Traduction et optimisation globale dans les langages de classes

Olivier Zendra

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAE + M

Traduction et optimisation globale dans les langages de classes

THÈSE

présentée et soutenue publiquement le 30 Octobre 2000

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Olivier Zendra

Composition du jury

Président :	Michel Cosnard
Rapporteurs :	Karel Driesen Jean-Marc Jézéquel Amedeo Napoli
Examinateurs :	Dominique Colnet Claude Godart

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

On dit toujours, assez justement, qu'une thèse est un travail personnel, un effort solitaire. Ceci est tout à fait vrai, mais néanmoins réducteur. Une thèse, si elle n'est pas vraiment un travail d'équipe, est aussi la résultante d'interactions avec de nombreuses personnes, et n'aboutit que grâce à mille et un petits détails souvent trop vite oubliés. Je souhaite ici remercier les personnes qui ont contribué, à des titres très divers, à ma thèse.

Dire que M. Michel Cosnard, Professeur à l'Ecole Normale Supérieure de Lyon, est quelqu'un dont les responsabilités lui prennent beaucoup de temps serait un euphémisme. Il a pourtant accepté de m'écouter et a su croire en moi à un moment particulièrement difficile vers le début de cette thèse, qui sans lui n'aurait jamais pu avoir lieu. Je tiens à lui exprimer ici ma plus sincère et durable gratitude. Je le remercie également vivement d'avoir sans hésitation répondu favorablement à ma proposition de faire partie du jury chargé d'évaluer mon travail de thèse et de m'avoir fait l'honneur de le présider.

M. Karel Driesen, Professeur à McGill University, m'a fait l'honneur de s'intéresser à mes travaux et d'accepter ensuite d'être rapporteur de cette thèse. Il n'a pas hésité à venir spécialement de Montréal pour la soutenance, à une période pourtant très chargée pour lui. Ses remarques, avant la rédaction de ce mémoire aussi bien que pendant, ont été très précieuses et continueront à influer sur mes travaux de recherche futurs. De tout cela, ainsi que d'avoir accepté de relire ce mémoire en français, je lui suis extrêmement reconnaissant.

M. Jean-Marc Jézéquel, Professeur à l'Université de Rennes 1, est l'un des utilisateurs les plus assidus de SmallEiffel, qu'il a été parmi les premiers à utiliser, dans le cadre de ses propres travaux de recherche. A ce titre, ses commentaires et suggestions ont été des plus judicieux et ont grandement contribué à faire passer SmallEiffel du stade de prototype à celui de produit utilisé dans le monde entier. Pour cela, pour sa patience avec les imperfections des premières versions de SmallEiffel et pour avoir accepté d'être rapporteur de mon travail de thèse, je lui adresse mes plus vifs et sincères remerciements.

M. Amedeo Napoli, Chargé de Recherche au Loria, m'a fait l'honneur d'accepter d'être rapporteur de ce travail, bien qu'il soit quelqu'un de déjà extrêmement occupé. Je suis certain que sa grande expertise — tout particulièrement dans le domaine des langages à objets — et la qualité de ses remarques m'auront permis d'améliorer sensiblement ce manuscrit. Pour tout cela, je lui adresse mes plus vifs remerciements.

M. Dominique Colnet, Maître de Conférences à l'Université Henri Poincaré, aura été pour moi plus qu'un directeur de thèse. Son optimisme permanent, sa confiance et ses conseils pertinents ont été une aide très précieuse durant ce travail de thèse et m'ont permis d'avancer dans les inévitables moments de doute. Son esprit critique et son recul ont grandement contribué à nos nombreuses discussions et à la qualité de ma thèse. Ma passion pour la recherche et le travail bien fait lui doivent beaucoup. Je tiens à l'en remercier tout spécialement et chaleureusement.

M. Claude Godart, Professeur à l'Université Henri Poincaré, ESSTIN, a eu la gentillesse de m'accueillir au sein de l'équipe ECOO du Loria. Il a toujours fait preuve d'une grande disponibilité pour répondre à mes questions aussi nombreuses que variées. Son expérience et ses conseils particulièrement avisés m'ont ainsi été très profitables et ont contribué au déroulement harmonieux de ma thèse. Je lui en suis extrêmement reconnaissant.

Les membres du projet ECOO du Loria méritent une mention particulière pour avoir été non seulement mes collègues mais aussi mes amis durant cette thèse. Leur chaleur et leur bonne humeur auront fortement contribué à la bonne ambiance dans laquelle s'est déroulée cette thèse.

Je remercie très chaleureusement mes amis Hala Skaf-Molli et Pascal Molli, dont les compétences et la grande curiosité intellectuelle ont permis de nombreuses discussions et débats passionnants. Ils ont activement contribué à me faire prendre un peu de ce recul qui manque tant lorsqu'on est plongé dans sa thèse.

Toute ma reconnaissance à Gérôme Canals et François Charoy pour leur jovialité et leurs nombreux conseils pertinents. Gérôme m'a également fourni une somme d'informations importante à propos de mon séjour post-doctoral, facilitant ainsi sensiblement ma fin de thèse.

Merci à Philippe Coucaud qui a travaillé avec moi sur les problèmes de ramasse-miettes. J'ai pu apprécier sa motivation et son efficacité au cours de nombreuses discussions et de longues heures de développement.

Mes remerciements à Daniela Grigori, avec qui j'ai longtemps partagé un bureau, dont la timidité et la gentillesse n'ont d'égal que le calme, que j'ai fort apprécié en période de rédaction de thèse.

Un grand merci aussi aux personnes dont le travail de fourmi a permis à cette thèse de se dérouler dans de bonnes conditions matérielles: les services généraux et administratifs, le centre de documentation et les moyens informatiques du Loria.

Alain Filbois, ingénieur système qui partageait avec moi la particularité d'être au laboratoire a des heures indues, mérite des remerciements très chaleureux. Il a toujours accepté de se pencher rapidement et avec compétence sur les divers problèmes que je pouvais rencontrer en utilisant les systèmes informatiques du laboratoire à ces heures. Le garde-manger et la parapharmacie de son bureau m'auront également rendu service plus d'une fois.

Je tiens aussi à exprimer ma gratitude envers les utilisateurs de SmallEiffel, qu'ils soient étudiants, individus, chercheurs ou industriels. Leur constant support, leurs suggestions et leurs commentaires ont sans cesse permis d'améliorer et de déverminer SmallEiffel. Les membres du projet PAMPA de l'IRISA méritent à ce propos une mention particulière, notamment Alain Le Guennec, grand explorateur des tréfonds de SmallEiffel.

Merci aux membres du département informatique de l'Université Henri Poincaré et de l'ESIAL, notamment Brigitte Jaray, Jocelyne Rouyer et Jacques Guyard, pour leur amical soutien dans mon rôle d'enseignant.

Je dois beaucoup, pour ma formation, aux enseignants que j'ai eu, depuis maintenant pas mal d'années. Je suis tout particulièrement reconnaissant aux enseignants qui m'ont fait découvrir les langages à objets ou la compilation, notamment Martine Camonin-Gautier, Karol Proch et Pierre Bouchet. Ils ont fortement contribué à me mettre l'eau à la bouche et à me donner une soif de connaissances qui m'a été bien utile dans le cadre de ma thèse.

Je dois également beaucoup à mes amis Alain "D'Artagnan", Arnaud "Nono", Benoît, Dominique, Emmanuelle, Jacques "Jack", Jean-Charles "Lamimi", Jean-Marc "Gars Humbert", Marjorie, Thierry "Garti" et Vincent, qui, sans toujours en être conscients, m'ont soutenu durant ma thèse.

Un très grand merci à mes parents, qui m'ont soutenu pendant ces nombreuses années et sans qui rien de tout cela n'aurait pu avoir lieu. Leur confiance en moi aura été sans faille.

Ma plus tendre reconnaissance à Anne-Céline. Elle aura su non seulement me supporter mais aussi me soutenir pour ma thèse, tout particulièrement pendant la délicate période de rédaction de ce mémoire et de préparation de la soutenance. Sa gentillesse et son altruisme m'ont sans cesse émerveillé et redonné le moral pendant les inévitables moments difficiles. Sa contribution, bien que très discrète vue du laboratoire, aura été déterminante.

Enfin, merci à toutes les autres personnes qui, à quelque titre et quelque moment que ce soit, ont pu m'aider dans le cadre de mon travail de thèse. J'ai essayé de les citer toutes, mais il est possible que je n'y sois pas arrivé. Que celles que j'ai oubliées ne m'en tiennent pas rigueur.

A mon père.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Avant-propos et historique

Ce mémoire présente un travail de thèse débuté fin 1997 mais dont les motivations sous-jacentes remontent à bien plus loin.

En effet, j'ai commencé à me passionner pour les problèmes liés à la compilation dès 1993-1994, alors qu'étudiant je les découvrais à travers divers projets d'initiation que j'avais choisis. C'est vers septembre 1994, au début de ma maîtrise, que j'ai commencé à travailler, sur mon temps libre, avec Dominique Colnet sur ce qui n'était alors que l'embryon du projet SmallEiffel.

Il s'agissait pour nous d'une part de nous intéresser aux problèmes de la compilation et d'autre part de fournir un compilateur pour le langage Eiffel qui soit gratuit, rapide, de grande qualité et dont le code source soit librement accessible [1], critères auxquels ne répondaient pas les compilateurs Eiffel disponibles à l'époque. Afin de nous focaliser sur les aspects de plus haut niveau, nous avons décidé que ce compilateur produirait du code C ANSI, que nous considérons comme un excellent macro-assembleur portable. Après environ un an d'efforts à temps partiel, nous avons eu la joie de célébrer notre premier *bootstrap* en juillet 1995. Durant près d'un an et demi, de fin 1995 à début 1997, j'ai continué à travailler sur ce projet, toujours pendant mon temps libre, étant occupé et éloigné par mon service national en tant que coopérant scientifique. Les toutes premières versions — encore fort incomplètes — diffusées auprès du public l'ont été à partir de février 1996.

Ce n'est qu'après mon retour du service national et le début de ma thèse en 1997 que j'ai pu me consacrer à plein temps à ce projet, sous la direction de Dominique Colnet. Offrant des performances considérablement meilleures que celles des compilateurs Eiffel commerciaux existants, notre compilateur a rapidement eu une diffusion importante. En juillet 1997, nous avons diffusé la première version offrant la possibilité de générer au choix du code C ANSI ou du *bytecode* Java. En 1998, SmallEiffel s'est vu reconnaître le titre de compilateur Eiffel officiel GNU par la *Free Software Foundation* [2] et est donc devenu *SmallEiffel, The GNU Eiffel Compiler*. Enfin, en avril 1998, SmallEiffel est devenu capable de produire automatiquement un ramasse-miettes adapté à l'application compilée.

Nos améliorations successives ont ainsi transformé notre prototype initial en un véritable *produit*, utilisé non seulement en recherche mais aussi dans l'industrie, et cela dans le monde entier. L'effort consacré à ce projet a bien entendu été à la mesure de ces résultats, le projet SmallEiffel représentant en juin 2000 près de 300 classes totalisant 80000 lignes de code, plus 2500 autres classes utilisées pourla validation comprenant 200000 lignes, sans compter de nombreux bancs d'essais plus spécifiques.

Ce mémoire de thèse de doctorat présente la synthèse de mes travaux de recherches dans le cadre de ce projet.

1. SmallEiffel, ses bibliothèques et son code source sont en effet téléchargeables depuis `http://SmallEiffel.loria.fr`
2. `http://www.fsf.org` ou `http://www.gnu.org`

1.2 Problématique et contexte

De nouveaux langages apparaissent régulièrement qui répondent à de nouveaux besoins, permettent aux développeurs de mieux exprimer leurs idées et introduisent de nouveaux concepts — en général de plus haut niveau. Ceci implique fort logiquement la réalisation de nouveaux compilateurs, toujours plus puissants.

De plus, l'accroissement de la complexité des tâches effectuées par les programmes informatiques nécessite des puissances de calcul toujours plus élevées, qui sont fournies en partie par l'augmentation de la puissance du matériel (microprocesseurs) et en partie par les progrès réalisés par les logiciels, dans le domaine de la compilation et plus précisément de l'optimisation des programmes.

C'est à ce niveau, *l'optimisation des programmes*, que se situe la problématique générale de cette thèse, avec pour objectif de répondre, — au moins partiellement, car il semble qu'une réponse définitive soit utopique — à cette éternelle question: *Comment améliorer les performances des programmes compilés, c'est à dire comment faire en sorte que les programmes soient plus rapides et moins consommateurs de ressources?*

Une question à la fois aussi simple et aussi vaste peut être abordée de façons très différentes et avoir des réponses multiples, dépendant du domaine traité.

Les recherches menées durant cette thèse se sont donc focalisées sur *la compilation et l'optimisation pour les langages de classes*. Ces derniers sont un sous-ensemble des *langages à objets* [Masini *et al.*, 1989] et comprennent entre autres Smalltalk [Goldberg et Robson, 1983], Objective C [Cox, 1983], Object Pascal [Schmucker, 1986], C++ [Stroustrup, 1986], Eiffel [Meyer, 1992] et Java [Gosling *et al.*, 1996]. Comme indiqué dans l'avant-propos, nous avons plus spécifiquement effectué nos recherches sur et avec le langage Eiffel, langage de classes particulièrement élégant, facile d'utilisation et puissant, offrant des constructions telles qu'héritage multiple, types paramétriques avec généricité contrainte ou non contrainte, assertions pour le support de la conception par contrat [Meyer, 1988], etc. Dans le cadre de ce travail de thèse, nous avons également eu pour objectif de proposer des optimisations de haut niveau et qui soient *portables*.

La première partie de notre travail de thèse, du point de vue chronologique comme du point de vue de l'importance des efforts fournis et des contributions apportées, concerne donc la conception et l'implantation d'un système de compilation efficace, c'est à dire capable de compiler très rapidement les programmes tout en générant un code spécialisé très fortement optimisé. Nous nous sommes, dans cette première partie, focalisés sur la compilation et la prise en compte des aspects *spécifiques aux langages de classes*, en mettant tout particulièrement l'accent sur *l'optimisation de la liaison dynamique*. En effet, cette dernière est un aspect des langages de classes qui est fondamental non seulement du point de vue conceptuel mais aussi en ce qui concerne les performances. Afin de *spécialiser* autant que possible l'ensemble du programme compilé, nous avons adopté une *approche globale* au système. Nous nous appuyons ainsi sur des algorithmes d'analyse globale, de prédiction de type et d'expansion en ligne (*inlining*) puissants. Nous proposons une nouvelle méthode d'implantation de la liaison dynamique, plus performante que les techniques actuelles, qui constitue une contribution majeure de cette thèse. Ces algorithmes permettent d'effectuer de façon automatique des optimisations ciblées sur les points importants pour les langages de classes.

Un second axe pour nos recherche a été le problème de la gestion mémoire automatique des applications compilées. Nous avons choisi de confier la génération et l'optimisation du ramasse-miettes au compilateur, afin d'avoir un processus aussi automatisé que possible, facilitant ainsi le travail du développeur d'applications. Nous nous sommes donc penchés sur la conception et l'intégration au processus de compilation d'un système de *génération* de ramasse-miettes *automatiquement adaptés* aux applications compilées. Nous proposons un système qui optimise les ramasse-miettes en étendant et généralisant au problème de la gestion automatique de la mémoire les algorithmes et techniques de nos premiers travaux, notamment l'analyse globale, la prédiction de type et la duplication-spécialisation de code. Nous montrons que ce système produit des ramasse-miettes optimisés ayant de très bonnes performances, du niveau des meilleurs ramasse-miettes actuels.

Enfin, nous avons exploré une troisième direction de recherche, complémentaire aux précédentes, ayant bien entendu toujours pour but d'optimiser les programmes. Il s'agit de l'étude de *l'aliasing* dans un langage de classes et de son impact sur le processus de compilation. L'aliasing est en effet souvent considéré comme néfaste et devant être évité. Il peut cependant présenter un certain nombre d'avantages,

notamment en termes de confort de développement et de performances à l'exécution, qui font qu'il peut également être très précieux. Nos recherches ont eu pour but de tenter de caractériser et de mesurer l'impact de l'aliasing, ce qui est assez rarement fait, ainsi que de tirer parti des avantages de l'aliasing tout en maîtrisant ses inconvénients. Nos recherches dans ce domaine ont été moins profondes que celles mentionnées précédemment, mais elles présentent des résultats initiaux très intéressants et prometteurs.

1.3 Organisation du document

L'organisation de ce document reflète les directions prises jusqu'à présent par nos recherches sur la compilation et l'optimisation de programmes écrits dans un langage de classes.

Le chapitre 2, page 5, porte sur la première partie de notre travail de thèse, à savoir une compilation rapide et générant un code très optimisé, notamment par compilation et optimisation de la liaison dynamique dans un contexte d'analyse statique globale. Après avoir rappelé certains concepts de base et passé en revue les techniques connues d'implantation de la liaison dynamique, ce chapitre détaille notre propre technique et montre son efficacité dans le cadre du compilateur SmallEiffel.

Le chapitre 3, page 53, présente une extension importante des recherches présentées dans le premier chapitre. Il rappelle tout d'abord les algorithmes classiques de gestion automatique de la mémoire, puis détaille notre système de génération automatique d'un ramasse-miette spécialisé pour l'application compilée. Il présente des résultats permettant de valider expérimentalement notre approche.

Le chapitre 4, page 89, porte sur nos travaux sur l'aliasing dans un langage de classes et son impact sur le processus de compilation. Après avoir brièvement présenté l'aliasing, ce chapitre décrit notre méthode pour utiliser l'aliasing de façon disciplinée et expose nos résultats, qui montrent l'efficacité de notre méthode et l'impact positif de l'aliasing.

Enfin, le chapitre 5, page 101, fait le bilan de nos travaux et présente les perspectives de recherches à l'issue de cette thèse.

Chapitre 2

Compilation de la liaison dynamique dans un contexte d'analyse statique globale

2.1 Introduction

La programmation à objets est devenue une tendance majeure en informatique. En effet, l'usage intensif de l'héritage et de la liaison dynamique tend à rendre le code plus extensible et réutilisable. Cependant, certaines préoccupations demeurent en ce qui concerne les performances des programmes à objets, notamment à cause de la liaison dynamique.

Afin d'atteindre des performances similaires à celles des langages traditionnels comme C, les systèmes à objets doivent implanter cette liaison dynamique efficacement. Les travaux récents sur la prédiction de type représentent un premier pas dans cette direction en permettant le remplacement de nombreux appels polymorphiques par des appels monomorphiques directs. La méthode utilisée pour implanter les appels polymorphiques restants a également un impact important en termes de vitesse d'exécution.

Bien évidemment, les techniques d'optimisation des programmes à objets doivent non seulement produire des exécutables rapides, mais doivent également être elles-mêmes suffisamment peu coûteuses pour ne pas se voir cantonnées à la dernière compilation des applications juste avant livraison. Des compilateurs rapides sont en effet indispensables pour permettre de développer avec des langages à objets de façon incrémentale.

Dans ce chapitre, nous présentons les résultats obtenus dans ces domaines durant nos recherches [Collin *et al.*, 1996; Collin *et al.*, 1997; Zendra *et al.*, 1997; Colnet et Zendra, 2000] dans le cadre du projet SmallEiffel. Ce projet, appliqué et fortement utilisé pour la recherche aussi bien qu'hors de la recherche, nous a amenés à la conclusion quelque peu surprenante que la compilation séparée n'est pas indispensable pour faire du développement incrémental. Nous affirmons en effet que des compilateurs intégrant une prédiction de type *globale* au système, combinée avec une implantation efficace de la liaison dynamique, peuvent être suffisamment rapides pour être utilisés dans le cadre d'environnements de développement incrémental.

Dans le système SmallEiffel, nous avons implanté et validé un algorithme de prédiction de type globale permettant un haut degré de spécialisation du code généré. Nous avons également proposé et expérimenté une méthode nouvelle d'implantation de la liaison dynamique, offrant d'excellentes performances. Cette méthode consiste à supprimer totalement toutes les tables de pointeurs de fonctions — *Virtual Function Tables*, ou VFT — ou les structures similaires, en les remplaçant par de simples, mais très efficaces, tests de type. Comme tous les appels indirects sont remplacés par des appels directs, cette implantation de la liaison dynamique permet d'importantes optimisations supplémentaires telles que l'expansion de code en ligne (*inlining*), et ceci même au sein des appels polymorphiques. Ces optimisations, qui portent sur l'implantation de concepts fondamentaux dans les langages de classe, sont très importantes car elles

permettent aux développeurs de se concentrer sur une bonne programmation à objets sans crainte de surcoûts excessifs. Notre méthode, détaillée plus loin dans ce chapitre, n'est pas limitée à Eiffel [Meyer, 1992], mais peut au contraire s'appliquer à tout langage de classes [Masini *et al.*, 1989] sans création ou modification dynamique des classes. Elle peut donc par exemple être appliquée à C++ [Stroustrup, 1986] mais pas à Smalltalk [Goldberg et Robson, 1983] ni Self [Ungar et Smith, 1987].

La suite de ce chapitre est organisée de la façon suivante. Nous rappelons tout d'abord, en section 2.2 les concepts fondamentaux associés à la liaison dynamique. Nous abordons ensuite la méthode de compilation que nous avons développée dans le cadre de cette thèse. La section 2.3 explique comment le système à compiler est exploré par analyse statique globale de façon à rassembler un maximum d'informations à son sujet. Puis la section 2.4 présente globalement notre méthode de génération de code optimisé par duplication-spécialisation. La section 2.5 passe ensuite en revue les techniques classiques d'implantation de la liaison dynamique, après quoi nous détaillons en section 2.6 notre propre technique pour implanter la liaison dynamique de façon particulièrement efficace. Notre validation expérimentale et les résultats obtenus avec cette méthode sont présentés en section 2.7. Enfin, la section 2.8 passe en revue les travaux liés à l'optimisation de la liaison dynamique et la section 2.9 conclut et présente les perspectives.

2.2 Le concept de liaison dynamique

Le concept de *liaison dynamique* (*late binding*), connu aussi sous les expressions *envoi de message* (*message send*), *dynamic dispatch* ou *dynamic binding*, est fondamental pour les langages à objets. Il est indissociablement lié à des notions avec lesquelles il est parfois — et à tort — confondu, qui sont les notions de *type statique*, *type dynamique* et *polymorphisme*.

2.2.1 Type statique et type dynamique

Dans un langage de programmation procédural "classique", comme Pascal [Wirth, 1971] ou C [Kernighan et Ritchie, 1978], une variable, ou de façon plus générale une expression quelconque, a un *type* et un seul, statique, qui ne variera pas au cours de l'exécution. Ce type est connu lors de la compilation, il s'agit de celui qui jouxte le nom de la variable là où elle est déclarée. Ainsi, par exemple, dans les déclarations C suivantes,

```
int mon_entier;
struct client mon_client;
```

`mon_entier` est de type entier (`int`) alors que `mon_client` est une structure de type client (`struct client`). Ceci reste immuable et permet au compilateur de générer le code approprié au type de la variable considérée, ainsi que d'effectuer des vérifications de type pour détecter les erreurs commises par le programmeur[3].

Dans un langage à objets, ce type, celui de la déclaration, qui apparaît dans le code source à côté du nom de la variable, est appelé *type statique*. Dans l'exemple de déclarations de types Eiffel suivant,

```
mon_reel: REAL;
mon_point: POINT;
```

`mon_reel` est de type statique REAL et `mon_point` a pour type statique POINT.

Le type statique dans un langage à objets joue globalement le même rôle que pour un langage classique, en ce sens qu'il est utilisé pour déterminer à la compilation la validité d'une instruction d'affectation (explicite ou par passage de paramètre), en vérifiant la compatibilité du type de la source avec le type de la destination. Il sert également, dans les langages à objets, à vérifier qu'une méthode appelée sur un type existe bel et bien dans ce type.

En revanche, dans les langages à objets, une variable, ou toute expression, possède également un *type dynamique*. Comme son nom l'indique, celui-ci peut changer dynamiquement, lors de l'exécution. Le type dynamique d'une variable à un instant précis de l'exécution est en fait le type de l'objet attaché à cette variable à cet instant.

3. Nous ne considérerons pas ici les *casts* C, qui permettent de forcer le type de la source d'une affectation et qui — bien que parfois nécessaires — sont source de nombreuses erreurs.

2.2.2 Le polymorphisme

Le fait qu'une variable puisse avoir différents types dynamiques au cours de l'exécution est appelé *polymorphisme* (du grec "poly" plusieurs et "morphê", forme).

En deux mots, et de manière un peu abstraite, ceci permet d'accéder aux primitives d'un (objet d'un) certain type de la même façon, syntaxiquement parlant, qu'à celles d'un (objet d'un) de ses sous-types. Ceci permet de substituer à un type l'un de ses sous-types, sans que ce changement n'apparaisse au niveau des clients.

Pour donner une explication plus concrète, considérons les classes POINT, POINT_CARTESIEN et POINT_POLAIRE suivantes[4], écrites en Eiffel:

```
deferred class POINT

feature
    x: REAL is deferred end
    y: REAL is deferred end
    alpha: REAL is deferred end
    d: REAL is deferred end

    is_origin: BOOLEAN is
       do
          Result := (x = 0.0) and (y = 0.0)
       end

invariant
    x_ok:        x = d * alpha.cos;
    y_ok:        y = d * alpha.sin;
    d_ok:        d = (x^2 + y^2).sqrt;
    origin_zero: is_origin implies (x = 0.0 and y = 0.0 and alpha = 0.0 and d = 0.0)
end; -- class POINT


class POINT_CARTESIEN
inherit POINT
creation
    make

feature
    make (abscisse, ordonnee: REAL) is
       do
          x := abscisse;
          y := ordonne
       end

    x: REAL
    y: REAL

    alpha: REAL is
       do
          if is_origin then Result = 0.0 else Result := (y/x).atan.to_real
       end

    d: REAL is
       do
          if is_origin then Result = 0.0 else Result := (x^2 + y^2).sqrt
```

4. Pour simplifier, nous ne tenons pas compte dans cet exemple des problèmes de représentation et d'erreurs d'arrondi des nombres en virgule flottante.

```
       end
end -- class POINT_CARTESIEN


class POINT_POLAIRE

inherit POINT
creation make

feature
    make (angle, distance: REAL) is
       require
          positive_distance: distance >= 0.0
       do
          alpha := angle;
          d := distance
       end

    x: REAL is
       do
          Result := d * alpha.cos
       end

    y: REAL is
       do
          Result := d * alpha.sin
       end

   alpha: REAL
   d: REAL

end; -- class POINT_POLAIRE
```

Supposons que nous ayons une routine cliente comme suit:

```
   do_something (p:POINT) is
      require
         p_exists: p /= Void
      do
         if p.is_origin then
            -- faire quelque chose
         else
            -- faire autre chose
         end
      end
```

Remarquons brièvement que ce code illustre l'utilisation des *assertions* en Eiffel [Meyer, 1992]. Les assertions sont des propriétés — des expressions booléennes — qui doivent être vérifiées lors de l'exécution des programmes[5]. La classe POINT comporte quatre *invariants* (mot-clé `invariant`), qui sont des assertions devant être vérifiés en permanence sur les instances du type POINT, sauf bien sûr durant la phase de création de l'objet. Ces invariants sont aussi valables pour les instances des sous-types POINT_POLAIRE et POINT_CARTESIEN. La routine `do_something` contient également une *précondition* (mot-clé `require`), assertion devant être vérifiée au début l'appel de cette routine.

Examinons maintenant l'exemple proprement dit. La routine `do_something` prend un POINT en paramètre et lui applique la fonction `is_origin`. Ici, le type statique de `p` est POINT, mais son type dynamique peut être aussi bien POINT_CARTESIEN que POINT_POLAIRE, les deux descendants concrets de POINT. Ainsi, `p` est une *variable polymorphe*. Dans notre exemple, comme l'indique le mot-clé Eiffel `deferred`,

5. Cette vérification est gérée par le compilateur. Les assertions peuvent être activées ou désactivées sélectivement par le développeur Eiffel à l'aide d'options de compilation et/ou d'exécution.

la classe POINT est une *classe abstraite*, c'est à dire contenant des primitives abstraites (ou *pure virtual*, selon la terminologie C++), et par conséquent ne peut pas être instanciée. Il ne sera donc possible d'avoir que des objets de type POINT_CARTESIEN ou POINT_POLAIRE attachés à la variable polymorphe `p`, ce qui revient à dire que le type dynamique de `p` ne peut être que POINT_CARTESIEN ou POINT_POLAIRE.

Comme la routine `is_origin` existe au niveau de POINT, elle existe aussi dans ses descendants. Il n'est donc pas nécessaire, dans le code de `do_something`, de se soucier du type dynamique de `p`, car on peut substituer à POINT n'importe lequel de ses descendants et la routine `is_origin` adéquate sera appelée. `p.is_origin` est donc un appel statiquement valide.

Ce processus de sélection, lors de l'exécution, de la primitive adéquate en fonction du type dynamique du receveur, s'appelle justement la *liaison dynamique*.

2.2.3 La liaison dynamique

Dans l'exemple ci dessus, les trois types de points partagent tous la même routine `is_origin` (celle définie dans POINT). Il n'est donc pas nécessaire de faire appel à la liaison dynamique: l'appel à la bonne routine peut être codé correctement lors de la compilation. Supposons en revanche que l'on ait le code client suivant:

```
do_something (p:POINT) is
   require
      p_exists: p /= Void
   do
      if p.d < 1.0 then
         -- faire quelque chose
      else
         -- faire autre chose
      end
   end
```

L'appel à `p.d` est statiquement valide puisque POINT, qui est le type statique de `p`, comprend bien une routine `d`. Cependant, celle-ci est codée différemment selon que l'on a affaire à un POINT_CARTESIEN ou à un POINT_POLAIRE. Il est donc nécessaire à l'exécution "d'aiguiller" l'appel `p.d` sur la routine correspondant au type dynamique de `p`. Si `p` est un POINT_POLAIRE, il s'agira d'un simple accès à un champ de l'objet `p`; si `p` est un POINT_CARTESIEN, il s'agira d'un appel de fonction. On dit donc de l'appel de primitive `p.d`, comme de tout appel sujet à liaison dynamique, qu'il s'agit d'un *appel polymorphique*. Dans le cas d'un appel polymorphique où le nombre de types possibles est important (à partir de 17, selon [Aigner et Hölzle, 1996]), on dira qu'il s'agit d'un *appel mégamorphique*. A contrario, un appel de primitive où un seul type dynamique est possible pour le receveur est appelé *appel monomorphique*.

Du point de vue de l'implantation, il est nécessaire afin de pouvoir effectuer l'opération de liaison dynamique de disposer à tout moment du type des objets. Ceux-ci se voient donc attacher à leur création une information supplémentaire qui est leur identifiant de type. Cette information consiste le plus souvent en un pointeur vers la description du type, mais peut prendre d'autres formes, par exemple un entier codant le numéro du type.

En revanche, dans un langage classique, non objet, aucune trace explicite de ce type n'est nécessaire dans la représentation des données à l'exécution et l'information correspondante est donc omise. Ainsi, le code généré l'a été en connaissant le type des données qu'il manipule, mais rien ne permet plus, au simple vu des données, de déterminer à quel type elles correspondent.

On voit donc que les langages à objets offrent une bien plus grande flexibilité, une meilleure facilité d'utilisation et un pouvoir expressif très supérieur aux langages classiques, au prix d'une légère baisse des performances (qui, rappelons-le, peut être gommée par de bons compilateurs). Il s'agit là d'une tendance générale en informatique, puisqu'on évolue sans cesse de langages de bas niveau, proche de matériel, vers des langages de plus haut niveau d'abstraction, un peu plus gourmands en ressources machines mais permettant de faire des économies importantes en temps de développement.

2.3 Analyse statique et prédiction de type

Le but global de la prédiction de type appliquée aux langages à objets est l'extraction d'informations, notamment pour la vérification de type et l'optimisation des applications. Notons que nous avons choisi dans le cadre de ce mémoire d'utiliser l'expression *prédiction de type dynamique* — ou, plus brièvement, *prédiction de type* — pour caractériser nos travaux et non pas *inférence de type*. En effet, cette dernière nous semble plus appropriée aux systèmes comme CAML [Weis *et al.*, 1989], par exemple, où aucun type n'est explicité et où tous les types doivent donc être déduits, inférés. Dans notre travail, en revanche, il s'agit de compléter et d'affiner les informations de types déjà disponibles en les propageant à l'ensemble du système, de façon à pouvoir prédire de façon plus précise quel sera lors de l'exécution le type des données à un certain point du programme.

La thèse de doctorat de Ole Agesen [Agesen, 1996] contient une revue complète des systèmes d'inférence de type. Les systèmes passés en revue vont des plus théoriques [Vitek *et al.*, 1992] à des systèmes utilisés couramment par une vaste communauté [Milner, 1978], en passant pas des systèmes partiellement implantés [Suzuki, 1981; Suzuki et Terada, 1984] et d'autres implantés pour de "petits" langages [Graver et Johnson, 1990; Palsberg et Schwartzbach, 1991; Palsberg et Schwartzbach, 1992].

Alors que beaucoup de travaux de recherche ont porté sur l'inférence de type [Cartwright et Fagan, 1991; Agesen *et al.*, 1993; Plevyak et Chien, 1994; Agesen et Hölzle, 1995; Dean *et al.*, 1995; Eifrig *et al.*, 1995], avec d'intéressants résultats, les compilateurs couramment utilisés en production ne semblent guère s'appuyer sur les avantages de puissants algorithmes de prédiction de type. En effet, les implantations de tels algorithmes ne sont parfois pas assez rapides pour être utilisées dans le cadre du développement incrémental. De plus, elles requièrent souvent la connaissance de l'ensemble du système, ce qui rend la compilation séparée problématique. Nous avons néanmoins choisi d'utiliser la prédiction de type dans nos travaux afin de rassembler des informations nous permettant d'optimiser tant le processus de compilation que le code généré. Notre algorithme d'analyse et de prédiction de type, que nous allons présenter dans les deux sections suivantes, peut être considéré comme une variation de l'algorithme RTA (*Rapid Type Analysis*) de Bacon et Sweeney [Bacon et Sweeney, 1996].

2.3.1 Analyse globale au système

Le but que nous avons assigné à notre analyse statique est bien entendu de rassembler des informations sur le système compilé. Nous avons cependant poussé ce choix plus loin, en décidant de rassembler le *maximum* d'informations possible sur l'ensemble du système, ce qui nous a naturellement conduit à développer une *analyse globale*.

Ceci va à l'encontre des pratiques actuellement les plus courantes dans le cadre de la compilation des langages à objets (ou plus largement, des langages procéduraux), qui consistent à utiliser la *compilation séparée* de fichiers sources partiels et indépendants en fichiers binaires, suivie d'une édition de liens permettant de mettre à jour les tables de symboles et de produire un seul exécutable. En effet, la compilation séparée est souvent considérée, pour des raisons historiques tenant essentiellement à la lenteur des machines disponibles, comme la seule technique permettant des (re)compilations rapides, grâce à son incrémentalité intrinsèque. Elle a de plus l'avantage d'être parfaitement adaptée à l'utilisation de bibliothèques logicielles du marché fournies sans leur code source (pour des raisons de protection de la propriété intellectuelle), qui sont intégrées lors de l'édition de liens. En revanche, le principal inconvénient de la compilation séparée est qu'elle ne permet d'optimiser que *localement*, à l'échelle du fichier source compilé, ou, au mieux, de l'ensemble des fichiers sources compilés. Elle ne permet ainsi pas de prendre en compte dans l'optimisation les fichiers et bibliothèques déjà compilés sous forme binaire[6].

