have been allocated to run the Queens program and that 90% of the allocations are lists.

12.3.2 Dynamic call graphs

Since KPROF uses regular profiling facilities, it can browse the dynamic call graph. For instance, it can report which functions are allocating a large fraction of the CONSes. Examining the functions that call CONS would report that the Scheme function MAP-REC is responsible for more than 42% of the calls. KPROF may also compute estimates of the number of allocator calls by a function and its (direct and indirect) callees. We refer to these as *indirect allocations*. For instance, consider the CONCMAP function of Queens :

```
(define (concmap f l)
   (if (null? l)
       '()
       (append (f (car l)) (concmap f (cdr l)))))
```

This function does not directly call any allocator, but obviously the functions it calls (its callees, notably APPEND) call CONS. Figure 12.3 presents the profile information computed by KPROF. The Direct allocs section is empty, as expected. But we discover that the CONCMAP callees are responsible for 99% of the calls to CONS (Indirect allocs, %called column). These calls account for 90.3% of the overall allocations invoked from CONCMAP (Indirect allocs, %allocs column).

KPROF can display the dynamic paths that exist from one function to another, for instance, the paths that go from CONCMAP to CONS as presented in Figure 12.4. Each edge label stands for the percentage of the caller calls devoted to that callee. For instance, 20% of the calls made by TESTCOL are calls to FILTER and 8.47% of FILTER calls are calls to CONS.

12.3.3 Memory profiling

KPROF can then display the heap configurations recorded by each garbage collection during program execution, as in Figure 12.5. Here, execution time is measured in GCs. We learn that eighteen GCs were required to execute Queens. For each GC, we learn the heap size (that varies here from 1.05MB to 1.20MB). KPROF also displays the stack size (the maximum is reached on GC 9 with about 0.25MB). The other low curve represents the number of live objects after each

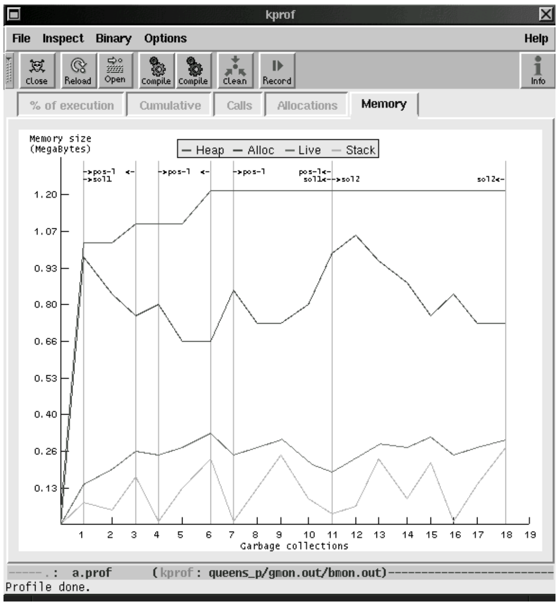


FIG. 12.5 – Queens's memory configurations

collection. That one goes from about 0.13MB to about 0.30MB with a maximum reached at GC 6 with about 0.30MB of live objects. The fourth curve takes into account the number of objects that have been allocated since the previous GC (about 1.0MB of allocations to GC 12).

During the study of such a profile it may be convenient to focus on some specific parts of the program. For instance, our Queens example includes two different implementations of the same strategy. It could be interesting to compare these different implementations at a glance. This is permitted by our Scheme extension : the `profile` form. Its syntax is :

```
(profile <label> <s-expr>)
```

Its evaluation is equivalent to `(begin s-expr)` but it will force KPROF to report data for a pseudo-function named *lbl*, with all costs associated with the evaluation of *s-expr* reported as part of the execution of *lbl*. It will thus be possible to determine the number of allocations executed during the evaluation of *s-expr*. In addition, the evaluation period for *lbl* is displayed on KPROFś memory profile. Figure 12.5 includes the result of several `profile` forms, labelled `SOL1`, `SOL2`, and `POS-L`, indicated immediately above the memory usage graph.

12.3.4 Kprof implementation

Bigloo, our Scheme compiler, translates from Scheme to C. The generated C code conforms to the standard C coding style [31]. In short, Scheme functions are compiled into C functions and Scheme variables are compiled into C variables. Because a direct correspondence exists between the produced C code and the initial Scheme source file, it is possible to reuse standard C tools to profile Scheme source programs. KPROF is designed as a layer surrounding the standard Unix GPROF tool [98, 75]. This technique is described in a forthcoming paper [209]. KPROF decodes GPROF information (in particular, KPROF demangles GPROF symbols) and uses the result to compute allocation profiles.

Inherited features

Since KPROF is a front-end to GPROF it inherits some of its facilities and, alas, some of its inaccuracies. These are described in some details in the GNU-gprof documentation [75]. We shortly summarize them in this section.

- Run-time figures are based on a sampling process. To determine the time spent in each function GPROF samples the hardware program counter at regular intervals (*e.g.*, every 0.01 seconds). The entire time interval is charged to the corresponding function. To produce reliable execution time estimates, the overall execution time of the application must be much larger than the sampling period.
- The number of calls are exact. They are computed by inserting an additional call to an accounting function into the prelude of every profiled function. This function is responsible for recording in an in-memory call graph table both its parent routine and its parent's parent.
- Callees run times presented in the call graph reports are probabilistic. The record made of an execution does not contain any information relative to the dynamic call graph. Here is an excerpt of the GPROF documentation describing the technique :
  "The assumption made is that the average time spent in each call to any function `foo` *is not correlated with who called* `foo`*. If* `foo` *used 5 seconds in all, and 2/5 of the calls to* `foo` *came from* `a`*, then* `foo` *contributes 2 seconds to* `a`*s callees time, by assumption.*
  This assumption is usually true enough, but

for some programs it is far from true. Suppose that `foo` *returns very quickly when its argument is zero ; suppose that* `a` *always passes zero as an argument, while other callers of* `foo` *pass other arguments. In this program, all the time spent in* `foo` *is in the calls from callers other than* `a`*. But* GPROF *has no way of knowing this ; it will blindly and incorrectly charge 2 seconds of time in* `foo` *to the children of* `a`*."*

Allocation estimates

KPROF reports on the number of *direct* and *indirect* calls to allocators. The *direct* calls are restricted to those made by the functions themselves. The *indirect* calls are the calls made by the functions themselves and their callees. The *direct* calls are exact values but the *indirect* ones are estimated.

The assumption made to compute *indirect* allocations is similar to the one used to compute callees run times. The indirect allocations of a function $\mathcal{F}$ are the direct allocations of that function plus a percentage of the allocations performed by its callees. For each callee $\mathcal{C}$, this percentage is calculated by dividing the time spent in $\mathcal{C}$ when called by $\mathcal{F}$ by the overall time spent in $\mathcal{C}$. For instance, let us suppose that a function `F1` calls an allocator `A1` n times and also calls a function `F2`. `F2` calls the allocator `A1` m times and no other functions. Now let us suppose that 30% of the run time of `F2` is spent when it is invoked by `F1`. KPROF reports that the number of *indirect* calls to `A1` from `F1` is $n + 0.3 \times m$.

The algorithm is partially validated by the fact that KPROF correctly reports that the entry point of a Scheme program indirectly calls 100% of the allocators (a whisker away because of floating point errors). However, as we will see in Section 12.5.2 these estimates may be imprecise. Nevertheless, since they are fast to compute, they can be used as a first indication that thorough investigation is needed. In many cases even these estimates may unveil obviously wrong behaviors. We will show in Section 12.5.1 that the estimates have permitted drastic reductions in the memory consumption of the Bigloo compiler itself.

12.3.5 Kprof and code generation

To "record" an execution, the source code has to be compiled in "profile" mode. That is, the compiler has to introduce some extra instructions for producing profile output file used by KPROF. In this section we describe that code.

Profiling and optimizations

Optimizations that change the initial structure of the source code have to be disabled in order to make profile information accurate. Inlining is one of these optimizations, since it replaces function calls with the bodies of the called functions. Since functions are the smallest entities the profiler reports on, it is important that user functions are not inlined when in profile mode. However it has been demonstrated that inlining is an important optimization, especially for functional languages [162]. We are thus facing a dilemma : should the profile compilation mode enable aggressive optimizations such as inlining even if this reduces the accuracy of the profiler reports, or should it disable optimizations ? One should notice that the second solution also effectively reduces the accuracy of the profiler, since the measured program is likely to behave differently from the final version.

We don't think there is a "best" solution for this problem, and instead we provide the user with two different profiling modes. One enables aggressive optimizations, the second disables all of them. Figure 12.1 presents the profile with optimizations enabled. Note that in the current version, Bigloo never inlines `CONS` since the code duplication introduces by this inlining has not shown any execution speedup.

Profiling with local functions and macros

As in most functional languages, Scheme has local, possibly anonymous, functions. KPROF reports on these as it does on global functions. Because several local functions may have the same name and reside inside the same module, the profiler prefixes their name with the name of the enclosing function.

For anonymous functions, the compiler generates a name from the source file location. For instance, let us suppose an excerpt of a file `F.scm` :

```
150 :  (define (foo x)
151 :      ... (map (lambda
       lambda (y) (+ x y)) ...) ...)
```

The function of line *151* will be named `F.scm :151 :7347`, where `7347` is a stamp that avoids name collision. When the user invokes the editor on that function, the profiler invokes the editor on the correct file.

Macro names are not used in the expanded code so macros are not present in KPROF́s output. That is, the expanded code is taken into account but name of the macros are discarded during compilation.

12.3.6 Kprof limitations

Allocation profiles are not sufficient to track down memory leaks. For instance, in Figure 12.1 the "live objects" curve seems to reveal an increase of live objects in the `sol1` stage. When `sol1` completes there are still some live objects and the difference between the number of live objects at the beginning and at the end of `sol1` is positive. KPROF provides no information as to whether this increase is normal or if it is due to a memory leak. It reports on *allocations* whereas memory leaks concern *deallocations*. To address this problem, we have designed and implemented a second tool named KBDB.

12.4 Kbdb : a heap inspector for Scheme programs

KBDB is an interactive heap inspector. It displays the live objects and the chains of pointers that link these objects in the heap. KBDB is embedded into our regular Scheme debugger. Each time an execution is suspended (for instance, when a breakpoint is reached) the heap may be inspected.

When KPROF reveals a *suspect* increase of live objects KBDB can be used to discover if this is due to a memory leak. Obviously this requires thorough knowledge of the source code. To suspect that a computation leaks memory, it must be known that this computation *should not* increase the number of live objects. KBDB can only answer the question "what is the amount of memory still in use after evaluating that particular expression of the source code ?".

12.4.1 Plain heap inspection

To start with, KBDB acts as a regular Scheme debugger. It enables suspension and resumption of execution. When an execution is suspended the variables and the stack of the computation may be inspected. In addition to these traditional features, KBDB can also display a snapshot

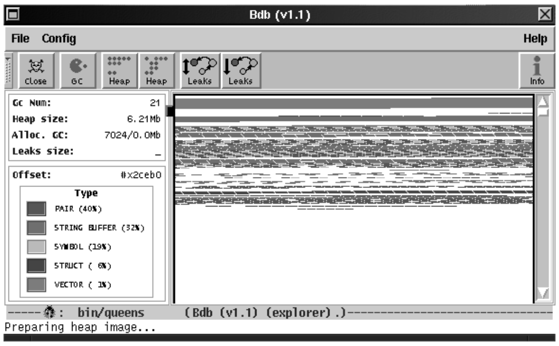


FIG. 12.6 – An heap view according to object types

of the heap as a 2 dimensional picture in which each pixel is associated with a memory location. Unused memory locations are left blank. Objects are distinguished by their color. Currently two color schemes have been implemented. In the first, objects are colored according to their type. For instance, all strings are displayed in blue, the pairs in red, and so on. The second classification uses the age of the objects to determine their color. Figure 12.6 is a snapshot of KBDB displaying a heap during the execution of the **queens** program, with objects classified by types.

Objects are represented by horizontal stripes ended with a white pixel. The larger an object is, the longer is its associated stripe. Sections of the heap can be magnified to make selection of specific objects accurate. Detailed information about a clicked object is then reported. For instance, clicking on a `PAIR` stripe could display :

```
Holder    : FILTER, frame 4 (local L) ;; The value holder
type      : PAIR (0)                  ;; The object type
Producer  : CONCMAP                   ;; The producer
Age       : 1                         ;; The GC birth date
Size      : 12                        ;; The byte size
   [PAIR]   (1)                       ;; The holder chain links
   [PAIR]   (2)
(bdb:MAIN) display (0)
$63 = (8)
(bdb:MAIN) display (2)
$63 = (6 7 8)
```

The `PAIR` we have clicked on has been allocated in the function `FILTER` before the sixth collection and its size is 12 bytes. The producer is the "first" user function that calls the allocator. That is, library defined functions are not reported as producers. For instance, the producer of the pairs allocated by `APPEND` in the `CONCMAP` function of Section 12.3.2 will be reported as allocated by `CONCMAP`, not `APPEND`. The `PAIR` was held by the local variable `L` of the function `FILTER`. An "holder" is either a global variable, a local variable, or simply a stack frame (when the value is not stored in any local variable but simply passed to another function). In the remainder of the paper, we will indistinguishly name an holder, a GC root. The stack frame of the `FILTER` invocation that holds `L` is the fourth one in the stack (`frame 4`). The *holder chain links* represent the pointers chain from the holder (*i.e.*, `FILTER`'s `L` local variable) to the inspected object (here a `PAIR`). The labels of the holder chain links can be used to explore or display the objects of that chain.

12.4.2 Memory leaks

Provided with a heap inspector, memory leak detection is easy. The framework for finding leaks in a suspect expression $\mathcal{E}$ is the following :

1. Stop the execution before the evaluation of $\mathcal{E}$. The number of already completed collections is GC_n.
2. Trigger a garbage collection.
3. Resume the execution.
4. Stop the execution when evaluation of $\mathcal{E}$ is completed.
5. Trigger a new garbage collection.

The current collection number is now GC_m. Leaking objects are those that have been allocated during the evaluation of $\mathcal{E}$ that are still live, *i.e.* live objects that have been allocated after GC_n and before GC_m.

To find a memory leak using the current KBDB interface, two breakpoints have to be set. One before the suspected expression and one after. When the execution reaches the first breakpoint, a simple click on the $\boxed{\texttt{leaks}}$ icon triggers the previously described steps 2 to 5. A picture consisting only of the live objects is then displayed as in Figure 12.7. In this snapshot of the heap only the "leaking" objects, *i.e.* newly allocated and still live objects, are displayed. The roots causing the leaks are displayed in a different color than the leaking objects. As reported on the left side of Figure 12.7, the entire leak size is of 8480 bytes in this case. Leaking objects may be selected as before. Clicking on one of these objects could produce :

```
Holder    : COUNT
Type      : PAIR   (0)
Producer  : NSOLN
Age       : 2
Size      : 12
   [PAIR]       (1)
   [CELL]       (2)
   [PROCEDURE]  (3)
```

This object can be displayed :

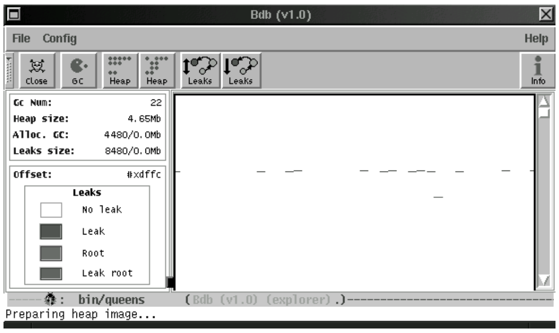


FIG. 12.7 – Memory leaks unveiled

```
(bdb:MAIN) display (0)
(2 3 4 5 6 7 8)
```

Even if this is not obvious in the gray scale display, there is only *one* GC root that is culprit for the entire leak. That root is the end of the root chain of the object we have selected. It can also be selected :

```
(bdb:MAIN) explore (5)
Holder    : COUNT
Type      : PROCEDURE   (0)
Producer  : COUNT
Age       : 2
Size      : 20
```

We can now draw some conclusions from this inspection of the **Queens** program : *i)* As we suspected, the evaluation of the `SOL1` form leaks memory. *ii)* That leak is composed of small lists (the entire leak size is 8.3KB). *iii)* All lists are accessible from the same root. *iv)* The root is the anonymous closure (a `PROCEDURE` type) that has been allocated in the `COUNT` function.

Actually, the `COUNT` function is a memo function. It allocates lists and stores them in a table that is never reset. In Section 12.3 we suspected a memory leak. KBDB has demonstrated that this leak really exists.

12.4.3 Kbdb implementation

We modified the Boehm-Demers-Weiser garbage collector to provide back-pointer information, as part of the debug information that could already be associated with individual objects. Each allocated object is provided with additional slots to store the source code location of the allocation and one *back pointer* slot that is filled by the collector during the marking process.

The contents of the back pointer slot point to the location of the pointer that caused the object in question to be marked. If the object is reachable by more than one path, the one that happened to be followed by the collector will be reflected in the KBDB output. In the, usually infrequent, cases where the conservative collector follows a stale stack pointer, or a misidentified pointer, that fact will also be accurately reflected in the chain of back pointers. Thus even such problems become debuggable.

As discussed in the related work section, there are other ways to display backward reference chains. This technique is the second one we have implemented, and by far the simplest. It was suggested by Alan Demers.

Details of the collector backtracing interface can be found in the collector distribution. In this section, we focus on the implementation of the heap picture construction and the KBDB architecture.

The debugged applications

Debuggable applications embed special library functions that are in charge of constructing the heap display. When KBDB is to display a heap, it requests that the debugged application produces a file on disk containing the picture. The picture file is constructed without additional memory consumption by means of simple linear scans of the heap. The garbage collector is able to report, for each address of the heap, if it is part of a live object, and to retrieve the size of that object. The algorithm to build a heap picture is :

```
make-picture( F ) =
 let min-addr = The heap min address
     max-addr = The heap max address
     i = 0
     pic = create-picture()
   while i + min-addr < max-addr do
     if i + min-addr is the address of a live object ?
     then let size = GC-object-size( i + min-addr )
              stop = size + i
              type = SCM-type( i + min-addr )
          while i < stop do
            set-pixel-color( pic, i, F( type ) )
            i = i + 1
     else set-pixel-color( pic, i, "white" )
          i = i + 1
   return pic
```

The argument $\mathcal{F}$ is a parameter of the picture construction. It is a function that maps Scheme objects to colors. It enables various coloring schemes to be applied to the heap construction (such as the type based or the age based ones).

The key point of this algorithm is that a picture file is directly dumped during a heap traversal. There is no need to allocate a memory area of the size of the inspected heap because of the direct mapping from heap addresses to picture pixels. This is possible only if the garbage collector is able to report information about random addresses in the heap. It is not clear how exact collectors could implement this.

File generation is straightforward. Leak detection and age-based coloring do not require pre-computation. On the other hand, association between types and color requires an additional first linear traversal of the heap in order to allocate colors to the most frequently used types (only those are allocated specific colors, infrequent types are all displayed with one unique color). Then, during a second linear traversal, the picture is built according to the `make-picture` algorithm.

Kbdb display generation

When a debugged application has generated a picture file, KBDB reads that file. Note that a picture file is often much smaller than the inspected heap because one pixel represents a memory word (*i.e.*, 4 or 8 bytes). For a monochrome picture, 8 pixels can be stored in a single byte, making a monochrome picture about 32 times smaller than the inspected heap. A four color picture is 16 times smaller than the heap. Four colors are sufficient for memory leak displays.

We have concentrated on the compactness of the heap representation. We think this is a central issue. If the representation requires too much memory, the system cannot be used to inspect large heaps. We have successfully applied KBDB to our Scheme compiler, demonstrating that it can be used on heaps larger than 8MB.