Notre analyse globale, elle, se focalise précisément sur l'optimisation du système dans son ensemble. Elle a pour but de permettre au compilateur de mieux définir, de préciser le *contexte* dans lequel chaque élément du code source (expression, instruction, variable, méthode, etc.) va être compilé. Ceci permet de diminuer le degré d'incertitude lors de la génération de code et de passer ainsi de la génération d'un

6. Notons que ceci n'est plus tout à fait vrai lorsque des éditeurs de liens optimisants sont utilisés. Ces derniers sont néanmoins encore peu courants, assez délicats à mettre au point, peu portables, et disposent en général de beaucoup moins d'informations sémantiques qu'un compilateur travaillant directement sur le code source, ce qui limite leurs possibilités d'optimisation.

code le plus général possible, pour prendre en compte tous les cas possibles, à la production de code plus spécialisé, prenant en compte un ensemble de cas plus précis, plus restreint, et donc potentiellement plus efficace, comme nous l'expliquons dans les sections suivantes. De façon à avoir une analyse globale aussi rapide que possible, il nous parait important de ne pas compiler tout le code source accessible (y compris dans les bibliothèques) sans discrimination, mais seulement le code nécessaire au système, c'est à dire le code *vivant* de ce système.

2.3.2 Calcul du code vivant

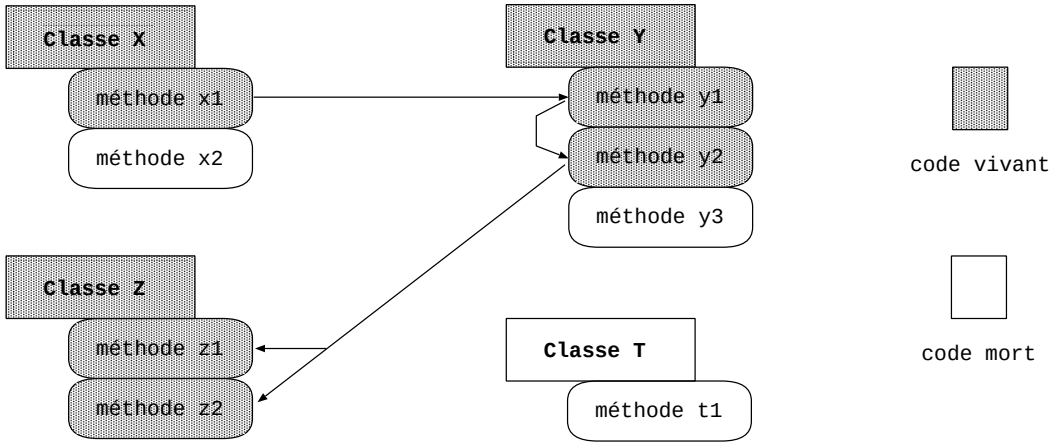


FIG. 2.1 – *Calcul du code vivant.*

En Eiffel, le point de départ d'un programme, appelé la *racine* de l'application [Meyer, 1992], est spécifié par une paire composée d'une classe initiale et de l'une de ses procédures de création. Le premier objet créé est donc une instance de la classe racine et la toute première opération consiste à lancer la procédure racine avec pour receveur l'objet racine.

La première étape de notre processus d'analyse statique globale calcule quelles parties du code source Eiffel sont *vivantes* (accessibles depuis la racine du système) ou *mortes* (inaccessibles). Partant de la racine du système, il calcule récursivement le *code vivant*, c'est à dire le code qui peut être exécuté. Le code qui ne peut être exécuté, parce qu'inaccessible, est appelé *code mort* (voir figure 2.1).

Ceci s'effectue pour l'instant de façon totalement statique et sans analyse de flot. Lorsqu'une instruction conditionnelle est rencontrée, toutes les alternatives sont a priori considérées comme pouvant être exécutées et donc vivantes. Le résultat de ce premier calcul est donc indépendant de l'ordre des instructions dans le programme.

Le calcul du code vivant est bien entendu étroitement lié à l'inventaire des types qui peuvent être instanciés à l'exécution. Par exemple, considérons la racine suivante:

```
class ROOT creation root feature
   a: A;
   root is
      do
         !!a;                              -- (1)
         a.add(123);                       -- (2)
         a.sub(321);                       -- (3)
      end;
end -- ROOT
```

De toute évidence, la procédure `root` de la classe ROOT appartient au code vivant et permet donc d'initialiser le processus. Comme cette procédure contient une instruction d'instanciation pour la classe A en ligne `(1)`, des instances du type A peuvent être créées. Les routines `add` et `sub` du type A peuvent être appelées (en lignes `(2)` et `(3)`). Par conséquent, ces deux routines sont ajoutées dans le code vivant, à partir duquel notre algorithme continue récursivement son exécution. Finalement, si le code des routines

`add` et `sub` ne contient aucune instruction d'instanciation et n'appelle aucun code en contenant, seules deux classes peuvent être instanciées à l'exécution de cette racine: la classe A et la classe ROOT. Par analogie avec les termes *code mort* et *code vivant*, une *classe vivante* est une classe pour laquelle au moins une instruction d'instanciation est dans le code vivant. A est donc une classe vivante. Au contraire, une *classe morte* est une classe dont aucune instance ne peut être créée dans le code vivant.

Un *type concret* est un type correspondant à une classe instanciable dans l'absolu, c'est à dire non abstraite. Un *type concret vivant* correspond à une classe qui peut effectivement être instanciée à l'exécution du programme. L'exploration du code vivant comme indiqué précédemment permet donc de calculer en même temps l'ensemble des types vivants, dont nous avons besoin à l'étape suivante.

Un intérêt immédiat du calcul du code vivant, avant même de raisonner en termes de collection d'informations, est que seul ce code *vivant* sera totalement compilé et donnera lieu à génération de code. Les classes mortes ne seront pas analysées du tout, alors que le code mort présent dans une classe vivante sera traité uniquement par l'analyse syntaxique (afin de vérifier que le code de la classe est syntaxiquement correct). Ceci revient donc à faire de l'élimination de code mort [Dhamdhere, 1991; Knoop *et al.*, 1994] de la façon la plus efficace qui soit, en minimisant le travail effectué sur le code mort. Ainsi, sur ce dernier, seules les premières étapes de la compilation sont effectuées, à savoir les vérifications syntaxiques pour contrôler la cohérence d'une classe.

Un inconvénient de cette méthode est qu'ainsi, il est possible que certaines erreurs sémantiques présentes dans des classes compilées ne soient pas détectées, car elle se trouvent dans des zones de code mort. Ces erreurs risquent donc de se révéler plus tard, lorsque l'évolution du développement du système rendra vivantes des zones de code précédemment mortes.

Notre calcul du code vivant contraste avec de nombreux compilateurs, qui compilent (partiellement ou en totalité) le code mort inutilement, l'éliminant seulement a posteriori, lors de la phase de génération de code. L'avantage majeur de notre système de *compilation à la demande* est donc qu'il permet un gain de temps non négligeable (cf. section résultats 2.7). De plus, l'élimination du code mort est également un facteur à prendre en considération car la taille des exécutables d'une application livrée revêt une certaine importance.

Remarquons que dans le cas des types paramétriques, implantés en Eiffel à l'aide de classes dites *génériques*, nous considérons tous les types qui sont des dérivations d'une même classe générique comme des types *distincts*. Ainsi, si l'on prend pour exemple la classe générique ARRAY, qui sert à implanter des tableaux flexibles, il est possible que dans un système de classes donné, les types ARRAY[CHARACTER] et ARRAY[FRUIT] soient vivants mais pas ARRAY[INTEGER].

Puisqu'il n'analyse pas le flot d'exécution et peut donc considérer comme vivantes des zones de code qui ne le sont pas, cet algorithme très simple est limité. Il représente cependant à notre avis un bon compromis, car son coût en temps d'exécution reste très raisonnable. De plus, comme l'information liée aux instanciations de classes semble plus importante que l'information basée sur le flot [Bacon et Sweeney, 1996], le coût d'une analyse de flot apparaît très élevé en regard du gain qu'elle pourrait apporter.

Enfin, notons que notre analyse n'implique aucun surcoût à l'exécution, puisqu'elle est réalisée lors de la compilation.

2.4 Duplication et spécialisation du code généré

Une fois le code source analysé globalement, les informations collectées, notamment celles se rapportant aux types et au code vivant, vont pouvoir être utilisées de façon à générer un code cible optimisé, grâce à une méthode à base de duplication-spécialisation.

2.4.1 La duplication de code

Un aspect fondamental du concept d'héritage est que, lorsqu'une primitive `prim` est présente dans le code source d'une classe ancêtre A, elle est également conceptuellement présente et disponible dans les descendants de A, même si elle n'apparaît pas explicitement dans leur code source. Ainsi considérons l'arbre d'héritage de la figure 2.2, où est définie une primitive `prim` dans la classe A, sans aucune redéfinition dans ses descendants. On note $\texttt{prim}_X$ la primitive `prim` dans la classe X. En fait, dans cet exemple,

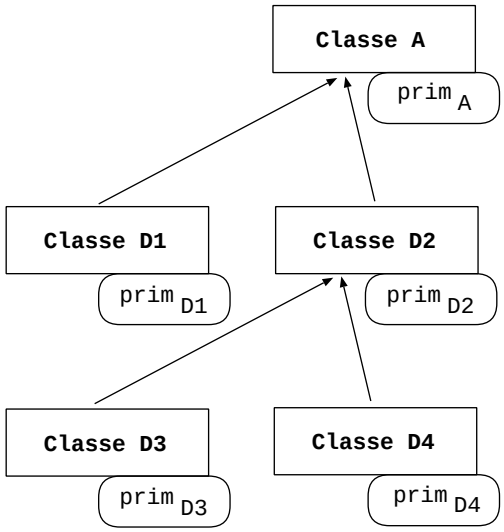


FIG. 2.2 – *Héritage d'une primitive.*

seule $\texttt{prim}_{\texttt{A}}$ existe vraiment, et $\texttt{prim}_{\texttt{D1}} = \texttt{prim}_{\texttt{D2}} = \texttt{prim}_{\texttt{D3}} = \texttt{prim}_{\texttt{D4}} = \texttt{prim}_{\texttt{A}}$. Des mécanismes comme la liaison dynamique (cf. section 2.5) permettent de faire en sorte qu'un appel à `prim` sur un descendant soit correctement exécuté, par appel de la primitive de l'ancêtre.

La duplication du code [Chambers et Ungar, 1989; Chambers et Ungar, 1990] consiste à *recopier* un exemplaire de cette primitive dans chacune des classes qui héritent de A. Cette recopie reste conceptuelle au niveau du code source (elle n'est pas effectivement réalisée), mais elle est faite physiquement au niveau du code cible généré lors de la compilation.

Lorsqu'il y a duplication, les cinq primitives $\texttt{prim}_{\texttt{D1}}$, $\texttt{prim}_{\texttt{D2}}$, $\texttt{prim}_{\texttt{D3}}$, $\texttt{prim}_{\texttt{D4}}$ et $\texttt{prim}_{\texttt{A}}$ de la figure 2.2 existent donc toutes réellement et individuellement dans le code généré, même si elles contiennent un code identique. De façon imagée, la duplication revient à "mettre à plat" chaque classe vivante, faisant "tomber" les primitives (routines ou attributs) le long du graphe d'héritage, depuis les ancêtres ("haut") vers les descendants ("bas"). Tout au long de cette "chute", les éventuels renommages de primitives — autorisés en Eiffel pour éviter certains problèmes posés par l'héritage multiple, comme les collisions de noms (*name clashes*) — sont pris en compte par notre algorithme, tout comme bien sûr les redéfinitions (*overriding*) et les fusions de primitives — dans le cas d'un héritage répété. De même, les assertions sont accumulées dans les héritiers, en suivant les règles de la conception par contrat (*design by contract*, [Meyer, 1988]).

Rappelons que, comme nous l'avons mentionné dans la section 2.3 sur la prédiction de type, des dérivations différentes d'un même type générique sont considérées comme des classes différentes. En conséquence, des ensembles de code spécifiques à chaque dérivation vivante d'une classe générique, par exemple ARRAY[INTEGER] et ARRAY[CHARACTER], sont produits.

Bien entendu, la simple duplication de code ne présente pas grand intérêt. Elle présente même un inconvénient important, qui est l'augmentation de la taille du code généré, et donc de la taille des programmes résultants. En revanche, la duplication de code prend tout son sens lorsqu'elle est complétée par une *spécialisation* dudit code.

2.4.2 La spécialisation de code

La spécialisation de code (*code customization*, [Chambers et Ungar, 1989; Chambers et Ungar, 1990]) consiste à considérer le code de chaque routine — potentiellement hérité d'une super-classe — et à l'adapter en fonction des informations connues sur le contexte de la classe courante, c'est à dire essentiellement le type exact du receveur. Il est ainsi possible de générer du code cible optimisé, notamment par calcul *statique* de certaines expressions [Collin *et al.*, 1997].

Le calcul statique des expressions est une optimisation classique dans les compilateurs et qui n'est ni propre aux langages à objets, ni limitée au code spécialisé. Dans un langage qui n'est pas à objets, ceci consiste à propager des constantes et à calculer tout ce qui peut l'être à la compilation au lieu de le faire à l'exécution. Dans un langage à objets, le calcul statique des expressions est limité par le polymorphisme (cf. section 2.2, qui empêche la propagation de certaines constantes, introduisant un certain degré d'incertitude. Avec la duplication et la spécialisation, le polymorphisme et l'incertitude "diminuent", ce qui permet une meilleure propagation des constantes et donc un meilleur calcul statique des expressions.

Un bon exemple de l'intérêt de la duplication et de la spécialisation est l'utilisation de *template methods*, qui sont très courantes dans les langages à objets [Gamma *et al.*, 1995; Jézéquel *et al.*, 1999]. En effet, il arrive souvent que tout ou partie d'une *template method* ne puisse être calculée statiquement dans le contexte de la classe où elle est définie mais qu'elle puisse l'être dans certains de ses descendants, où plus d'information est disponible. Considérons par exemple la classe abstraite suivante:

```
deferred class FRUIT

   stone_fruit: BOOLEAN is  deferred end;

   display_more is
      do
         if stone_fruit then                -- (1)
            print("This is a stone fruit."); -- (2)
         else
            print("Not a stone fruit.");     -- (3)
         end;
      end
```

La méthode `display_more` est une *template method* qui utilise la primitive abstraite `stone_fruit` de la classe FRUIT. Si l'on suppose que l'on a deux descendants de FRUIT, nommés PEACH et APPLE, `stone_fruit` est définie comme la constante `true` dans PEACH et comme la constante `false` dans la classe APPLE. Comme toute autre procédure, la *template method* `display_more` est dupliquée et spécialisée pour chaque type auquel elle appartient. Lorsque le receveur est un objet de type PEACH, l'expression conditionnelle `stone_fruit` de la ligne `(1)` est toujours vraie. La ligne `(2)` est donc toujours exécutée, alors que la ligne `(3)` n'est jamais atteinte. Ainsi, l'expression conditionnelle qui constitue le corps de la routine `display_more` peut être, dans PEACH, évaluée statiquement et remplacée par l'instruction d'affichage de la ligne `(2)`, alors que dans `apple` elle est remplacée par l'instruction d'affichage de la ligne `(3)`.

Un autre exemple spécifique aux langages à objets est le test qui doit être effectué lors de l'exécution pour vérifier qu'un appel de primitive s'effectue bien avec un receveur non nul [7]. Ce test n'est pas nécessaire lorsque le receveur d'un appel est l'objet `Current` [8], qui bien entendu ne peut jamais être `Void` [9]. Ce cas est fréquent, puisque de nombreux appels intra-classe se font sur le receveur `Current`. Le même raisonnement s'applique lorsqu'on a affaire à un receveur qui est une variable contenant un objet Eiffel `expanded`, c'est à dire passé par valeur. Ceci est également courant, puisqu'en plus des objets définis comme `expanded` par l'utilisateur, tous les types de base comme INTEGER, REAL, DOUBLE, CHARACTER, etc. sont `expanded` et sont bien évidemment soumis à de nombreux appels. Il est donc important de bien spécialiser et optimiser ces deux situations.

Lors de la spécialisation, lorsqu'un site d'appel (potentiellement polymorphique) est rencontré, l'ensemble des types vivants possibles pour le receveur est connu, grâce à l'algorithme de prédiction de type présenté précédemment à la section 2.3. Si cet ensemble a pu être réduit à un seul type, l'appel devient monomorphique et peut être codé par un appel direct. Le bénéfice de notre méthode est donc maximal dans ce cas. Les deux types d'appels sur `Current` ou sur des objets `expanded` mentionnés précédemment

7. Ceci est en fait valable pour les méthodes d'instances, mais ne s'applique pas aux méthodes de classes, dites routines *statiques* en C++ ou Java. Dans la définition du langage Eiffel, il n'y a pas de notion de méthode de classe, toutes les méthodes étant des méthodes d'instance. Notre compilateur est cependant capable de détecter des routines qui ne dépendent pas du receveur et sont, de fait, des méthodes de classes.

8. `Current` est la notation Eiffel pour l'objet courant, comme `self` en Smalltalk [Goldberg et Robson, 1983], ou `this` en Java [Gosling *et al.*, 1996] ou C++ [Stroustrup, 1986].

9. `Void` est en Eiffel la référence nulle, équivalente au `NULL` de C/C++ ou `null` de Java.

sont justement des cas triviaux où le type du receveur est connu avec certitude, ce qui résulte en un appel direct.

Dans le cas où l'ensemble des types possibles est vide, car aucune instruction d'instanciation pour le type statique du receveur ou l'un de ses sous types n'est présente dans le code vivant, notre algorithme signale, à la compilation, une erreur de cohérence globale du système. Il s'agit là d'une fonctionnalité de notre compilateur qui n'est à notre connaissance fournie dans aucun autre compilateur Eiffel.

Nous utilisons bien sûr les informations fournies par l'analyse statique du code source (cf. section 2.3) pour ne compiler, dupliquer et spécialiser que les primitives qui font partie du code vivant. Ceci contribue grandement à limiter la quantité de travail effectué lors de la compilation, ainsi que la taille du code généré.

Évidemment, les 100 % de prédiction correcte du type du receveur constituent une limite qui ne peut être atteinte pour tous les programmes: un simple tableau rempli avec des objets mélangés, de types différents et connus seulement à l'exécution, suffit à mettre en échec n'importe quel système d'inférence de type. La plupart des programmes contiennent donc des sites réellement polymorphiques, même après le meilleur des algorithmes de prédiction de type. Lorsqu'on considère tous les bancs d'essais de [Diwan *et al.*, 1996], on peut remarquer que les sites polymorphiques sont appelés 26 fois plus souvent que les sites monomorphiques. Il est donc crucial d'optimiser au mieux les sites polymorphiques restants.

2.5 Techniques d'implantation de la liaison dynamique

Dans cette section, nous présentons les techniques classiques d'implantation de la *liaison dynamique*. Remarquons que dans toute la suite de ce mémoire, nous nous focalisons sur les méthodes simples, pour lesquelles le polymorphisme ne concerne que le type dynamique du receveur (*single dispatching*). L'étude des multiméthodes (*multiple dispatching*) ne rentre pas dans ce cadre.

Nous nous intéressons tout d'abord aux *techniques statiques*, c'est à dire basées sur des calculs faits lors de la compilation. Nous passons ensuite en revue des *techniques dynamiques*, basées sur des informations calculées lors de l'exécution.

2.5.1 Techniques statiques d'envoi de message

Comme nous l'avons détaillé dans la section 2.2 précédente, la liaison dynamique est un point capital dans les langages à objets. son implantation est donc un facteur important, qui a fait l'objet de nombreux travaux. Nous faisons ici un bref rappel des techniques de liaison dynamique classiquement implantées dans les compilateurs [Driesen *et al.*, 1995; Driesen, 1999].

Il existe deux principaux types de techniques d'envoi de message: les techniques statiques et les techniques dynamiques.

Cette section porte sur les techniques *statiques* d'implantation de la liaison dynamique, c'est à dire les techniques précalculant leurs données et structure de code à la compilation afin de minimiser le travail fourni lors de l'exécution. La section 2.5.2 suivante présente les techniques *dynamiques*, basées sur des structures de données calculées lors de l'exécution du programme.

Les diverses techniques statiques qui ont été inventées pour compiler la liaison dynamique peuvent globalement être divisées en deux catégories: les techniques de parcours de graphe, et les techniques à base de tables.

2.5.1.1 Recherche de message

La méthode la plus simple qui vient à l'esprit pour implanter la liaison dynamique est la recherche de message (*Message Lookup* ou *Dispatch Table Search — DTS*). Elle consiste à associer à chaque objet un descripteur de classe contenant (entre autres) les méthodes associées aux objets de cette classe (figure 2.3). Lors de l'exécution d'un appel polymorphique, la méthode est cherchée dans la table de fonctions correspondant au type de l'objet receveur. Si elle n'est pas trouvée, la recherche se poursuit dans la ou les super-classe(s). Le graphe d'héritage est donc parcouru lors de l'exécution. Lorsqu'aucune méthode n'est trouvée, un message d'erreur peut être émis.

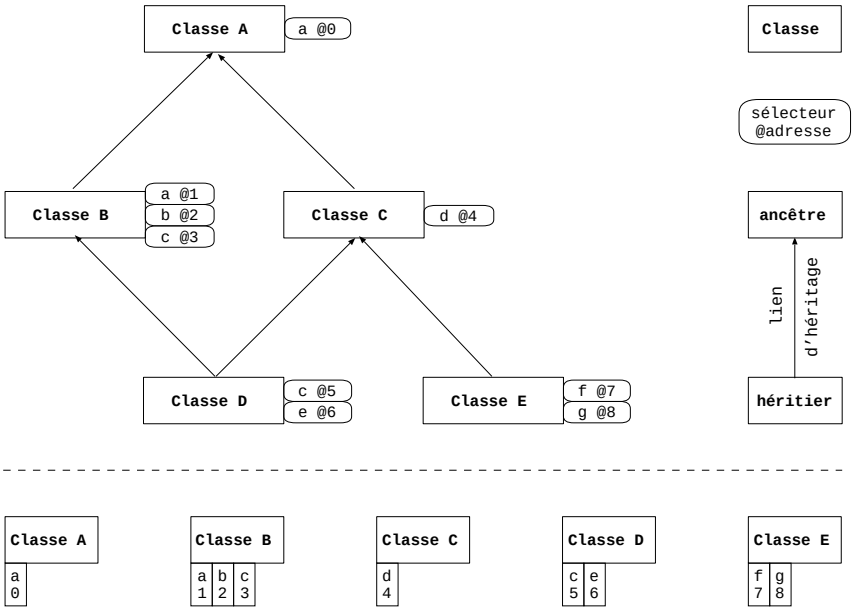


FIG. 2.3 – *Recherche de message (Message Lookup ou Dispatch Table Search).*

Haut: Hiérarchie de classes, avec noms des sélecteurs sélecteur dans les classes où ils sont définis et les adresses de leur code symbolisées par des entiers.
Bas: Tables de liaison dynamique correspondant à cette hiérarchie.

Cette technique basique est facile à compiler et nécessite peu de mémoire, mais elle est coûteuse en temps d'exécution du programme compilé. Notons également que dans la version de base de cette technique, le temps de recherche d'une méthode est variable, le graphe d'héritage devant potentiellement être parcouru en totalité dans le cas le plus défavorable. Des versions optimisées font appel à une recherche de méthode plus rapide avec hachage.

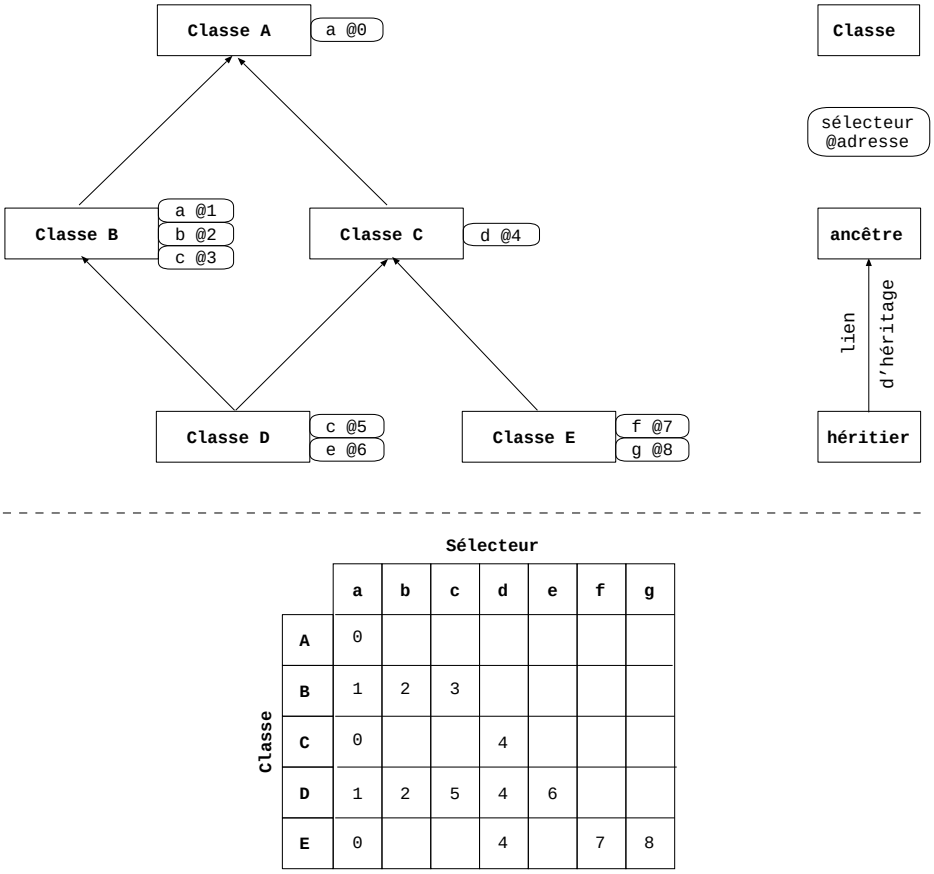
2.5.1.2 Table indexée par sélecteur

Conceptuellement, comme nous l'avons dit, la liaison dynamique revient à associer une méthode (implantation) à un couple composé d'un type (classe) et d'un sélecteur (nom de méthode, ou plus généralement signature). Il est donc assez naturel de vouloir l'implanter à l'aide d'une table à deux dimensions, globale au programme, contenant des références sur le code des méthodes, appelée *table de sélecteurs* (voir figure 2.4).

L'avantage principal des techniques à base de *table indexée par sélecteur* (*Selector Table Indexing — STI*) est que contrairement à la recherche dans un graphe, l'accès à une méthode s'y fait en temps constant.

L'implantation la plus simple est hélas coûteuse en place mémoire, puisque la taille de la table est le produit du nombre de classes par le nombre de sélecteurs dans le système. De plus, le gaspillage de place est important, puisque la plupart des cellules de la table — 95 % dans le système Smalltalk [Driesen, 1993] — seront vides, ou plus précisément correspondront au message d'erreur indiquant l'absence de méthode. En effet, un sélecteur n'est, en général, défini que dans un faible nombre de classes du système.

Des techniques ont donc fort logiquement été cherchées pour améliorer ce système de table de sélecteurs. On peut les classer en deux catégories: les techniques qui visent à optimiser une table globale, et celles qui éclatent la table globale en sous-tables locales.



Classe	a	b	c	d	e	f	g
A	0						
B	1	2	3				
C	0			4			
D	1	2	5	4	6		
E	0			4		7	8

FIG. 2.4 – *Table indexée par sélecteur (Selector Table Indexing).*

Haut: Hiérarchie de classes, avec noms des sélecteurs dans les classes où ils sont définis et les adresses de leur code symbolisées par des entiers.
Bas: Table globale indexée par sélecteur. La méthode `a` choisie pour la classe D est celle venant de B, ce qui correspond à la méthode *dominante* en cas d'héritage virtuel en C++ [Ellis et Stroustrup, 1990; Lippman, 1993].

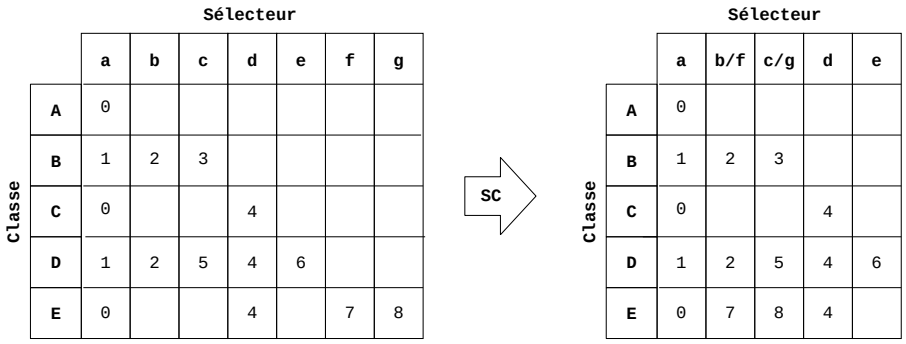


FIG. 2.5 – *Table à coloration de sélecteurs (Selector Coloring Table).*

Gauche: Table globale indexée par sélecteur. Les adresses du code des sélecteurs sont symbolisées par des entiers.
Droite: La même table, compactée par coloration de sélecteurs.

2.5.1.3 Coloration de sélecteurs

La *coloration de sélecteurs* (*Selector Coloring*, [Dixon *et al.*, 1989; André et Royer, 1992]) est une technique permettant de compacter une table de sélecteurs globale.

Soient deux sélecteurs `s1` et `s2`. Soit $C(x)$ l'ensemble des classes où le sélecteur `x` est présent. Si $C(s1) \cap C(s2) = \emptyset$, `s1` et `s2` partagent la même couleur. Sinon, `s1` et `s2` ont des couleurs différentes. Lors de la construction de la table de sélecteurs, les sélecteurs ayant la même couleur peuvent être implantés dans la même ligne/colonne. Il est ainsi possible de compacter sensiblement la table, même si celle-ci reste encore très creuse (40 % d'entrées à vide, dans un système Smalltalk [Driesen, 1993]).

La figure 2.5 page 17 donne un tel exemple de compactage de table par coloration de sélecteur, le sélecteur `f` pouvant se voir attribuer la même couleur que le sélecteur `b`, et le sélecteur `g` la même couleur que le sélecteur `c`.

Notons qu'il est possible d'affecter les couleurs à faible coût, à l'aide d'algorithmes rapides donnant une bonne approximation de la solution optimale.

2.5.1.4 Déplacement de lignes

Le *déplacement de lignes* (*Row Displacement*) est une autre technique permettant de compacter des tables de sélecteurs. On peut dire qu'elle constitue une généralisation de la coloration de sélecteurs présentée précédemment.

Le déplacement de lignes consiste, en partant de la table à deux dimensions (avec sélecteurs en lignes et classes en colonnes), à faire "glisser" les lignes de façon à ce que toute cellule vide d'une ligne corresponde à une — et une seule — cellule non vide d'une autre ligne (voir figure 2.6). Chaque ligne débute à un décalage différent par rapport à la base du tableau, ce qui permet d'adresser le bon sélecteur. Ainsi, on peut obtenir un tableau à une dimension assez bien rempli. Ce taux de remplissage dépend de l'algorithme de déplacement des lignes, ainsi que de l'algorithme d'affectation des identifiants de classes. Notons que le découpage du tableau en lignes de sélecteurs s'avère plus efficace que le découpage en lignes de classes, et permet de réduire le nombre d'entrées vides à moins de 1 % de la taille du tableau [Driesen et Hölzle, 1995].

2.5.1.5 Table indexée par sélecteur compacte

Cette technique de *table indexée par sélecteur compacte*, ou *Compact Selector-Indexed Dispatch Table*, permet également de compresser la table globale des sélecteurs.

Elle calcule tout d'abord deux tables (voir figure 2.7 page 20). L'une est associée aux sélecteurs "avec conflit" (*conflict selectors*), c'est à dire les sélecteurs dont plusieurs définitions *indépendantes* existent à divers endroits de la hiérarchie (par exemple le sélecteur `d` dans la figure 2.7). L'autre sert pour les sélecteurs normaux, qui sont définis à un seul endroit de la hiérarchie des classes (avec éventuellement des redéfinitions dans les descendants). Ces tables peuvent bien entendu être compressées avec des techniques classiques comme la coloration de sélecteurs (voir ci-dessus la section 2.5.1.3, et l'étape (2) au bas de la figure 2.7), la fusion des lignes ayant les mêmes entrées (étape (3)), etc.

Le point le plus important de cette technique est qu'elle permet la fusion non seulement des entrées identiques de la table, mais aussi de certaines entrées différentes, ce qui augmente ainsi la compacité de la table. C'est un *critère de similarité* paramétrable qui permet de choisir quelles entrées non identiques sont ainsi fusionnées (*surchargées*).

Les étapes (2) et (3) du bas de la figure 2.7 montrent un tel exemple de surcharge pour les classes B et D, dont les entrées ne présentent qu'une seule différence (adresse 3 ou 5 pour le sélecteur `c`). Le code d'aiguillage pour les sélecteurs `a` et `b` est le même pour la classe B et la classe D. En revanche, pour le sélecteur `c`, dont l'entrée est surchargée, il est nécessaire de faire la distinction entre les deux types de receveur possibles en exécutant un petit fragment de code d'aiguillage vérifiant le type dynamique. Notons qu'un autre critère de similarité aurait pu choisir de fusionner l'entrée des classes A et C et celle de la classe E, qui ont un même comportement vis-à-vis du sélecteur `a`.

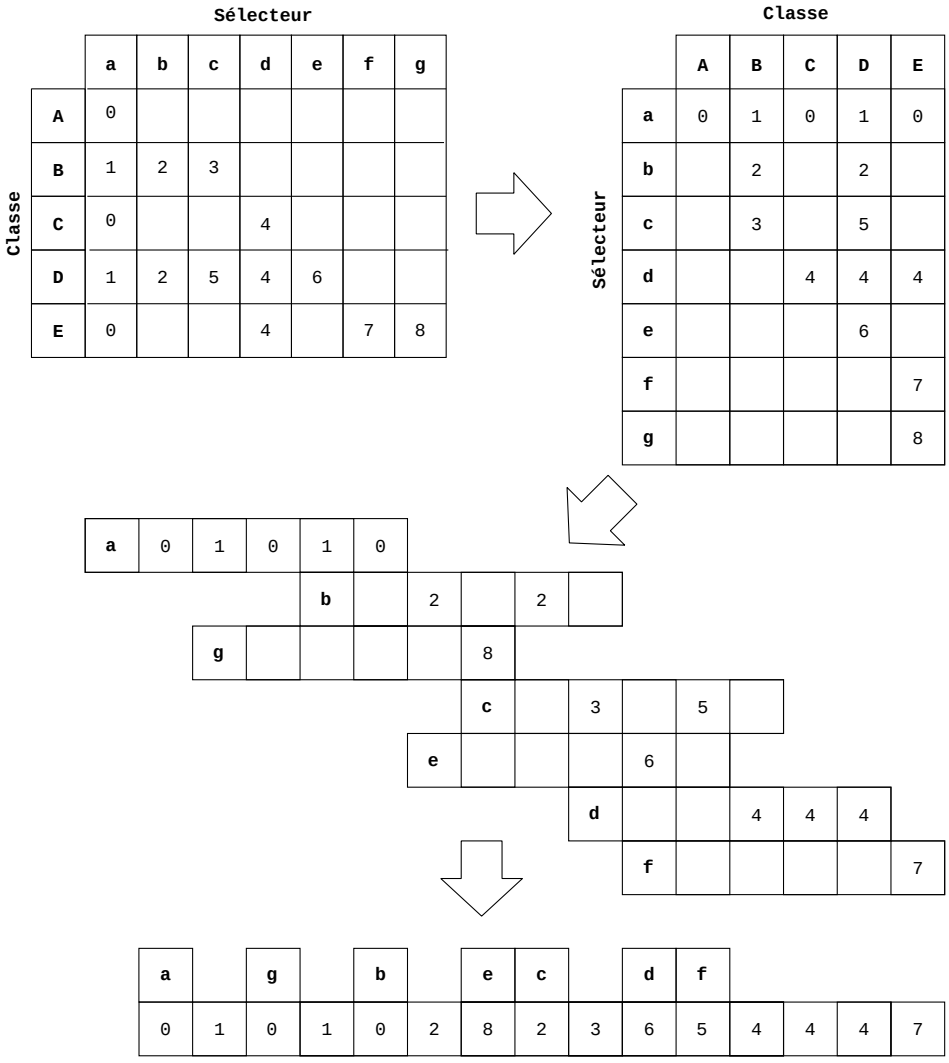


FIG. 2.6 – *Déplacement de lignes (Row displacement).*

Haut-Gauche: Table globale indexée par sélecteur. Les adresses du code des sélecteurs sont symbolisées par des entiers.
Haut-Droite: La transposée de cette table.
Milieu: L'algorithme de décalage des lignes détaillé.
Bas: Le tableau compact résultant.

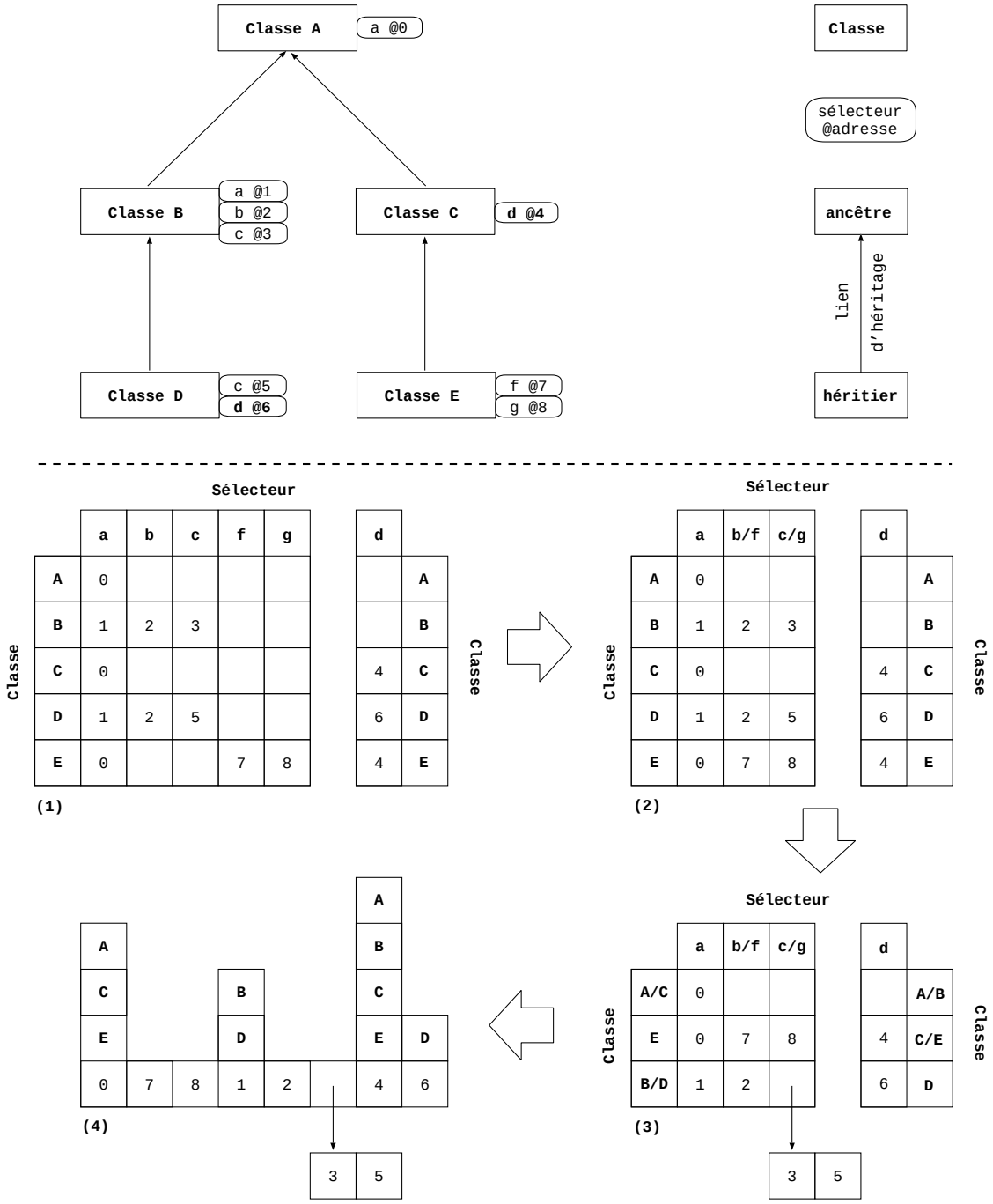


FIG. 2.7 – *Table indexée par sélecteur compacte (Compact selector-indexed dispatch table).*

Haut: Hiérarchie de classes, avec noms des sélecteurs dans les classes où ils sont définis et les adresses de leur code symbolisées par des entiers.
Bas: Construction de la table compacte correspondante: (1) Table principale et table de conflit pour `d`. (2) Compactage par aliasing de `b` avec `f` et de `c` avec `g`. (3) Compactage par partage des tables de conflit entre A et B et entre C et E, plus partage des tables principales entre A et C et entre B et D. La table principale pour B et D est surchargée pour les sélecteurs `c` et `g`. (4) Compactage en un seul tableau.

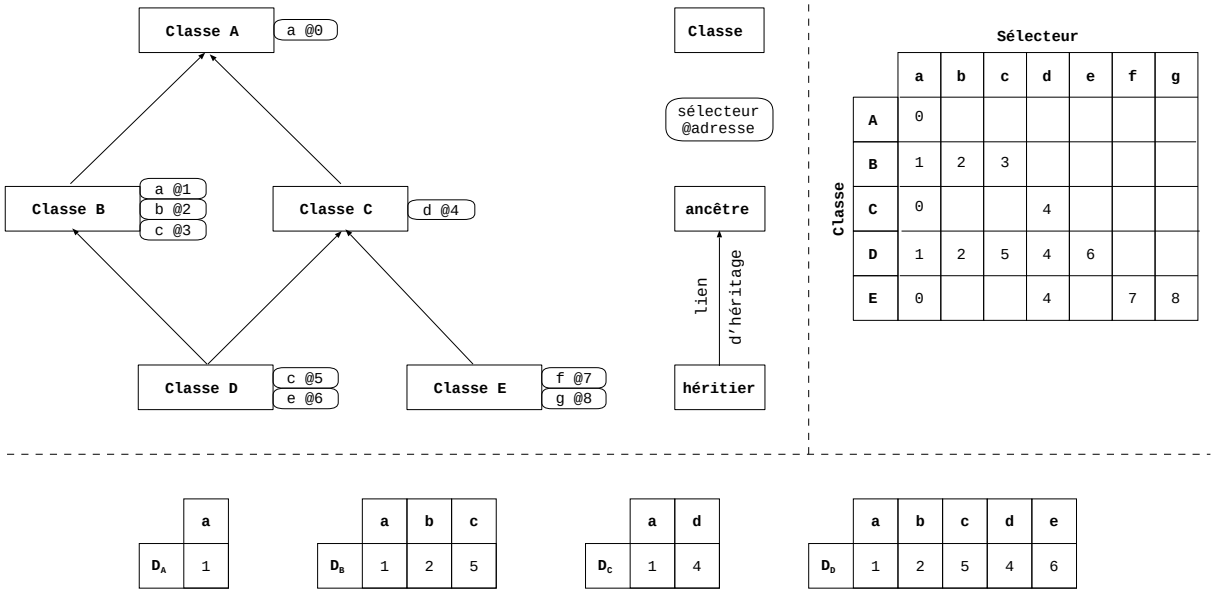


FIG. 2.8 – *Tables de fonctions virtuelles (Virtual Function Tables — VFTs).*

Haut-Gauche: Hiérarchie de classes, avec noms des sélecteurs dans les classes où ils sont définis et les adresses de leur code symbolisées par des entiers.
Haut-Droite: La table indexée par sélecteur correspondante.
Bas: Les tables de fonctions virtuelles pour la classe D.

2.5.1.6 Tables de fonctions virtuelles (VFT)

Les *tables de fonctions virtuelles* (*Virtual Function Tables — VFTs ou VTBLs* [Ellis et Stroustrup, 1990; Driesen *et al.*, 1995; Driesen et Hölzle, 1996]) constituent actuellement la méthode la plus couramment utilisée pour implanter la liaison dynamique en Java et C++. Les explications que nous donnons ci-dessous baseront leurs exemples sur C++.

La technique des VFTs consiste à éclater la table des sélecteurs en sous-tables, en affectant les numéros des sélecteurs localement à une classe. Chaque (descripteur de) classe référence donc directement ses propres tables de sélecteurs, correspondant à la classe elle-même et à ses ancêtres.

La figure 2.8 page 21 montre un exemple de VFT en présence d'héritage multiple. Notons qu'on se place dans le cas le plus général et le plus puissant en C++, c'est à dire l'héritage répété avec classe de base *virtuelle* (*virtual base class*), appellée parfois classe de base *partagée* (*shared base class*) [Ellis et Stroustrup, 1990; Lippman, 1993]. Cela correspond également à la façon dont l'héritage est conçu en Eiffel.

Dans l'exemple de la figure 2.8, la classe D a trois ancêtres, A, B et C. Elle a donc une VFT D_A, utilisée pour faire la liaison dynamique sur les objets de type dynamique D vus au travers d'une expression de type statique A, une VFT D_B, utilisée lorsque ces objets sont considérés via une expression de type statique B, une VFT D_C, utilisée lorsque ces objets sont considérés via une expression de type statique C et une VFT D_D utilisée lorsque les objets de type dynamique D sont manipulés via une expression de même type statique D. Ainsi, une représentation possible en mémoire des objets de type D et de leurs VFTs est celle de la figure 2.9, page 22. Lorsqu'un objet de type dynamique D est référencé via une variable de type statique D, la référence pointe au début de l'objet (`ptrD`). Lorsqu'il est référencé via une variable de type C — et subit donc une conversion vers le type C — la référence pointe au début de la partie C de l'objet (`ptrC`), et est donc décalée de `décalage(C)` par rapport au début de l'objet. Il en est de même pour A et B. Notons que comme la partie B se trouve au début de l'objet, `ptrB` et `ptrD` sont égaux, `décalage(B)` et `décalage(D)` étant tous deux égaux à zéro.

Pour illustrer le code implantant la liaison dynamique à l'aide de VFTs, ainsi que pour expliciter les

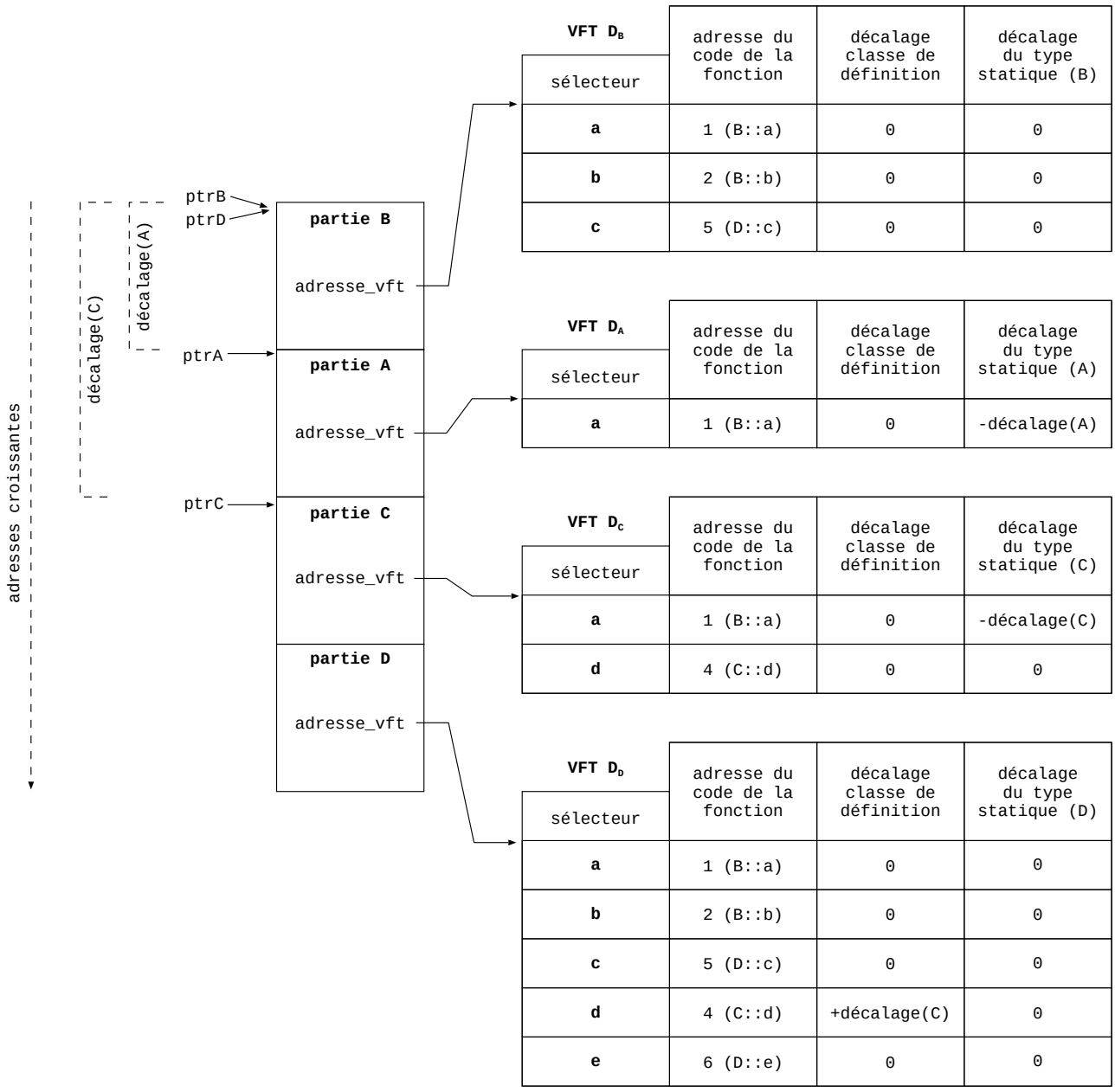


FIG. 2.9 – *Représentation mémoire et VFTs d'un objet sujet à héritage multiple.*

A gauche, représentation mémoire d'une objet de type D. A droite, ses VFTS.

différents éléments de la figure 2.9, considérons l'appel de méthode suivant:

```
objet.selecteur(arguments)
```

L'appel ci-dessus se traduit par le (pseudo-)code suivant:

```
vft = objet.adresse_vft;                    /* (1) */
entre_vft = vft[selecteur];                 /* (2) */
objet += entree_vft.decalage;               /* (3) */
*(entree_vft.fonction)(objet, arguments);   /* (4) */
```

L'instruction `(1)` consiste à charger la VFT associée au receveur de l'appel (et donc à son type dynamique), compte tenu du type statique de la variable `objet` qui le contient. Par exemple, supposons que le receveur, de type dynamique D, est référencé via une variable de type (statique) C. Dans ce cas, c'est la VFT D_C qui est chargée.

L'instruction `(2)` accède à l'entrée de cette VFT qui correspond au `selecteur` de l'appel. Considérons pour notre exemple qu'il s'agit du sélecteur `a`. Cette entrée contient à la fois l'adresse de la fonction à appeler (dans notre exemple, c'est l'adresse symbolisée par la valeur 1), et le décalage nécessaire à l'ajustement du pointeur sur le receveur.

Ce décalage est fait en instruction `(3)`. En effet, la fonction appelée doit avoir pour paramètre un pointeur sur un receveur (`this` en C++) correspondant au type dans lequel la fonction a été définie. Mais l'objet peut très bien être référencé via une variable d'un autre type. Ainsi, dans notre exemple, le sélecteur `a` adéquat (d'adresse 1) est celui défini dans la classe B, et attends donc un pointeur sur un objet de type B, ou sur la partie B d'un objet D (pointeur `ptrB` sur la figure 2.9). Or dans notre exemple l'objet l'objet receveur est référencé via une variable de type C, qui pointe donc sur la partie C de l'objet (pointeur `ptrC` dans la figure 2.9). Il est donc nécessaire de réajuster l'adresse de `this` de façon à la faire pointer au début de la partie B, en soustrayant à cette adresse le décalage de la partie C (type de la variable) et en y ajoutant le décalage de la partie B (type du fournisseur de la méthode). Comme ces deux décalages sont calculés statiquement (lors de la compilation), il suffit de stocker dans la VFT leur somme algébrique, ce qui explique que l'ajustement soit fait par la seule instruction `(3)`. Notons qu'il existe d'autres façons d'ajuster le receveur, mais qui sont moins efficaces [Ellis et Stroustrup, 1990, section 10.8c].

Une fois le receveur ajusté, l'instruction `(4)` effectue l'appel proprement dit, par appel de fonction indirect à l'adresse stockée dans l'entrée de la VFT.

Notons que comme les pointeurs sur les parties B et D de l'objet sont identiques (`ptrB` et `ptrD`), il est possible de fusionner les VFTs D_B et D_D. De façon générale, un objet d'une sous-classe (*derived class*) a besoin d'une VFT pour chaque ancêtre (*base class*), plus une pour la classe dérivée elle-même, mais peut partager cette dernière VFT avec celle de sa première classe de base [Ellis et Stroustrup, 1990, section 10.8c].

2.5.2 Techniques dynamiques d'envoi de message

En plus des techniques *statiques* de liaison dynamique vues ci-dessus section 2.5.1, il existe de nombreuses techniques *dynamiques*, permettant de résoudre la liaison dynamique à l'aide d'informations et structures basées sur des informations d'exécution. Ces dernières sont collectées soit en continu pendant l'exécution du programme soit au cours de pré-exécutions servant à établir des profils d'exécution (*profiling*, [Hölzle *et al.*, 1991; Aigner et Hölzle, 1996]) suivies d'une recompilation-optimisation partielle.

Ces techniques dynamiques sont en général basées sur divers types de caches à l'exécution [Deutsch et Schiffman, 1984; Ungar et Patterson, 1987], et peuvent donc éventuellement être couplées avec les techniques statiques vues à la section 2.5.1 afin d'en améliorer les performances. Ces techniques dynamiques semblent donc, en particulier, pouvoir être complémentaires à la technique que nous présentons dans la section 2.6.

2.5.2.1 Recherche de message avec cache global

Comme nous l'avons indiqué précédemment en section 2.5.1.1, la technique de recherche de message simple (ou de *lookup* global), implantée notamment dans Smalltalk, peut avoir un coût important.

Les premières implantations de Smalltalk ont donc cherché à en améliorer les performances en utilisant un cache global. Celui-ci consiste en une table de hachage indexée par un code de hachage calculé à partir du nom du sélecteur et du type dynamique du receveur. Les entrées de la table sont des paires, contenant une clé formée d'un type et du nom d'un sélecteur, et une valeur qui est (l'adresse de) la méthode associée à la clé.

A chaque appel, le code de hachage correspondant est calculé, fournissant l'index d'une entrée de la table. Dans le cas où la classe et le sélecteur de l'appel correspondent bien à ceux de l'entrée, la méthode référencée par le cache peut directement être exécutée.

Dans le cas contraire (collision de code de hachage), une recherche de message classique est effectuée. Le résultat de la recherche est utilisé pour mettre à jour l'entrée du cache en accord avec la stratégie de mise à jour choisie. Dans le cas classique, c'est la stratégie *MRU – Most Recently Used* qui est appliquée, les données du dernier appel — type dynamique du receveur, sélecteur et adresse de la méthode retrouvée par la recherche de message — remplaçant celles de l'entrée correspondante de la table.

Cette technique de *cache global de recherche de message* (*Global Lookup Cache*) augmente fortement les performances de la liaison dynamique, notamment lorsque les collisions dans la table de hachage sont limitées, et elle a donc été très utilisée dans les premières implantations de Smalltalk [Goldberg et Robson, 1983; Krasner, 1983]. Elle a par contre l'inconvénient d'imposer un calcul de code de hachage à chaque appel polymorphique, ce qui s'avère relativement coûteux [Driesen, 1999].

2.5.2.2 Caches en ligne

Cet inconvénient du cache global de recherche de message, ainsi que le fait que le cache global ne permet pas de profiter pleinement des phénomènes de localité dans les programmes, ont conduit à développer des techniques basées sur des caches *locaux*. En effet, on sait que pour un site d'appel donné, le type du receveur varie peu au cours de l'exécution d'un programme. En Smalltalk, par exemple, des études [Deutsch et Schiffman, 1984; Ungar et Patterson, 1987; Ungar, 1987] ont montré qu'il restait constant 95 % du temps.

On associe donc à chaque site d'appel un petit *cache en ligne* (*Inline Cache*) qui sert à stocker le type du receveur et l'adresse de la méthode correspondant au précédent appel observé à ce site. Si l'appel suivant s'effectue sur un objet de même type (*cache hit*), la méthode cachée est directement appelée. Dans le cas où le type du receveur change par rapport au précédent appel (*cache miss*), une recherche de méthode classique est effectuée et le cache mis à jour avec le nouveau type et l'adresse de la nouvelle méthode.

Cette méthode à base de caches locaux améliore fortement la précédente avec cache global. En effet, elle est très peu coûteuse dans le cas d'un *cache hit*: une simple comparaison de types remplace notamment l'accès à une table de hachage. Elle est en revanche coûteuse dans le cas d'un *cache miss*, puisqu'au code de recherche de méthode classique vient s'ajouter le surcoût du maintient du cache. Fort heureusement, les taux de succès des accès aux caches locaux sont sont en général très bons, atteignant 90 à 99 % sur des programmes typiques en Smalltalk [Ungar, 1987] ou Self [Hölzle et Ungar, 1994]. Ceci peut se traduire par des améliorations en termes de performances de l'ordre de 33 % [Ungar et Patterson, 1987]. Les caches en ligne ont donc remplacé le cache global dans de nombreuses implantations actuelles de Smalltalk, ainsi que dans les implantations initiales de Self [Chambers et Ungar, 1989].

2.5.2.3 Caches en ligne polymorphiques

Les caches en ligne fonctionnent très bien lorsque le type du receveur à un site d'appel donné reste globalement constant, ce qui est le cas comme nous l'avons vu d'une majorité de d'appels. En revanche, lorsque sur un site d'appel polymorphique plusieurs types alternent de façon régulière, le type du receveur peut changer à chaque fois, ce qui met systématiquement en échec la prédiction du cache en ligne. Ce dernier a donc pour effet, dans ce cas, de ralentir le processus de liaison dynamique au lieu de l'accélérer. Ainsi, des études ont montré qu'un système Self très efficace pouvait passer jusqu'à 25 % de son temps [Chambers et Ungar, 1989; Hölzle *et al.*, 1991] à traiter les prédictions incorrectes d'un cache en ligne.

Il est donc apparu nécessaire d'accroître les capacités des caches en ligne. La technique des *caches en ligne polymorphiques* (ou *Polymorphic Inline Caches — PICs*) [Hölzle *et al.*, 1991] consiste à générer

dynamiquement le cache polymorphique approprié à chaque site d'appel. Au lieu de mémoriser seulement le *dernier* type du receveur à chaque site, *tous* les types déjà rencontrés pour ce site sont mis en cache. Dans le cas d'un site monomorphique, le PIC est identique à un cache en ligne classique.

Dans le cas d'un site effectivement polymorphique, le cache est étendu au fur et à mesure de l'exécution par les types des receveurs rencontrés. Ceci permet de gommer le problème des caches en ligne par rapport aux sites polymorphiques, puisque petit à petit tous les types se retrouvent dans le cache et évitent un *lookup*. Cependant, comme le code chargé de la liaison dynamique doit chercher dans un cache dont la taille dépend du nombre de types possibles pour le receveur au site d'appel, il est possible que, dans le cas d'un site mégamorphique, la recherche dans le cache finisse par faire perdre du temps. Fort heureusement, de tels sites restent relativement rares dans les programmes à objets, et certaines méthodes peuvent être utilisées pour traiter les sites mégamorphiques de façon satisfaisante [Hölzle *et al.*, 1991].

Enfin, il faut noter que l'augmentation de la taille du programme due à l'utilisation des PICs est très peu sensible, puisqu'en Self elle représente typiquement moins de 2 % [Hölzle *et al.*, 1991]. Les PICs partagés (*shared PICs*) constituent justement une variante qui permet de diminuer la place prise par les caches en utilisant le même PIC pour plusieurs sites d'appels, cachant pour un nom de message donné tous les types de receveurs déjà rencontrés.

2.6 Une nouvelle technique d'envoi de message

Comme nous l'avons mentionné à plusieurs reprises, la liaison dynamique est un aspect fondamental des langages à objets, et les programmes à objets contiennent donc en général de très nombreux points de liaison dynamique, qu'il est important d'optimiser. Pourtant, malgré un certain nombre de travaux qui se sont intéressés à son implantation efficace, et que nous avons présentés dans la section 2.5 précédente, ce problème reste toujours d'actualité dans les implantations des langages à objets maintenant disponibles.

La présente section détaille donc nos propres travaux sur le sujet[10], à savoir la conception d'un nouvelle technique statique d'implantation de la liaison dynamique, que nous avons appliquée au langage Eiffel et expérimentée avec succès dans le compilateur SmallEiffel.

Dans la suite de cette section, nous présentons tout d'abord notre méthode de liaison dynamique par arbre de branchement binaire (section 2.6.1), puis nous expliquons l'attribution des identifiants de types et son impact sur la recompilation incrémentale (section 2.6.2). Nous détaillons ensuite les très importantes opportunités d'optimisation par expansion en ligne (section 2.6.3) créées par notre méthode, la suppression de point de liaison dynamique supplémentaires (section 2.6.4) et la suppression de l'identifiant de type de certains objets (section 2.6.5). Enfin, nous terminons par une discussion de l'intégration à notre technique statique de liaison dynamique de techniques plus dynamiques (section 2.6.6).

2.6.1 La liaison dynamique par arbre de branchement binaire

Nous avons décidé d'implanter la liaison dynamique selon le même principe que celui que nous avons appliqué pour la génération du reste du code, c'est à dire de façon à prendre avantage de la spécialisation du code permise par notre algorithme de prédiction de type afin de générer un code aussi efficace que possible. En effet, grâce à l'analyse statique du code du système (voir section 2.3), qui calcule l'ensemble des types vivants à l'exécution, le nombre de types possibles pour le receveur de chaque site d'appel est réduit, ce qui, même lorsque l'appel est réellement polymorphique (i.e. ne peut être remplacé par un appel monomorphique), permet certaines optimisations.

La tendance la plus courante actuellement est d'implanter la liaison dynamique à l'aide de méthodes à base de tables, la plupart du temps avec des VFTs (voir sections 2.5.1.2 à 2.5.1.6). Nous avons choisi au contraire de *ne pas* utiliser de table d'indirection pour notre implantation de la liaison dynamique.

Contrairement aux méthodes à base de VFTs, pour lesquelles la structure représentant chaque objet contient un *pointeur vers sa VFT*, ou au moins vers le descripteur de classe correspondant, notre technique de liaison dynamique implique la présence dans cette structure d'un *identifiant de type entier*.

10. Comme indiqué lors de la présentation des techniques existantes d'implantation de la liaison dynamique (section 2.5), nous ne considérons pas ici le cas des multiméthodes, et le polymorphisme traité ne concerne donc que le type dynamique du receveur.

Nous générons donc, pour tout point de liaison dynamique, un code chargé de comparer explicitement l'identifiant de type dynamique stocké dans la structure de l'objet receveur avec les différents types possibles à ce point. Cette suite de tests de type explicites est codée sous forme *dichotomique*, et non pas séquentielle, de façon à en améliorer les performances. Une fois le type du receveur reconnu, la méthode correspondante est appelée *directement*.

A titre d'exemple, supposons que `x.f` est un site polymorphique pour lequel notre analyse globale indique que le receveur a quatre types concrets possibles T_A, T_B, T_C et T_D. Prenons comme hypothèse que les identifiants (IDs) des types T_A, T_B, T_C et T_D sont respectivement 19, 12, 27 et 15. Le (pseudo-) code de sélection pour l'appel `x.f` est donc l'arbre de branchement binaire présenté en figure 2.10 page 26.

```
if id_x ≤ 15 then
     if id_x ≤ 12 then f_B(x)
          else f_D(x)
     endif
   else
     if id_x ≤ 19 then f_A(x)
          else f_C(x)
     endif
endif
```

FIG. 2.10 – *Pseudo-code de liaison dynamique par arbre de branchement binaire.*

On peut constater que ce code présente plusieurs avantages. Tout d'abord, les identifiants de types (attribués selon une méthode présentée en section 2.6.2, page 27) sont codés "en dur", ce qui permet des comparaisons directes très rapides. En effet, ces dernières sont traduites, au niveau assembleur, en des comparaisons avec constante, voir comparaisons avec valeur immédiate si l'identifiant de type est assez petit. Ensuite, une fois le type du receveur reconnu, l'appel à la méthode correspondante est un appel direct, d'une rapidité nécessairement équivalente à — et souvent meilleure que — celle d'un appel indirect, via un pointeur de fonction. Enfin, ce codage explicite et direct des appels des méthodes sélectionnées permet des optimisations supplémentaires très importantes, sous la forme d'*inlinings*[11]. En effet, lorsqu'un appel Eiffel polymorphique est résolu, il ne débouche pas nécessairement sur un appel de *fonction* C. Comme le type dynamique concret du receveur est connu exactement à chaque feuille ($f_A(x)$, $f_B(x)$, $f_C(x)$ et $f_D(x)$ dans l'exemple de la figure 2.10) de l'arbre binaire de tests, le code correspondant peut être expansé en ligne (*inliné*). Nous expliquons en détails ces *inlinings* en section 2.6.3, page 29.

Bien entendu, la taille et donc potentiellement le temps d'exécution du code de recherche de type détaillé en figure 2.10 croît en fonction du nombre de types possibles pour le receveur. Cependant, le coût moyen d'une recherche de type parmi P types possibles (c'est à dire la "profondeur" moyenne d'une recherche) est, en supposant les P types équiprobables, $log_2(P)$ comparaisons. On a donc avec notre méthode un coût qui croît *logarithmiquement*, et qui ainsi reste faible même lorsque le nombre de types possibles est important.

L'exemple de la figure 2.10 montre le code correspondant à notre méthode en mode de compilation optimisé. Lorsque des options de déverminage sont utilisées, le code généré est un `switch` C dont la dernière branche (`default:`) traite le cas où le type du receveur n'est pas possible à ce site d'appel. Dans une telle situation, un message d'erreur (équivalent au *does not understand* de Smalltalk) est émis.

Notons que ce schéma de génération de code de liaison dynamique n'est pas limité à l'héritage simple. Il s'applique de la même façon en présence d'héritage multiple, sans accroissement en complexité du code généré et sans aucun surcoût à l'exécution. Ceci est possible grâce à l'analyse globale (cf. section 2.3.1) du système, qui permet de se placer lors de la génération de code dans un cadre bien délimité et à l'incertitude réduite. Au contraire, les méthodes actuelles à base de VFTs (cf. section 2.5.1.6), ayant été

11. Dans ce mémoire, pour des raisons de commodité d'expression, nous utiliserons indifféremment le vocable *expansion en ligne* ainsi que ses dérivés (comme *expansé en ligne*) et les vocables anglais ou même "franglais" correspondants (*inlining*, *inliné*), qui bien que parfois incorrects, sont néanmoins très souvent utilisés par de nombreux informaticiens.

conçues dans le cadre de la compilation séparée des fichiers sources, paient un surcoût en cas d'héritage multiple [Ellis et Stroustrup, 1990; Driesen, 1999].

L'exemple de la figure 2.10 met aussi en évidence le fait que, contrairement aux méthodes à base de table, où un appel polymorphique est toujours codé par la même indirection, notre méthode génère pour chaque site un code d'aiguillage dont la taille croît en fonction du nombre de types de receveur possibles. Il est donc clair que, dans un programme à objets comportant de nombreux sites d'appel polymorphiques, le programme généré risque de voir sa taille augmenter de façon importante à cause de ce code de liaison dynamique. De plus, la compilation du code C généré risque d'être sensiblement ralentie.

Pour pallier ce problème, au lieu de générer l'arbre de branchement binaire complet à chaque site, nous factorisons ce code de liaison dynamique, en le déportant dans des *fonctions d'aiguillage*. A chaque paire `(sélecteur, ensemble de types possibles calculé statiquement)` correspond une *routine d'aiguillage spécialisée*, qui sert à faire la liaison dynamique pour tous les *sites* d'appels polymorphiques concernant cette même paire `(sélecteur, ensemble de types possibles calculé statiquement)`, ce qui permet de produire un code cible plus compact. Il ne s'agit donc pas d'une fonction la plus générale possible, capable de faire la liaison dynamique pour tous les sites, non optimisée, mais bien de fonctions spécialisées, optimisées pour un sous-ensemble des sites d'appel du programme.

Si l'on reprend l'exemple précédemement détaillé, en supposant que T_B, T_C et T_D sont des descendants de T_A, le pseudo-code d'aiguillage de la figure 2.10 se retrouve dans une fonction `switchAf` ("liaison dynamique pour le sélecteur `f` du type A et ses sous-types"), comme le montre la figure 2.11 ci-dessous:

```
switchAf(A * c) {
   if id_c ≤ 15 then
        if id_c ≤ 12 then f_B(c)
             else f_D(c)
        endif
      else
        if id_c ≤ 19 then f_A(c)
              else f_C(c)
        endif
   endif
}
```

FIG. 2.11 – *Pseudo-code d'une routine spécialisée effectuant la liaison dynamique par arbre de branchement binaire.*

Les sites d'appel polymorphiques `x.f` où les types concrets possibles sont T_A, T_B, T_C et T_D se voient donc codés par un simple appel à la routine d'aiguillage présentée ci-dessus: `switchAf(x);`.

2.6.2 Attribution des identifiants de types et recompilation incrémentale

Comme nous venons de l'expliquer, notre méthode de liaison dynamique est basée sur un code de branchement binaire implantant une série de tests dichotomiques sur l'identifiant de type de l'objet receveur. Cet identifiant entier est attribué à chaque type pendant la compilation, ce qui permet de trier l'ensemble des identifiants des types concrets possibles à chaque site d'appel afin de générer le code binaire approprié. Cette attribution des identifiants à la compilation, comme nous le verrons bientôt, influe sur l'incrémentalité et donc la vitesse de la recompilation d'un système. Nous allons donc présenter brièvement la façon dont le code cible C ANSI est généré et (re)compilé, puis nous détaillerons l'impact de l'attribution des identifiants de types sur ce processus.

2.6.2.1 Recompilation incrémentale

L'analyse de l'ensemble du système est de nature à inspirer certaines réserves en ce qui concerne les temps de compilation. Comme nous allons le montrer dans nos résultats, section 2.7, la traduction d'Eiffel vers C, en partant de zéro, est extrêmement rapide. Au contraire, la compilation du code C généré prend un temps important et doit donc être minimisée autant que possible. Afin d'éviter de devoir recompiler

tous les fichiers générés, nous utilisons un procédé étonnamment simple et efficace, qui consiste en trois étapes.

Tout d'abord, tous les fichiers C et les fichiers objets associés qui ont été produits durant la précédente compilation sont sauvés. Notons que chaque fichier C généré contient une partie de l'ensemble du code C correspondant au système compilé, sans que ce découpage soit en relation avec la hiérarchie des classes Eiffel. Soient $old_1.c$, $old_2.c$, ..., $old_n.c$ les noms des fichiers C sauvegardés, et $old_1.o$, $old_2.o$, ..., $old_n.o$ les noms de ces fichiers binaires correspondants sauvegardés. Ensuite, l'ensemble du code source Eiffel est analysé et les nouveaux fichiers C correspondants sont générés, nommés $new_1.c$, $new_2.c$, ..., $new_n.c$. Enfin, pour chaque paire ($new_i.c$, $old_i.c$), les contenus des fichiers sont comparés octet par octet. Si le fichier C n'a pas changé, l'ancien fichier objet $old_i.o$ est réutilisé, ce qui évite la recompilation C.