12.5 Applying Kprof and Kbdb

This section presents the results of our first attempt to apply KPROF and KBDB to a real application : our Scheme compiler. The aim of this section is to show that KPROF and KBDB are useful in practice. For this experiment we have looked at bootstrapping the compiler, that is, compiling a part of the compiler with an instrumented version of the compiler. We have arbitrarily time bounded that effort by allocating

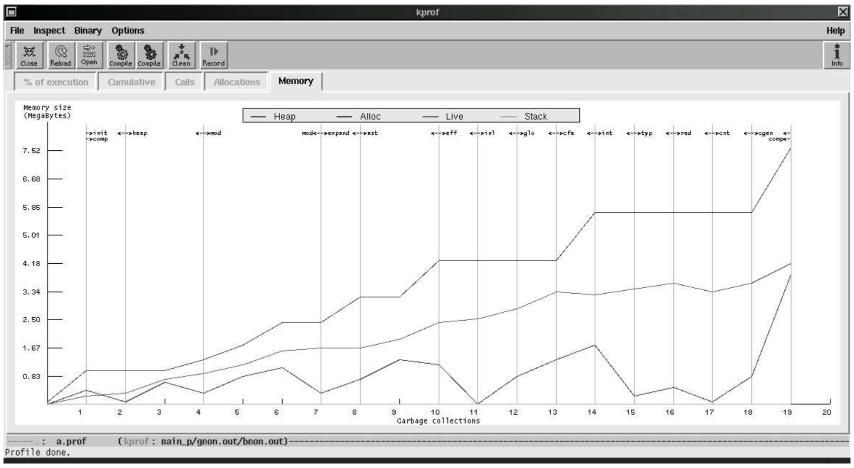


FIG. 12.8 – Profiling Bigloo

only two days to profiling. This section reports on what we have learned about the compiler during these two days and concludes with some measurements that show the execution overhead of profiling and debugging.

12.5.1 The Bigloo compiler

Bigloo compiles Scheme into C. It is written in Scheme and compiled by itself. Bigloo consists of 47,000 lines of code. It reads the program to be compiled and builds an abstract syntax tree (henceforth AST) to represent the program. This tree contains a structure of 23 different node types. There are nodes for constants, variable assignments, conditionals, function calls, etc. The compiler is made up of stages, each of which can be seen as a process that modifies the AST. The driver is a Scheme function that looks like :

```
(define (compiler src)
   (let ((ast (build-ast src)))
      (macro-expand! ast)        ;; 1st stage
      (function-inline! ast)     ;; 2nd stage
      ...
      (code-generate! ast)))     ;; 20th stage
```

This rigid structure, in which each stage acts as a stand alone program, helps the implementation and maintenance of the compiler. It also helps with profiling the compiler. Because of that structure it is easy to let KPROF report on allocations stage by stage. It only requires instrumenting the compiler driver with profile forms such as
`(profile ast (build-ast src))`.

Reducing compiler memory allocation

Figure 12.8 presents the heap profile as reported by KPROF when bootstrapping a module of the compiler. The file chosen for that experiment is the one that compiles into the largest C file. That file contains most of the classes representing Bigloo's AST. In Bigloo's object system, class instance slots are fetched and changed via getter and setter functions. These functions are automatically generated by the compiler. The module we are studying produces a large C file because it contains most of the getters and setters of the AST.

The overall allocation for compiling that module is 17.3MB amongst which, 56% are `CONS`es. This was a surprise to us. The AST of the compiler is represented by a class hierarchy and we thought we have successfully avoided `CONS`ing in the code of the compiler. Actually, variable size data structures are implemented using `CONS`es. For instance, the list of formal parameters of a function is represented by a Scheme list, and sequences of expressions are also stored in lists (that is the `sequence` node of the AST holds several values, one of which is a list of expressions). KPROF shows that these lists are much more numerous than we suspected. It is possible to reduce allocation for such lists. It would, for instance be possible to use vectors instead. This would reduce memory requirements since vectors are more compact than lists (at least when they contain more than two elements).

KPROF points out several other surprising allocations. We focus here on the most significant ones. The last compiler stage (namely CGEN) writes the C code on a disk file. This stage only *dumps* the AST. It is not supposed to allocate memory. However, KPROF demonstrates that CGEN actually *does* allocate ! Inspecting CGEN allocations using the allocator profiler shows that CGEN is responsible for more than 13% of the calls to CONS. Studying the dynamic paths that go from CGEN to CONS shows that nearly all the CONSes allocated during the evaluation of CGEN are called by functions that write on the disk. That is, the CONSes are allocated when the compiler uses the standard Scheme output functions (DISPLAY, NEWLINE, ...). These CONSes are allocated because, as reported in Section 12.1.2, variable arity functions allocate, and standard Scheme output functions accept an optional output port. In order to remove these allocations, we have implemented a simple source-to-source transformation. When a call to a regular Scheme output function is detected, it is replaced with a call to a specialized function accepting one or two arguments, depending on the nature of the call. We don't claim that this optimization is "general purpose", we only claim that with a one-hour effort, we have been able to reduce the memory allocation to 14.3MB, which is a reduction of 21%. The proportion of CONSes for the compilation of the compiler dropped to 47%.

Chasing bootstrap memory leaks

Some stages of the compiler are expected to increase the live memory (such as the AST stage that constructs the AST, or INL that implements inlining optimization). On the other hand, some stages implement optimizations or analyses that should reduce the size of the AST. Amongst these stages is EFF (GC 10 to GC 11) which is a pass that computes the side effect property for each function of the AST. The results of EFF are later used to implement regular optimizations such as data optimizations (RED stage). Surprisingly KPROF reports that EFF slightly increases the number of live objects (2.45MB to 2.55MB). We used KBDB to inspect memory leaks of the EFF stage. KBDB reported that EFF is responsible for 210KB of leaks due to only two GC roots. One is located in a function called MAKE-SIDE-EFFECT! and the other in a function called FUN-CALL-GRAPH!. Inspecting the source code of these functions has revealed the nature of the two leaks : both functions use a table that is not reset when EFF is completed.

12.5.2 Impact of profiling and debugging

The performance difference of programs compiled in profile mode, debug mode or optimization mode should be as small as possible. If the difference is very noticeable, the profiler or debugger would be tedious to use. Even more seriously, if the performance degradation is too substantial, large programs can simply not be profiled or debugged. In this section we present some time and allocation measurements for the different versions. We have used three different programs : a small one (**Queens**, a 100 lines long program we previously discussed), a mid-size one (**Eval** the 500 lines long Bigloo Scheme interpreter), and a large one (**Bootstrap**, the 46,000 lines long Scheme compiler itself). Figure 12.9 compares the compilation time, execution time and the heap size of these three programs using different compilation flags, all other things being equal.

Comp is the compilation time, Size the size of the binary file. In order to present the impact of profiled compilation on the executable size, we have decided not to strip (that is not to remove the symbols table) the binary files in optimization mode. However, one should be aware that on our working architecture stripping an executable shrinks it by about a factor of three. In addition, all binaries are linked against static Bigloo libraries. (There is no issue here because all modes support shared libraries.) Run is the execution time (the minimum of user plus system time for three consecutive runs). Alloc is the amount of memory allocated during the execution.

The differences in size of the executables are important. Profiled executables are about four times larger. Debuggable executables are up to 10 times bigger (for **Bootstrap**). There is no way to avoid this increase because it does not depend on the Bigloo C generated code but on the assembly code generated by the C compiler. For instance, app.scm is one of the compiler source files. The generated C file app.c is 56KB long. When compiled with C optimization enabled, the object file app.o is 10KB in size When compiled in C debug mode, it enlarges to 51KB !

Debugged and profiled programs run slower than optimized programs. Independent of ins-

	Queens				Eval				Bootstrap			
Comp. modes	**Comp.**	**Size**	**Run**	**Alloc**	**Comp.**	**Size**	**Run**	**Alloc**	**Comp.**	**Size**	**Run**	**Alloc**
Optimized	2.3s	185k	1.3s	15.5MB	3.6s	167k	1.8s	7MB	6.6s*	898k	5.3s*	9.3MB*
Profiled	2.2s	494k	5.5s	15.5MB	3.41s	500k	6.11s	7MB	4.1s*	4643k	8.2s*	9.3MB*
Debugged	3.4s	620k	11.7s	55.2MB	7.4s	209k	6.8s	34.7MB	5.8s*	15929k	15.6s*	38.4MB*

FIG. 12.9 – Impact of the profiling and debugging compilations. Hardware configuration : K6/200, Linux 2.0.x, 64MB. The compilation time, run time and allocation size figures for the compiler **Bootstrap** (*) have been gathered when compiling only *one* module of the source code of the compiler.

trumentation, disabling optimization slows down programs by a ratio of 1.5 to 2. For small programs, the difference in performance for instrumented programs is important (a factor of 9 for **Queens**). Experience seems to show that the larger the programs are, the smaller are the gaps between optimized and instrumented applications : A factor 3.8 for **Eval** and a factor 2.9 for **Bootstrap**.

As we have previously stated, KPROF does not increase memory consumption. In contrast KBDB does ! The allocation growth factors are : 3.6 times for **Queens**, 4.9 for **Eval** and 4.1 for **Bootstrap**. That is, the memory overhead introduced by KBDB cannot be neglected. For our current implementation of the current run time system, the overhead for each allocated cell is of 4 words : 1 for a back pointer, 1 for producer information, 1 for age, and one other words used internally by the collector, primarily to support C debugging. These 4 additional words explain why the allocation size grows so much. Without debugging information, a Bigloo `CONS` is two words. That is, the run time type information is stored in the pointer to the pair (*i.e.*, there is no header word for pairs). This technique is no longer available in debug mode because all objects must have the very same data layout. As a consequence, in debug mode, a pair is 7 words large. Because of hardware alignment constraints this turned to 8 words, that is, 4 times larger. As reported by KPROF `CONS` allocations are dominant, it is thus not surprising that the overall heap usage increases by about a factor of 4.

12.5.3 Profiling validation

KPROF is implemented on top of Gprof. That is, KPROF re-targets the Gprof C sampling technique for Scheme. It is usually accepted that the accuracy of the C approximations delivered by Gprof is sufficient. However, it is conceivable that, due to differences in programming style, the Gprof technique applied to Scheme delivers inaccurate results. For instance, if Scheme programs make use of numerous smaller functions, higher sampling rates could be needed than for C. In order to show that it applies equally well to C and Scheme we have conducted another experiment. We have measured the number of calls per second for optimized Scheme and C programs. The number of calls have been gathered using exact Gprof function call counting. We use the three Scheme programs from Section 12.5.2. It is extremely difficult to compare Scheme and C programs, especially because it is nearly impossible to establish a set of representative benchmark programs. As much as possible we have tried to use C programs that perform the same kind of computation as our Scheme programs. The first one, **Amd** is a test released by AMD, which uses it to estimate the speed of its processors. The second, **Li** is a Lisp interpreter implemented in C, which is part of the Spec 95 benchmark suit. The last one, **Gcc** is the special version of the GNU-C compiler that is also included in the Spec 95 suite.