La comparaison d'un fichier C avec son ancienne version est une opération relativement peu coûteuse, nettement plus rapide qu'une compilation C du fichier. En effet, le travail effectué comprend, dans le premier cas, deux lectures de fichiers ($old_i.c$ et $new_i.c$) plus la comparaison lexicale, et dans l'autre une lecture de fichier ($new_i.c$), une compilation (avec analyses lexicale, syntaxique, sémantique et génération de code) et une écriture (plus coûteuse qu'une lecture) du nouveau fichier binaire $new_i.o$. Dans le cas où le fichier C est resté le même, le surcoût dû à la comparaison (inutile) est donc faible, tandis que s'il a changé, le temps économisé en faisant une comparaison au lieu d'une compilation est important. Comme d'une recompilation du système à l'autre il est de toute évidence extrêmement fréquent que des fichiers changent, il apparaît rentable de faire systématiquement cette comparaison.

Un avantage de cette technique simple est d'éviter d'avoir à maintenir une base de données sur le projet en cours, contenant des informations qui sont déjà, sous une forme cependant plus implicite, dans le code cible C. Le travail de traduction du compilateur d'Eiffel vers C est donc réduit et simplifié, ce qui contribue à ses performances. Ce système donne de très bons résultats, car la compilation Eiffel, bien que non incrémentale, est extrêmement rapide, alors que la compilation C, beaucoup plus lente, est gérée de façon incrémentale, à un coût très raisonnable.

2.6.2.2 Attribution des identifiants de types

On voit donc que la non recompilation d'un fichier C généré dépend de la préservation de son contenu à l'identique, à l'octet près. Les identifiants de types ont sur ce point une influence très importante. Nous avons donc expérimenté plusieurs méthodes pour le calcul de cet identifiant[12].

La première consiste à les produire séquentiellement, en incrémentant une valeur à chaque fois qu'un type est rencontré. Ceci est la façon la plus simple de faire et présente l'avantage d'être extrêmement peu coûteuse, tout en garantissant bien l'unicité de l'identifiant. Elle a de plus l'avantage de commencer par générer des valeurs d'identifiants petites, qui seront plus facilement codées dans les champs *valeur immédiate* des instructions du microprocesseurs, ce qui permettra de meilleures performances. En effet, les comparaisons pourront être implantées avec une seule instruction assembleur contenant la comparaison et la valeur, alors qu'avec une valeur plus grande, il peut être nécessaire d'avoir une instruction de chargement de la valeur en registre, suivie de l'instruction de comparaison proprement dite.

Cette méthode d'attribution présente cependant un inconvénient important, du point de vue ce l'incrémentalité de la recompilation du système généré. En effet, lorsqu'un système déjà compilé subit une modification, même légère, il est très fréquent que l'ordre des types rencontrés par le compilateur change. Ceci provoque donc un changement d'ordre d'attribution des identifiants de types, et par conséquent des valeurs de ces identifiants (pour tous les types rencontrés "après" la modification). Ainsi, le code cible généré, qui contient ces identifiants "en dur", est modifié à de très nombreux endroits, à chaque fois qu'un identifiant de type est présent, c'est à dire dans des créations d'objets, dans des appels polymorphiques, dans les routines d'un type (dont le nom contient l'identifiant de type), dans les affectations et passages de paramètres (où le type C est *casté*), etc. Ceci impose en général la recompilation d'une très grande partie du système, la plupart des fichiers C générés étant modifiés alors même que le changement dans le source Eiffel a pu être minime.

Une seconde méthode plus robuste de ce point de vue a donc été expérimentée. Elle consiste à produire l'identifiant de type par une technique à base de code de hachage, calculé à partir du nom du type concret

12. Notons que pour des raisons pratiques, la valeur de l'identifiant de type de quelques types de base, comme les entiers, booléens, caractères, chaînes de caractères, est prédéfinie "en dur".

complet (c'est à dire incluant les paramètres dans le cas de types génériques, ou types paramétriques). En cas de collision sur ce code de hachage, l'identifiant est incrémenté de 1, jusqu'à ce qu'il corresponde à un identifiant non encore affecté, de façon à garantir l'unicité de chaque identifiant de type. Le coût de cette méthode est donc plus important que pour la précédente, mais il reste néanmoins raisonnable. Le gros avantage de cette méthode est que l'attribution des identifiants ne dépend plus de l'ordre dans lequel le compilateur rencontre les noms de types, sauf lorsqu'il y a des collisions de code de hachage. Ainsi, lorsqu'un changement de code minime est fait sur un système déjà compilé, aucun identifiant de type n'est affecté, sauf ceux concernés par des collisions de code de hachage. La quantité de code recompilé dans ce cas est donc nettement plus faible qu'avec la première méthode d'attribution des identifiants, mais une partie du code peut encore être recompilée inutilement.

La méthode de génération des identifiants de types qui donne les meilleurs résultats, et qui est celle actuellement présente dans le compilateur SmallEiffel, est en fait une variation de la première, ajoutant à la génération d'identifiants par incrémentation la sauvegarde de ceux-ci dans un fichier. A chaque recompilation, sauf lors de la première, le fichier des identifiants précédemment attribués est donc relu, ce qui permet de réaffecter les *mêmes* identifiants à *tous* les types déjà rencontrés. Ainsi, les fichiers de code cible C déjà générés pour des types qui n'ont pas été modifiés sont préservés et n'ont pas besoin d'être recompilés. Les types nouvellement apparus depuis la dernière compilation voient leurs identifiants attribués en partant du plus grand identifiant déjà attribué relu depuis le fichier des identifiants de types, et en incrémentant cette valeur de 1 à chaque nouveau type. Bien qu'elle impose la lecture et l'écriture d'un fichier contenant les noms des types et leurs identifiants à chaque compilation, cette méthode est la plus efficace puisqu'elle évite tous les changement d'identifiants déjà attribués, qui, notamment sur de gros systèmes, imposent des temps de recompilation importants. Elle conserve également l'avantage de la génération d'identifiants de types correspondant à de petites valeurs qui permettent des comparaisons par valeur immédiate, rapides car potentiellement basées sur une seule instruction assembleur. De plus, elle présente l'avantage d'être elle aussi très simple à mettre en oeuvre. On peut enfin remarquer que cette méthode d'attribution des identifiants est issue d'un raisonnement que nous avons appliqué à la plupart des techniques que nous avons développées, et qui consiste à éviter de faire deux fois le même travail.

Notons qu'actuellement, le système d'attribution des identifiants de types est indépendant du système de génération des routines d'aiguillage: les identifiants sont attribués aux types sans tenir compte des sites d'appel dans lesquels ils apparaissent. Le tri nécessaire à la génération du code de branchement binaire est fait a posteriori, lors de la production de chaque fonction d'aiguillage. Enfin, remarquons également que la génération statique de ces identifiants de types, et donc de notre code de liaison dynamique par arbre de branchement binaire, n'est possible que parce que notre algorithme de compilation explore la totalité du système à compiler.

2.6.3 Les expansions en ligne induites

Grâce à notre algorithme de prédiction de type, qui bien qu'assez simple offre des scores de prédiction élevés (voir section 2.7), ainsi que grâce à notre technique nouvelle d'implantation de la liaison dynamique, notre méthode de compilation fait apparaître de nombreux sites d'expansion en ligne (*inlining*) potentiels aux feuilles de l'arbre de branchement binaire. Il est important de se rappeler que les méthodes classiques à base de VFTs (tableaux de pointeurs) sont au contraire un obstacle à l'expansion en ligne, car le compilateur ne peut prédire quelle fonction sera appelée. Les expansions en ligne supplémentaires permises par notre méthode en font donc partie intégrale et jouent un rôle significatif dans l'optimisation globale du code généré. De plus, en permettant, comme nous le verrons, de supprimer ou au moins de très fortement diminuer le coût de la liaison dynamique et le coût dû à l'encapsulation, ces optimisations favorisent l'utilisation de ces techniques. Ainsi, se focalisant sur des aspect directement liés aux langages à objets, ce type d'optimisations est réellement utile dans ce cadre et facilite une bonne conception objet.

Nous avons cherché à identifier et à implanter les divers *schémas* selon lesquels l'expansion en ligne pouvait être réalisée. La plupart des schémas d'expansion en ligne que nous avons catégorisés et implantés dans SmallEiffel sont présentés dans cette section. L'impact de ces différents types d'expansion en ligne sera évalué dans en section 2.7, dont les figures reprendront les noms et abréviations que nous avons choisis pour nommer chacun de ces schémas.

ARI L'Expansion en ligne sur Lecture d'Attribut (ou *Attribute Reader Inlining*) est réalisée lorsque la fonction consiste en une simple lecture de la valeur d'un attribut, c'est à dire lorsqu'elle ne contient qu'une instruction qui est un accès à un attribut. Tous les sites d'appel d'une telle fonction sont expansés en ligne et la fonction elle-même n'a plus besoin d'être définie. Ainsi, considérons le code suivant:

```
class CLIENT
...
     do_it is
        local
           an_integer: INTEGER
           a_supplier: SUPPLIER
        do
            ...
            an_integer := a_supplier.get_integer_attribute;   -- (1)
            ...
        end
end -- class CLIENT

class SUPPLIER

feature

   integer_attribute: INTEGER;

   get_integer_attribute: INTEGER is
      do
         Result := integer_attribute;
      end

end -- class SUPPLIER
```

Dans ce système, le type dynamique de `a_supplier` ne peut être que SUPPLIER. L'instruction `(1)` est donc un appel monomorphique, qui représente un cas d'ARI simple et est traduite en le code C suivant:

```
_an_integer = ((SUPPLIER*)_a_supplier)->_integer_attribute;   /* (1) */
```

Ceci montre que les sites monomorphiques sont directement expansés en ligne, alors que dans le cas des sites d'appels polymorphiques, les expansions en ligne sont faites dans la branche correspondant à l'identifiant de type, dans la routine d'aiguillage de l'appel polymorphique. Considérons le code suivant, dans lequel on suppose que SUPPLIER_A et SUPPLIER_B sont deux types vivants:

```
class CLIENT
...
     do_it is
        local
           an_integer: INTEGER
           a_supplier: POLY_SUPPLIER
        do
            ...
            an_integer := a_supplier.get_integer_attribute; -- (2)
            ...
        end
end -- class CLIENT

deferred class POLY_SUPPLIER
feature
   get_integer_attribute: INTEGER is
      deferred
      end
```

```
end -- class POLY_SUPPLIER

class SUPPLIER_A inherit POLY_SUPPLIER
feature
...
   integer_attribute_a: INTEGER;
   get_integer_attribute: INTEGER is
      do
         Result := integer_attribute_a;
      end
end -- class SUPPLIER_A

class SUPPLIER_B inherit POLY_SUPPLIER
feature
...
   integer_attribute_b: INTEGER;
   get_integer_attribute: INTEGER is
      do
         Result := integer_attribute_b;
      end
end -- class SUPPLIER_B
```

Dans ce système, l'instruction (2) constitue un cas d'ARI polymorphique, qui est traduit par:

```
_an_integer = switchPOLY_SUPPLIERget_integer_attribute(_a_supplier);   /* (2) */
```

avec la routine d'aiguillage suivante:

```
int switchPOLY_SUPPLIERget_integer_attribute(POLY_SUPPLIER* C) {
    int id = C->id;
    if (id <= ID_OF_SUPPLIER_A)
       return = ((SUPPLIER_A*)C)->_integer_attribute_a;
    else
       return = ((SUPPLIER_B*)C)->_integer_attribute_b;
}
```

Ainsi, l'utilisation de fonctions de lecture d'attribut dans le code source Eiffel n'implique aucun surcoût à l'exécution du code généré. Ceci est un point capital, puisque l'accès aux valeurs des attributs d'un objet est une opération extrêmement fréquente dans les programmes à objets.

Dans un code source Eiffel, du point de vue syntaxique, l'accès à un attribut se fait en général directement, via son nom, sans routine d'accès[13]. Ainsi, on ne peut en Eiffel différencier un accès à un attribut d'un appel à une fonction sans paramètre (principe d'accès uniforme). Ces accès aux attributs "syntaxiquement directs" dans le code Eiffel sont bien entendu implantés de façon directe dans le code généré (ils peuvent en fait être considérés comme des cas d'ARI triviaux).

Par contre, le nombre et l'utilité des ARI non triviaux en Eiffel semblent plus limités. Cependant, suite à l'héritage, il n'est pas rare de voir un corps de routine d'un ancêtre redéfini (ou défini, si le corps était vide) sous la forme d'un simple accès à un attribut dans certains descendants. Les ARI prennent alors tout leur intérêt.

En revanche, dans le cadre d'autres langages, comme Java ou C++, la distinction entre appel de fonction sans paramètre et accès à un attribut est faite à un niveau syntaxique (présence de parenthèses vides ou non). De plus, l'accès à un attribut exporté peut s'y faire non seulement en lecture mais aussi en écriture[14]. Il devient donc indispensable, pour préserver dans ce type de langages l'encapsulation des objets, de définir dans le code source des fonctions d'accès aux attributs. Les ARI apparaissent alors comme capitaux pour assurer des performances décentes sur ces opérations très fréquentes, en évitant

13. On dit parfois, de façon relativement "parlante" mais néanmoins incorrecte, que l'attribut et la routine d'accès ont le même nom.

14. Sans vouloir débattre des mérites respectifs des différents langages, notons toutefois que ce choix peut être considéré comme peu judicieux du point de vue conceptuel.

la génération de ces nombreuses fonctions d'accès, remplacées dans le code généré par des accès directs beaucoup plus rapides et compacts.

AWI L'Expansion en ligne sur Ecriture d'Attribut (ou *Attribute Writer Inlining*) est le pendant de l'ARI pour l'écriture d'attributs. Elle est faite lorsqu'une procédure est définie pour simplement affecter à un attribut une valeur passée en paramètre, c'est à dire lorsqu'une procédure n'a qu'un seul argument et comprend une seule instruction où l'argument est affecté à l'attribut. L'AWI est effectué aussi bien sur une procédure "locale" (dont le receveur, explicite ou non, est `Current`) qu'une procédure appliquée à un autre objet. Ainsi, le code Eiffel suivant:

```
class AWI_CLIENT
   ...
   do_something is
      do
         set_integer_attribute(12);               -- (1)
         awi_supplier.set_supplier_attribute(34); -- (2)
      end
   set_integer_attribute (ia: INTEGER) is
      do
         integer_attribute := ia;
      end
   integer_attribute: INTEGER
end -- class AWI_CLIENT


class AWI_SUPPLIER
   ...
   set_supplier_attribute (sa: INTEGER) is
      do
        supplier_attribute := sa;
      end;
   supplier_attribute: INTEGER
end -- class AWI_SUPPLIER
```

sera traduit par le code C qui suit:

```
   (((AWI_CLIENT*)C)->_integer_attribute) = 12;                  /* (1) */
   (((AWI_SUPPLIER*)_awi_supplier)->_supplier_attribute) = 34;   /* (2) */
```

Bien entendu, dans le cas où la procédure d'affectation à un attribut est sujette à liaison dynamique, ce même type d'*inlinings* est fait au niveau des branches de la routine d'aiguillage.

Comme dans le cas des ARI, tous les sites d'AWI sont donc expansés en ligne. Les procédures d'écriture d'attribut présentes dans le code source ne causent donc aucun surcoût, puisqu'elles ne sont même pas définies dans le code généré, étant remplacés par des écritures directes. Ceci est bien entendu capital pour les performances et la compacité du code, puisque l'encapsulation favorise grandement la prolifération et l'utilisation de telles routines dans les langages à objets.

DRI L'Expansion en ligne sur Relais Direct (ou *Direct Relay Inlining*) s'applique lorsque le corps de la routine considérée a seulement une instruction qui fait le "relais" en appliquant une autre méthode sur le même receveur. Un tel cas se produit notamment lorsque l'on souhaite faire un appel relais vers une routine de la même classe (ce qui est une façon de définir des routines synonymes, ou alias). Par exemple, supposons que l'on ait dans une classe COLLECTION[X] la définition Eiffel suivante:

```
   push(element: X) is
      do
         Current.add_first(element); -- ou simplement: "add_first(element);"
      end;
```

Comme `add_first` est appelée avec le même receveur que `push`, à savoir l'objet courant, l'appel à la procédure `push` est expansé en ligne.

Ainsi, tous les sites d'appels représentant des relais directs sont expansés et ce type de relais ne cause là non plus aucun surcoût au niveau du code généré, la définition d'une routine supplémentaire (le synonyme) étant même évitée.

Le DRI est également applicable quand les arguments de l'appel relayé sont calculables statiquement. Par exemple, cette fonction de la classe STRING de la bibliothèque standard de SmallEiffel est elle aussi expansée selon le schéma DRI:

```
first: CHARACTER is
   do
      Result := Current.item(1);
   end;
```

L'appel à `first` a donc exactement le même coût qu'un appel direct à `item(1)`. Ceci permet donc d'améliorer l'encapsulation et la lisibilité du code, capitales pour la maintenance, sans aucun coût à l'exécution.

DARI L'Expansion en ligne sur Relais Direct vers Attribut (ou *Direct Attribute Relay Inlining*) utilise le fait que quand le type concret de l'objet courant est connu, il est possible que les types concrets de ses attributs le soient également.

Un cas de DARI est détecté lorsque les trois conditions suivantes sont vraies:

- Tout d'abord, la routine doit avoir une seule instruction qui est un appel.
- Ensuite, le receveur de cet appel est un attribut d'instances de la classe contenant cette routine.
- Enfin, cet appel est monomorphique, le type du receveur étant unique.

Par exemple, la fonction `item` de la classe STRING, qui permet d'accéder à un caractère de la chaîne par son indice, est définie de la façon suivante dans la bibliothèque standard de SmallEiffel:

```
item(index: INTEGER): CHARACTER is
   do
      Result := storage.item(index - 1);
   end;
```

L'appel à `item` est monomorphique car l'attribut `storage` n'a qu'un seul type concret possible: il s'agit toujours d'un tableau de caractères de bas niveau (NATIVE_ARRAY[CHARACTER]). Un appel à `item` tel que celui-ci:

```
some_character := local_string.item(123);
```

est donc toujours expansé et traduit de la façon suivante:

```
_some_character = (((STRING*)(_local_string))->_storage)[123-1];
```

On voit ici que bien que correspondant à une situation très spécifique, le DARI n'en est pas moins très important, car il se produit sur des routines comme `item` de STRING et ARRAY, qui sont extrêmement souvent appelées.

PRI L'Expansion en ligne sur Résultat Précalculable (ou *Predictable Result Inlining*) est utilisée dans le cas où le résultat d'une fonction a une valeur connue lors de la compilation. Ce résultat peut être n'importe quelle expression statiquement calculable, y compris les appels de fonctions imbriqués. Ainsi, dans l'exemple suivant:

```
two: INTEGER is
   do
      Result := 1 + 1;
   end;
three: INTEGER is
   do
```

```
      Result := 1 + Current.two; -- ou simplement two
   end;

compute is
   local
      some_integer: INTEGER
   do
      some_integer := three; -- (1)
      ...
   end
```

les fonctions `two` et `three` sont toutes deux expansées lors de l'appel de l'instruction `(1)`, ce qui produit le code C suivant:

```
_some_integer = 1+(1+1);  /* (1) */
```

Ce code peut ensuite très facilement être optimisé par calcul statique et peut lui-même faire partie d'un PRI. Encore une fois, les routines `two` et `three` étant ainsi expansées, elles ne sont même pas définies dans le code cible généré.

Cet exemple est trivial, mais il est important de se rappeler que chaque routine vivante est spécialisée en fonction du type vivant concret (voir section 2.4.2 et [Collin *et al.*, 1997; Zendra *et al.*, 1997]). Ainsi, l'appel à `two` — qui est ici calculable statiquement — peut parfaitement être redéfini dans un descendant. Cette nouvelle définition sera considérée séparément et peut elle aussi être calculable statiquement.

RCI L'Expansion en ligne sur Résultat `Current` (ou *Result is Current Inlining*) est réalisée lorsqu'un corps de fonction n'a qu'une instruction qui retourne `Current`. La fonction suivante est donc expansée et remplacée par un simple accès à `Current`:

```
foo(bar: Y): X is
   do
      Result := Current;
   end;
```

Comme nous le montrerons dans nos résultats (section 2.7), ce type de fonction n'est pas très fréquent, mais il existe.

EPI L'Expansion de Procédure Vide (ou *Empty Procedure Inlining*) est faite quand une procédure ne contient aucune instruction, et peut donc être tout simplement supprimée. Par exemple, le `do_nothing` du standard Eiffel est un cas d'EPI. De façon surprenante, nous avons constaté que de tels cas, bien qu'assez peu nombreux, étaient plus fréquents que l'on pouvait s'y attendre. Ceci s'explique par le fait qu'il arrive qu'une routine fournie dans un ancêtre soit redéfinie de différentes façons dans les différents descendants et que pour certains d'entre eux, n'ayant plus vraiment lieu d'être, elle soit redéfinie avec un corps vide.

OEI Des Expansions Spécifiques à Eiffel (ou *Other Eiffel Inlinings*) sont également implantées par le compilateur SmallEiffel. Nous ne décrirons pas ces expansions ici car elles sont étroitement liées au langage Eiffel (par ex., l'expansion des fonctions `once` précalculées). Contrairement aux expansions précédentes, l'OEI n'est donc pas directement transposable à C++ ni à Java. Le lecteur intéressé pourra trouver des détails supplémentaires dans [Colnet et Zendra, 1999].

2.6.4 Suppression de points de liaison dynamique supplémentaires

La suppression de points de liaison dynamique supplémentaires est encore possible même après la fin de l'algorithme de prédiction de type. En effet, quand toutes les branches d'un arbre binaire codant un point d'envoi de message polymorphique ont exactement le même comportement — i.e. le code produit dans chaque branche a exactement le même effet, même s'il n'est pas identique — aucune liaison dynamique

n'est nécessaire, et l'appel polymorphique se réduit à un appel monomorphique. Là encore, il s'agit de faciliter la tâche du développeur en lui assurant qu'une bonne programmation à objets sera efficacement compilée.

Voici la classification que nous faisons des cas où des sites supplémentaires de liaison dynamique ont pu être supprimés.

ARR La Suppression sur Lecture d'Attribut (ou *Attribute Read Removal*) s'effectue lorsque deux conditions sont réunies. Premièrement, toute branche du code binaire d'aiguillage est un cas d'ARI, c'est à dire constitue un simple accès en lecture d'un attribut. Deuxièmement, tous les attributs ont le même décalage par rapport au début de la structure codant l'objet, pour tous les types concrets possibles pour le receveur. Même quand tous les attributs ont le même nom, il est important de vérifier cette seconde condition car, à cause de l'héritage multiple, un type concret peut hériter de différents attributs. Pour les langages à héritage simple, cette condition peut être omise.

AWR La Suppression sur Écriture d'Attribut (ou *Attribute Write Removal*) est le cas semblable au précédent, mais pour l'écriture d'un attribut. Elle se fait également à deux conditions. Tout d'abord, que pour chaque type concret possible du receveur, la branche correspondante du code binaire d'aiguillage soit un AWI, c'est à dire un simple accès en écriture d'un attribut. Ensuite, que tous les attributs aient le même décalage par rapport au début de l'objet. Comme dans le schéma de suppression précédent, cette dernière condition n'est pas nécessaire dans le cas où le langage n'a pas d'héritage multiple.

DARR La Suppression sur Relais Direct sur Attribut (ou *Direct Attribute Relay Removal*) est faite lorsque toutes les branches du code binaire d'aiguillage sont des relais (DARI) vers la même routine.

OER Des Suppressions Spécifiques à Eiffel (ou *Other Eiffel Removals*) permettent de diminuer encore le nombre de sites d'appels polymorphiques. Elles ne seront pas décrites ici car elles sont étroitement liées à l'interface de SmallEiffel vers les tableaux de bas niveau. Comme les OEI, les OER peuvent ne pas être applicables à d'autres langages comme C++ ou Java.

2.6.5 Suppression de l'identifiant de type de certains objets

Comme nous l'avons expliqué, notre méthode de liaison dynamique par arbre de branchement binaire implique la présence d'un champ identifiant de type dans les structures représentant les objets. Cependant, il arrive que des objets ne soient pas sujets à liaison dynamique, et donc que ce champ identifiant de type ne soit absolument pas nécessaire.

En effet, grâce à l'analyse statique globale du système et à l'algorithme de prédiction de type, de nombreux sites d'appels polymorphiques sont remplacés par des appels monomorphiques. Il arrive donc que, pour un type donné, seuls des appels monomorphiques soient présents dans le code vivant, et que les instances de ce type ne soient donc jamais sujettes à liaison dynamique.

A titre d'exemple, on peut citer le type STRING, qui, dans le code source du compilateur SmallEiffel lui-même illustre bien ce genre d'objets sans champ identifiant de type. En effet, dans le code vivant du compilateur, d'une part la classe STRING est une classe feuille (sans sous-classe) et d'autre part, toujours dans le code vivant du compilateur, une expression de type STRING est toujours affectée (directement ou lorsqu'elle est passée en argument) vers ce même type STRING. Ainsi, n'importe quelle expression de type STRING correspond effectivement et assurément à un objet instance de cette classe[15].

De même, nous avons mentionné en section 2.4.2 l'existence en Eiffel d'une catégorie d'objets qui, de par leur nature même, ne sont eux non plus jamais sujets à liaison dynamique: les objets `expanded`, qui sont toujours passés par valeur (INTEGER, etc.).

Ainsi, dans ces deux cas, l'identifiant de type n'est aucunement utilisé lors de la liaison dynamique. Comme le code produit par SmallEiffel est spécialisé, il est possible dans ces cas de ne générer que le strict minimum nécessaire, à savoir une structure d'objets ne comportant pas le champ identifiant de type. Ceci permet un gain en mémoire équivalent à un mot machine (un entier) par rapport aux objets

15. A l'exception du cas particulier `Void`.

normaux. Notons que ceci n'est possible que parce que notre compilateur génère du code et des structures de données spécialisés basés sur une analyse globale du système. Contrairement aux apparences, ce gain n'est globalement pas négligeable, puisque qu'il peut s'appliquer à des objets extrêmement utilisés comme les chaînes de caractères, etc...

Remarquons également que les types de base, comme les entiers, flottants, caractères, etc., de par leur nature même, ne sont pas implantés à l'aide d'une structure mais sont traduits directement en des types de base du langage cible (`int`, `float`, `char`, etc.), et n'ont donc pas non plus de champ identifiant de type associé.

Cette optimisation permet donc de garantir au développeur que le surcoût en terme de mémoire dû à l'utilisation de la liaison dynamique sera minimal.

2.6.6 Apports possibles des techniques dynamiques

Bien que notre méthode de liaison dynamique soit totalement statique, c'est à dire voit ses données et code entièrement calculés lors de la compilation, elle peut bénéficier d'apports des techniques dynamiques, pour améliorer encore ses performances. Par exemple, les méthodes à base de caches en ligne sont tout à fait "compatibles" avec notre technique. Il est en effet possible d'ajouter, juste avant notre code de liaison dynamique par branchement binaire, du code supplémentaire gérant un cache permettant de retrouver immédiatement le dernier type de receveur observé à un certain site d'appel. Les caches en ligne sont donc efficaces sur des sites où le receveur ne varie pas, ou très peu. Notre méthode l'est également, notamment si le site est faiblement polymorphique. Dans le cas d'un site mégamorphique, l'intérêt du cache semble augmenter, puisqu'il évitera plusieurs aiguillages dans notre code de branchement binaire.

Il faut cependant garder à l'esprit que ces aiguillages peuvent être prédits avec exactitude (quand le type du receveur ne varie pas) par les systèmes de prédiction de branchements directs des microprocesseurs (*Branch History Table — BHT*, voir [Hennessy et Patterson, 1996]). De même, les branchement indirects sont eux aussi prédictibles, grâce aux prédicteurs sur branchements indirects (*Branch Target Buffers — BTB*). On voit donc que, dans une certaine mesure, il est difficile d'estimer le comportement d'une technique de liaison dynamique sans tenir compte de l'architecture de la machine sur laquelle s'exécutent les programmes. Comme chaque architecture a des performances différentes sur les prédictions de branchement direct et les prédictions de branchement indirect, il n'est pas certain que l'adjonction de caches en ligne à notre méthode puisse l'améliorer.

Comme les caches en ligne dont ils sont issus, les caches en ligne polymorphiques (PICs) pourraient être couplés à notre méthode de liaison dynamique, servant de "première étape" de notre algorithme. Cependant la version classique des PICs consiste en une recherche *séquentielle* sur les types dynamiques rencontrés. Comme notre méthode consiste en un arbre binaire de recherche, il est fort possible que les PICs soient en fait plus lents que notre méthode, notamment pour les sites mégamorphiques où le type du receveur est très variable. Ici encore, le comportement réel dépend en partie de l'architecture de la machine et notamment de l'efficacité des prédicteurs de branchement.

On voit donc que des études descendant plus en profondeur dans les détails des microprocesseurs seraient nécessaires, et ce sur une large gamme d'architectures. De façon plus générale, nous considérons qu'il serait indispensable de disposer d'un moyen de spécifier et de simuler des exécutions en fonctions de différentes architectures.

2.7 Résultats expérimentaux

Tout au long de notre travail, nous avons bien entendu cherché à évaluer au mieux les techniques que nous proposons et leur impact. Nous avons donc implanté ces techniques dans SmallEiffel, notre compilateur Eiffel, afin de pouvoir les tester en situation réelle et afin de pouvoir les diffuser au sein de la communauté informatique mondiale. Nous avons été particulièrement sensibles aux effets des techniques que nous avons étudiées sur les performances, qu'il s'agisse des temps d'exécution ou de la mémoire consommée, tant lors de la compilation d'un programme que lors de l'exécution d'un programme généré avec SmallEiffel.

		SmallEiffel	Eiffel/S
(1)	Taille en modes `-boost` et `-O`	622 KO	884 KO
(2)	Taille en mode `-no_check`	2.3 MO	3.3 MO
(3)	Taille en mode `-require_check`	2.7 MO	3.3 MO

FIG. 2.12 – *Taille des exécutables générés: SmallEiffel face à Eiffel/S.*

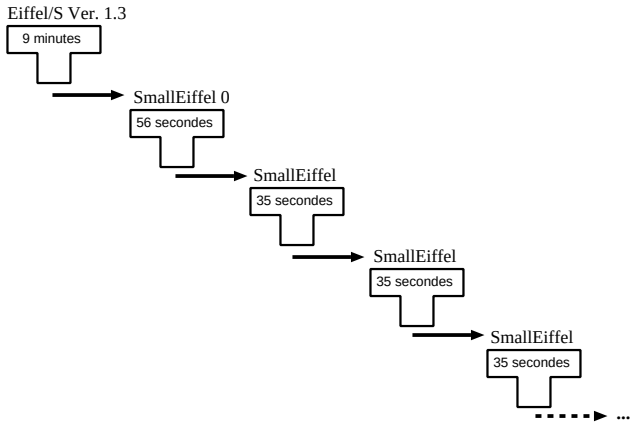


FIG. 2.13 – *Le processus de bootstrap: SmallEiffel face à Eiffel/S.*

Nous présentons dans cette section un certain nombre des résultats que nous avons obtenus grâce à la méthode de compilation que nous avons détaillée précédemment.

2.7.1 SmallEiffel: l'auto-compilation

Nous nous intéressons tout d'abord à l'observation des performances globales de SmallEiffel.

SmallEiffel nous a non seulement servi d'outil dans lequel nous avons pu implanter et expérimenter nos idées, il a aussi constitué un sujet d'observation extrêmement intéressant. En effet, SmallEiffel est complètement écrit en Eiffel, mis à part quelques éléments de base très localisés, et est donc auto-compilé (*bootstrapped*). Il représentait dans sa version `-0.86`, qui nous a servi pour faire la plupart de nos mesures[16], environ 50000 lignes d'Eiffel pour environ 300 classes. Il est donc clair que ce programme constitue un banc d'essai tout à fait significatif, que nous avons pu utiliser pour expérimenter et évaluer nos techniques, comme nous le rapportons dans la suite de cette section.

La figure 2.12 compare la taille des codes générés en compilant SmallEiffel[17] avec lui-même (*bootstrap*) et en le compilant avec le compilateur Eiffel/S[18].

La figure 2.13 quant à elle détaille le processus de *bootstrap* qui était effectué au début du développement de SmallEiffel, lorsque le compilateur Eiffel/S était encore utilisé comme amorce initiale (actuellement, c'est bien entendu SmallEiffel qui sert d'amorce primaire). Les temps indiqués sont les temps de génération du code ANSI C à partir du source Eiffel.

De façon à permettre une comparaison sensée, l'ensemble des résultats montrés dans ces deux figures a été obtenu à partir de tests effectués sur la même machine (HP 9000/887), avec le même compilateur C et les mêmes options de compilation (`cc -O`).

La taille du code optimisé donné en ligne (1) de la figure 2.12 correspond au mode `-boost` de SmallEiffel (optimisation maximale) et à l'option `-O` du compilateur Eiffel/S. Les comparaisons de taille des lignes (2)

16. Les versions de SmallEiffel sont numérotées à partir de -1 et tendent vers 0. La première version publique de SmallEiffel portait le numéro `-0.99`; la version `-0.80` est donc la vingtième.

17. Version -0.99, la première mise à disposition des utilisateurs.

18. Version 1.3, de SiG Computer GmbH, maintenant Object Tools.

(avec test de receveur non `Void`) et (3) (avec préconditions) sont données uniquement à titre informatif puisque les comparaisons ne sont vraiment significatives que sans le code servant à tracer les erreurs à l'exécution. Ce code est différent d'un compilateur à l'autre et varie avec le niveau de détail de la trace. Nous pouvons cependant remarquer que l'accroissement de taille du code est constant avec Eiffel/S, alors que la taille du code généré par SmallEiffel croît avec le niveau d'assertions choisi. Ceci est bien entendu dû au fait que SmallEiffel produit *seulement* le code qui est strictement nécessaire au mode de compilation spécifié.

Afin d'évaluer le gain directement issu de notre technique de liaison dynamique il est utile d'examiner plus en détail le processus d'auto-compilation (*bootstrap*). La figure 2.13 montre comment SmallEiffel est produit, par une succession de compilations de la même base de code (le code source de SmallEiffel). Le compilateur produit par chaque étape n de la compilation est utilisé pour compiler le code source à l'étape suivante $n+1$ du processus d'auto-compilation, jusqu'à stabilisation par production d'un même exécutable à l'étape n et à l'étape $n+1$, ce qui doit normalement arriver dès $n = 2$ [Aho et Ullman, 1977]. Dans le cas où ceci n'intervient pas dès l'étape 2, on sait que le compilateur contient des erreurs qui le rendent instable.

La première compilation (9 minutes) correspond à une exécution du compilateur Eiffel/S sur le code de SmallEiffel, et produit un compilateur `SmallEiffel0`. `SmallEiffel0` contient donc les algorithmes de SmallEiffel, avec l'implantation (code généré) d'Eiffel/S. Ainsi, par exemple, la liaison dynamique y est faite par l'algorithme d'Eiffel/S (à base de VFTs). La seconde compilation du code de SmallEiffel (56 secondes), utilisant le compilateur `SmallEiffel0`, produit un compilateur `SmallEiffel`. Ce dernier est donc bien un compilateur SmallEiffel "pur", puisqu'il contient les algorithmes du code de SmallEiffel, avec l'implantation de SmallEiffel (présente dans `SmallEiffel0`). Ainsi, la liaison dynamique y est faite par notre technique à base d'arbre de branchement binaire (cf. section 2.6). La troisième compilation (35 secondes), sert à vérifier la stabilisation du processus de compilation comme expliqué ci-dessus, et génère elle aussi (sauf erreur) un même exécutable `SmallEiffel`.

La comparaison des performances de `SmallEiffel` avec celles de `SmallEiffel0` (35 secondes au lieu de 56) montre l'accroissement de vitesse dû à nos choix d'implantation (des structures des objets, de la liaison dynamique, de la gestion automatique de la mémoire, etc.). Notre implantation est $56/35 = 60\,\%$ plus rapide que celle d'Eiffel/S sur ce banc d'essai de 50000 lignes.

La comparaison des performances de `SmallEiffel0` (nos algorithmes avec l'implantation d'Eiffel/S) avec Eiffel/S (algorithmes et implantation Eiffel/S) permet d'avoir une idée de l'efficacité relative de nos algorithmes de compilation par rapport à ceux d'Eiffel/S (56 secondes au lieu de 540, soit plus de 8 fois plus rapides).

Enfin, la comparaison de la vitesse de `SmallEiffel` avec celle d'Eiffel/S (35 secondes au lieu de $9 \times 60 = 540$) montre l'efficacité de l'ensemble du processus de compilation implanté dans SmallEiffel (plus de 14 fois plus rapide).

2.7.2 Le programme SmallEiffel en détails

Après une analyse globale des performances de SmallEiffel, nous nous sommes attachés à les examiner plus en détail, en accordant une attention toute particulière à celles de la liaison dynamique.

Le compilateur SmallEiffel a été développé en utilisant au maximum les concepts objets et les possibilités qu'ils offrent. On pourrait ainsi dire que SmallEiffel est vraiment "orienté objets". L'ensemble de son code source fait donc un usage intensif de la liaison dynamique, notamment la partie chargée d'analyser un code source Eiffel à compiler (analyseur syntaxique). Un bon exemple de cet usage important de la liaison dynamique est la façon dont sont manipulées les différentes expressions rencontrées dans le code source d'un programme Eiffel compilé par SmallEiffel. En effet, il n'existe pas moins de 48 sous-types de la classe abstraite EXPRESSION concrets et *vivants*, qui représentent les différents types d'expressions rencontrées dans un source Eiffel. Bien entendu, c'est grâce à la que le comportement approprié à chaque (type d')expression peut être sélectionné dans le compilateur.

Le tableau suivant donne quelques exemples du degré d' "orientation objet" (*"Object-Orientedness"*

[Dean *et al.*, 1996]) de SmallEiffel:

Nombre de sous-types concrets	Classe abstraite
48	`EXPRESSION`
27	`CALL`
20	`TYPE`
17	`FEATURE`
16	`INSTRUCTION`
12	`NAME`
8	`ROUTINE`
8	`CONSTANT_ATTRIBUTE`
...	.....

Pour bien saisir l'importance de la liaison dynamique dans SmallEiffel, il faut également garder à l'esprit qu'Eiffel est un "vrai" langage à objets [Meyer, 1988], c'est à dire que chaque routine peut être appelée polymorphiquement (est *virtual*, en utilisant la terminologie C++). En effet, toutes les méthodes Eiffel sont des méthodes d'instance, contrairement à Java ou C++, par exemple, où existent des méthodes de classes.

Après avoir montré dans la section 2.7.1 certains résultats initiaux reliés au processus d'auto-compilation de SmallEiffel (*bootstrap*), nous allons maintenant présenter des résultats plus détaillés, basés sur de nouveaux tests de performances et de nouvelles analyses[19].

2.7.2.1 SmallEiffel sur différentes architectures

Comme SmallEiffel analyse l'ensemble du système pour produire son code cible, on peut être préoccupé par le temps de compilation. Après avoir expliqué en section 2.6.2 page 27 notre processus de recompilation, nous présentons ici quelques résultats qui montrent que ces craintes n'ont pas raison d'être.

La figure 2.14 page 40 montre le comportement du processus d'auto-compilation sur différentes architectures. Durant ce processus, SmallEiffel compile son propre code source avec toutes les optimisations précédemment indiquées activées. Le temps total de compilation est divisé en deux parties. La partie supérieure montre le temps nécessaire pour produire le code C à partir du source Eiffel, en démarrant de zéro, et en incluant le temps de comparaison entre les anciens et les nouveaux fichiers C. La partie intermédiaire indique le temps pour produire l'exécutable à partir du code C précédemment généré, en utilisant le compilateur `gcc` avec l'option de compilation `-O6`.

Ces graphiques montrent que la compilation par SmallEiffel de l'Eiffel en C est extrêmement rapide: il faut par exemple moins de 10 secondes[20] pour traduire 50000 lignes d'Eiffel en 65000 lignes de code C optimisé. A titre indicatif, cette première étape, qui comprend toutes les techniques d'optimisation précédemment décrites, est environ 10 à 20 fois plus rapide que la traduction du code C en un exécutable...

Le temps total de compilation à partir de zéro jusqu'à l'exécutable est de seulement 2 minutes environ, ce qui est très raisonnable. De plus, en cas de recompilation incrémentale (pas à partir de rien, donc), seuls certains fichiers C doivent être recompilés, comme nous l'avons expliqué. Ainsi, à titre indicatif, avec un changement mineur dans un programme de 50000 lignes d'Eiffel, il suffit d'environ 15 secondes pour reconstruire un nouvel exécutable, ce qui est tout à fait performant. Ceci démontre clairement qu'une analyse globale du système peut aisément être utilisée dans un cadre de développement incrémental.

La figure 2.14 donne aussi la taille du fichier exécutable produit (le compilateur SmallEiffel). Comparé à d'autres compilateurs Eiffel commerciaux, ou avec `gcc`, l'exécutable de SmallEiffel est petit (environ 660 KO). Ceci démontre qu'une intensive spécialisation de méthodes et de nombreux *inlinings*, combinés à une puissante élimination de code mort, n'augmentent pas sensiblement la taille des exécutables.

Afin d'avoir des résultats plus précis, nous avons instrumenté le code du compilateur de façon à comptabiliser exactement différents types d'appels rencontrés en compilant un code source (celui de SmallEiffel par exemple).

19. Réalisés avec la 14$^{\text{ème}}$version, numérotée `-0.86`, qui prend en compte des algorithmes plus complets et plus optimisés, notamment ceux décrits dans les sections 2.6.3 et 2.6.4.

20. Mesures faites avec la version `-0.86` sur un PC Pentium Pro 200 MHz avec 32 MO de RAM.

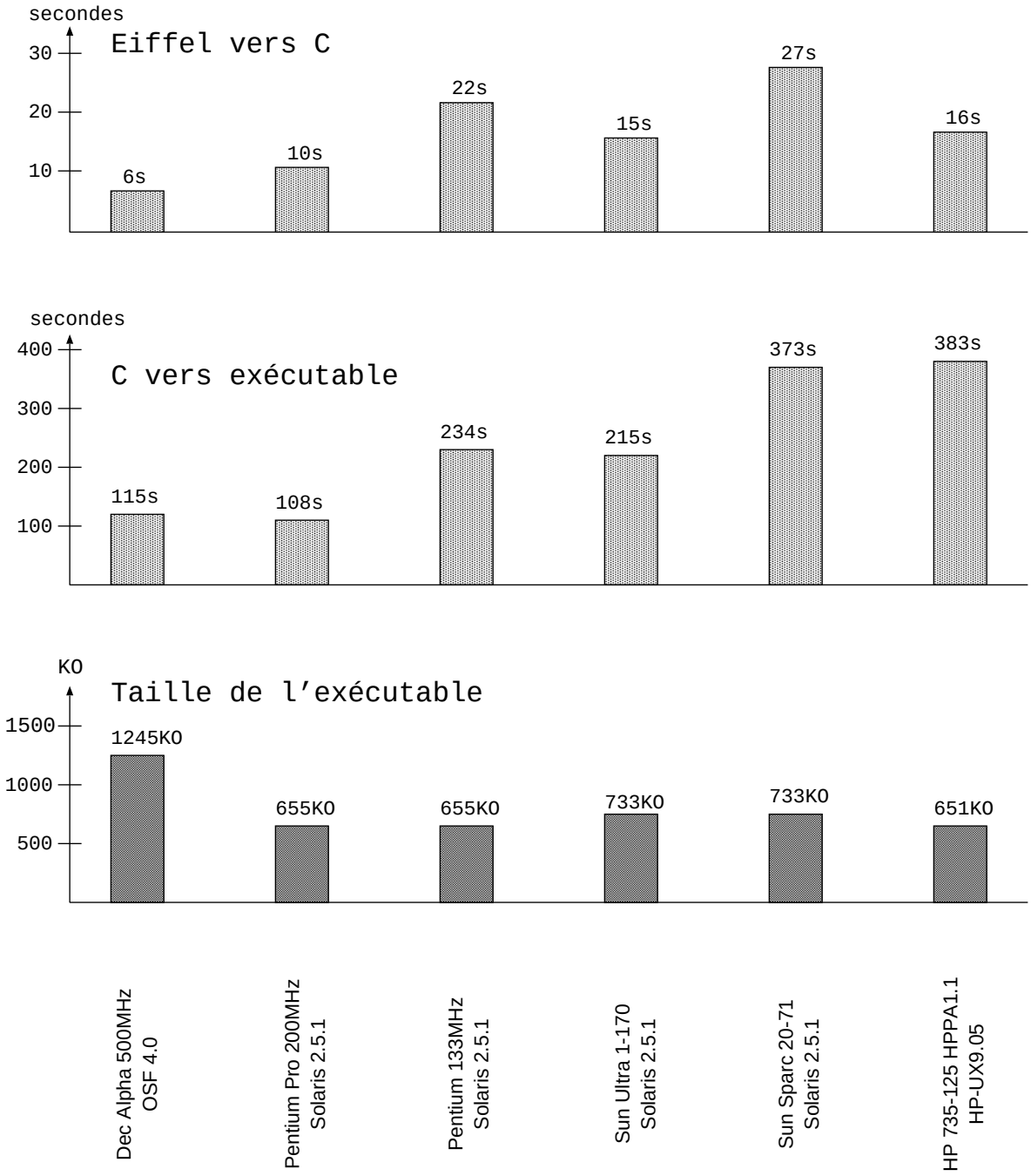
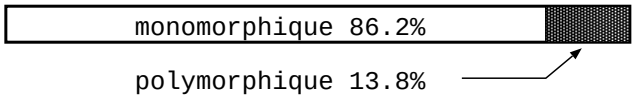


FIG. 2.14 – *Auto-compilation de SmallEiffel sur diverses architectures.*

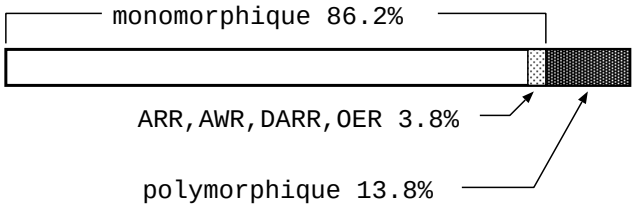
Haut: Temps total d'une recompilation complète pour obtenir 65000 lignes de code C optimisé à partir de 50000 lignes d'Eiffel.
Milieu: Temps total de compilation du code C pour produire un exécutable (édition de liens comprise).
Bas: Taille des exécutables autonomes.

2.7.2.2 Résolution statique des sites polymorphiques

Pour l'ensemble du compilateur, le score final de prédiction de type et de spécialisation de la liaison dynamique est excellent. En effet, sur l'ensemble des 28958 appels présents dans le programme — a priori tous polymorphiques, du fait de l'absence de méthodes de classe en Eiffel — il reste seulement 3983 sites d'appels polymorphiques, alors que 24975 sites d'appels ont pu être résolus comme des appels monomorphiques. La proportion d'appels résolus statiquement est donc considérable, comme le montre ce graphique:

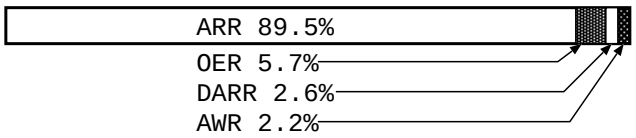


Sur les 24975 sites polymorphiques devenus monomorphiques, la majorité d'entre eux, soit 23877, ont été résolus par l'algorithme d'analyse globale, de prédiction de type et de duplication-spécialisation présenté dans les sections 2.3 et 2.4. Les 1098 sites supplémentaires ont été résolus plus tard dans la compilation, lors de l'étape de suppression des sites où chaque branche du code d'aiguillage contient le même code (voir la section 2.6.4 sur les ARR, AWR, DARR et OER). Remarquons que comme ces 1098 suppressions de sites polymorphiques supplémentaires arrivent après la fin de l'algorithme de prédiction de type, le bénéfice de 3,8 % correspondant n'est pas négligeable.



Soulignons que ces scores globaux de résolution de sites polymorphiques en sites monomorphiques, de l'ordre de 86 %, dont 83 % directement grâce à la prédiction de type, sont très représentatifs. Nous avons en effet constaté sur la plupart des programmes compilés que des taux de plus de 80 % étaient la norme avec notre méthode.

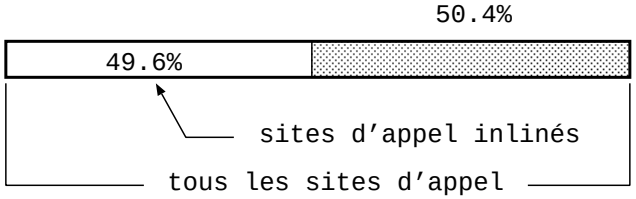
En considérant plus attentivement les ARR, AWR, DARR et OER, on s'aperçoit que la grande majorité des sites polymophiques supprimés proviennent des lectures d'attributs (ARR):



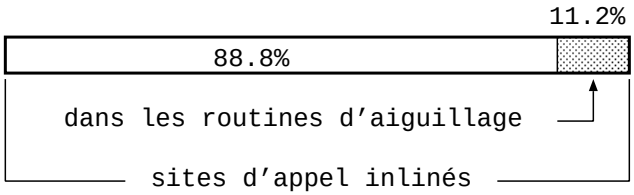
Remarquons que ce type d'informations peut très probablement servir à d'autres développeurs de compilateurs, notamment en leur permettant de faire des choix d'optimisations plus fins et plus focalisés.

2.7.2.3 Les *inlinings*

Le nombre total de sites *inlinés* est de 14376, sur un total de 28958 sites d'appels, soit près de la moitié:

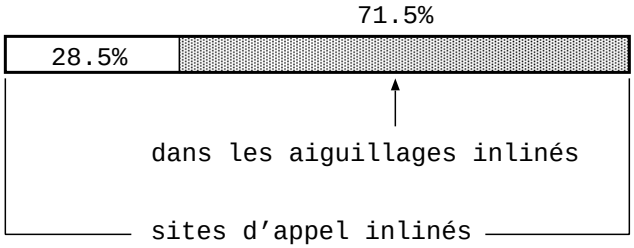


Comme nous l'avons vu précédemment (cf. section 2.6.3), l'expansion en ligne peut se produire sur les sites monomorphiques aussi bien qu'à l'intérieur des fonctions d'aiguillage des sites polymorphiques (aux branches du code binaire). Sur les 14376 sites expansés en ligne, 1603, soit 11,2 %, se trouvent ainsi à l'intérieur de routines d'aiguillage, et 12773, soit 88,8 %, viennent d'*inlinings* de sites monomorphiques:



Pour les 3983 sites d'appels polymorphiques non résolus statiquement et restants dans le programme, 199 routines d'aiguillage sont créées. Le nombre moyen de sites d'appel par routine (ou si l'on préfère, le nombre moyen d'appels *statiques*, présents dans le code) est donc de 20 environ. En conséquence, l'impact de l'expansion en ligne dans une routine d'aiguillage d'un site polymorphique peut être considéré comme *20 fois plus important* que celui de l'expansion de code pour un site d'appel monomorphique.

Si l'on suppose que les routines d'aiguillage elles-mêmes sont toutes *inlinées*, ce qui produirait $1603 \times 20 = 32060$ sites d'expansion en ligne liés à la liaison dynamique pour les sites polymorphiques, à comparer aux 12773 *inlinings* de sites monomorphiques, le ratio devient le suivant:



Ce genre d'*inlining* peut sembler intéressant pour éviter les appels aux routines d'aiguillage, et donc accélérer encore l'exécution. Néanmoins, ces routines représentent environ 8000 lignes de code C, sur un total de 65000 générées pour l'ensemble du compilateur. Comme chacune de ces routines est appelée en moyenne 20 fois, leur expansion en ligne produirait 160000 lignes supplémentaires, amenant le programme à un total de 217000 lignes ($65000 - 8000 + 20 \times 8000$). Ceci représenterait un triplement de la taille du code C généré, ce qui aurait pour conséquence de ralentir encore le processus de compilation du C vers l'exécutable, déjà nettement plus lent que la compilation d'Eiffel vers C.

De plus, comme l'appel à une routine d'aiguillage est un appel direct, statique, il ne brise pas le flot d'exécution. L'accélération potentielle due à de telles expansions en ligne supplémentaires semble donc limitée. Ceci, combiné avec l'importante augmentation de la taille du code généré, nous a conduit à choisir de ne *pas inliner* le code de ces fonctions d'aiguillage. On peut noter que le compilateur C, lui, reste libre d'effectuer de telles expansions lors de la génération du code binaire exécutable. Ainsi, nous avons par exemple vérifié que le compilateur `gcc` version 2.7.2 effectuait bien ces *inlinings*, mais seulement pour les fonctions d'aiguillage de petite taille.

La figure 2.15 page 43 présente la distribution des schémas d'*inlining* précédemment décrits, sans tenir compte de leur emplacement (à l'intérieur ou l'extérieur de fonctions d'aiguillage).

L'expansion en ligne sur lecture d'attribut (ARI) est de loin la plus commune. Ceci n'est pas surprenant du tout, car l'accès à un attribut est une opération élémentaire et fréquente dans les langages à objets. De plus, aucune fonction d'accès à un attribut n'est jamais générée, puisque comme nous l'avons expliqué précédemment — page 30, section 2.6.3 — tous les sites d'appel à de telles fonctions sont toujours expansés en ligne. Par conséquent, il n'y a aucun surcoût dû à la lecture d'un attribut via une fonction d'accès: performance maximale et sécurité via l'encapsulation peuvent être atteintes simultanément. Au contraire, les méthodes implantant la liaison dynamique à base de VFTs imposent l'utilisation de pointeurs de fonctions, qui empêchent l'expansion en ligne de toute fonction polymorphique.

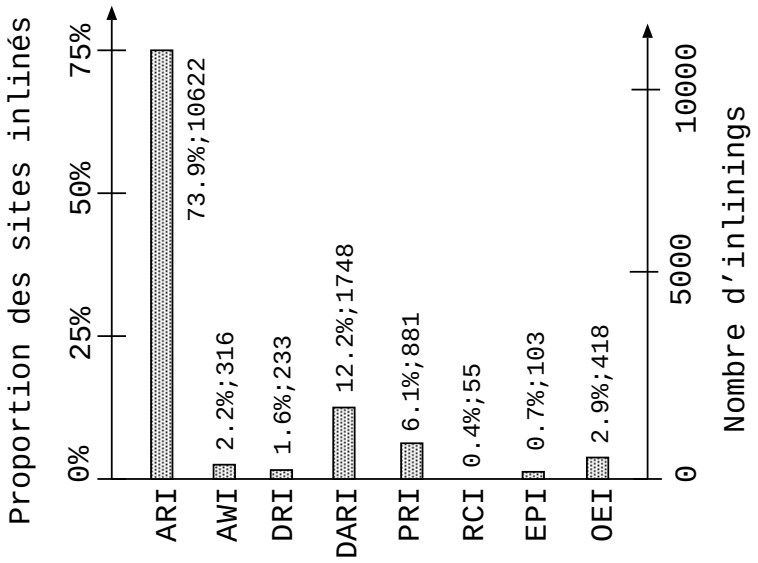


FIG. 2.15 – *Répartition des différents types d'inlinings.*

Les schémas DARI (*Direct Attribute Relay Inlining*) et PRI (*Predictable Result Inlining*) représentent aussi une part non négligeable des expansions en ligne. Après avoir étudié le code généré, nous avons constaté que les DARI viennent souvent de l'encapsulation. En effet, il n'est pas rare qu'une classe serve de relais, encapsulant des services (routines) fournis par d'autres classes, attributs privés de la première. De plus, comme nous l'avons mentionné dans la section 2.6.3, les DARI concernent en particulier certaines routines très souvent appelées, comme `item` de et .

Une explication de l'importance du PRI peut être déduite de la figure 2.16, qui montre — pour chaque type d'expansion en ligne — le ratio entre le nombre d'expansions à l'intérieur de routines d'aiguillage et le nombre total d'expansions de code.

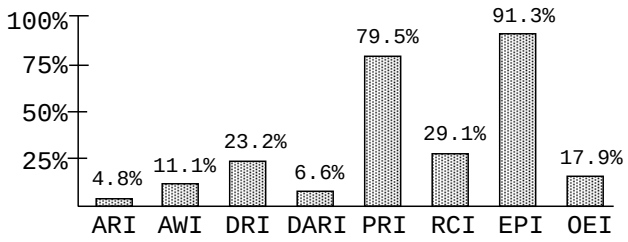


FIG. 2.16 – *Part d'inlinings au sein des routines d'aiguillage.*

La plupart des PRI sont à l'intérieur des routines d'aiguillage. Un examen plus approfondi du code généré révèle que ce phénomène est lié à un usage classique de l'héritage dans la programmation à objets: une routine "virtuelle" dans une classe abstraite est implantée au moyen d'un résultat constant dans la plupart de ses sous-types. Ainsi, la fonction `number_of_wheels` de la classe VEHICULE a la valeur constante 4 dans sa sous-classe CAR et 2 dans sa sous-classe MOTORCYCLE.

L'importance des EPIs (*Empty Procedure Inlinings*) à l'intérieur des routines d'aiguillage résulte d'une cause similaire. Il n'est en effet pas rare de définir un sélecteur `do_something` ayant un corps vide, ou au contraire d'avoir un comportement par défaut qui ne fait rien et qui est redéfini pour faire quelque chose dans seulement quelques sous-types.

Il s'agit là d'exemples typiques de petites *template methods* (voir section 2.4.2).

2.7.2.4 Objets sans identifiant de type

La suppression des identifiants de type pour les objets qui ne sont pas sujets à liaison dynamique (voir section 2.6.5) est effectuée très souvent. En effet, d'un point de vue statique, sur les 321 types vivants trouvés lors de la compilation de SmallEiffel[21], 160, soit 50 %, peuvent être générés sans identifiant de type.

Il est plus délicat d'observer ceci d'un point de vue dynamique, c'est à dire en considérant les objets créés à l'exécution. En effet, s'il est possible d'obtenir des informations sur les objets "réellement" créés, en instrumentant le code du ramasse-miettes intégré au compilateur (voir chapitre 3), il est nettement plus difficile d'en obtenir sur les objets non gérés par le ramasse-miettes. Ceux-ci comprennent les objets `expanded`, qui sont alloués en pile (à l'exception du cas particulier des tableaux de bas niveau), et qui sont toujours sujets à la suppression de l'identifiant de type.

Ainsi, si l'on observe le nombres d'objets crées dynamiquement, lors de l'exécution d'un *bootstrap* de SmallEiffel, on constate que 407269 objets gérés par le ramasse-miettes — c'est à dire des objets passés par référence ou des tableaux de bas niveau (NATIVE_ARRAY) — sont créés. Parmi ceux-ci, 92294 soit environ 23 %, le sont sans identifiant de type. Ce chiffre ne rend cependant pas bien compte de la réalité, puisqu'il ne compte pas les objets `expanded` (à l'exception des tableaux bas niveau), qui comprennent notamment toutes les classes de base comme INTEGER, CHARACTER, etc... Le nombre total d'objets créés sans identifiant de type est donc de toute évidence bien supérieur.

La mesure en détail de l'impact à l'exécution (et sur différentes exécutions) de ces suppressions d'identifiants de type semble pouvoir constituer une extension intéressante à nos travaux.

2.7.3 Liaison dynamique et surcoût des VFTs

Afin de quantifier l'impact du code de branchement binaire comparé aux VFTs, nous avons mesuré les performances d'un banc d'essai écrit à la fois en C++ et en Eiffel. Il comprend un seul site polymorphique, inclus dans une boucle exécutée un grand nombre de fois. Il utilise une classe abstraite X ayant trois sous-types concrets, A, B et C. La fonction virtuelle abstraite `do_it` de X est redéfinie différemment dans chaque classe concrète. Ces redéfinitions sont bien sûr choisies de façon à éviter toute possibilité de suppression (voir section 2.6.4) par SmallEiffel.

En conséquence, le programme suivant comporte toujours un site de liaison dynamique après , ce qui nous permet de bien mesurer le coût de la méthode de liaison dynamique. En voici la boucle correspondante écrite en C++:

```
{X *x;
 A *a = new A();
 B *b = new B();
 C *c = new C();
 int i;
 for (i = 100000000; i != 0; i--) {
    switch (i % 3) {
    case 0 : x = a; break;
    case 1 : x = b; break;
    default: x = c;
    }
   x->do_it(); //  Le site d'appel polymorphique
  }
};
```

Comme il alterne entre les trois branches, ce banc d'essai rend difficile la prédiction du type du receveur, aussi bien au niveau du compilateur qu'au niveau du . Des prédicteurs évolués des microprocesseurs peuvent cependant capturer ce type de régularité [Hennessy et Patterson, 1996].

Cette version C++, compilée avec `g++`, implante la liaison dynamique avec une VFT, alors que le code C généré en standard par SmallEiffel utilise une routine avec un aiguillage sous forme d'arbre de

21. Compilation de `compile_to_c`, version -0.76beta1.

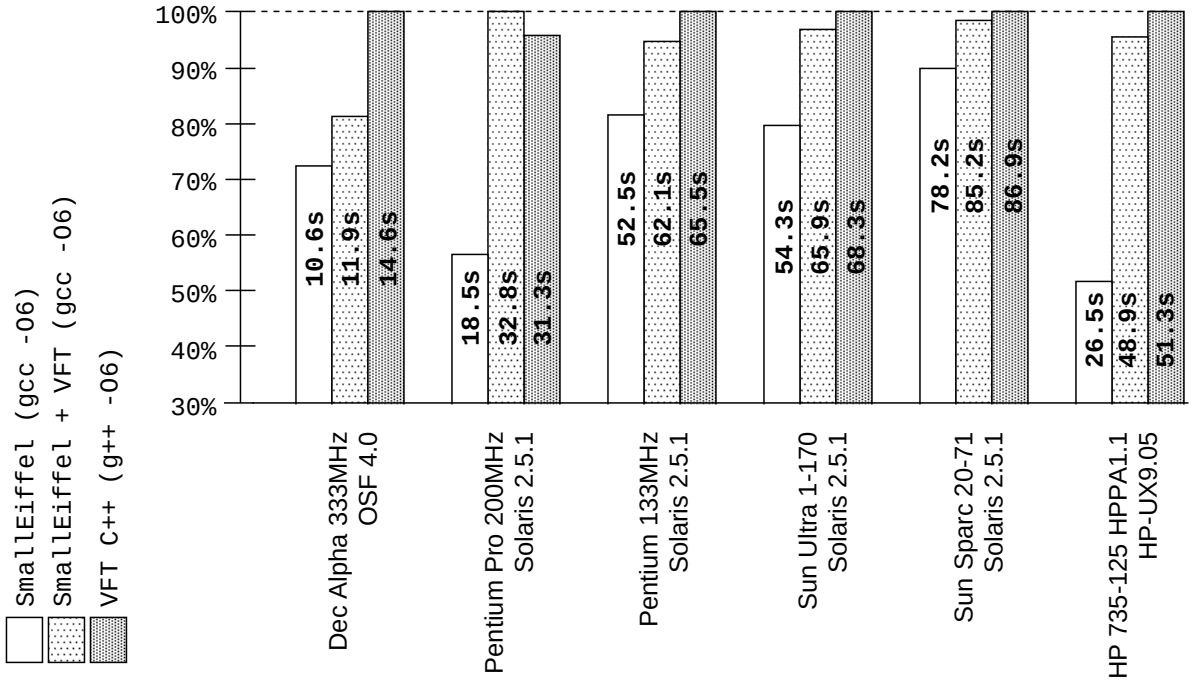


FIG. 2.17 – *Comparaison des temps d'exécution pour un site d'appel polymorphique cyclique à 3 possibilités: VFT vs. arbre binaire.*

tests binaire. Afin de nous assurer que la seule différence remarquable venait bien de l'implantation de la liaison dynamique, nous avons ajouté une troisième version du programme de test. Celle-ci a été obtenue en modifiant manuellement le code C produit par SmallEiffel en remplaçant le site d'appel par un appel via VFT écrit en C, toutes choses restant inchangées par ailleurs.

La figure 2.17 montre les temps d'exécution de ces trois versions du programme de test sur différentes architectures. Bien entendu, notre but n'est pas ici de comparer les processeurs, mais bien de démontrer que nos résultats ne sont pas dépendants d'une architecture spécifique et donc que notre méthode de liaison dynamique est portable.

On constate clairement que le code C non modifié produit par SmallEiffel est toujours le plus rapide. Sur les plates-formes où nous avons fait nos tests, SmallEiffel est dans le pire des cas 10 % plus rapide que la version C++, alors qu'il est jusqu'à 48 % plus rapide dans le meilleur des cas. Ceci montre que, lorsqu'on a affaire a des sites d'appel polymorphiques réguliers avec un faible degré de polymorphisme (peu de types possibles pour le receveur), notre implantation de la liaison dynamique dans SmallEiffel est nettement meilleure. De plus, les temps obtenus avec C++ et la version modifiée du code généré par SmallEiffel sont consistants. Comme ils implantent tous deux la liaison dynamique avec des VFTs, ceci confirme que les implantations à base de VFT sont moins efficaces que la méthode que nous présentons dans ce mémoire.

Cependant, la méthode de liaison dynamique par VFT a un temps d'exécution constant quel que soit le nombre de types concrets possibles pour un site d'appel polymorphique donné. Au contraire, le coût de la méthode que nous avons implantée dans SmallEiffel n'est pas constant car le nombre de branches dans la routine d'aiguillage croît avec le nombre de types concrets possibles. Comme nous avons choisi un code d'aiguillage basé sur un arbre de tests binaires énumérant tous les types possibles, le nombre de branches est de $log_2(P)$ où P est le nombre de types concrets possibles. La question de la capacité de notre méthode à "monter en charge", c'est-à-dire à supporter des sites d'appels *mégamorphiques* — i.e. des sites d'appels polymorphiques avec de nombreux types de receveur possibles [Driesen *et al.*, 1995] — se pose donc fort naturellement. Afin de comparer les temps d'exécution des deux méthodes pour un site d'appel *mégamorphique*, nous avons étendu le banc d'essai précédent à 50 sous-classes. Rappelons que ceci est considérable, puisque selon [Aigner et Hölzle, 1996], un site avec un total de "seulement" 17 types

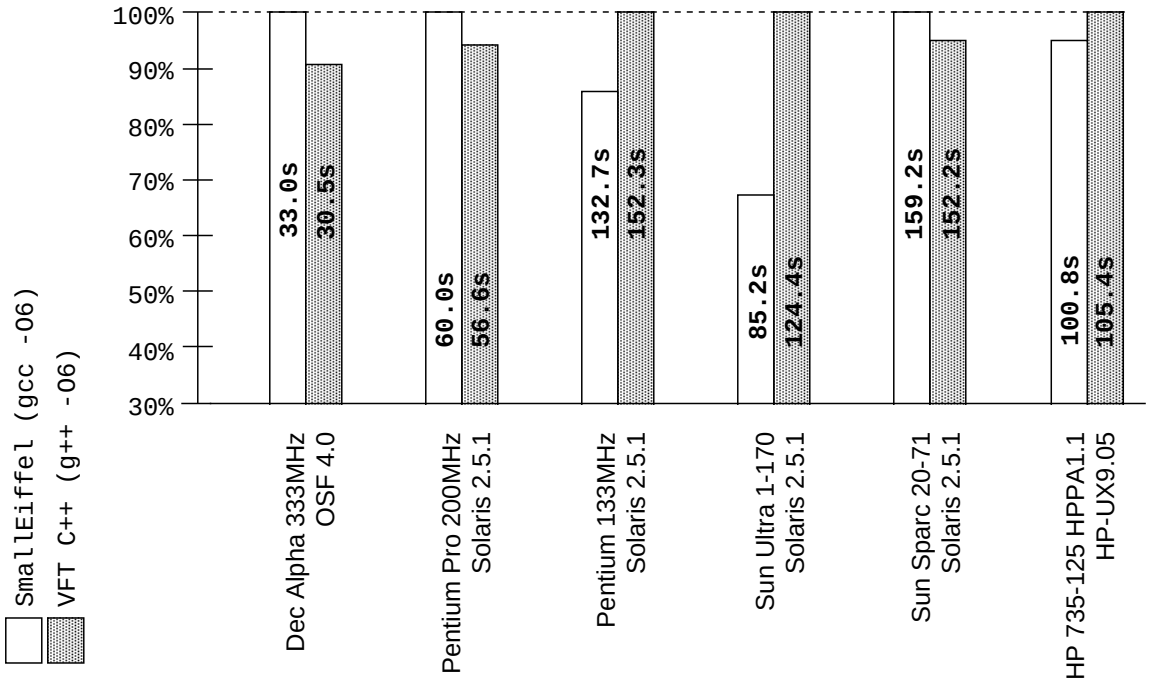


FIG. 2.18 – *Comparaison des temps d'exécution pour un site mégamorphique cyclique à 50 possibilités: VFT vs. arbre binaire.*

concrets possibles peut déjà être considéré comme mégamorphique. Remarquons que dans SmallEiffel, gros programme faisant un usage intensif de la liaison dynamique, le site d'appel ayant le degré de polymorphisme le plus élevé a 48 types concrets possibles (les descendants de la classe EXPRESSION, voir table page 39).

La classe X précédemment évoquée possède donc maintenant 50 sous-classes, ce qui aboutit à un aiguillage à 50 branches. La figure 2.18 montre les résultats obtenus avec un banc d'essai où le code alterne entre ces 50 possibilités, rendant l'appel polymorphique très difficilement prédictible tant au niveau logiciel que matériel.

A l'opposé, la variante testée dans la figure 2.19 ne fait pas alterner le receveur parmi plusieurs types, invoquant toujours la méthode `do_it` sur le même receveur, ce qui rend l'appel parfaitement et aisément prédictible.

La figure 2.18 montre que, pour un site mégamorphique (quasiment) imprédictible, la méthode d'aiguillage à arbre de tests binaire et la méthode à base de VFT ont des résultats similaires. En effet, sur trois des architectures, le code à base de VFT est plus rapide que le code de tests binaires, de 4 % à 7 %, alors que dans trois autres cas, c'est l'arbre de tests qui est le plus rapide, de 4 % à jusqu'à 31 %. Le code de branchement à arbre de tests binaire apparaît donc efficace, voire meilleur que les VFTs, même dans des cas mégamorphiques qui lui sont théoriquement défavorables. Comme tous les types de receveur sont équiprobables, des techniques basées sur des profils d'exécution [Aigner et Hölzle, 1996] ou des caches en ligne [Deutsch et Schiffman, 1984; Ungar et Patterson, 1987] ne peuvent améliorer ces résultats.

La figure 2.19 montre que les résultats sont encore meilleurs sur un site d'appel parfaitement prédictible, puisque l'arbre de tests binaire dépasse le code à base de VFT sur quatre architectures.