Prgm	**Function call per second**
Amd (c)	2,787,920 cs^{-1}
Queens (scm)	3,524,345 cs^{-1}
Spec95 Li (c)	5,280,070 cs^{-1}
Eval (scm)	1,581,905 cs^{-1}
Spec95 Gcc (c)	1,445,991 cs^{-1}
Bootstrap (scm)	1,453,832 cs^{-1}

These time figures show that the function call frequency is similar for Scheme and C. In particular, the frequency is astonishingly close for **Bootstrap** and **Gcc**, both of which are compilers. Consequently, the execution time spent in each Scheme function is proportionally close to the time spent in each C function. Thus, there is no a priori reason to believe that Scheme requires different sampling techniques than C.

The second step of our validation was measurement of the accuracy of KPROF's allocation profiler. For this, we have built a special version of the Bigloo runtime system that reports exactly on heap allocation performed by each function. In that version, each time an allocator a is called, every active function on the execution stack is marked as calling a. When execution completes, all this information is dumped into a file. This experimental runtime system is unrealistic because it is far too slow. With this version, execution of the **Bootstrap** benchmark requires about 8 hours on our hardware configuration. However, this slow implementation is still sufficient for estimating the accuracy of the indirect allocations reported by KPROF.

For the small and medium sized programs we have tested (including **Queens** and **Eval**), KPROF allocation profiling is very precise. We observed that function allocation estimates computed by the algorithm presented in Section 12.3.4 have an error rate of less than 5 %.

For larger programs, the accuracy of the estimates varies. If we inspect a function f that indirectly calls an allocator a, the quality of the estimate is highly dependent on the *length* of the path from f to a, that is, how many functions calls are needed to reach a from f. For instance, in Figure 12.4 the longest path from `CONCMAP` to `CONS` is 6, the shortest is 2. If shortest paths from f to a count a lot, that is shortest paths are more frequently traversed (in Figure 12.4 the shortest path is important because the vertex from `CONCMAP` to `APPEND-2` represents 33.3 % of all the calls made by `CONCMAP`) then the estimate for f is reliable. Otherwise, it is imprecise. In the compiler bootstrap benchmark the paths are relatively small, KPROF reports that the control flow analysis (`cfa` stage on Figure 12.8) allocates 1% of all closures and 1.2 % of the `CONS` cells, while actually it is responsible for 1.25 % of procedures and 2 % of the `CONS` cells. On the other hand, when paths are longer the estimates are imprecise. For the AST construction (the `ast` label on Figure 12.8), KPROF estimates the number of `CONS` cells to be 4.5 % and the number of procedures to be 7.1 %, while exact measures report 16 % of `CONS` cells and 12 % of procedures. The errors in KPROF's estimates are inherent to the lack of exact information about the dynamic paths. Using Gprof, we don't think it is possible to produce more reliable results. However, in addition to the current estimates, KPROF could report on their accuracy. This new information could be computed from the number of paths and their length from a function to an allocator and it could be presented on the same window as the estimates, or it could be based on a call stack sampling technique.

12.6 Related work

KPROF is implemented on top of GPROF [98, 75]. Thus, KPROF inherits GPROF's run-time instrumentation and its computation of the dynamic call graph. In the past, alternative techniques to gather profile information have been proposed [10]; however these techniques seem less accurate than GPROF's ones.

`Mprof` [264] is an allocation profiler. It relies on techniques which are similar to those of KPROF but with a different implementation. Instead of re-using GPROF results, `mprof` implements its own monitoring and textual displayer. Instrumented applications record all the call chains that lead to allocation sites. In order to avoid overly large record files, `mprof` compacts its records with a technique slightly more accurate than the GPROF's one.

The Haskell community has been fairly active at exploring heap profiling for lazy functional languages. Some of this work concentrates on studying the graphical means to display profiling information [187]. Other research concentrates on providing a semantics to the evaluation of lazy languages with profiling [194]. Yet other work focuses on issues close to memory leak detection [186, 185].

The graphical display advocated in an early paper by Runciman and Wakeling [187] is used in all other Haskell studies. This representation is totally different from the one we use. It displays, over the whole execution of a program, the heap composition. That is, the heap size and the percentage of pairs, strings, vectors, etc. It could be that such a nice representation enables faster understanding of the memory allocations of a program, in particular when it is used to display a *producer* view. A producer is a function that calls some allocators. This view definitively helps in understanding who's responsible for the allocations of a program and, more importantly, how long the objects live. We think this representation could point out memory leaks. However, we don't think it helps much with fixing these leaks. To fix a leak, one has to understand why objects fail to be reclaimed, which is not reported by heap profiling.

A more recent paper by Röjemo and Runciman [179] presents a study that could help in understanding memory leaks. They present the so-called *lag*, *drag*, *void* and *use* phases. They characterize wasted space as that occupied by objects that have passed their last use, or have been allocated but are not yet in use. They present tools to analyze the presence of such objects. Execution has to be completed before any profiling information can be reported. This approach is orthogonal to the one presented here and could be usefully combined with it.

`Profile` forms are close to the `ghc` Haskell implementation's "set cost center" construction (`scc` in short) [193, 192, 194]. Cost centers are more central to Haskell because they form the basis of profiling for this lazy programming language. To KPROF, `profile` is only a convenience that enhances the presentation of the generated profiles. In addition, both cost center construct and `profile` forms allow fine grain tuning. The more they are introduced, the more optimization is inhibited. These two forms require source code changes, but they allow exact control of profiling.

KBDB has been inspired by the work of De Pauw and Sevitski [62], in which the authors present Jinsight, a tool for visualizing reference patterns and tracing memory leaks in Java programs. Jinsight operates on an instrumented JVM that keeps track of *back pointers*, linking all the live objects of the heap. To avoid displaying individual objects they classify objects using their type (their Java class). By successively refined requests, it is possible to determine which objects are responsible for memory leaks. KBDB owes much to Jinsight : the definition of a memory leak and the basic framework for using it comes from that work. However, KBDB differs greatly in its representation of objects and in its runtime overhead. Jinsight creates records describing the profiled heap. These records are very large. Three examples are presented for which the records are respectively 32, 46 and 49 times larger than the profiled heaps (110Kb heap is recorded in a 3.5Mb record file, 20Kb in a 0.9Mb record and 25Kb in a 1.2Mb). Even worse, the memory needed by Jinsight is much larger than the recorded files (in the better case, Jinsight uses 370 times more memory than the heap profiled). To visualize our 17Mb heap (the heap size of the **Bootstrap** of Section 12.5.1) we would have required a 6Gb heap !

A very primitive pointer backtracing facility for leak detection was incorporated in the Xerox Portable Common Runtime by the second author around 1995. Although it required no per object space overhead, this relied on a full search of the heap to trace back one pointer level in the reference chain. In spite of the obvious performance issues, it proved useful in tracking down real problems in a large system. However the implementation was tricky, in that it tended to suffer from "accidental reflection" : It was easy to mistreat the local variables used by the backtracing code itself as roots.

12.7 Future work

As we have reported, for large programs such as Bigloo itself, KPROF supplies rough estimates. We are currently exploring another strategy for allocation profiling. We are developing a technique that computes exact figures. Because it will be slower than the current one it won't replace it. It will be an additional tool that could be deployed when the current KPROF fails.

KBDB is a heap visualization tool. It displays live cells in the heap. Currently, three different visualizations are implemented : a type based classification, an age based classification and a leak detection view. KBDB could be used to display information *à la* Haskell. That is, we could implement a new KBDB module that displays producer graphs such as the one presented in [187]. Our run time provides enough information for this. In addition, we think it could be interesting to provide the user with a statistical information about the heap, such as the average number of GCs survived by certain objects. We think a GC is a wonderful tool for profiling and debugging because a GC keeps track of all pointers. Since a GC is able to scan an entire heap, it can answer questions such as "how many objects are pointing to that other object". This could be used by the debugger to exhibit, on an on-demand basis, sharing properties.

Conclusion

In that paper we have presented two memory profilers for the Scheme programming language. The first one KPROF, reports on allocations that take place in program executions. It acts as a regular profiler. A sampling execution is recorded and, afterwards, allocation profiling information is reported. KPROF can point out which functions consume memory. KPROF imposes a low run time overhead and in particular it does not enlarge memory consumption of profiled executions. The second tool, KBDB acts as a debugger. Programs are stepped and, on a on-demand basis, heaps can be visualized. The heaps are then represented by 2 dimensional pictures in which each live cell of a heap is represented by pixels. Zooming in that picture enables cells selection. KBDB is used to fix memory leaks. It slows down executions of about 10 times and it enlarges heap size of about 5 times. However, KBDB is still efficient enough to be practical at inspecting the 17MB heap of the bootstrap of our Scheme compiler.

Acknowledgments

Many thanks to Simon Peyton-Jones, Erick Gallesio, Céline and to Brian Lynn for their helpful feedbacks on this work and to Al Demers for his suggestion about the back-pointers implementation.

Annexe A

This program implements the solution of the queens problem. It explores all the possible configurations for queens on a chess board. Each configuration is represented by a Scheme list. Amongst the configurations the algorithm selects the one that are solution to the problem. Most of the execution of this program is spent allocating and collecting lists.

The `queens` Scheme module :

```
;; Queens problem from an early implementation in Lazy
ML by L. Augustsson.
(module queens
   (main main))

(define (mmap f)
   (define (map-rec l)
      (if (null? l) '() (cons (f (car l)) (map-rec (cdr l)))))
   map-rec)

(define (filter p l)
   (cond
      ((null? l)
       '())
      ((p (car l))
       (cons (car l) (filter p (cdr l))))
      (else
       (filter p (cdr l)))))

(define count
   (let ((memo '()))
      (lambda (from to)
         (define (inner from to)
            (if (>fx from to)
                '()
                (cons from (inner (+fx 1 from) to))))
         (let* ((key (cons from to))
                (cell (assoc key memo)))
            (if (pair? cell)
                (cdr cell)
                (let ((new (inner from to)))
                   (set! memo (cons (cons (cons from to) new) memo))
                   new))))))

(define (concmap f l)
   (if (null? l)
       '()
       (append (f (car l)) (concmap f (cdr l)))))

(define (nsoln nq)
   (define (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))))))))
   (define (ok-n l)
      (if (null? l) #t (safe 1 (car l) (cdr l))))
   (let ((pos-l (count 1 nq)))
      (define (testcol b)
         (filter ok-n (profile pos-l ((mmap (lambda
lambda (q) (cons q b))) pos-l))))
      (define (gen n)
         (if (=fx n 0) '(()) (concmap testcol (gen (-fx n 1)))))
      (length (gen nq))))

(define (nsoln-a nq)
   (define (ok l)
      (if (null? l)
          #t
          (let* ((x (car l)) (l (cdr l)))
             (define (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))))
   (define (gen-n n)
      (if (=fx n 0)
          '(())
          (concmap (lambda (b)
                      (filter ok ((mmap (lambda (q) (cons q b)))
                                  (count 1 nq))))
                   (gen-n (-fx n 1)))))
   (length (gen-n nq)))

(define (main argv)
   (let* ((sol1 (profile sol1 (nsoln 10)))
          (sol2 (profile sol2 (nsoln-a 10))))
      (print "queens(" 10 "): " sol1 " " sol2)))
```

Bibliographie

[1] . **Special Track on Debugging**. *Communications of the ACM*, 40(4), April 1997.