Un examen plus détaillé du code assembleur généré pour le code de branchement binaire et pour le code à base de VFT amène à certaines observations intéressantes. Tout d'abord, les dépendances de données sont importantes dans le code à base de VFT (voir par exemple [Driesen et Hölzle, 1996]), alors qu'elles le sont moins dans le code de branchement binaire. Néanmoins, dans ce dernier, de plus nombreuses dépendances de contrôle apparaissent. Celles-ci ne représentent probablement pas un problème sur les architectures utilisant un mécanisme de table de prédiction de branchements directs(*Branch History Table*, ou BHT), qui permettrons à de plus nombreux branchements conditionnels d'être prédits avec

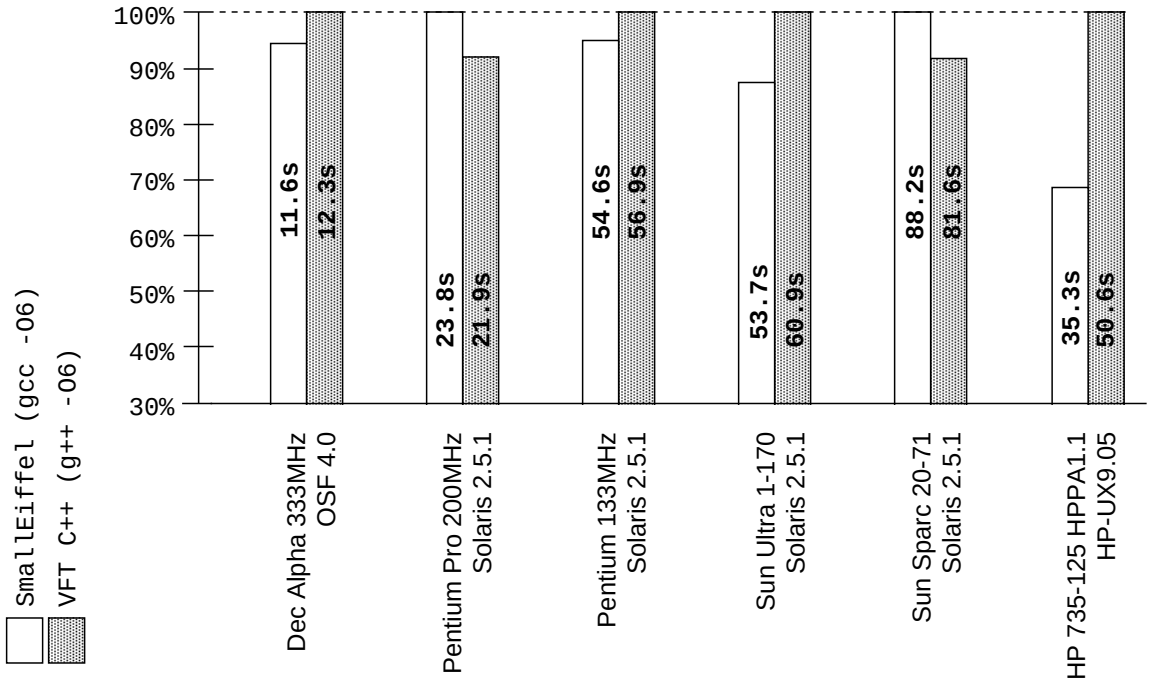


FIG. 2.19 – *Comparaison des temps d'exécution pour un site d'appel mégamorphique à 50 possibilités totalement prédictible.*

exactitude [Hennessy et Patterson, 1996]. En effet, une propriété reconnue du polymorphisme est qu'en général le type du receveur à un site d'appel polymorphique donné varie peu. La BHT fait donc office de mémoire du dernier type du receveur, ce qui peut être considéré comme une forme de cache en ligne géré par le microprocesseur.

2.7.4 Comparaison avec d'autres compilateurs C++ et Eiffel

Dans cette section, basée sur [Wolz, 1997], nous présentons les résultats obtenus sur un algorithme calculant la colimite d'un diagramme de signature (une construction de la théorie des catégories qui est utile pour les concepts de paramétrisation dans la spécification et les langages de programmation, ainsi que pour les transformations de graphes). Cet algorithme a été implanté à la fois en Eiffel et en C++, et testé sur différents compilateurs et/ou bibliothèques.

Le programme de tests Eiffel consiste en 13 classes dont l'une — des tableaux statiques, héritière de la classe standard ARRAY — a été soigneusement adaptée aux différents compilateurs pour optimiser ses performances. Le programme C++ utilise une structure similaire et est basé sur la Standard Template Library (STL). Les tests ont été réalisés sur une machine avec processeur Pentium 200 MHz avec 512 KO de cache et 192 MO de RAM, sous système Linux à noyau 2.0.12 et avec les compilateurs `gcc` et `g++` version 2.7.2.

La figure 2.20 a été obtenue en mesurant les performances des deux versions du programme sur un gros diagramme comportant 2 millions de symboles. Les résultats incluent les temps de compilation et d'exécution, dans la partie supérieure du graphique, ainsi que la taille de l'exécutable et celle occupée en mémoire lors de l'exécution, dans la partie inférieure.

Ce banc d'essai est une illustration frappante de la vitesse et l'efficacité de la méthode que nous avons précédemment décrite et qui est implantée dans SmallEiffel.

Tout d'abord, l'exécutable généré par SmallEiffel est parmi les plus rapides, que ce soit comparé aux autres compilateurs Eiffel ou aux compilateurs C++. De plus, les temps de compilation avec SmallEiffel sont les meilleurs, quel que soit le compilateur considéré. Il faut se rappeler que comme SmallEiffel a

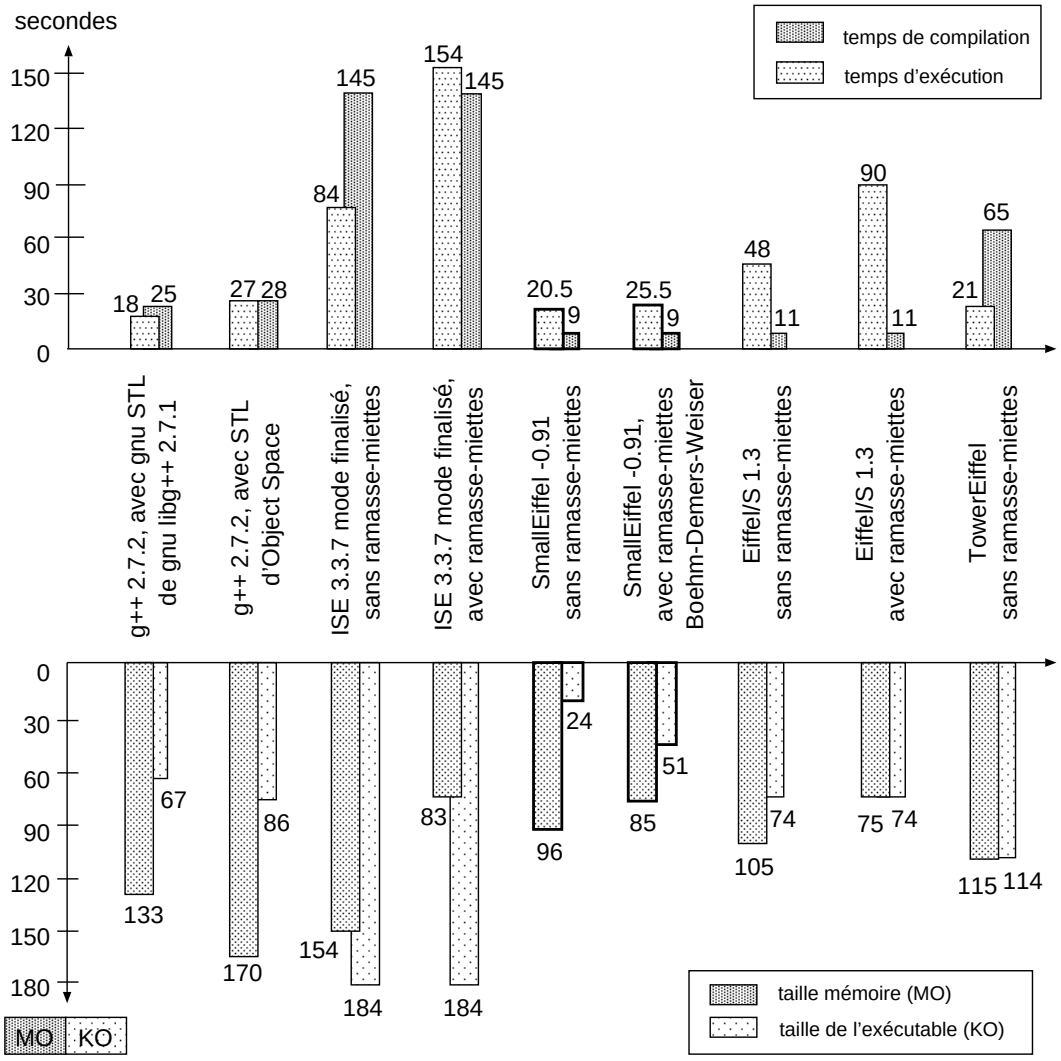


FIG. 2.20 – *Comparaison des performances de SmallEiffel face à d'autres compilateurs.*

Haut: Temps de compilation et temps d'exécution.
Bas: Taille de l'exécutable et mémoire occupée à l'exécution.

été auto-compilé, son code exécutable a lui-même été optimisé par les techniques détaillées ci-dessus. Ses bons résultats sont donc une confirmation supplémentaire de l'efficacité de la méthode.

La figure 2.20 montre également clairement que SmallEiffel génère les exécutables les plus petits, en dépit de la duplication du code spécialisé et de l'utilisation de tests de type explicites au lieu de pointeurs de fonctions pour l'implantation de la liaison dynamique.

Afin d'interpréter les résultats en termes de mémoire, il est important de prendre en compte le fait que le programme C++ utilise une gestion manuelle de la mémoire, mais que certaines fuites demeurent. L'occupation mémoire avec C++ n'est donc pas significative, mais la comparaison entre les différents compilateurs Eiffel, elle, est parfaitement valide.

Comme la version de SmallEiffel utilisée pour ces mesures était l'une des premières (`-0.91`), elle ne possédait pas encore de ramasse-miettes intégré (cf. chapitre 3). Ses exécutables ont donc été liés à la bibliothèque de ramasse-miettes Boehm-Demers-Weiser (BDW 4.10) [Boehm et Weiser, 1988][22]. Cette bibliothèque, utilisant un algorithme heuristique et générique [Boehm, 1993; Boehm et Shao, 1993], n'avait aucunement été adaptée à SmallEiffel et n'avait aucune connaissance de la représentation interne des objets générés par SmallEiffel. Pourtant, l'utilisation mémoire avec la version produite par SmallEiffel lié au BDW est la seconde meilleure, derrière celle du compilateur Eiffel/S.

Comme nous l'expliquons en détail dans le chapitre 3, page 53, consacré à la gestion automatique de la mémoire dans SmallEiffel, celui-ci intègre maintenant son propre ramasse-miettes, produit à l'aide de notre algorithme d'analyse statique suivie de duplication-spécialisation et dont les performances sont tout à fait à niveau avec celles du BDW.

2.8 Autres travaux liés à l'optimisation de la liaison dynamique

Calder et Grunwald [Calder et Grunwald, 1994] ont étudié la prédiction du type de receveur à l'aide de profils d'exécution pour éliminer les appels de fonctions indirects. Ils ajoutent des tests explicites pour éviter le système de liaison dynamique standard pour les types de receveur les plus courants. Le code généré par SmallEiffel n'est pas sans rapport avec celui de Calder et Grunwald, puisqu'il consiste à tester explicitement le type du receveur et à utiliser des appels de fonctions directs. Cependant, une différence importante est que le test explicite du type est étendu dans notre technique à tous les types de receveur possibles, pas seulement les plus courants. Ceci permet l'élimination totale de tout mécanisme du type VFT pour l'implantation de la liaison dynamique.

Utilisant à la fois des optimisations tirées de profils d'exécution et une analyse de la hiérarchie des classes, Aigner et Hölzle montrent comment améliorer l'efficacité de la liaison dynamique en C++ [Aigner et Hölzle, 1996]. SmallEiffel utilise un algorithme de prédiction de type plus complet et implante les sites de liaison dynamique restants sans aucune VFT. Aigner et Hölzle montrent également que les *inlinings* accroissent marginalement la taille du code généré et que pour la plupart des programmes le taux d'échec des accès au cache (*cache miss rate*) d'instructions n'augmentait pas sensiblement. Nos résultats sont consistants avec les leurs aussi bien pour la taille des exécutables que le temps d'exécution.

Dans les deux études précédentes, les tests de type explicites sont *inlinés*, avec un coût en taille limité, puisque seuls les types de receveur les plus courants sont testés. Dans nos travaux, au contraire, nous factorisons ces tests de type dans une routine d'aiguillage, car comme nous l'avons expliqué en section 2.7, il ne serait pas raisonnable pour des raisons de place d'*inliner* ces tests de types de receveur lorsque *tous* les types de receveur possibles sont testés. Une prédiction du type du receveur basée sur des informations extraites de profils d'exécution pourrait être ajoutée à notre méthode afin par exemple de trier partiellement et d'optimiser le code de branchement binaire que nous utilisons pour la liaison dynamique.

Dans [Dean *et al.*, 1995], Dean *et al.* décrivent l'algorithme d'analyse de la hiérarchie de classes (*Class Hierarchy Analysis*, ou CHA), utilisé pour supprimer les sites d'appel polymorphiques. Ils montrent que le CHA est assez rapide pour être implanté dans des environnements de développement interactifs. Ils indiquent aussi qu'en dépit du fait que le CHA a besoin d'informations sur l'ensemble du programme et est donc une analyse globale, il peut être adapté pour autoriser l'incrémentalité. Nos résultats confirment également ceci, grâce à la grande vitesse du compilateur SmallEiffel.

22. Cette bibliothèque est disponible depuis `http://reality.sgi.com/boehm_mti/gc.html`

Bien que notre algorithme de prédiction de type soit plus développé que le seul CHA, l'algorithme du produit cartésien (*Cartesian Product Algorithm*, ou CPA) d'Agesen [Agesen, 1995] est lui-même plus puissant que le notre. En effet, notre algorithme n'effectue actuellement aucune analyse de flot de données [Corney et Gough, 1994], qu'elle soit intra- ou inter-procédurale. L'incorporation du CPA à notre méthode nous permettrait d'éliminer encore plus de sites de liaison dynamique, augmentant encore la rapidité du code généré par SmallEiffel. Des études supplémentaires sont néanmoins nécessaires afin d'évaluer l'impact possible cette intégration. Il est en effet courant que la prise en compte du flot ait un coût important, notamment en termes de vitesse de compilation, pour une amélioration relativement limitée de la qualité des informations disponibles sur le programme et de la vitesse du programme généré. Dans le cadre d'une application autre qu'un prototype, ceci doit être considéré afin d'atteindre un compromis satisfaisant entre complétude et performances.

Hölzle et Ungar [Hölzle et Ungar, 1995] soulèvent l'intéressante question de savoir si les langages à objets requièrent du matériel spécifique. Ils concluent que les tests de liaison dynamique ne peuvent pas être facilement améliorés à l'aide de matériel spécifique et dédié, et que la façon la plus prometteuse de réduire le coût de la liaison dynamique est via des optimisations faites par le compilateur. SmallEiffel en est un exemple, ses optimisations éliminant environ 80 % des sites d'appels polymorphiques et générant uniquement des instructions très simples et donc d'exécution rapide.

Bacon et Sweeney [Bacon et Sweeney, 1996] ont étudié la capacité de trois catégories d'analyses statiques à améliorer les programmes C++ en résolvant les appels de fonctions virtuelles et réduisant la taille du code compilé. Leur meilleur algorithme, RTA (dont CHA est un composant de base) supprime environ 71 % des sites d'appels virtuels, ce qui est comparable aux 80 % atteints par SmallEiffel avec notre méthode. Le score supérieur de cette dernière peut provenir en partie des suppressions qui interviennent après la prédiction de type (ARR, AWR, DARR et OER, voir section 2.6.4).

Dans [Rose, 1988], Rose propose une technique de table dite "grasse" (*fat table*) pour implanter la liaison dynamique, en plus des classiques tables "maigres" (*thin tables*) du type VFT. Ces tables grasses contiennent du code *inliné* pour les petites fonctions, au lieu des pointeurs de fonctions, ce qui permet un branchement direct depuis la table. Cependant, comme le confirme Rose, les tables grasses peuvent seulement contenir de très courtes méthodes qui s'exécutent sans transfert de contrôle supplémentaire. Cette technique reste donc très limitée et n'offre pas les avantages de la notre, puisqu'elle est difficilement applicable à des méthodes autres que de très petite taille et en l'absence de polymorphisme.

Enfin, notons que Chambers et Chen ont récemment [Chambers et Chen, 1999] réutilisé notre technique comme partie de leur système adaptatif choisissant automatiquement parmi divers algorithmes pour implanter efficacement la liaison dynamique dans le compilateur pour le langage Cecil [Chambers, 1992; Chambers, 1993], dans le cadre du *single dispatching*, du *multiple dispatching* et du *predicate dispatching*. Ceci conforte donc les excellents résultats que nous présentons dans ce mémoire.

2.9 Conclusions et perspectives

Nous décrivons une méthode permettant d'implanter la liaison dynamique avec une très grande efficacité.

Sa première étape consiste en un puissant algorithme de prédiction de type analysant l'ensemble du système pour remplacer les sites d'appels polymorphiques par des appels directs, statiques. Cet algorithme atteint des scores en moyenne élevés: plus de 80 % des sites polymorphiques sont réduits en sites monomorphiques dans nos tests. Les sites de liaison dynamique restants sont alors codés efficacement, en éliminant toute table de type VFT et même tout pointeur de fonction, et en les remplaçant par un arbre binaire de tests statiques. Ces deux étapes permettent à la troisième, l'expansion de code en ligne (*inlining*), d'être faite de façon très intensive, à la fois pour les sites d'appels monomorphiques et au sein des branches du code d'aiguillage des appels polymorphiques restants.

Cette méthode est implantée dans un nouveau compilateur pour le langage Eiffel nommé SmallEiffel, et qui est devenu *The GNU Eiffel Compiler*. A notre connaissance, c'est la première fois qu'une telle méthode est appliquée à un tel projet, non limité à un prototype.

Nous démontrons la validité de la méthode tant en ce qui concerne la vitesse d'exécution que la taille du code généré, même dans le cas pathologique d'appels peu prédictibles et fortement polymorphiques

(*mégamorphiques*). Nous montrons que le surcoût dû à la VFT est évité par cette méthode, sans qu'il soit remplacé par aucun coût supplémentaire.

Nous soulignons l'utilité pour la conception à objets des optimisations permises par notre méthode et implantées dans SmallEiffel. En effet, il n'est pas rare, avec les compilateurs C++ par exemple, que des développeurs hésitent à utiliser "à fond" les concepts des langages à objets (tels qu'objets, liaison dynamique, polymorphisme, encapsulation), par crainte du surcoût à l'exécution introduit par ces mécanismes de haut niveau. Ils font donc un compromis entre bonne conception et bonnes performances, ce qui nuit fortement à la qualité de leur production. Notre algorithme de compilation se charge de l'optimisation du code lié aux concepts puissants mentionnés ci-dessus, se focalisant sur des optimisations faisant disparaître le surcoût potentiel dû à la programmation à objets, ce qui assure ainsi des performances excellentes sans que les développeurs aient à transiger sur la qualité de leur conception et leur apporte un confort de développement non négligeable.

Un inconvénient intrinsèque apparent de notre méthode est qu'elle requiert la connaissance de l'ensemble du système, ce qui semble rendre la compilation séparée et la production de bibliothèques précompilées plus délicates. Ceci peut être considéré comme un problème pour du développement incrémental. Néanmoins, dans ce contexte, le point le plus important pour le développeur est la vitesse de (re)compilation. Notre méthode globale permettant justement une très grande vitesse de compilation, comme montré en section 2.7, nous sommes convaincus qu'elle peut aisément être intégrée à des compilateurs incrémentaux utilisés en production. Nous avons nous-mêmes pu apprécier, en tant qu'utilisateurs de SmallEiffel dans le cadre de nos travaux, cette rapidité de compilation et de recompilation, qui nous a permis de développer de façon incrémentale (ce qui n'aurait pas été le cas avec d'autres compilateurs Eiffel commerciaux).

En fait, les programmes générés avec SmallEiffel peuvent interagir avec les bibliothèques existantes (faites par exemple avec des compilateurs C/C++ classiques), à l'aide du mécanisme d'*externals*[23] d'Eiffel. Notre technique peut également être utilisée pour produire des bibliothèques pré-compilées (*bibliothèques statiques*) ainsi que des *bibliothèques partagées* et chargées à l'exécution (*bibliothèques dynamiques*). Comme notre compilateur requiert la connaissance de la totalité du système à compiler et qu'il ne compile que le code vivant accessible depuis une ou plusieurs racines (voir section 2.3.2), il est nécessaire de lui indiquer quels sont les "points d'entrée" dans une bibliothèque susceptibles d'être utilisés par le code client extérieur. Ceci peut se faire grâce au mécanisme CECIL[24] (*C-Eiffel Call-In Library*) d'Eiffel. Néanmoins, ce code pré-compilé pouvant comporter de nombreuses racines (point d'entrées externes), il risque fort d'être moins spécialisé et donc moins optimisé que le code d'un exécutable autonome compilé à partir d'un système connu entièrement. Ce problème de "frontière" entre système connu et système inconnu, son impact sur la production de bibliothèques, sur la réutilisation de composants binaires, et plus généralement sur la compilation, nous semble être un sujet de recherche tout à fait intéressant. Il se rapproche d'ailleurs des nombreuses recherches menées actuellement sur la *compilation dynamique*.

La technique de compilation que nous décrivons ici pour Eiffel peut être appliquée à n'importe quel langage de classes — même avec typage générique et héritage multiple — ne permettant pas la création ou la modification dynamique de classes. C'est le cas de C++, alors qu'il n'est pas certain que la méthode puisse être appliquée à Java, à cause de certains aspects plus dynamiques de ce langage. Ce point semble néanmoins pouvoir faire l'objet d'études intéressantes.

Notre méthode utilise seulement des informations statiques pour améliorer l'efficacité du code — notamment la liaison dynamique. L'adjonction à notre compilateur de la prise en compte du flot d'exécution augmenterait les performances du code généré, en permettant le remplacement d'un plus grand nombre de sites de liaison dynamique par des appels monomorphiques directs, statiques. Ceci peut néanmoins provoquer une augmentation conséquente des temps de compilation.

Comme nous l'avons vu précédemment dans la section 2.6.4, lorsque dans une fonction d'aiguillage toutes les feuilles de l'arbre de branchement binaire correspondent au même code, les sites d'appels polymorphiques correspondants sont complètement supprimés. Il est également possible d'optimiser encore

23. Voir `http://SmallEiffel.loria.fr/man/external.html`
24. Sans rapport avec le *langage à objets* Cecil. Voir `http://SmallEiffel.loria.fr/man/cecil.html`

plus l'arbre de tests binaires en fusionnant sélectivement plusieurs branches lorsqu'elles mènent au même code. Une rapide analyse des arbres d'aiguillage générés semble confirmer que de nombreuses branches pourraient ainsi être fusionnées. Afin de pouvoir maximiser le nombre de ces fusions, l'attribution des identifiants de type est un point crucial. Il s'agit là d'un problème complexe que nous n'avons pour l'instant pas encore étudié.

D'autre améliorations peuvent aussi venir de méthodes heuristiques, basées sur des informations dynamiques, rassemblées lors de l'exécution, comme les caches en ligne (*Inline Caches*).

Enfin, une voie de recherche prometteuse semble être l'étude des interactions entre logiciel et matériel afin d'optimiser au mieux des programmes écrits à l'aide de langages à objets, notamment au niveau de l'implantation de la liaison dynamique. Un problème important consiste en effet à arriver à caractériser et modéliser les interactions entre les différentes techniques logicielles de liaison dynamique et le matériel, afin de comprendre leur comportement, avec pour objectif l'amélioration de leur performances en tenant compte de l'architecture matérielle sous-jacente.

Nous pouvons donc conclure que l'ensemble des techniques de compilation que nous avons présentées dans ce chapitre, et tout particulièrement notre analyse globale et notre technique de liaison dynamique par arbre de branchement binaire, offrent de très bons résultats, mais peuvent encore faire l'objet de nombreux travaux visant à les améliorer ou s'en servant comme d'une base de départ.

Chapitre 3

Spécialisation automatique d'un système de gestion mémoire

3.1 Introduction

La mémoire disponible pour l'exécution des programmes a toujours été une ressource précieuse et, en général, rare. En effet, bien que la baisse des prix et l'augmentation des capacités des matériels informatiques en général, et de leur mémoire en particulier, soient devenues des poncifs, il n'en reste pas moins vrai que les besoins des programmes ne cessent eux aussi d'augmenter. On assiste donc a une course, probablement sans fin, entre augmentation des ressources et augmentation des besoins qui font que la gestion judicieuse de ces ressources reste un point délicat. Nous nous intéressons dans ce chapitre à la gestion de la mémoire consommée par un programme.

Historiquement, les premières techniques de gestion mémoire ont été les techniques manuelles, qui consistent à laisser le développeur indiquer explicitement les allocations (comme avec `malloc` en C, par exemple) et surtout les restitutions de mémoire (avec `free` en C, par exemple). Les techniques automatiques, plus complexes à mettre en oeuvre, ne sont arrivées que plus tard. Pour cette raison, et à cause d'une certaine inertie chez les développeurs, les techniques automatiques étaient considérées avec réticence. Le reproche qui leur était le plus souvent fait était leur coût "inacceptable" par rapport à la gestion manuelle de la mémoire.

Cette critique était relativement justifiée avec les premières implantations des ramasse-miettes, qui pouvaient représenter une part non négligeable du temps d'exécution total des programmes dont ils géraient la mémoire. Ainsi, les études menées dans les années 1975 à 1985 montraient que typiquement le ramasse-miettes prenait jusqu'à 40 % du temps d'exécution des gros programmes Lisp [Steele, 1975; Foderaro et Fateman, 1981; Gabriel, 1985]. Depuis, fort heureusement, les techniques de gestion automatique de la mémoire se sont constamment et graduellement améliorées, amenant vers 1995 à des coûts moyens de l'ordre de 10 % et à des algorithmes incrémentaux, parallèles, ou distribués, fournissant ainsi une large gamme de solutions aux développeurs [Wilson, 1992; Wilson *et al.*, 1995; Jones et Lins, 1996]. La gestion automatique de la mémoire semble maintenant être préférée à la gestion manuelle dans la plupart des langages de programmation modernes [Ungar et Smith, 1987; Meyer, 1992; Gosling *et al.*, 1996]. En effet, l'expérience montre, lorsqu'on impose au développeur une gestion manuelle de la mémoire de son application, qu'il existe un risque non négligeable que cette gestion soit incorrecte. Les deux erreurs classiques sont que le développeur oublie de restituer la mémoire qui n'est plus utile au programme (c'est ce qu'on appelle *une fuite de mémoire*), ou qu'il restitue trop tôt la mémoire correspondant à des données encore utilisées (ce qui crée un pointeur vers "n'importe quoi", appelé en anglais *dangling pointer*). La gestion automatique présente l'avantage considérable de supprimer cette catégorie d'erreurs particulièrement fréquentes et difficiles à corriger.

Il n'en reste pas moins que certains développeurs continuent à penser que les meilleures performances ne peuvent être atteintes qu'en s'appuyant sur une gestion manuelle de la mémoire. En effet, ils considèrent que cela leur permet de mieux répondre aux besoins spécifiques à leur application. Néanmoins, la

communauté des chercheurs en gestion automatique de la mémoire travaille depuis longtemps sur l'adaptation des ramasse-miettes à chaque application (spécialisation, tenant ainsi compte des préoccupations des adeptes de la gestion manuelle de la mémoire.

Beaucoup de ces ramasse-miettes adaptés [Bartlett, 1990; Edelson, 1992; Boehm et Shao, 1993; Attardi *et al.*, 1995] requièrent encore une intervention du développeur, par exemple la fourniture explicite d'informations sur les objets gérés. Nous présentons dans ce chapitre une implantation d'un système complètement automatique dans lequel le support de la part du compilateur permet la génération d'un ramasse-miettes très performant en temps d'exécution car adapté à l'application compilée, sans pour autant imposer le moindre travail supplémentaire au développeur de l'application.

Ce chapitre est organisé de la façon suivante. La section 3.2 introduit brièvement le concept de gestion automatique de la mémoire et présente les algorithmes classiques du domaine. La section 3.3 indique ensuite les principes adoptés dans SmallEiffel pour générer automatiquement des ramasse-miettes adaptés aux applications compilées. Le section 3.4 explique plus précisément la méthode que nous avons utilisée pour adapter le code du ramasse-miettes à l'aide du support du compilateur et détaille ensuite la gestion des objets de taille fixe. Les objets redimensionnables, quant à eux, sont considérés dans la section 3.5. La section 3.6 décrit brièvement diverses optimisations plus spécifiques, dépendant du langage. Un ensemble de résultats expérimentaux est présenté en section 3.7. Enfin, la section 3.8 passe en revue les travaux du domaine comparables à notre méthode, tandis que la section 3.9 conclut.

3.2 Les algorithmes de gestion automatique de la mémoire

La gestion automatique de la mémoire fait partie intégrante de nombreux langages de programmation de haut niveau. De très complètes revues de ce domaine peuvent être trouvées dans [Wilson, 1992] et [Jones et Lins, 1996].

Un programme dont la mémoire est gérée par un ramasse-miettes se compose en fait de deux parties. L'une est le système de gestion automatique de la mémoire (ramasse-miettes). L'autre est le programme "utile" lui-même, appelé dans le contexte de la gestion automatique de la mémoire le *mutateur* puisque, du point de vue du ramasse-miettes, son seul effet est de modifier, le graphe des structures de données actives en mémoire. Nous présentons dans cette section les principales méthodes de mise en oeuvre des ramasse-miettes. Les trois méthodes classiques de récupération de la mémoire sont le comptage de références (*reference counting*), le marquage-balayage (*mark-and-sweep*) et la recopie (*copying*). Tous les algorithmes de ramasse-miettes existants sont basés sur ces trois archétypes et leurs dérivés.

3.2.1 Le comptage de références

L'algorithme de comptage de références est l'un des systèmes de gestion mémoire automatique les plus anciens [Gelernter *et al.*, 1960; Collins, 1960] et les plus simples, qui a été l'objet de nombreuses variantes [Jones et Lins, 1996].

Il consiste à associer à chaque cellule mémoire — ou, dans le cadre des langages à objets, à chaque objet — un compteur indiquant *combien de références pointent vers cet objet* (voir figure 3.1 page 55).

Ce compteur, qui est initialisé à 0 à la création de l'objet, est maintenu par le système de gestion mémoire. A chaque fois qu'une affectation crée une nouvelle référence (ou, pour reprendre la terminologie utilisée dans notre chapitre 4, un nouvel *alias*) vers l'objet, le compteur de références de ce dernier est automatiquement incrémenté de 1 (cf. objet `c` de la figure 3.2). Similairement, à chaque fois qu'une référence vers un objet est détruite, soit par affectation d'un nouvelle référence, soit parce que la variable qui la contenait a été détruite (variable locale en fin de routine), le compteur de références est décrémenté de 1 (cf. objet `b` de la figure 3.2). Lorsque le nombre de référence vers un objet atteint zéro, le système sait que l'objet n'est plus référencé depuis l'application et qu'il peut donc être désalloué ou recyclé (cf. objet `d` de la figure 3.2).

Lors de l'allocation, le système de gestion mémoire prend une partie de la mémoire libre dont il dispose pour en faire un nouvel objet et met son compteur de références à zéro. Puis, à chaque affectation de référence, il maintient les compteurs de références des objets impliqués comme indiqué ci-dessus. Enfin, à

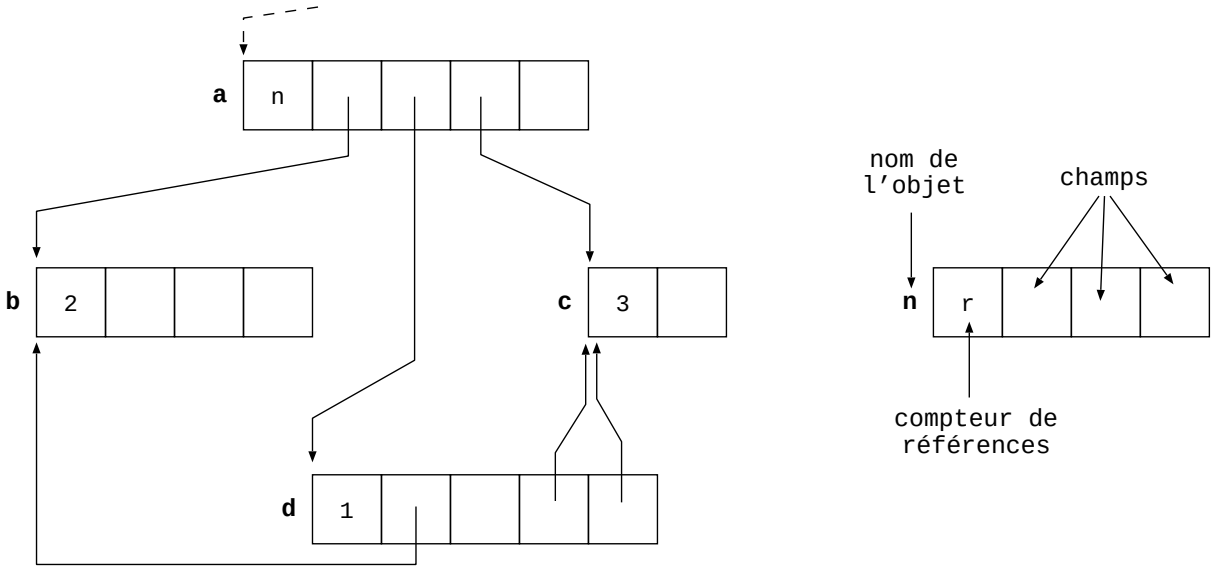


FIG. 3.1 – *Le comptage de références.*

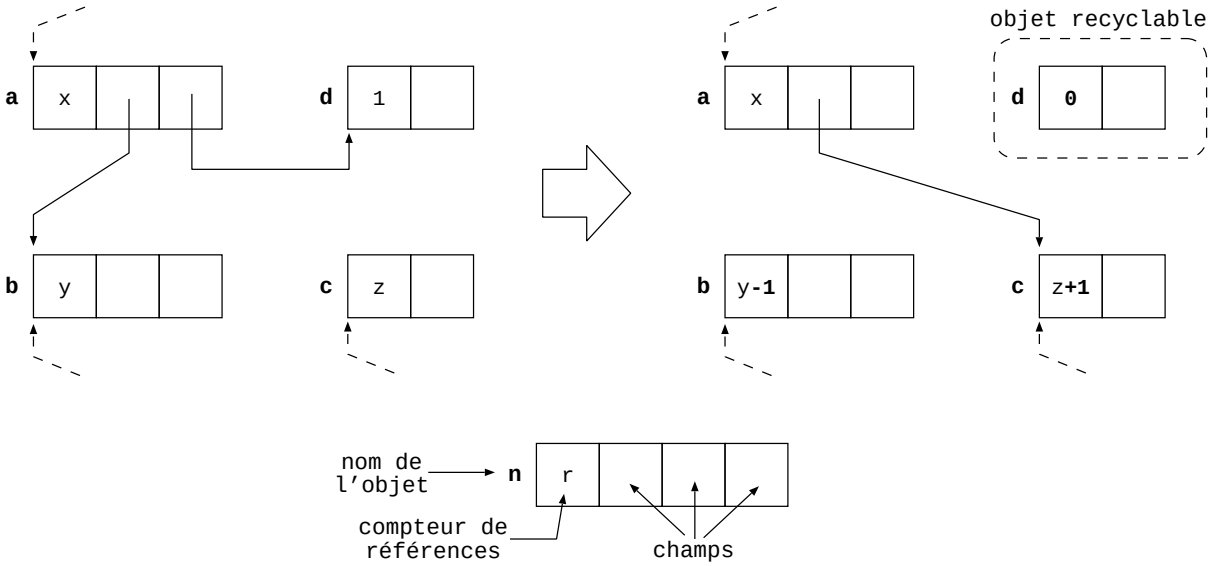


FIG. 3.2 – *Mise à jour des compteurs de références.*

Gauche: Graphe d'objets. **a** pointe sur **b** et **d**.
Droite: Grpahe d'objets après mise à jour du premier champ référence de **a** et nullification du second champ référence de **a**.

la mort de l'objet, celui-ci est retourné dans la mémoire libre (immédiatement, dans les versions classiques de l'algorithme, plus tard dans certaines autres).