[2] A. Adl-Tabatabai, M. Cierniak, G-Y. Lueh, V. Parikh, and J. Stichnoth. **Fast, Effective Code Generation in a Just-In-Time Java Compiler**. In *Conference on Programming Language Design and Implementation*, pages 280–190, June 1998.

[3] A. Aho, R. Sethi, and J. Ullman. **Compilers : Principles, Techniques and Tools**. Addison-Wesley, 1986.

[4] M. Alberganti. **L'insoutenable légèreté de l'informatique**. *Le Monde*, 55(17081), 26-27 Décembre 1999.

[5] AMD. **AMD-K6-2, Processor Code Optimization, *Application Note***. Technical Report 21924, Rev B, Advanced Micro Devices, Inc., May 1998.

[6] A. Appel. **Garbage Collection can be faster than stack allocation**. *Information Processing Letters*, 25(4) :275–279, June 1987.

[7] A. Appel. **Runtime Tags Aren't Necessary**. Technical Report CS-TR-142-88, Princeton University, 1989.

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

[9] A. Appel. **Loop Headers in λ-calculus or CPS**. *Lisp and Symbolic Computation*, 7 :337–343, December 1994.

[10] A. Appel, F. Duba, D. MacQueen, and A Tomach. **Profiling in the Presence of Optimization and Garbage Collection**. Technical Report CS-TR-197-88, Princeton University, November 1988.

[11] A. Appel and Z. Shao. **Empirical and Analytic Study of Stack versus Heap Cost for Languages with Closures.** *Journal of Functional Programming*, 6(1) :47–74, 1996.

[12] Eastern Research Apple Computer and Technology. **Dylan, An object-oriented dynamic language**. Apple Computer, Inc., April 1992.

[13] M. Ashley. **The effectiveness of flow analysis for inlining**. In *Int'l Conf. on Functional Programming*, pages 99–111, June 1997.

[14] G. Attardi. **The Embeddable Common Lisp**. Technical Report unpublished, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy, 1994.

[15] A. Ayers. **Abstract Analysis and Optimization of Scheme**. PhD thesis, Massachusetts Institute of Technology, September 1993.

[16] A. Ayers, R. Schooler, and R. Gottlieb. **Aggresive Inlining**. In *Conference on Programming Language Design and Implementation*, pages 134–145, June 1997.

[17] H. Baker. **Inlining Semantics for Subroutines which are recursive**. *ACM Sigplan Notices*, 27(12) :39–46, December 1992.

[18] H. Baker. **CONS Should Not CONS Its Arguments, Part II : Cheney on the M.T.A <1>**. Technical Report TR-Nbl-94-01, , 1994.

[19] A. Banerjee and D. Schmidt. **Stackability in the Simple-Typed Call-By-Value Lambda Calculus**. In *1fst Static Analysis Symposium*, pages 131–146, Namur, Belgium, September 1994.

[20] J.F. Bartlett. **Compacting Garbage Collection with Ambiguous Roots**. Research Report 88/2, Digital Western Research Laboratory, Palo Alto, CA, February 1988.

[21] J.F. Bartlett. **Mostly-Copying Garbage Collection Picks Up Generations and C++**. Technical Note TN-12, Digital Western Research Laboratory, Palo Alto, CA, October 1989.

[22] J.F. Bartlett. **Scheme->C a Portable Scheme-to-C Compiler**. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, CA, January 1989.

[23] B. Blanchet. **Escape Analysis for Object-Oriented Languages. Aplication to Java**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, November 1999.

[24] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. **Common lisp object system specification**. In *special issue*, number 23 in SIGPLAN Notices, September 1988.

[25] H.J. Boehm. **Space Efficient Conservative Garbage Collection**. In *Conference on Programming Language Design and Implementation*, number 28, 6 in SIGPLAN Notices, pages 197–206, 1993.

[26] H.J. Boehm. **Simple Garbage-Collector-Safety**. In *Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 1996.

[27] H.J. Boehm and M. Weiser. **Garbage Collection in an Uncooperative Environment**. *Software — Practice and Experience*, 18(9) :807–820, September 1988.

[28] F. Bourdoncle. **Abstract Debugging of Higher-Order Imperative Languages**. In *Conference on Programming Language Design and Implementation*, pages 46–55, 1993.

[29] F. Bourdoncle and S. Merz. **Type Checking Higher-Order Polymorphic Multi-Methods**. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[30] P. Breton. **La tribu informatique**. Éditions A.M. Métailié, 5 rue de Savoie 75006 Paris, 1990.

[31] L. Cannon, R. Elliot, L. Kirchoff, J. Miller, J. Milner, R. Mitze, E. Schan, N. Whittinton, D. Spencer, H. Keppel, and M. Brader. **Recommended C Style and Coding Standards**, June 1990.

[32] M. Carlsson and T. Hallgren. FUDGETS **- A Graphical User Interface in a Lazy Functional Language**. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.

[33] G. Castagna. **Object-Oriented Programming – A Unified Foundation**. Birkhäuser, Boston, 1997.

[34] E. Chailloux. **Compilation des langages fonctionnels : CeML un traducteur ML vers C**. Thèse de doctorat d'université, Université Paris 7, Paris (France), November 1991.

[35] E. Chailloux. **An efficient way to compile ML to C**. In *Workshop on ML and its applications*, San Francisco, California, USA, 1992. ACM SIGPLAN.

[36] J. Chailloux. **La machine LLM3**. Technical Report 55, INRIA, France, June 1985.

[37] J. Chailloux, M. Devin, F. Dupont, J.M. Hullot, B. Serpette, and J. Vuillemin. **Le_Lisp version 15.2. Le manuel de référence**. Technical Report L-003, INRIA-Rocquencourt, France, 1986.

[38] C. Chambers. **The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages**. Technical report stan-cs-92-1240, Stanford University, Departement of Computer Science, March 1992.

[39] C. Chambers. **The Cecil Language : Specification and Rationale**. Technical Report 93-03-05, University of Washington, Department of computer Science and Engineering, March 1993.

[40] C. Chambers. **Predicate Classes**. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[41] C. Chambers. **Towards Diesel, a Next-Generation OO Language after Cecil**. In *Int'l Workshop on Foundations of Object-oriented Languages*, 1998. invited talk.

[42] C. Chambers and D. Ungar. **Customization : Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language**. In *Conference on Programming Language Design and Implementation*, Portland, OR, USA, June 1989.

[43] D. Chase. **Safety considerations for storage allocation optimizations**. In *Conference on Programming Language Design and Implementation*, Atlanta, Georgia, USA, June 1988.

[44] S. Chiba. **A Metaobject Protocol for C++**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1995.

[45] J-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. **Escape Analysis for Java**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, November 1999.

[46] K. Claessen, T. Vullinghs, and E. Meijer. **Structuring Graphical Paradigm in TkGofer**. In *Int'l Conf. on Functional Programming*, 1997.

[47] W. Clinger. **Macros that work**. In *Symposium on Principles of Programming Languages*, 1991.

[48] W. Clinger. **Proper Tail Recursion and Space Efficiency**. In *Conference on Programming Language Design and Implementation*, June 1998.

[49] W. Clinger and J. Rees. **The Revised^4 Report on the Algorithmic Language Scheme**, November 1991.

[50] G. Collins. **A method for overlapping and erasure of lists**. *Communications of the ACM*, 3(12) :655–657, 1960.

[51] K. Cooper, M. Hall, and L. Torczon. **Unexpected Side Effects of Inline Substitution : A Case Study**. *ACM Letters on Programming Languages and Systems*, 1(1) :22–31, 1992.

[52] P. Cousot and R. Cousot. **Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints**. In *Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, January 1977.

[53] J. Cox. **Object-Oriented Programming : an Evolutionary Approach**. Addison-Wesley, 1986.

[54] R. Cridlig. **An optimizing ML to C compiler**. In *Workshop on ML and its applications*, San Francisco, California, USA, 1992. ACM SIGPLAN.

[55] M. Cusumano and D. Yoffie. **What Netscape Learned From Cross-Platform Software Development**. *Communications of the ACM*, 42(10) :72–78, October 1999.

[56] M. Dalheimer. **Programming with Qt**. O'Reilly, 1st edition, april 1999.

[57] M. Dalheimer and T. Dalheimer. **KDE. Anwendung und Programmierung**. O'Reilly, June 1999.

[58] L. Damas and R. Milner. **Principle Type Inference for Functional Programs (extended abstract)**. In *9th ACM Symposium on Principle of Programming Languages*, pages 207–212, 1982.

[59] J. Davidson and A. Holler. **A Study of a C Function Inliner**. *Software — Practice and Experience*, 18(8) :775–790, August 1988.

[60] J. Davidson and A. Holler. **Subprogram Inlining : A Study of its Effects on Program Execution Time**. *IEEE Transactions on Software Engineering*, 18(2) :89–101, February 1992.

[61] H. Davis, P. Parquier, and N. Séniak. **Sweet Harmony : the Talk/C++ Connection**. In *Conference Record of the 1994 ACM Conference on Lisp and Functional Programming*, pages 121–127, Orlando, Florida, USA, 1994.

[62] W. De Pauw and G. Sevitski. **Visualizing Reference Patterns for Solving Memory Leaks in Java**. In *Proceedings ECOOP'99*, pages 116–134, Lisbon, Portugal, June 1999.

[63] F. Dean, D. Grove, and C. Chambers. **Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis**. In *European Conference on Object-Oriented Programming*, pages 77–101, June 1995.

[64] G. Dean, J. DeFouw, D. Grove, V. Litvinov, and C. Chambers. **Vortex : An Optimizing Compiler for Object-Oriented Languages**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1996.