Cet algorithme présente un certain nombre d'avantages. Tout d'abord, il est très simple à comprendre et à mettre en oeuvre. Il est ensuite tout naturellement incrémental, puisque l'ensemble du travail fourni pour gérer la mémoire est distribué tout au long de la vie du programme géré, et non pas concentré sur quelques périodes particulières. Il ne cause donc pas de pauses dans l'exécution du programme, ce qui est un atout fondamental dans le cas de programmes dont les temps de réponses sont critiques, comme par exemple les systèmes temps réel. Enfin, d'un point de vue plus technique, cet algorithme présente de très bonnes propriétés de localité, puisque les objets manipulés par le ramasse-miettes à un instant donné sont justement ceux utilisé par le programme. Ainsi, mis à part en phase d'allocation et de désallocation, le ramasse-miettes ne travaille que sur des objets déjà présents en mémoire et dans les caches, puisqu'actifs du point de vue du programme utilisateur. Ces avantages expliquent que des algorithmes basés sur le comptage de références aient été choisis pour l'implantation de nombreux langages, comme Smalltalk [Goldberg et Robson, 1983], Modula2+ [DeTreville, 1990], SISAL [Cann *et al.*, 1992], Perl [Wall et Schwartz, 1991] ainsi que de nombreuses applications, telles que les systèmes de gestions de fichiers de divers systèmes d'exploitation (UNIX, pour déterminer si un fichier peut être détruit), Adobe Photoshop, Netscape Communicator et sa version "logiciel libre" Mozilla [Mozilla.org, 1998], l'utilitaire UNIX `awk` [Aho *et al.*, 1988], etc... Le comptage de référence est également un système de ramassage de miettes qui très utilisé dans un contexte distribué (objets distribués en Modula-3 [Birrell *et al.*, 1994] et en Java [Sun, 2000a]), puisque les communications qu'il implique sont locales aux objets concernés par la mise à jour [Plainfossé et Shapiro, 1995; Jones et Lins, 1996].

En revanche, le comptage de références comporte un certain nombre d'inconvénients importants qui font qu'il est assez souvent écarté au profit d'autres techniques. La principale faiblesse du comptage de références est que cet algorithme est incapable, dans ses versions classiques, de récupérer les structures d'objets cycliques [McBeth, 1963]. En effet, lorsqu'une telle structure n'est plus référencée par le programme, les compteurs de références de ses objets ne sont pas à zéro, du fait des références entre objets de la structure (voir figure 3.3). Il est donc nécessaire d'adjoindre au comptage de références une autre technique qui, elle, est capable de gérer les cycles, telle que celles que nous expliquons en sections 3.2.2 et 3.2.3. En termes de mémoire, le comptage de référence nécessite dans le pire des cas un champ supplémentaire par objet, pour le compteur de références. Ce champ doit de plus être assez grand pour pouvoir représenter un nombre égal au total de références présentes dans le système. Ce coût n'est donc pas négligeable, surtout lorsqu'il s'applique à de très petits objets. En pratique, néanmoins, il est dans la majorité des implantations plus petit et réduit par diverses stratégies [Jones et Lins, 1996]. Le coût important en termes de temps d'exécution est par contre un reproche valide souvent fait au comptage de références. En effet, chaque fois qu'une affectation de référence est faite, les compteurs des objets source et destination doivent également être mis à jour, ce qui revient en gros à tripler le coût de l'affectation de référence. Enfin, on peut remarquer que le comptage de référence est un système très étroitement couplé au programme mutateur, du fait de la mise à jour à chaque affection de référence, et donc potentiellement plus fragile que des systèmes plus indépendants.

3.2.2 Le marquage-balayage

L'algorithme de marquage-balayage (*mark-and-sweep* ou parfois *mark-scan*) est le premier algorithme de gestion automatique de la mémoire développé [McCarthy, 1960], toujours très utilisé et objet lui aussi de nombreuses variantes.

Il consiste à traverser l'ensemble des objets *vivants*, actifs, et à les marquer pour ainsi pouvoir repérer les objets *morts*, inactifs, dont la mémoire peut être libérée ou recyclée. Plus précisément, le marquage-balayage simple est un algorithme en deux phases: d'abord le *marquage* qui traverse l'ensemble des objets vivants, puis le *balayage* qui récupère les objets non utilisés.

A chaque objet est associé un marqueur booléen, indiquant si l'objet est vivant (vient d'être marqué) ou non. A chaque fois qu'un objet doit être alloué, le système de gestion mémoire regarde s'il dispose de la mémoire nécessaire dans sa liste de "cellules" mémoire libres. Si oui, il fournit la mémoire nécessaire à l'objet, en initialisant son marqueur à faux (non marqué) et le programme continue son exécution

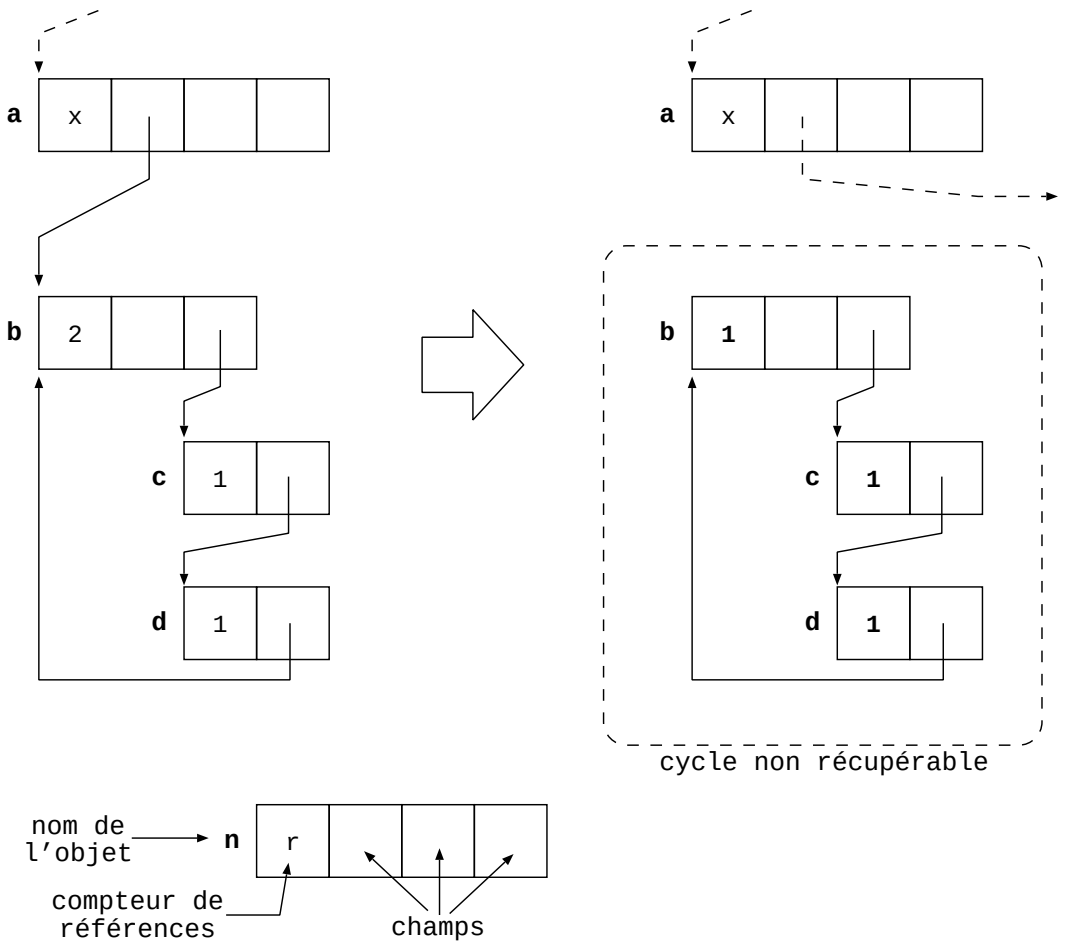


FIG. 3.3 – *Comptage de références et structures de données cycliques.*

Gauche: Graphe d'objets initial. Le champ référence de **a** pointe sur **b**.
Droite: Graphe d'objets après modification du champ référence de **a**.

sans interruption. Si le système ne peut immédiatement fournir la mémoire, un cycle de ramassage est déclenché qui interrompt l'exécution du mutateur, afin de récupérer la mémoire nécessaire. Ce cycle s'effectue en deux phases: le *marquage* puis le *balayage*. La figure 3.4 page 59 illustre ce processus.

Une *racine* du graphe d'objets vivants est une référence vers un objet que le mutateur peut *directement* manipuler. Ces références peuvent se trouver dans les registres du processeur, dans la pile d'exécution qui contient les variables locales et temporaires, ainsi que dans des variables globales. Le *marquage* consiste, à partir de l'ensemble des racines du graphe d'objets, à calculer la fermeture transitive du graphe d'objets en suivant les références d'un objet à un autre. Chaque objet rencontré ainsi est marqué *vivant*, et le marquage continue en suivant les références contenues dans cet objet afin de trouver de nouveaux objets vivants. Les objets `a`, `b`, `e` et `f` de la figure 3.4 sont ainsi marqués de proche en proche. Bien entendu, lorsqu'une référence pointe vers un objet déjà marqué, il n'est pas nécessaire de relancer le marquage à partir de cet objet, puisque cela a déjà été fait. C'est par exemple le cas de la référence de `f` vers `e`, l'objet `f` ayant nécessairement été marqué après l'objet `e`. Une fois cette phase de marquage terminée, tous les objets vivants ont été marqués, alors que tous les objets non marqués sont *morts*, c'est à dire non utilisés par le programme, et peuvent être recyclés. Dans l'exemple de la figure 3.4, les objets `c`, `d` et `g` sont recyclables car non marqués.

La phase de *balayage* débute alors. Elle consiste à parcourir linéairement l'ensemble de la portion de mémoire gérée par le ramasse-miettes (voir le bas de la figure 3.4) et à examiner quels sont les objets (ou cellules) qui sont non marqués. La mémoire correspondante est récupérée, c'est à dire mise dans une liste de cellules libres, pouvant resservir à l'allocation de nouveaux objets. Les objets marqués, vivants, ne sont pas touchés, à part leur marqueur qui est repositionné à faux (non marqué), de façon à préparer le marquage du prochain cycle de ramassage de miettes. A la fin de la phase de balayage, le système de gestion mémoire alloue l'objet initialement demandé par le mutateur en puisant dans sa liste de cellules libres (fraîchement libérées) et rend la main au mutateur.

On voit donc que contrairement à l'algorithme à comptage de références décrit au 3.2.1, le marquage-balayage ne récupère pas la mémoire dès qu'un objet devient mort, mais de façon asynchrone, c'est à dire plus tard, lorsqu'un nouveau cycle de ramasse-miettes sera déclenché. Les exécutions du mutateur et du ramasse-miettes sont donc nettement moins entrelacées que dans le cas du comptage de référence.

Cet algorithme présente de grands avantages sur le comptage de références, qui ont mené à son adoption pour divers systèmes, comme certaines implantations des langages Miranda [Turner, 1985], Prolog [Appleby *et al.*, 1988], Lisp [Zorn, 1989], C/C++ [Boehm et Weiser, 1988] et Java — JDK de SUN [Sun, 2000c].

Le premier avantage est que cet algorithme gère naturellement et sans aucun problème les cycles dans le graphe d'objets, et donc les structures de données présentant de tels cycles. Ceci simplifie la réalisation d'un système complet, puisque contrairement au comptage de références qui réclame un système d'appoint pour les cycles, le marquage-balayage suffit pour gérer tous les graphes d'objets qui peuvent se présenter, que ce soit avec ou sans cycles. De plus, et surtout, aucun surcoût n'est associé à la manipulation des pointeurs, ce qui pénalise moins le mutateur. Enfin, le travail en phases et non pas durant l'exécution du mutateur fait que le mutateur et le système de gestion de la mémoire sont un peu plus découplés.

Le marquage-balayage souffre cependant d'un certain nombre d'inconvénients. Tout d'abord, comme nous l'avons indiqué, il interrompt totalement le mutateur pendant tout le cycle de ramassage de miettes (marquage plus balayage). La durée de ces cycles croissant en fonction de la mémoire active du mutateur, plus ce dernier est gros et plus la part de ressources processeur qui lui est réservée diminue (phénomène dit "de *trashing*"). Ceci peut conduire à des pauses (temps pendant lequel le mutateur ne s'exécute pas) de longue durée, ce qui peut être inacceptable pour des programmes hautement interactifs ou temps-réel. De plus, cet algorithme tend à fragmenter la mémoire, puisqu'il la découpe en cellules (objets) qui lorsqu'elles sont réutilisées peuvent elles aussi être redécoupées pour des objets plus petits. Ceci peut rend l'algorithme de marquage-balayage indésirable dans le cadre d'utilisations très prolongées, par exemple avec des serveurs.

3.2.3 La recopie

Les ramasse-miettes par recopie (*copying garbage collectors*) représentent la dernière grande catégorie de systèmes de gestion mémoire automatique.

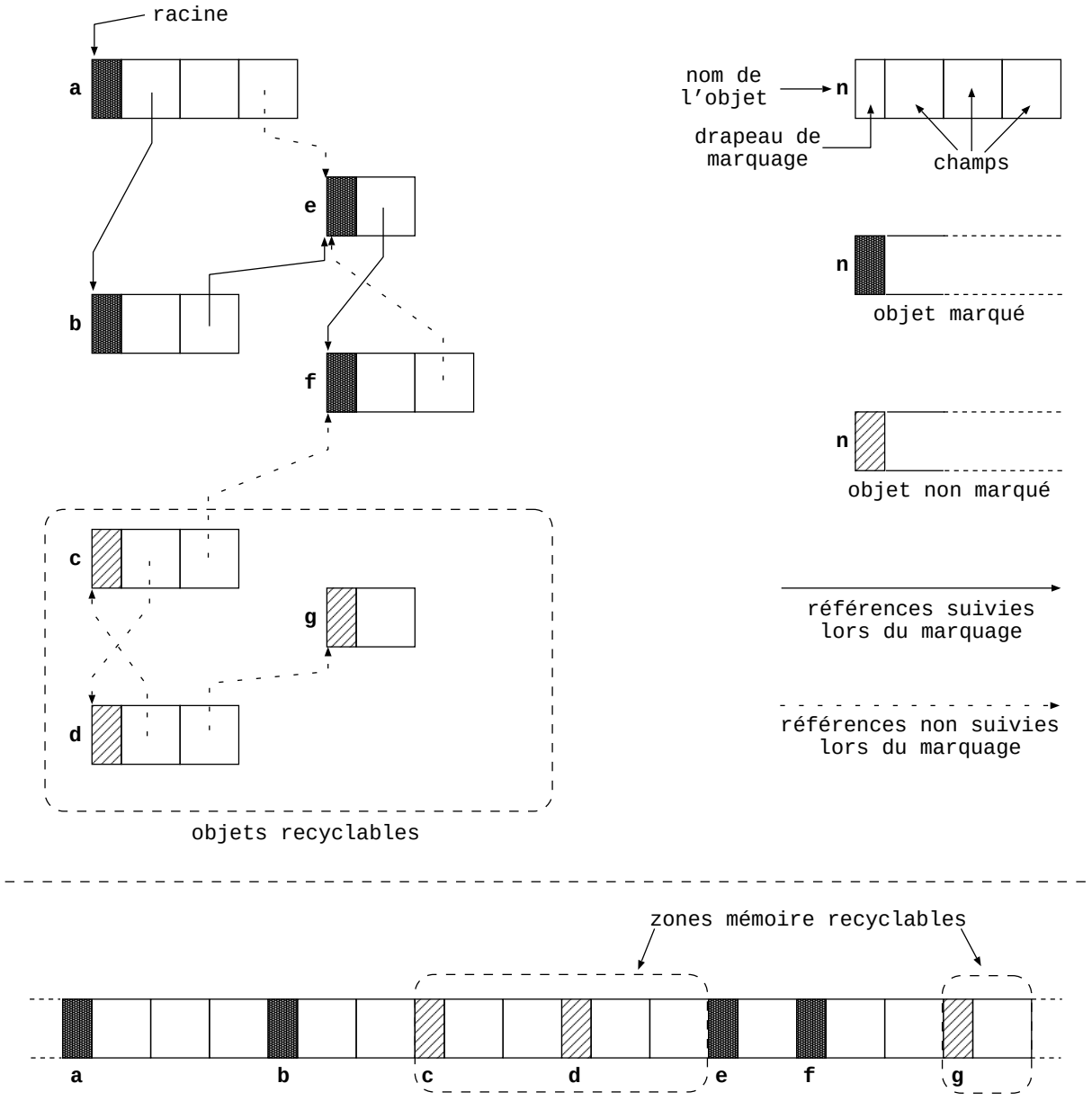


FIG. 3.4 – *Le marquage-balayage.*

Haut: Graphe d'objets après la phase de marquage.
Bas: Représentation de la mémoire correspondant à ce graphe.

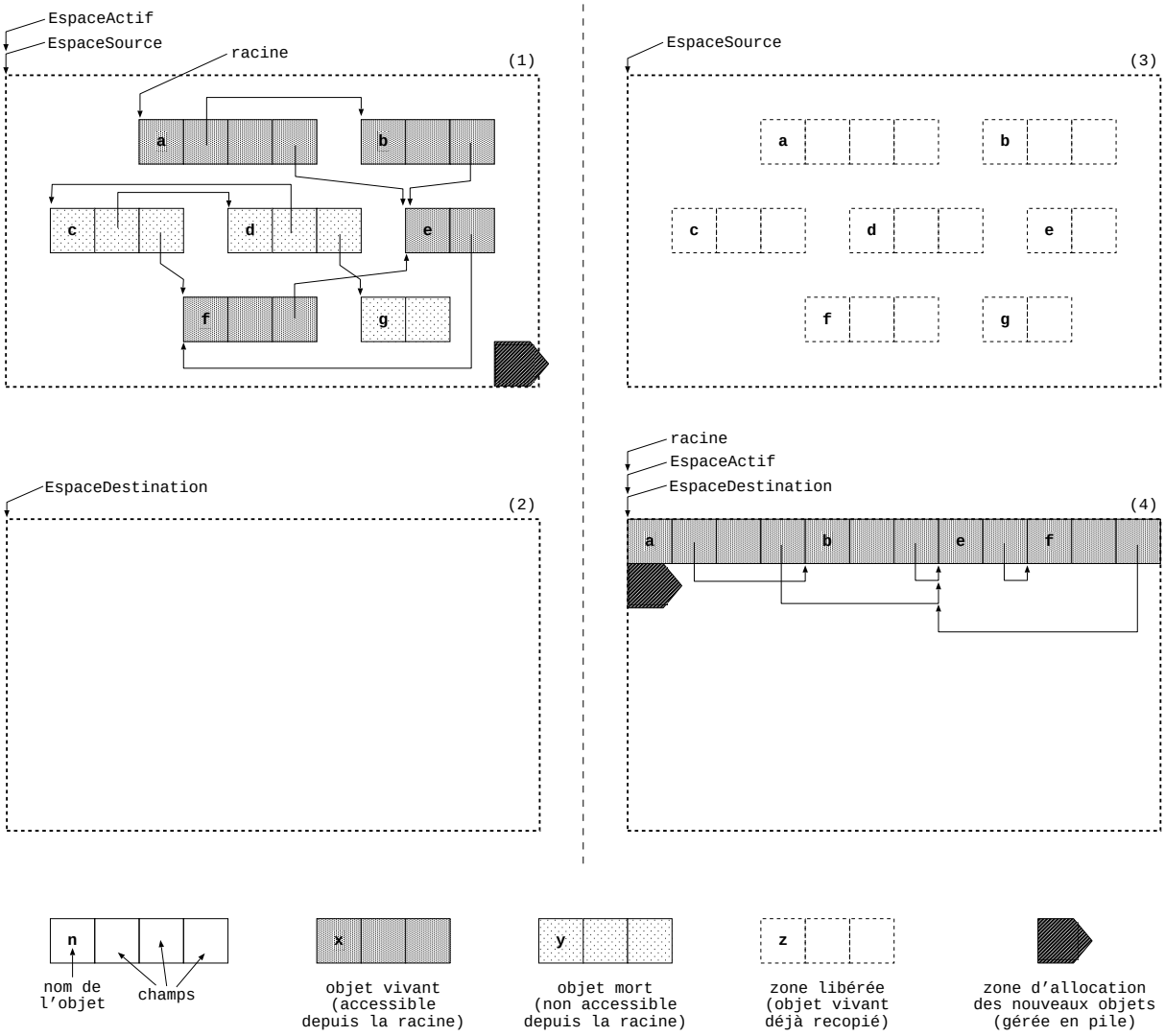


FIG. 3.5 – *Le ramassage de miettes par recopie.*

Gauche: État de la mémoire juste avant la recopie.
Droite: État de la mémoire juste après la recopie (les pointeurs `EspaceSource` et `EspaceDestination` n'ont pas encore été échangés).

L'algorithme par recopie d'objets consiste à doubler l'espace mémoire géré par le ramasse-miettes et à le diviser en deux sous-espaces, l'un actif, l'autre mort. Les objets vivants de l'espace actif sont recopiés vers l'espace mort, qui devient à son tour l'espace actif, et ainsi de suite (voir figure 3.5 page 60).

Plus précisément, les objets utilisés par le mutateur sont alloués dans l'espace actif, géré classiquement sous forme d'une pile. Lorsqu'une allocation ne peut plus être effectuée parce que l'espace actif semble plein (hauteur de pile maximale atteinte), un cycle de ramassage de miettes est déclenché, interrompant l'exécution du mutateur (voir partie (1) de la figure 3.5).

Celui-ci consiste à tracer, de façon similaire au marquage de l'algorithme de marquage-balayage (section 3.2.2), les objets encore vivants dans l'espace actif (espace source, ou *FromSpace*) et à les recopier, — soit au fur et à mesure du marquage, soit après — dans dans le second sous-espace mort (espace destination, ou *ToSpace*), lui aussi géré en pile (voir parties (1) et (4) de la figure 3.5). Comme le sous-espace destination ne contenait pas d'objets alloués au mutateur, les objets de l'espace source peuvent sans danger être recopiés vers le "bas" de l'espace destination (lui aussi géré comme une pile) puis en montant, de façon contiguë.

Le premier objet vivant rencontré, quelle que soit sa position dans l'espace source, est donc recopié en bas de l'espace destination. Le second objet vivant trouvé, toujours quelle que soit sa place dans l'espace source, est recopié dans l'espace destination juste au dessus du précédent, et ainsi de suite. Ainsi, même si les objets vivants ne sont pas contigus dans l'espace source, étant entrelacés avec les objets morts, ils se retrouvent rangés de façon contiguë et donc sans perte de place dans l'espace destination (voir partie (4) de la figure 3.5). Ce dernier contient, à la fin du parcours des objets vivants, une réplique du graphe d'objets vivants du mutateur, les objets morts n'étant bien sûr pas recopiés. La mémoire correspondant à la place des objets morts se retrouve au dessus des objets vivants dans l'espace destination, alors qu'elle était dispersée entre les objets vivants dans l'espace destination. Elle peut donc être immédiatement réutilisée, en inversant les rôles de l'espace source et de l'espace destination et en reprenant une allocation mémoire en sommet du nouvel espace actif (c'est à dire en inversant `EspaceSource` et `EspaceDestination` dans les parties (3) et (4) de la figure 3.5). Bien entendu, il est capital lors de cette recopie des objets vivants de l'espace source vers l'espace destination de maintenir la cohérence des références entre les objets. Les pointeurs qu'ils contiennent doivent donc être mis à jour afin de ne plus pointer vers les objets de l'espace source mais vers leur copie dans l'espace destination. Les détails de cette gestion des références dans le cas des ramasse-miettes qui bougent les objets ne présentant pas d'intérêt particulier dans le cadre de cette thèse, ils ne seront pas présentés ici. Le lecteur intéressé pourra trouver les précisions nécessaires dans la bibliographie, notamment dans [Jones et Lins, 1996].

L'avantage le plus évident — et qui est la raison d'être — des algorithmes de gestion mémoire par recopie, est l'absence de fragmentation, quelle que soit la durée d'exécution du mutateur. En effet, la recopie faisant office de compactage de la mémoire, la fragmentation potentiellement apparue lors de l'exécution du mutateur disparaît totalement au cours du cycle de ramassage suivant. Les ramasse-miettes par recopie sont donc considérés comme très sûrs, notamment pour les programmes de type "serveurs". Un autre avantage non négligeable est le fait que le coût d'une allocation réussie est extrêmement bas: il s'agit simplement d'une comparaison de pointeurs afin de savoir si la mémoire disponible (entre sommet courant de l'espace destination et sommet maximum) est suffisante pour l'objet à allouer, suivie d'une incrémentation de ce sommet courant de la taille de l'objet. Ce coût reste de plus constant même lorsque les objets à allouer ont des tailles différentes ou sont de très grande taille. Enfin, comme le coût de la recopie est proportionnel à la quantité de mémoire vivante, ce type d'algorithmes est très approprié pour les nombreux systèmes où le taux de survie des objets est faible. Les avantages des algorithmes de gestion mémoire par recopie d'objets ont donné lieu à leur adoption pour les implantations de divers langages et systèmes, tels que Lisp [Minsky, 1963], ML [Appel, 1992], Eiffel [ISE, 1999], Java (machine virtuelle HotSpot du JDK 1.3 de SUN) [Sun, 2000b], etc...

Cependant, les algorithmes de gestion mémoire par recopie présentent au moins un grand inconvénient, lui aussi évident. Il s'agit de la mémoire requise avec ce système, qui est du double de la mémoire nécessaire au mutateur, puisqu'au moment du ramassage de miettes il est nécessaire d'avoir en plus de l'espace actif un espace destination de même taille. Un autre inconvénient est que, lorsqu'on considère deux cycles de ramassage de miettes, l'algorithme par recopie touche à *toutes* les zones mémoires des deux espaces (actif et inactif), c'est à dire à deux fois plus de zones que les algorithmes précédents. Il y a donc de plus forts risques de délais dû à des défauts de pages mémoire dans un système à recopie d'objets. La

localité des objets manipulés par un système à recopie est également nettement moins bonne que celle des objets manipulés par les algorithmes précédents, ce qui impose une pénalité supplémentaire en temps d'exécution. Ces problèmes de performances liés aux problèmes de défauts de caches et de défauts de pages sont notamment étudiés dans [Zorn, 1991], qui indique que sur ces aspects les algorithmes à recopie sont sensiblement moins performants que les algorithmes à marquage-balayage.

3.2.4 Autres algorithmes et variantes

Les trois types d'algorithmes de gestion automatique de la mémoire présentés ci-dessus sont bien entendu des catégorie assez générales, pour lesquelles seul l'algorithme "de base" a été détaillé. Il en existe des variantes extrêmement nombreuses, qui tentent de pallier les défauts que nous avons mentionnés. Il serait beaucoup trop long et hors du cadre de cette thèse d'expliquer chacune de ces variantes, nous n'en indiquerons ici que les grandes tendances. Des précisions supplémentaires et relativement exhaustives peuvent être trouvées dans [Jones et Lins, 1996].

De nombreuses variantes des algorithmes à comptage de références ont pour but de pouvoir collecter les cycles d'objets. Pour ce faire, le comptage de référence est généralement couplé à un second algorithme, de traçage, soit de type marquage-balayage, soit à recopie. Celui-ci est généralement déclenché périodiquement.

Les variantes du marquage-balayage s'attellent souvent au problème de la fragmentation de la mémoire. La solution la plus naturelle consiste à associer au marquage-balayage un second algorithme fonctionnant par recopie, qui pourra par exemple être appelé périodiquement, ou seulement lorsqu'une trop grande fragmentation est détectée. Une importante évolution du marquage-balayage, appelée marquage-compactage (*mark-and-compact*), est en fait un type d'algorithme hybride, intégrant une recopie partielle au marquage-balayage. Le marquage-compactage recopie les objets, mais n'utilise pas deux sous-espaces disjoints comme le font les algorithmes à recopie. Seul l'espace utilisé par le programme est effectivement compacté, en bouchant les trous laissés par les objets morts ou les objets vivants déplacés à l'aide d'objets vivants recopiés d'ailleurs. Ceci peut imposer de bouger de nombreux objets vivants, éventuellement en plusieurs passes, afin d'atteindre le niveau de "remplissage" souhaité. Le compactage est donc plus compliqué et potentiellement plus cher que la recopie simple des algorithmes à plusieurs espaces, mais permet de n'utiliser qu'un seul espace mémoire et donc de diminuer l'empreinte du programme.

Les variantes des algorithmes à recopie se sont focalisées sur la réduction du coût de la recopie des objets actifs. La principale évolution des algorithmes à recopie est le ramassage de miettes à générations (*generational garbage collection*). Le but des générations est de tenir compte de la "localité temporelle" des objets et de la transposer spatialement. En effet, de nombreuses observations montrent que les objets se divisent globalement en deux catégories: les objets temporaires, qui vont être alloués et mourront très peu de temps après, et les objets "permanents", qui ont une très grande durée de vie. Ces derniers, devant être recopiés à chaque cycle, sont une charge importante pour les ramasse-miettes à recopie. De plus, de nombreuses recherches [Deutsch et Bobrow, 1976; Foderaro et Fateman, 1981; Wilson, 1994; Zorn, 1989; Sansom et Jones, 1993; Hayes, 1991; Appel, 1992; Barett et Zorn, 1993] ont montré que dans la plupart des langages la très grande majorité des structures de données ou objets alloués sont des données temporaires, ce qui tend à valider l'*hypothèse générationnelle faible* [Ungar, 1984], selon laquelle *la majorité des objets meurent jeunes.*

Il apparaît donc intéressant de *séparer* physiquement ces deux catégories d'objets, en les mettant dans des sous-espaces (des générations) différents. Ainsi, l'algorithme de collection est souvent déclenché sur l'espace contenant des objets jeunes, où l'on sait que les chances de récupérer la mémoire sont maximales et le coût de recopie minimal. Au contraire, la ramassage de miettes sur les objets vieux, donc à durée de vie potentiellement plus grande, est fait beaucoup plus rarement. Bien entendu, il est impossible, sauf dans des cas particuliers (information fournie explicitement par le développeur, à l'aide de constructions du langage, par exemple) de savoir lors de l'allocation d'un objet si celui-ci va vivre longtemps ou non. Un point important des algorithmes à générations est donc le problème de la promotion d'un objet d'une génération jeune vers une génération plus âgée. De très nombreuses heuristiques existent à cet effet, souvent basées sur la taille de l'objet, ainsi que sur son histoire (nombre de cycles de ramassage de miettes auquel l'objet a survécu...). Les algorithmes à générations sont en général assez complexes à mettre en oeuvre, mais ils peuvent donner de bonnes performances.

Un autre aspect qui a donné lieu à de nombreuses variantes des algorithmes de ramassage de miettes est l'*incrémentalité*, c'est à dire le fait que le coût du ramasse-miettes puisse être réparti sur l'ensemble de l'exécution du programme de façon à éviter d'interrompre trop longtemps le mutateur. Comme nous l'avons indiqué, les algorithmes à comptage de référence sont naturellement incrémentaux. Ceci n'est plus vrai lorsque le passage à zéro d'un compteur de références provoque la libération et la récupération d'un très grand sous-graphe d'objets. Des solutions ont donc été cherchées pour fractionner cette libération et résoudre ce problème d'incrémentalité. Dans les autres algorithmes en revanche, l'incrémentalité est un problème majeur et des variantes plus incrémentales ont vu le jour. Elles consistent en diverses techniques pour ne plus faire une collection en une seule phase, mais en limitant le temps maximal d'interruption du mutateur, ou en effectuant une partie du ramassage de miettes à chaque allocation. Bien entendu, comme le mutateur et le ramasse-miettes s'interrompent mutuellement, il est nécessaire que chacun informe l'autre dans le cas où il fait des modifications sur des objets manipulés par l'autre. Ceci ajoute donc une complexité non négligeable dans ces algorithmes, ainsi qu'une augmentation du coût global de la gestion mémoire.

Afin de résoudre, entre autre, ces problèmes d'incrémentalité, toute une catégorie d'algorithmes *concurrents*, c'est à dire s'exécutant *en même temps* que le mutateur, au lieu de l'interrompre, ont été l'objet de recherches. Celles-ci touchent non seulement à la gestion mémoire, mais aussi à de nombreux problèmes liés au parallélisme et, débordant clairement le cadre de cette thèse, ne seront pas abordées ici. De même, on peut mentionner les ramasse-miettes *distribués*, que nous avons brièvement évoqués, et qui, eux, en plus de la gestion mémoire et de certains problèmes de concurrence, s'attaquent aussi à des problèmes liés aux réseaux, et constituent donc un domaine à eux seuls.

Enfin, on peut mentionner des ramasse-miettes qui tentent de tirer parti de l'architecture du matériel, qui a une influence non négligeable sur les performances de ces algorithmes. Ces algorithmes sont malheureusement souvent liés à une architecture particulière et sont donc soit non portables, soit portables mais avec des performances qu'on ne peut généraliser d'une architecture à l'autre.

3.3 Schéma de génération automatique de ramasse-miettes spécialisé

Eiffel étant un langage de haut niveau récent, ses implantations comprennent un ramasse-miettes afin de faciliter le travail des développeurs. Comme en Java, de par la définition même du langage, la gestion manuelle de la mémoire par le développeur est proscrite. La charge de mettre au point le ramasse-miettes incombe donc aux concepteurs de compilateurs Eiffel.

Nous avons conçu pour SmallEiffel un système de gestion mémoire novateur, basé sur les résultats des travaux sur la génération de code spécialisé précédemment présentés dans ce mémoire. Ainsi, lorsqu'il compile une application, quelle qu'elle soit, SmallEiffel génère, en plus du code de ce mutateur, le code correspondant à un ramasse-miettes spécialement adapté à ce mutateur. Pour la génération de ce ramasse-miettes spécialisé, nous avons cherché à étendre et à généraliser à la gestion mémoire les concepts que nous avions développés dans le cadre de la liaison dynamique (cf. chapitre 2). Nous nous sommes ainsi intéressés à la spécialisation des routines du ramasse-miettes. De même, nous avons voulu utiliser au maximum les résultats de la prédiction de type présentés en section 2.3 pour adapter les structures de données du ramasse-miettes.

On a pu voir, d'après la présentation faite à la section 3.2 des algorithmes de ramasse-miettes qui existent actuellement, que tous les systèmes présentaient des avantages et inconvénients, et qu'il n'y avait aucun algorithme qui ressortait comme *le* meilleur algorithme. Le choix de l'un ou l'autre doit s'effectuer non seulement en fonction des caractéristiques intrinsèques de l'algorithme mais également en tenant compte de celles de l'application dont on veut gérer la mémoire. Les critères de développement sont aussi à considérer, comme la facilité d'implantation, les possibilités d'extension de l'algorithme, son potentiel d'optimisation, etc.

Le type d'algorithme que nous avons choisi de générer avec SmallEiffel est un ramasse-miettes à marquage-balayage. Ce choix se justifie par plusieurs raisons. Tout d'abord, l'implantation d'un ramasse-miettes représentant un effort considérable, nous avons préféré choisir un algorithme relativement simple,

moins complexe par exemple qu'un algorithme à générations. Nous avons cependant tenu à ce que cet algorithme ait de bonnes caractéristiques en termes de performances, ce qui est le cas des algorithmes à marquage-balayage, marquage-compactage et à recopie. Il nous est également apparu raisonnable, au moins dans un premier temps, de ne pas choisir un algorithme qui déplace les objets, c'est à dire d'éviter le marquage-compactage et les algorithmes à recopie. En effet, il est fréquent d'interfacer des programmes Eiffel avec du C, pour pouvoir utiliser certaines bibliothèques, ou pour certaines parties de bas niveau du compilateur. L'utilisation d'un ramasse-miettes bougeant les objets nous aurait obligé, et surtout aurait obligé les utilisateurs du compilateur, à protéger leurs pointeurs C par un niveau d'indirection supplémentaire. Ceci aurait introduit un fardeau et un coût en performances que nous souhaitions éviter. Enfin, le marquage-balayage, bien que de complexité raisonnable, connaît de nombreuses variantes plus ou moins perfectionnées. Il nous est donc possible avec ce choix de faire évoluer, si nécessaire, notre système de génération de ramasse-miettes, voire d'aller vers un système à marquage-compactage si les problèmes de fragmentation apparaissent trop importants.