[65] J. Dean and C. Chambers. **Towards Better Inlining Decisions Using Inlining Trials**. In *Conference on Lisp and Functional Programming*, pages 273–282, Orlando, Florida, USA, June 1994.

[66] D. Detlefs and O. Agesen. **Inlining of Virtual Methods**. In *Proceedings ECOOP'99*, pages 258–278, Lisbon, Portugal, June 1999.

[67] G. Dowek, A. Felty, H. Herbelin, and G. Huet. **The COQ proof assistant user's guide**. Technical Report 154, Inria-Rocquencourt, France, 1993.

[68] K. Dybvig. **Chez Scheme User's Guide**. Cadence Research Systems, 1998.

[69] K. Dybvig, D. Friedman, and C. Haynes. **Expansion-Passing Style : Beyond Conventional Macros**. In *Conference on Lisp and Functional Programming*, pages 143–150, 1986.

[70] K. Dybvig, R. Hieb, and C. Bruggeman. **Syntactic abstraction in Scheme**. *Lisp and Symbolic Computation*, 5(4) :295–326, 1993.

[71] Ericsson. **Erlang**. http ://www.erlang.se, 1998.

[72] M. Feeley, J. Miller, G. Rozas, and J. Wilson. **Compiling Higher-Order Languages into Fully Tail-Recursive Portable C**. Technical Report Rapport technique 1078, Université de Montréal, Département d'informatique et r.o., August 1997.

[73] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. **The DrScheme Project : An Overview**. SIGPLAN *Notices*, 1998.

[74] M. Felleisen and D. Friedman. **A Little Java, A Few Patterns**. MIT Press, Cambridge, Mass., USA, 1998.

[75] J. Fenlason and B. Baccala. **GNU-gprof : user manual**. Technical report, Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, 1997.

[76] R. Findler, C. Flanagan, M. Flatt, S. Krishnamuryhi, and M. Felleisen. **DrScheme : A Pedagogic Programming Environment for Scheme**. In *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, 1997.

[77] R. Findler and M. Flatt. **Modular Object-Oriented Programming With Units and**

Mixins. In *Int'l Conf. on Functional Programming*, Baltimore, MD, USA, September 1998.

[78] S. Finne and S. Peyton Jones. **Composing Haggis**. In *Fifth Eurographics Workshop on Programming Paradigm for Computer Graphics*, Maastricht, NL, September 1995.

[79] C. Flanagan. **Effective Static Debugging via Componential Set-Based Aanlysis**. PhD thesis, Rice University, 1997.

[80] C. Flanagan, M. Flatt, S. Krishnamuryhi, S. Weirich, and M. Felleisen. **Static Debugging : Browsing the Web of Program Invariants**. In *Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 1996.

[81] M. Flatt and M. Felleisen. **Units : Cool Modules for HOT Languages**. In *Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[82] M. Flatt, R. Findler, S. Krishnamuryhi, and M. Felleisen. **Programming Languages as Operating Ssystems (*or* Revenge of the Son of the Lisp Machine)**. In *Int'l Conf. on Functional Programming*, 1999.

[83] M. Flatt, S. Krishnamurthi, and M. Felleisen. **Classes and Mixins**. In *Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[84] Franz Inc. **Allegro Common 5.0**. http :-//www.franz.com/, 1998.

[85] J. Furuse and J. Garrigue. **A label-selective lambda-calculus with optional arguments and its compilation method**. Technical Report RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, October 1995.

[86] R. Gabriel. **Lisp : Good News, Bad News, How to Win Big**. http ://www.ai.mit.edu-/articles/good-news/good-news.html, 1991.

[87] E. Gallesio. **STk Reference Manual**. Technical Report RT 95-31a, I3S-CNRS/Univ. of Nice–Sophia Antipolis, July 1995.

[88] E. Gallesio. **Designing a Meta Object Protocol to Wrap a Standard Graphical Toolkit**. In *ISOTAS*, 1996.

[89] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. **Design patterns : Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.

[90] A. Goldberg. **Smalltalk-80 : The interactive programming environnent**. Addison-Wesley, 1983.

[91] A. Goldberg and D. Robson. **Smalltalk-80 : The Language and Its Implementation**. Addison-Wesley, 1983.

[92] B. Goldberg and G. Park. **Higher order escape analysis : Optimizing stack allocation in functional program implementations**. In *European Symposium on Programming*, number 432 in Lecture Notes on Computer Science, pages 152–160, May 1990.

[93] M. Goncalves and A. Appel. **Cache Performance of Fast-Allocating Programs**. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 293–305, 1995.

[94] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadworth. **A meta-language for interactive proofs in LCF**. In *Symposium on Principles of Programming Languages*, pages 119–130, 1978.

[95] J. Gosling, B. Joy, and G. Steele. **The Java™ Language Specification**. Addison-Wesley, 1996.

[96] J. Gosling, F. Yellin, and the Java Team. **The Java™ Application Programming Interface, Volume 2 : Window Toolkit and Applets**. Addison-Wesley, 1996.

[97] J. Goubault. **Generalized Boxings, Congruences and Partial Inlining**. In *1fst Static Analysis Symposium*, pages 147–161, Namur, Belgium, September 1994.

[98] S. Graham, P. Kessler, and McKusik M. **gprof : a call graph execution profiler**. In *Compiler Construction, SIGPLAN Notices 17(4)*, pages 120–126, 1982.

[99] D. Gudeman. **Representing Type Information in Dynamically Typed Languages**. Technical report, University of Arizona, Departement of Computer Science, Gould-Simpson Building, The University of Arizona, Tucson, AZ 85721, April 1993.

[100] J. Hamer. **Un-Mixing Inheritance with Classifiers**. In *Multiple Inheritance and Multiple Subtyping : Position papers of the ECOOP'92 Workshop*, LNCS 707, pages 6–9, Utrect, Netherlands, July 1992. Springer-Verlag.

[101] L.T. Hansen. **The impact of programming style on the performance of Scheme programs**. M.s. thesis, University of Oregon, August 1992.

[102] C. Hanson. **Efficient Stack Allocation for Tail-Recursive Languages**. In *Conference on Lisp and Functional Programming*, pages 106–118, Nice, France, 1990.

[103] Harlequin. **ML Works**. http ://www-.harlequin.com, 1998.

[104] R. Harper, R. Milner, and M. Tofte. **The Definition of Standard ML**. MIT-press, 1990.

[105] R. Harper and G. Morrisset. **Compiling polymorphism using intensional type analysis**. In *22 Annual* ACM *Symposium on Principles of Programming Languages*, pages 130–141, New York, NY, USA, January 1995.

[106] P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, P. Lee, X. Leroy, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, P. Weis, and P. Wentworth. **Pseudoknot : a Float-Intensive Benchmark for Functional Compilers**. *Journal of Functional Programming*, 6(4) :621–655, 1996.

[107] N. Heintze and D. McAllester. **On The Complexity of Set-Based Analysis**. In *Int'l Conf. on Functional Programming*, pages 150–163, 1997.

[108] F. Henglein. **Global Tagging Optimization by Type Inference**. In *Conference on Lisp and Functional Programming*, 1992.

[109] C. Hewitt and B. Smith. **Towards a Programming Apprentice**. *IEEE Transactions on Software Engineering*, 1(1) :26–45, March 1975.

[110] U. Hölzle, C. Chambers, and D. Ungar. **Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches**. In *European Conference on Object-Oriented Programming*, pages 22–36, 1991.

[111] P. Hudak. **A semantic model of reference counting and its abstraction**. In *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[112] P. Hudak and P. Wadler. **Report on the programming language Haskell**. YALEU/DCS/RR-777, Yale University, 1990.

[113] W. Hwu and P. Chang. **Inline Function Expansion for Compiling C Programs**. In *Conference on Programming Language Design and Implementation*, Portland, Oregon, USA, June 1989. ACM.

[114] IEEE Std 1178-1990. **IEEE Standard for the Scheme Programming Language**. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[115] ISO/IEC. **9899 Programming Language - C**. Technical Report DIS 9899, ISO, July 1990.

[116] S. Jagannathan and A. Wright. **Effective Flow Analysis for Avoiding Run-Time Checks**. In *2nd Static Analysis Symposium*, Lecture Notes on Computer Science, pages 207–224, Glasgow, Scotland, September 1995.

[117] S. Jagannathan and A. Wright. **Flow-directed Inlining**. In *Conference on Programming Language Design and Implementation*, Philadelphia, Penn, USA, May 1996.

[118] T. Johnson. ***Lambda Lifting* : Transforming Programs to Recursive Equations**. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, 1985.

[119] R. Jones and R. Lins. **Garbage Collection : Algorithms for Automatic Dynamic Memory Management**. John Wiley & Sons, Chichester, England, 1996.

[120] R. Kelsey, W. Clinger, and J. Rees. **The Revised^5 Report on the Algorithmic Language Scheme**. *Higher-Order and Symbolic Computation*, 11(1), September 1998.

[121] B. Kernighan. **A Descent into Limbo**. Technical report, Bell Labs, `www.cs.bell-labs.com/inferno`, July 1996.

[122] B. Kernighan and R. Pike. **The Unix Programming Environment**. Prentice-Hall, Englewood Cliffs, NJ, USA, 1984.

[123] G. Kiczales, J. des Rivières, and D. Bobrow. **the Art of the Metaobject Protocol**. MIT Press, Cambridge, Mass., USA, 1992.

[124] P. Kinnucan. **JDE User's Guide**. http :-//sunsite.auc.dk/jde/, 1998.

[125] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. **Hygienic macro expansion**. In *Conference on Lisp and Functional Programming*, pages 151–161, 1986.

[126] A. Krall, J. Vitek, and N. Horspool. **Near optimal hierarchical encoding of types**. In *Proceedings ECOOP'97*. Springer-Verlag, 1997.

[127] D. Kranz, R. Kesley, J. Rees, P. Hudak, J. Philbin, and N. Adams. **ORBIT : An Optimizing Compiler for Scheme**. In *Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM.

[128] D.A. Kranz. **ORBIT : An Optimizing Compiler For Scheme**. PhD thesis, Yale university, February 1988.

[129] S. Krishnamurthi, M. Felleisen, and D. Friedman. **Synthesizing Object-Oriented and Function Design to Promote Re-use**. In *Proceedings ECOOP'98*, pages 93–113, July 1998.

[130] M. Lee and A. Stepanov. **The Standard Template Library**. `http ://www.cs.rpi-.edu/musser/doc.ps`.

[131] D. Lenkov, M. Mehta, and S. Unni. **Type Iidentification in C++**. In *USENIX C++ Conference*, April 1991.

[132] X. Leroy. **Efficient data representation in polymorphic languages**. In P. Deransart and J.Małuszyński, editors, *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, volume 456 of *Lecture Notes on Computer Science*. Springer-Verlag, 1990. Also available as INRIA research report 1264.

[133] X. Leroy. **The ZINC experiment : an economical implementation of the ML language**. Technical Report RT-117, Inria-Rocquencourt, France, February 1990.

[134] X. Leroy. **Unboxed objects and polymorphic typing**. In *Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, January 1992.

[135] X. Leroy. **Applicative functors and fully transparent higher-order modules**. In *Symposium on Principles of Programming Languages*, pages 142–153, January 1996.

[136] X. Leroy. **Compilation techniques for function and object-oriented languages**. In *Conference on Programming Language Design and Implementation*, page tutorial, June 1998.

[137] X. Leroy. **Objects, Classes and modules in Objective Caml**. In *Int'l Conf. on Functional Programming*, September 1999. invited lecture.

[138] H. Liberman and C. Hewitt. **A Real-Time Garbage Collector Based on the Lifetimes of Objects**. *Communications of the ACM*, 26(6) :419–429, June 1983.

[139] T. Lindholm and F. Yellin. **The Java Virtual Machine**. Addison-Wesley, 1996.

[140] S. Lippman. **Inside The C++ Object Model**. Addison-Wesley, 1996.

[141] M. Lutz. **Programming Python**. O'Reilly & Associates, 1996.

[142] R. MacLachlan. **The Python Compiler for CMU Common Lisp**. In *Proceedings of the 1992* ACM *Conference on Lisp and Functional Programming*, pages 235–246, San Francisco, CA, USA, June 1992.

[143] D. MacQueen. **Modules for Standard ML**. In *Conference on Lisp and Functional Programming*, 1984.

[144] J. McCarthy. **Recursive functions of symbolic expressions and their computation by machine – I**. *Communications of the ACM*, 3(1) :184–195, 1960.

[145] J. McCarthy and *al.* **LISP 1.5 Programmer's Manual**. The M.I.T. press, 1962.

[146] B. Meyer. **Object-Oriented Software Construction**. Prentice Hall, 1988.

[147] T. Millstein and C. Chambers. **Modular Statically Typed Multimethods**. In *European Conference on Object-Oriented Programming*, pages 279–303, June 1999.

[148] D. Moon and D. Weinreb. **Lisp machine manual**. Technical Report, MIT Artificial Intelligence Laboratory, 1981.

[149] S. Muchnick. **Advanced Compiler Design & Implementation**. Morgan Kaufmann, 1997.

[150] Pierre-Alain Muller. **Modélisation objet avec UML**. Eyrolles, 1997.

[151] A. Myers, J. Bank, and B. Liskov. **Parametrized Types for Java**. In *Symposium on Principles of Programming Languages*, pages 132–145, January 97.

[152] R. Nobble and C. Runciman. **Functional Languages and Graphical User Interfaces – a review and a case study**. Technical Report 94-223, Department of computer Science, University of York, February 1994.

[153] M. Odersky and P. Wadler. **Pizza into Java : Translating theory into practice**. In *Symposium on Principles of Programming Languages*, pages 146–159, January 97.

[154] J. Ousterhout. **An X11 toolkit based on the Tcl Language**. In *USENIX Winter Conference*, pages 105–115, January 1991.

[155] J. Ousterhout. **Tcl and the Tk toolkit**. Addison-Wesley, 1994.

[156] D. Patterson and J. Hennessy. **Computer Architecture, a Quantitative Approach**. Morgan Kaufmann, 2nd edition, 1996.

[157] D. Patterson and J. Hennessy. **Computer Organization and Design *The hardware/software interface***. Morgan Kaufmann, 2nd edition, 1998.

[158] H. Pennington. **Gtk+/Gnome Application Development**. New Riders Publishing, 1999.

[159] F. Pessaux and X. Leroy. **Typed-based analysis of uncaught exceptions**. In *Symposium on Principles of Programming Languages*, 1999.

[160] S. Peyton Jones. **The Implementation of Functional Programming Languages**. Prentice-Hall, New York, 1987.

[161] S. Peyton Jones and J. Launchbury. **Unboxed Values as First Class Citizens in a Non-Strict Functional Language**. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, Cambridge, MA, USA, August 1991.

[162] S. Peyton Jones and S. Marlow. **Secrets of the Glasgow Haskell Compiler Inliner**. In *Implementation of Declarative Languages*, Paris, France, September 1999.

[163] Pitac. **President's Information Technology Advisory Committee Report to the President**. Technical Report http ://www-.ccic.gov, c/o National Coordination Office for Computing, Information, and Communications, February 1999.

[164] I. Piumarta and F. Riccardi. **Optimizing direct threaded code by selective inlining**. In *Conference on Programming Language Design and Implementation*, pages 291–300, June 1998.

[165] M. Plezbert and R. Cytron. **Does "Just In Time" = "Better Late Than Never"?** In *Symposium on Principles of Programming Languages*, pages 120–131, 1997.

[166] L. Prechelt. **Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences**. *Communications of the ACM*, 42(10) :109–112, October 1999.

[167] W. Press, B. Flannery, S. Teukolsky, and Vetterling W. **Numerical Recipes in C**. Cambridge University Press, 1988.

[168] C. Queinnec. **Compilation of Non-Linear, Second Order Pattern on S-Expressions**. In P. Deransart and J. Maluszyński, editors, *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, number 245 in LNCS, pages 340–357, August 1990.

[169] C. Queinnec. **Le filtrage : une application de (et pour) Lisp**. InterÉditions, Paris (France), 1990.

[170] C. Queinnec. **Designing** MEROON **V3**. In *Workshop on Object-Oriented Programming in Lisp*, 1993.

[171] C. Queinnec. **Lisp In Small Pieces**. Cambridge University Press, 1996.

[172] C. Queinnec and J-M. Geffroy. **Partial Evaluation applied to Symbolic Pattern Matching with Intelligent Backtrack**. In M Billaud, P Castéran, MM Corsini, K Musumbu, and A Rauzy, editors, *Workshop on Static Analysis*, number 81-82 in bigre, pages 109–117, Bordeaux (France), September 1992.

[173] C. Queinnec and N. Séniak. **Puzzling with current puzzles**. *Lisp Pointeurs*, 2(3), January 1989.

[174] C. Queinnec and B. Serpette. **A Dynamic Extent Control Operator for Partial Continuations**. In *Symposium on Principles of Programming Languages*, pages 174–184, 1991.

[175] G. Ramalingam and H. Srinivasan. **A Member Lookup Algorithm for C++**. In *Conference on Programming Language Design and Implementation*, pages 18–30, 1997.

[176] D. Rémy and J. Vouillon. **Obective ML : A simple object-oriented extension of ML**. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[177] F. Rivard. **Évolution du comportement des objets dans les langages à classes réflexifs**. 98-1-info, École des Mines de Nantes, 1998.

[178] N. Röjemo. **Generational Garbage Collection without Temporary Space Leaks**. In *Int'l Workshop on Memory Management*, 1995.

[179] N. Röjemo and C. Runciman. **Lag, drag, void and use – heap profiling and space-efficient compilation revised**. In *1fst Int'l Conf. on Functional Programming*, pages 34–41, Philadelphia, Penn, USA, May 1996.

[180] J. R. Rose and H. Muller. **Integrating the Scheme and C languages**. In *Conference Record of the 1992 ACM Conference on Lisp and Functional Programming*, pages 247–259, 1992.

[181] F. Rouaix. **A Web navigator with applets in Caml**. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28 :7–11, pages 1365–1371. Elsevier, May 1996.

[182] G.J. Rozas. **Liar, an Algol like compiler for Scheme**. S.b. thesis, Massachusetts Institute of Technology, Cambridge, Mass., January 1984.

[183] G.J. Rozas. **Taming the Y operator**. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 226–234. ACM, June 1992.

[184] C. Ruggieri and T. Murtagh. **Lifetime Analysis of Dynamically Allocated Object**. In *Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[185] C. Runciman and N. Röjemo. **Heap profiling for space efficiency**. In E. Meijer J. Launchbury and T. Sheard, editors, *LNCS Vol. 1129, 2nd Intl. School on Advanced Functional Programming*, pages 159–183, August 1996.

[186] C. Runciman and N. Röjemo. **New dimensions in heap profiling**. *Journal of Functional Programming*, 6, 1996.

[187] C. Runciman and D. Wakeling. **Heap profiling of lazy functional programs**. *Journal of Functional Programming*, 3(2) :217–245, 1993.

[188] M. Sage. **FranTk – A declarative GUI language for Haskell**. In *Int'l Conf. on Functional Programming*, Montréal, Québec, Canada, September 2000.

[189] E. Saint-James. **Recursion is more efficient than iteration**. In *ACM Symposium on Lisp and Functionnal Programming*, Austin Texas, USA, 1984.

[190] E. Saint-James. **La programmation applicative (de Lisp à la machine en passant par le λ-calcul)**. Hermès, Paris, 1993.

[191] E. Sandewall. **Programming in an Interactive Environment : the "Lisp" Experience**. *Computing Surveys*, 10(1) :35–71, March 1978.

[192] P. Sansom. **Time Profiling a Lazy Functional Compiler**. In *Functional Programming, Glasgow 1993, Workshop in Computing*, Glasgow, 1994. Springer Verlag.

[193] P. Sansom and S. Peyton Jones. **Profiling Lazy Functional Programs**. In *Functional Programming, Glasgow 1992, Workshop in Computing*, Glasgow, 1993. Springer Verlag.

[194] P. Sansom and S. Peyton Jones. **Time and space profiling for non-strict, higher-order functional languages**. In *22nd ACMSymposium on Principles of Programming Languages*, San Francisco, USA, January 1995.

[195] R.W. Scheifler. **An Analysis of Inline Substitution for a Structured Programming Language**. *CACM*, 20(9) :647–654, September 1977.

[196] R. Schmidt. **Dynamically Extensible Objects in a Class-Based Language**. In *TOOLS USA*, July 1997.

[197] U. Schultz, C. Lawall, J. Consel, and G. Muller. **Towards Automatic Specialization of Java Programs**. In *Proceedings ECOOP'99*, pages 367–390, Lisbon, Portugal, June 1999.

[198] N. Séniak. **Efficient Compilation of Local Functions using C as a Back-end**. Technical report, LIX, École Polytechnique, Palaiseau, France, 1990.

[199] N. Séniak. **Théorie et pratique de Sqil : un langage intermédiaire pour la compilation des langages fonctionnels**. PhD thesis, Université Pierre et Marie Curie (Paris VI), November 1991.

[200] M. Serrano. **Rgc : un générateur d'analyseurs lexicaux efficaces en Scheme**. In C. Queinnec, editor, *Avancées applicatives, Actes des journées JFLA*, Bigre 76–77, pages 235–252, February 1992.

[201] M. Serrano. **De l'utilisation des analyses de flot de contrôle dans la compilation des langages fonctionnels**. In Pierre Lescanne, editor, *Actes des journées du GDR de Programmation*, September 1993.

[202] M. Serrano. **Bigloo user's manual**. RT 0169, INRIA-Rocquencourt, France, December 1994.

[203] M. Serrano. **Using Higher Order Control Flow Analysis When Compiling Functional Languages**. In M. Hermenegildo and J. Penjam, editors, *6th International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes on Computer Science, pages 447–449, Madrid, Spain, September 1994.

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

[205] M. Serrano. **A Fresh Look to Inlining Decision**. In *4th International Computer Symposium,* (Invited paper), Mexico city, Mexico, November 1995.

[206] M. Serrano. **Control Flow Analysis : a Functional Languages Compilation Paradigm**. In *10th Symposium on Applied Computing*, pages 118–122, Nashville, Tennessee, USA, February 1995.

[207] M. Serrano. **Inline expansion : *when* and *how* ?** In *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, pages 143–147, Southampton, UK, September 1997.

[208] M. Serrano. **Wide classes**. In *Proceedings ECOOP'99*, pages 391–415, Lisbon, Portugal, June 1999.

[209] M. Serrano. **Bee : an Integrated Development Environment for the Scheme Programming Language**. *Journal of Functional Programming*, 10(2) :1–43, May 2000.

[210] M. Serrano and H-J. Boehm. **Understanding Memory Allocation of Scheme Programs**. In *Int'l Conf. on Functional Programming*, Montréal, Québec, Canada, September 2000.

[211] M. Serrano and M. Feeley. **Sorage Use Analysis and its Applications**. In *1fst Int'l Conf. on Functional Programming*, pages 50–61, Philadelphia, Penn, USA, May 1996.

[212] M. Serrano and P. Weis. **$1 + 1 = 1$: an optimizing Caml compiler**. In ACM *Sigplan Workshop on ML and its Applications*, pages 101–111, Orlando (Florida, USA), June 1994. Acm Sigplan, INRIA RR 2265.

[213] M. Serrano and P. Weis. **Bigloo : a portable and optimizing compiler for strict**

functional languages. In *2nd Static Analysis Symposium*, Lecture Notes on Computer Science, pages 366–381, Glasgow, Scotland, September 1995.

[214] A. Shalit. **The Dylan Reference Manual : The Definitive Guide to the New Object-Oriented Dynamic Language**. Addison-Wesley, 1996.

[215] Z. Shao. **Flexible Representation Analysis**. In *Proceedings of the SIGPLAN '97 Int'l Conf. on Functional Programming*, June 1997.

[216] Z. Shao and A. Appel. **A Type-Based Compiler for Standard ML**. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[217] Z. Shao, C. League, and Monnier. S. **Implementing Typed Intermediate Languages**. In *Int'l Conf. on Functional Programming*, pages 313–323, June 1998.

[218] O. Shivers. **Control Flow Analysis in Scheme**. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[219] O. Shivers. **Control-Flow Analysis of Higher-Order Languages or Taming Lambda**. CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[220] O. Shivers. **Atomic Heap Transactions and Fine-Grain Interrupts**. In *Int'l Conf. on Functional Programming*, Paris, France, September 1999.

[221] B. Shriver and B. Smith. **The Anatomy of a High-Performance Microprocessor – A System Perspective**. Ieee Computer Society Press, 1998.

[222] M. Slater. **Microprocessor Report**. Micro-Design Resources, Sunnyvale, CA 94086-6245. (M. Slater is Founder and executive editor).

[223] J. Smart. **wxWindows toolkit Reference Manual**. available at `http ://web-.ukonline.co.uk/julian.smart/wxwin`, 1992.

[224] A. Srivastava and A. Eustace. **ATOM : A system for building customized program analysis tools**. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[225] R. Stallman. **Using and Porting GNU CC**. for version 2.7.2 ISBN 1-882114-66-3, Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, November 1995.

[226] R.M. Stallman. **GDB Manual**. Second edition, Free Software Foundation, Inc., February 1988.

[227] G.L. Steele. **Rabbit : a Compiler for Scheme**. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[228] G.L. Steele and G. Sussman. **Scheme : an interpreter for extended lambda calculus**. MIT AI Memo 349, Massachusetts Institute of Technology, Cambridge, Mass., December 1975.

[229] P.A. Steenkiste and J. Hennessy. **Tags and Type Checking in LISP : Hardware and Software Approaches**. In *Architectural support for programming languages and operating systems*, pages 50–59, Palo Alto. CA US, 1987.

[230] D. Stefanovic, K. McKinley, and E. Moss. **Age-Based Garbage Collection**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 370–381, November 1999.

[231] J. Stichnoth, G-Y. Lueh, and M. Cierniak. **Support for Garbage Collection at Every Instruction in a Java Compiler**. In *Conference on Programming Language Design and Implementation*, pages 118–127, Atlanta, Georgia, USA, 1999.

[232] B. Stroustrup. **The C++ Programming Language**. Addison-Wesley, 2nd edition, 1991.

[233] B. Stroustrup. **The Design and Evolution of C++**. Addison-Wesley, May 1994.

[234] Sun Microsystems. **Java Language Specification**, 1995.

[235] Sun Microsystems, Inc., Mountain View, California. **The SPARC Architecture Manual**, August 1987.

[236] M. Superina. **Buildoo**. In *Actes des journées JFLA*, January 2000.

[237] Symbolics Inc. **Symbolics Software**, 1981.

[238] A. Taivalsaari. **Object-oriented programming with modes**. *Journal of Object-oriented programming*, pages 25–32, June 1993.

[239] D. Tarditi, A. Acharya, and P. Lee. **No assembly required : Compiling Standard ML to C**. *ACM Letters on Programming Languages and Systems*, 2(1) :161–177, 1992.

[240] D. Tarditi, G. Morrisset, P. Cheng, C. Stone, R. Harper, and P. Lee. **TIL : a type-directed optimizing compiler for ML**. In *Conference on Programming Language Design and Implementation*, pages 181–192, June 1996.

[241] W. Teitelman. **InterLISP Reference Manual**. Technical report, Xerox Palo Alto Research Center, California, USA, 1974.

[242] M. Tofte and J-P. Talpin. **Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions**. In *21st* ACM SIGPLAN-SIGACT *Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, USA, January 1994.

[243] D. Turner. **SASL Language Manual**. Technical report, St Andrews University, 1976.

[244] D. Turner. **Miranda : a non-strict functional language with polymorphic types**. In Springer Verlag, editor, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes on Computer Science, pages 1–16, 1985.

[245] D. Ungar. **The Design and Evaluation of a High Performance Smalltalk System**. MIT Press, Cambridge, Mass., USA, 1986.

[246] J. Vitek and Horspool. **Taming message passing : Efficient method look-up for dynamically-typed languages**. In *European Conference on Object-Oriented Programming*, 1994.

[247] J. Vitek, N. Horspool, and A. Krall. **Efficient Type Inclusion Tests**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1997.

[248] J. Vitek, M. Serrano, and D. Thanos. **Mobile Object Systems : Towards the Programmable Internet**, volume 1222 of *Lecture Notes on Computer Science*, chapter **Security and Communication in Mobile Object Systems**, pages 177–200. Springer Verlag, J. Vitek and C. Tschudin editors, Heidelberg, April 1997.

[249] T. Vullings, D. Tuijnman, and V. Schulte. **Lightweight GUIs for Function Programming**. In *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, 1995.

[250] P. Wadler. **Why No One Uses Functional Languages**. SIGPLAN *Notices*, 33(8) :23–27, August 1998.

[251] D. Wall. **Register Windows vs. Register Allocation**. Technical Report 87/5, Dec Wrl, December 1987.

[252] L. Wall, T. Christiansen, and R. Schwartz. **Programming Perl**. A Nutshell Handbook. O'Reilly, 1fst edition, 1991.

[253] K. Walrath and M. Campione. **The JFC Swing Tutorial : A Guide to Constructing GUIS**. Addison-Wesley, July 1999.

[254] S. Weeks. **MLton User's Guide**, July 1999.

[255] P. Weis and *al.* **The CAML Reference manual**. Technical Report 121, INRIA-Rocquencourt, 1991.

[256] P. Weis and X. Leroy. **Le langage CAML**. InterEditions, Paris, 1993.

[257] J. Whaley and M. Rinard. **Compositional Pointer and Escape Analysis for Java Programs**. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, November 1999.

[258] P. Wilson. **Uniprocessor garbage collection techniques**. *ACM Computing Surveys*, 0(0), September 1997.

[259] N. Wirth. **Programming in Modula-2**. In *Texts and Monographs in Computer Science*. Springer Verlag, 1993.

[260] A. Wright, S. Jagannathan, C. Ungureanu, and A. Hertzman. **Compiling Java to a Typed Lambda-Calculus : A Preliminary Report**. In *In Proceedings of the 2nd Int'l Workshop on Types in Compilation*, March 1998.

[261] A. Zeller and D Lütkhaus. **DDD – A Free Graphical Front-End for UNIX Debuggers**. Technical report, Abteilung Softwaretechnologie, Technische Universität Braunschweig, August 1995.

[262] B. Zorn. **Comparing Mark-and-sweep and Stop-and-copy Garbage Collection**. In *Conference on Lisp and Functional Programming*, pages 87–97, 1990.

[263] B. Zorn. **The Measured Cost of Conservative Garbage Collection**. *Software — Practice and Experience*, 23(7) :733–756, July 1993.

[264] B. Zorn and P. Hilfinger. **A Memory Allocation Profiler for C and Lisp Programs**. In *Usenix conference*, pages 223–237, 1998.