Le ramasse-miettes généré par SmallEiffel est donc un collecteur par marquage-balayage classique, partiellement *conservateur* (voir [Boehm, 1993]). Les informations fournies par l'algorithme de prédiction de type présenté dans la section 2.3, qui permet d'analyser lors de la compilation les relations inter-classes sont utilisées pour classer les objets en fonction de leur type et adapter statiquement la plus grande partie du code du ramasse-miettes généré. Ainsi, la gestion mémoire se base principalement sur des routines spécialisées en fonction du type des objets qu'elles manipulent et donc plus efficaces.

Dans les sections suivantes, nous expliquons en détail selon quel schéma notre compilateur génère automatiquement ce ramasse-miettes à marquage-balayage adapté au mutateur compilé. Cette présentation reprend en partie la synthèse de nos travaux que nous avons présentée [Colnet *et al.*, 1998a] à ISMM, conférence mondiale sur les ramasse-miettes ainsi que dans diverses publications [Colnet *et al.*, 1998c; Coucaud *et al.*, 1999a].

3.4 La gestion des objets de taille fixe

3.4.1 L'allocation

L'allocateur que nous avons mis en oeuvre prend avantage d'informations sur la structure des objets fournies par le mécanisme de prédiction de type de SmallEiffel. Comme cet allocateur sait statiquement quels types d'objets sont alloués, les objets de taille fixe sont regroupés (*segregated*) par type, plutôt que par simplement par taille comme dans la plupart des autres algorithmes à ségrégation.

A chaque type vivant est associée une collection spécifique de blocs mémoire typés. De cette façon, un bloc ne contient qu'un seul type d'objets (voir figure 3.6 page 65), dont la taille est connue lors de la compilation et codée "en dur" partout où elle intervient. Ainsi, le ramasse-miettes de SmallEiffel n'a pas besoin d'un champ supplémentaire pour stocker la taille de l'objet. Chaque bloc est une zone mémoire qui contient un nombre fixe de "places" pour les objets du type correspondant. Une entête de bloc comprend d'autres informations spécifiques au type, par exemple des pointeurs donnant accès aux fonctions de marquage et de balayage adaptées.

Après avoir expérimenté diverses configurations et mesuré leurs performances, il s'avère que des blocs de taille fixe correspondant à la taille d'une page représentaient un bon compromis entre une allocation rapide et une petite empreinte mémoire.

Chaque type a sa propre méthode d'allocation et est associé à un ensemble de blocs mémoire contenant des objets vivants de ce type. L'un de ces blocs (le dernier alloué depuis le système) est le Bloc d'Allocation Linéaire (BAL) pour ce type. Contrairement aux autres blocs associés au type, le BAL est géré à la manière d'une pile[25], avec un Pointeur d'Espace Libre (PEL) désignant le début de l'espace libre disponible pour les objets de ce type. Quand un objet doit être créé, la méthode d'allocation associée à son type essaie de l'allouer directement à partir du BAL correspondant, simplement en incrémentant le PEL du type par la taille de l'objet. Une telle allocation linéaire typée est probablement la plus rapide que l'on puisse imaginer.

25. *stricto sensu*, il ne s'agit pas d'une pile, puisque le PEL ne peut que monter et qu'il est possible que des "trous" apparaissent en fond de pile.

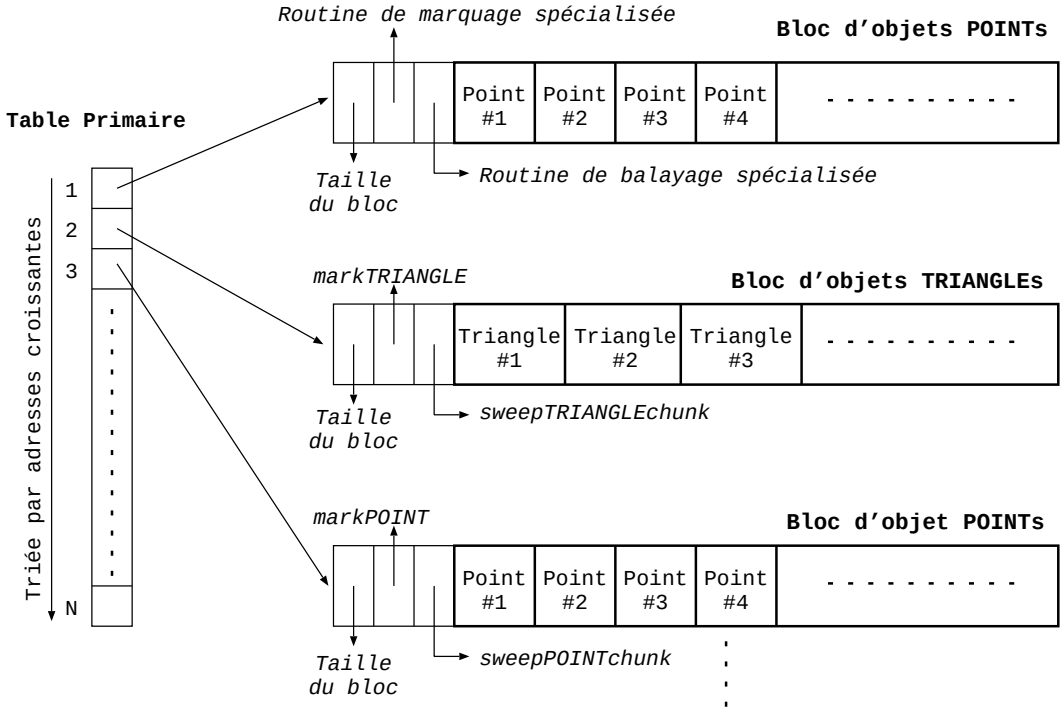


FIG. 3.6 – *Organisation mémoire générale.*

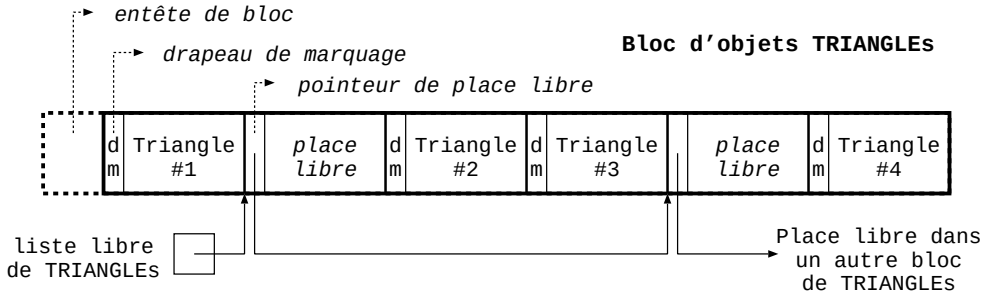


FIG. 3.7 – *Détail d'un bloc d'objets de taille fixe.*

Lorsque cette allocation "en pile" est impossible, car il n'y a plus assez de mémoire disponible au dessus du PEL, la méthode d'allocation regarde dans la liste libre du type, qui chaîne les "places" libres à travers l'ensemble des blocs associés au type (voir figure 3.7). Si la liste n'est pas vide, la première place sur laquelle elle pointe est enlevée de la liste et utilisée pour le nouvel objet. La ségrégation des objets par type permet ainsi de chercher une place libre pour un coût faible et constant.

S'il n'y a plus de place dans aucun des blocs correspondant au type, un cycle de ramassage de miettes peut être déclenché, afin de récupérer les objets non utilisés et donc de fournir une place pour le nouvel objet. Dans le cas où le cycle de ramassage de miettes ne fournit pas la mémoire demandée, un nouveau BAL pour les objets du type considéré doit être alloué (demandé au système), l'ancien BAL devenant alors un banal bloc associé au type.

Pour ne pas déclencher un cycle de ramassage de miettes complet à chaque fois qu'aucune mémoire libre ne peut être trouvée tant dans le BAL que dans la liste des objets libres du type, un critère additionnel — le "plafond" mémoire — est considéré. Il représente la marge libre pour les objets de taille fixe, c'est à dire la quantité de mémoire allouée en deçà de laquelle le ramasse-miettes n'est pas activé, un nouveau bloc mémoire étant alloué (demandé au système).

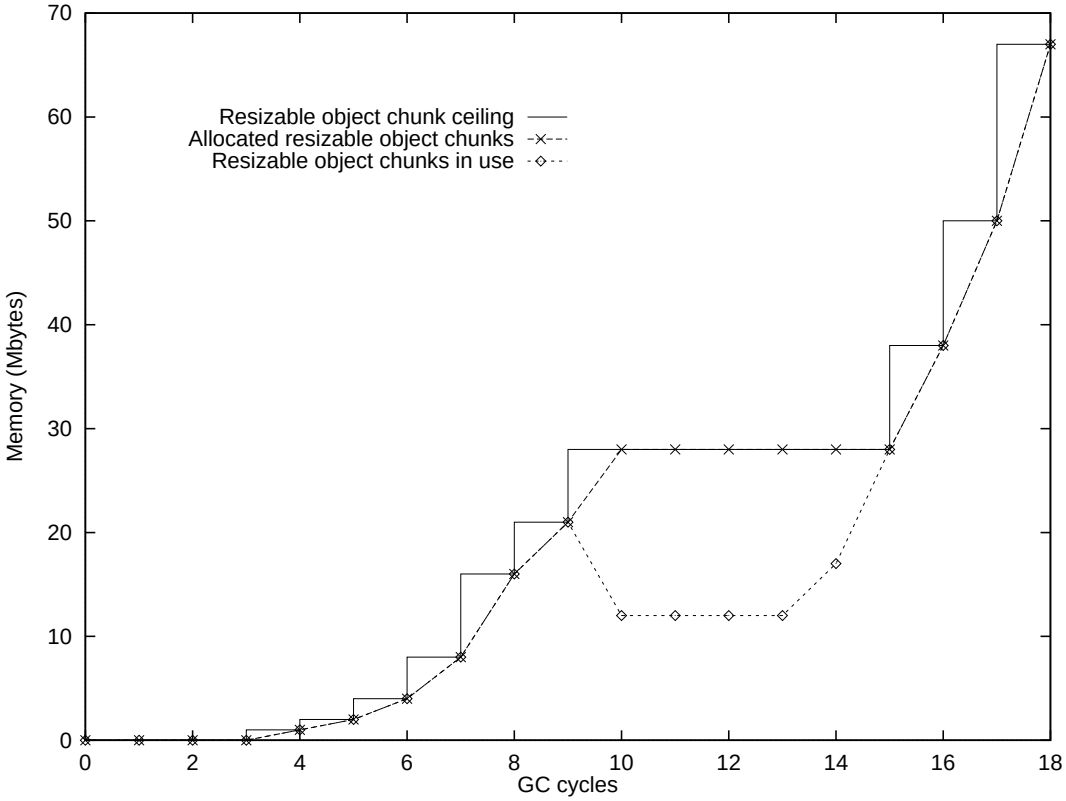


FIG. 3.8 – *Mise à jour du plafond d'allocation mémoire.*

Grâce à l'algorithme de prédiction de type exécuté lors de la compilation, une valeur initiale peut être affectée au plafond, en considérant le nombre de types vivants dans le système. Par exemple, un programme avec peu de types concrets vivants a un plafond plus bas qu'un autre en ayant beaucoup. En pratique, le plafond est proportionnel au nombre de types concrets vivants. Certains objets Eiffel ayant des propriétés spécifiques dans le système (l'unicité par exemple) sont gérés d'une façon particulière (voir la section 3.6).

Une incrémentation constante du plafond est bien sûr trop simpliste pour offrir de bonnes performances, particulièrement parce qu'elle ne considère pas l'historique des allocations précédentes. Une extrapolation polynomiale basée sur la mémoire allouée semble bien adaptée car elle permet de mettre à jour le plafond en fonction de l'évolution précédente des besoins en mémoire, même lorsqu'une augmentation très brutale se produit. Nous avons cependant obtenu les meilleurs résultats, tant en ce qui concerne la vitesse que la taille mémoire, avec un simple facteur de croissance. Ainsi, après chaque cycle de ramassage de miettes, le programme est autorisé à allouer de nouveaux blocs représentant jusqu'à une certaine proportion de la taille des blocs déjà utilisés, afin de lui garantir une marge suffisante.

La figure 3.8 illustre le comportement de ce plafond dans un programme de test qui comporte trois phases d'exécution différentes. Durant la première (cycles de collection 0 à 9), le programme alloue beaucoup de mémoire. Durant cette phase, le plafond est rapidement monté (il est doublé après chaque appel du ramasse-miettes quand la mémoire allouée est en deçà de 10 MO, et accru de 30 % après cette limite). Pendant la deuxième phase (cycles 10 à 14), le programme alloue des objets sans garder de référence sur eux. Le plafond n'est donc pas modifié, puisque le ramasse-miettes recycle suffisamment de blocs mémoire. Enfin, le programme entre dans une nouvelle phase d'allocation (cycles 15 à 18) qui provoque une nouvelle série d'augmentations du plafond de 30 %.

3.4.2 Le marquage

Le processus de marquage nécessite l'addition d'informations à chaque objet afin de savoir s'il est marqué ou non. Cette information supplémentaire est implantée sous la forme d'une entête ajoutée à chaque objet et d'une taille d'un mot machine. Elle contient le bit de marquage, que nous avons codé comme un drapeau de type entier pour des raisons de simplicité. Pour de futures versions, nous avons en vue une implantation plus compacte utilisant un champ de bits associé à chaque bloc.

La phase de marquage est basée sur deux étapes, distinguées par l'origine des références suivies pour marquer les objets: l'une traite les références racines trouvées dans la pile d'exécution, l'autre les références internes à la structure des objets[26].

La première est une étape *conservatrice*, c'est à dire qu'elle explore une zone (la pile d'exécution) en entier, afin d'y trouver les mots machines qui sont des références vers des objets. La seconde, au contraire, est *exacte* (ou *type-accurate*), car elle travaille avec des références dont l'emplacement est parfaitement connu. Ceci explique que nous caractérisons le ramasse-miettes que nous avons implanté dans SmallEiffel comme un ramasse-miettes *semi-conservateur*, contrairement, par exemple, au ramasse-miettes BDW, totalement conservateur [Boehm, 1993].

3.4.2.1 Recherche des racines

En effet, parmi toutes les données contenues dans la pile (valeurs scalaires, adresses de retour des fonctions, adresses d'objets, etc.), seules les références vers des objets présentent un intérêt pour le marquage.

En Eiffel, les objets normaux sont toujours alloués dans le tas et, dans le cas où un objet normal est référencé depuis une variable locale ou un argument de routine, seul un pointeur vers sa position en mémoire est empilé. En revanche, les objets `expanded` d'Eiffel [Meyer, 1992] peuvent être alloués directement dans la pile. Ils peuvent contenir aussi bien des références sur des objets que des objets eux-mêmes.

Donc en examinant l'ensemble de la pile il devrait être possible de retrouver toutes les références vers des objets vivants dans le tas. La recherche de références internes vers d'autres objets vivants est expliquée plus loin.

La profondeur de la pile[27] est un facteur important à prendre en considération. Un bon calcul de la zone de pile à explorer peut permettre d'économiser un certain nombre d'analyses de mots machine à chaque appel du ramasse-miettes. SmallEiffel considère l'adresse de l'objet associé à la classe racine comme le bas de la pile et la dernière variable locale allouée comme son sommet.

Comme il est possible que des données normalement en pile soient stockées dans les registres internes au processeur (cf. par exemple [Hennessy et Patterson, 1996]), des références sur des objets vivants peuvent ne pas être présentes en pile. Ainsi, en plus du processus d'examen de la pile, un autre est nécessaire pour accéder aux registres du processeur. Ce mécanisme, purement technique, basé sur des fonctions telles que le `setjmp`, dépend de la plate-forme et ne sera pas détaillé ici. Le lecteur intéressé peut se reporter au code source contenu dans le répertoire `gc_lib` de SmallEiffel[28].

Lors de l'accès à un mot mémoire de la pile (ou à un registre), on ne sait pas a priori s'il s'agit d'une référence Eiffel ou d'un autre type de données. En conséquence, tous les mots de la pile sont considérés comme des références potentielles [Boehm et Weiser, 1988]. Il est donc nécessaire d'identifier les véritables références, et cela de façon efficace. Comme dans [Chailloux, 1992] et [Kurihara *et al.*, 1990], quatre filtres successifs sont utilisés pour réduire les cas d'erreur d'identification.

Soient r la référence candidate et N le nombre total de blocs mémoire (quel que soit leur type). Nous notons $[B_x,E_x]$, avec $x \in [1,N]$, l'intervalle d'adresses incluses dans le bloc x, et $ObjectSize(x)$ la taille des places dans le bloc x.

Les quatre étapes de l'algorithme de filtrage sont les suivantes:

1. Puisque les adresses de tous les blocs créés sont stockées dans la table primaire (comme montré sur

26. Notons que dans le code généré par SmallEiffel, les pointeurs ne peuvent pas se trouver dans des zones statiques.

27. On considère ici que les adresses augmentent quand la pile croît. Bien entendu, SmallEiffel analyse la direction de croissance de la pile et gère à la fois les adresses croissantes et les adresses décroissantes.

28. Rappelons que ce code, ainsi l'ensemble du code source de SmallEiffel, peut être téléchargé depuis `http://SmallEiffel.loria.fr/`.

la figure 3.6), il est aisé de calculer l'intervalle des adresses acceptables pour un pointeur sur un objet:
r est acceptable si et seulement si $r \in [B_1,E_N]$.

2. On vérifie si la référence potentielle fait partie de l'intervalle des adresses offert par un bloc mémoire existant:
$\exists i \in [1,N] \mid r \in [B_i,E_i]$.
3. On vérifie si la référence potentielle pointe effectivement sur (le début d') un objet dans ce bloc i. Ce test est effectué rapidement par une fonction spécialisée associée au bloc mémoire et qui vérifie si la valeur du pointeur correspond à un décalage par rapport au début du bloc égal à un multiple entier de la taille des objets de ce bloc:
$\exists k \in \mathbb{N}^+ \mid r = B_i + k \times ObjectSize(i)$.
4. On vérifie si l'adresse pointée correspond à un objet alloué non marqué, grâce au drapeau de marquage de l'objet.

Si la référence candidate passe ces quatre tests, elle est considérée comme une référence racine valide et l'objet pointé est marqué vivant, en utilisant l'entête supplémentaire pour le ramasse-miettes.

Il est important de noter que même après les quatre tests ci-dessus, on ne sait toujours pas avec certitude si un mot issu de la pile est bien une référence sur un objet ou non. Une coïncidence peut se produire dans laquelle un mot de la pile contient une séquence de bits qui représentent une adresse valide, alors que ledit mot n'est pas du tout un pointeur, mais par exemple un entier. Dans ce cas d'erreur d'identification, l'objet est considéré vivant alors qu'il ne l'est pas et sa mémoire ne peut être réutilisée tant que la valeur prise pour son adresse ne disparaît pas de la pile. Ce phénomène de rétention mémoire peut causer une légère augmentation de la mémoire utilisée mais la technique décrite est la seule qui soit sûre car elle garantit la *complétude* de l'algorithme de marquage. Les expériences sur des programmes de tests relatées par Boehm dans [Boehm, 1993] ont montré que les erreurs d'identification provoquaient une rétention mémoire d'environ 10 % pour des collecteurs complètement conservateurs.

3.4.2.2 Suivi des références internes

Lorsqu'un objet `o1` est atteint et marqué vivant, il est nécessaire de propager le processus de marquage sur tous les objets auxquels `o1` fait référence (ses fournisseurs [Meyer, 1992]). Un marquage qui ne connaîtrait rien de l'objet excepté sa taille devrait faire le travail d'identification et de filtrage des références décrit précédemment pour la détection des racines. Bien entendu, dans probablement toutes les implantations de langages à objets, les objets contiennent une information concernant leur type dynamique — ne serait-ce que pour pouvoir réaliser la liaison dynamique — sous la forme d'un pointeur vers un objet descripteur ou sous celle d'un identifiant de type utilisé pour accéder à une table de description de type. Le processus de marquage est donc à même d'accéder à la description de la structure de l'objet, puis de suivre chaque référence interne sans avoir à vérifier si la référence est valide ou non. De nombreuses fuites de mémoire potentielles [Boehm, 1993] sont ainsi évitées.

Grâce aux capacités de prédiction de type et d'adaptation de code de notre méthode, SmallEiffel implante ce traitement des pointeurs internes de façon plus efficace.

En effet, SmallEiffel — qui est basé sur une forte adaptation du code — génère une routine de marquage spécialisée pour chaque type d'objet. Une telle routine est codée avec l'exacte connaissance de la structure des objets qu'elle traite et donc de l'emplacement des références vers d'autres objets ainsi que leur type. En conséquence, après le premier "saut incertain" depuis la pile vers un bloc `b`, la routine de marquage spécialisée associée au bloc `b` (voir figure 3.6) est appelée. Comme il s'agit d'une fonction spécialisée typée, le processus de marquage suit des pointeurs "sûrs", typés, et appelle statiquement les fonctions de marquage appropriées jusqu'à ce qu'une feuille (un objet sans fournisseur) soit atteint.

Supposons par exemple qu'une partie d'un système est composée d'objets de type TRIANGLE qui contiennent un INTEGER représentant leur couleur ainsi que trois références sur des objets POINT, ces derniers comprenant deux DOUBLEs pour leurs coordonnées. La fonction C de marquage qui est automatiquement générée correspond à:

```
void markTRIANGLE(Triangle *triangle) {
    if (triangle->mark_flag!=MARKED_FLAG){
        triangle->mark_flag=MARKED_FLAG;
```

```
        if (triangle->point1 != NULL)
           markPOINT(triangle->point1);
        if (triangle->point2 != NULL)
           markPOINT(triangle->point2);
        if (triangle->point3 != NULL)
           markPOINT(triangle->point3);
    }
}
```

Seuls les attribut de type référence sont pris en compte dans cette fonction afin de propager le marquage. Comme on peut le voir, l'attribut `color` de la classe TRIANGLE étant connu comme un champ qui n'est pas une référence, aucun code n'a besoin d'être généré pour son marquage dans `markTRIANGLE`.

Puisque la classe POINT possède seulement deux attributs DOUBLE et ne contient aucun attribut référence, aucun code de marquage ne doit être généré pour ses fournisseurs:

```
void markPOINT(Point *point) {
    point->mark_flag=MARKED_FLAG;
}
```

Notons également que l'absence de fournisseur fait qu'il devient inutile de tester si le drapeau est déjà marqué ou non. La fonction C `markPOINT` générée est ainsi réduite au strict minimum, et il est difficile d'envisager solution plus optimale.

Évidemment, à cause du polymorphisme, le type concret d'un fournisseur donné peut être n'importe quel descendant de son type statique, et une liaison dynamique sur la méthode de marquage adéquate est nécessaire. Comme nous l'avons indiqué, le mécanisme de prédiction de type de SmallEiffel (section 2.3) rend possible une diminution significative du coût de la liaison dynamique, en réduisant le nombre de types possibles à celui des types réellement vivants, ce qui participe à une liaison dynamique rapide (voir sections 2.6 et 2.7).

Supposons par exemple que la classe abstraite FRUIT ait deux (sous-)types concrets vivants: APPLE et PEACH. L'implantation de la liaison dynamique sur les fonctions de marquage pour les objets de type statique FRUIT se fait comme suit:

```
void markFRUIT(Fruit *fruit) {
    switch (fruit->id) {
      APPLEid: markAPPLE((Apple*)fruit); break;
      PEACHid: markPEACH((Peach*)fruit); break;
    }
}
```

Comme pour toute fonction de liaison dynamique, une telle méthode est en fait codée par SmallEiffel à l'aide d'un code de branchement dichotomique (voir section 2.6.1), qui résulte en une exécution très rapide, comme montré en section 2.7.

3.4.2.3 Suppression du marquage récursif

Comme on peut s'y attendre avec des algorithmes de marquage récursifs, les structures de données ayant un fort niveau d'imbrication, telles que des listes chaînées très longues, tendent à faire grossir la pile d'exécution démesurément et à finalement provoquer un débordement de pile. La technique classique pour résoudre ce problème consiste à transformer des appels récursifs en des parcours itératifs avec structures de données auxiliaires.

Comme le compilateur SmallEiffel connaît la structure des objets, il peut réordonner le marquage de leurs champ dans l'ordre le plus efficace, évitant ainsi l'utilisation de structures de données auxiliaires lorsqu'un seul de ces champ est le début d'une longue chaîne de références. Par exemple, une liste chaînée d'INTEGERs est marquée de la façon suivante:

```
void markLINK(Link *link) {
    do {
       if (link->mark_flag!=MARKED_FLAG){
         link->mark_flag=MARKED_FLAG;
         link = link->next;
       }
```

```
    } while (link != NULL);
}
```

Nous n'avons pas encore implanté cette technique pour des structures récursives plus complexes où plus d'un champ débute une longue chaîne de références. Nous pensons cependant que la généralisation à de tels cas est possible et qu'elle impliquerait l'utilisation d'une structure de données auxiliaire dont la taille serait limitée et connue grâce au mécanisme de prédiction de type de SmallEiffel.

3.4.3 Le balayage

La phase suivant le marquage, le balayage, consiste à examiner tous les objets alloués afin de collecter la mémoire utilisée par ceux qui n'ont pas été marqués vivants.

Il est donc nécessaire de parcourir tous les blocs mémoire de la table primaire (voir figure 3.6) pour collecter les objets qui ne sont plus référencés et dont le drapeau a été laissé `unmarked`. Ceci est fait efficacement dans SmallEiffel au moyen de fonctions de balayage adaptées à chaque type. Ainsi, les adresses où ces drapeaux se trouvent peuvent être facilement calculées en tenant compte du fait que tous les objets d'un bloc ont la même taille, connue à la compilation. La continuité de la mémoire gérée par un bloc est aussi de nature à garantir une localité des données plus importante lorsque le bloc est parcouru qu'avec des blocs mémoire chaînés et dispersés en mémoire. Voici un exemple simplifié de routine de balayage spécialisée pour les blocs de TRIANGLEs:

```
void sweepTRIANGLEchunk(TriangleChunk *tc) {
    Triangle*tp;
    for (tp=tc->first; tp <= tc->end; tp++){
       if (tp->mark_flag==MARKED_FLAG)
         tp->mark_flag=UNMARKED_FLAG;   /* (1) */
       else
         addToFreeListOfTRIANGLE(tp);   /* (2) */
    }
}
```

Chaque type possède sa propre liste libre contenant tous les objets libres associés à ce type, c'est à dire toutes les places libres pour des objets de ce type, quel que soit le bloc dans lequel elles sont. Quand un objet non marqué est trouvé, il est ajouté en tête de la liste libre correspondant à son type (ligne `(2)`). Celle-ci n'implique aucun surcoût mémoire, car c'est le mot mémoire qui constitue l'entête de l'objet dédié au ramasse-miettes — et qui a précédemment été utilisé pour marquer si l'objet était vivant ou non — qui peut maintenant être utilisé pour chaîner les objets dans la liste libre, comme le montre la figure 3.7 de la page 65. Notons que pour simplifier et accélérer le traitement des objets récemment devenus libres, cette liste d'objets libres est reconstruite à chaque balayage par l'instruction de la ligne `(2)`.

Tous les objets marqués vivants sont démarqués au fur et à mesure du balayage des blocs — ligne `(1)` — préparant ainsi le graphe d'objets pour le passage suivant du ramasse-miettes.

Remarquons enfin que lorsqu'un bloc ne contient que des places libres, il est remis dans la liste des blocs libres. Celle-ci n'est pas typée, ce qui permet un meilleur recyclage des blocs mémoire, qui ont tous la même taille.

3.4.4 Finalisation des objets

Avant qu'un objet soit collecté, une routine de finalisation [Hayes, 1992] peut être appelée sur cet objet.

Les routines de finalisation du ramasse-miettes de SmallEiffel sont générées comme n'importe quelle autre routine définie par le développeur. Elles sont donc produites seulement pour des classes qui définissent réellement une routine de finalisation effective.

Comme toutes ces routines sont définies lors de la compilation, le ramasse-miettes peut être adapté pour générer les appels correspondants au moment approprié. Ainsi, il n'est pas nécessaire de vérifier dynamiquement pour chaque objet s'il doit être finalisé. Les blocs contenant des objets possédant une routine de finalisation ont donc une routine de balayage spécialisée légèrement différente de celle des autres blocs, puisque juste avant qu'un objet soit libéré (remis dans la liste libre correspondant aux objets de ce type), la routine de finalisation des objets de ce type est appelée. De cette façon, le surcoût

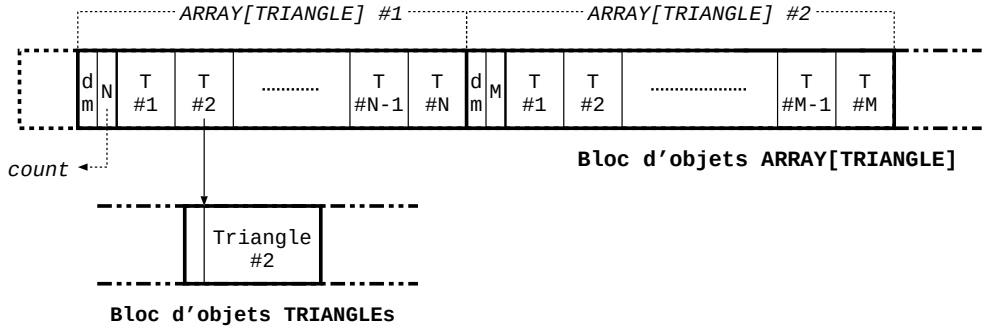


FIG. 3.9 – *Détail d'un bloc mémoire d'objets redimensionnables.*

induit par le support de la finalisation est très limité et ne diminue pas les performances du programme généré.

Voici ce que devient la routine de balayage C spécialisée pour les blocs de TRIANGLEs lorsque ces objets doivent être finalisés:

```
void sweepTRIANGLEchunk(TriangleChunk *tc) {
    Triangle*tp;
    for (tp=tc->first; tp <= tc->end; tp++){
       if (tp->mark_flag==MARKED_FLAG)
         tp->mark_flag=UNMARKED_FLAG;   /* (1) */
       else {
         if (tp->mark_flag==UNMARKED_FLAG)
            finalizeTRIANGLE();         /* (3) */
         addToFreeListOfTRIANGLE(tp);   /* (2) */
       }

    }
}
```

Notons que seuls les objets récemment libérés sont finalisés, en ligne `(3)`.

3.5 La gestion des objets redimensionnables

La gestion par le ramasse-miettes des objets redimensionnables — des contenants tels que tableaux et chaînes de caractères — est très similaire à celle des objets de taille fixe. Cependant, comme la taille de ces objets redimensionnables n'est pas calculable lors de la compilation, leur gestion est plus compliquée.

3.5.1 L'allocation des objets redimensionnables

Les objets redimensionnables de petite taille sont alloués comme les objets de taille fixe (voir section 3.4.1), dans des blocs mémoire typés et de taille fixe (voir figure 3.9). La taille de ces derniers, cependant, est plus importante que celle des blocs contenant des objets de taille fixe, afin de prendre en compte le fait que les objets redimensionnables sont généralement plus gros que ceux de taille fixe. L'allocation des objets redimensionnables maintient son propre "plafond" qui est géré de la même façon que celui les objets de taille fixe, la valeur initiale étant là aussi fonction du nombre de types concrets vivants calculé statiquement.

Les objets redimensionnables de très grande taille qui sont plus larges que la taille normale de ces blocs mémoires sont eux traités différemment. Chacun d'eux constitue à lui tout seul un bloc "à une place", dont la taille est simplement celle de l'objet redimensionnable lui-même.

Le nombre d'éléments contenus par l'objet redimensionnable est conservé dans son entête. Ceci permet de marquer les éléments du tableau, s'il s'agit de références vers des objets. Ceci permet également le

balayage des blocs contenant des objets redimensionnables, puisque la position dans le bloc de chacun de ceux-ci peut être aisément calculée.

3.5.2 Le marquage des objets redimensionnables

Le mécanisme de marquage des objets redimensionnables est très similaire à celui des objets de taille fixe (voir section 3.4.2). Cependant, pour chaque élément d'un objet redimensionnable, il n'est besoin d'aucune information supplémentaire. En effet, à la place, un seul drapeau de marquage est associé au contenant et s'applique à tous ses éléments. Ainsi, dans les tableaux et objets similaires, la place réservée à un élément correspond exactement à sa taille [29].

Comme pour les objets de taille fixe (voir figure 3.6), une routine de marquage spécifique est associée à chaque type concret vivant correspondant à un contenant redimensionnable. Ainsi, comme ARRAY[INTEGER] et ARRAY[TRIANGLE] sont deux types concrets distincts, ils requièrent différentes fonctions de marquage.

Quand le container contient des objets de type référence, la fonction de marquage générée doit propager le processus de marquage à chaque élément. Par exemple, la fonction de marquage pour un conteneur de TRIANGLEs consiste en une boucle qui propage le marquage sur les éléments:

```
void markContainerOfTRIANGLE(Triangles *c) {
    int i;
    if (c->mark_flag!=MARKED_FLAG){
      c->mark_flag=MARKED_FLAG;
      for (i=c->count - 1; i >= 0; i--) {
         Triangle *t = c->storage[i];
         if (t != NULL) markTRIANGLE(t);
      }
    }
}
```

Comme cette fonction a été spécialisée, elle est très efficace. En effet, le code le plus approprié compte tenu du type des éléments est utilisé au lieu d'un code générique: des appels directs à `markTRIANGLE` sont faits lorsqu'un conteneur de TRIANGLE est marqué.

Pour un conteneur renfermant des objets non-références, comme un tableau d'entiers, aucun marquage supplémentaire n'est nécessaire. La fonction de marquage positionne donc simplement le drapeau du conteneur et se termine:

```
void markContainerOfINTEGER(Integers *c) {
    c->mark_flag=MARKED_FLAG;
}
```

Sur un tel conteneur, l'algorithme de marquage spécialisé est plus rapide que celui d'un ramasse-miettes conservateur.

3.5.3 Le balayage des objets redimensionnables

Le balayage des objets redimensionnables est similaire à celui des objets de taille fixe (section 3.4.3), à la différence que le premier doit prendre en compte la taille réelle de chaque objet conteneur dans le bloc. Comme ce processus n'est pas basé sur le type des éléments du conteneur, la même fonction de balayage est utilisée pour tous les blocs d'objets redimensionnables [30].

Lorsqu'un objet redimensionnable est collecté, il est placé dans la liste libre d'objets correspondant à son type. Comme pour les objets de taille fixe, un bloc complètement libre est remis dans une liste de blocs libres non typée et peut être réutilisé pour contenir des objets de n'importe quel type.

Quand un très gros objet redimensionnable est libéré, le bloc correspondant est également intégré à la liste des blocs libres. Ce bloc peut par la suite être "découpé" afin d'être utilisé comme un bloc de taille normale pour de petits objets redimensionnables. Une conséquence de ce découpage est que le reste du bloc peut constituer un bloc plus petit que la taille normale. Pour éviter une fragmentation excessive,

29. Les octets d'alignement mis à part

30. Cette fonction de balayage est donc écrite une fois pour toutes et peut être consultée dans le fichier `gc_lib.c` de la distribution de SmallEiffel.

une opération de compactage et fusion est déclenchée périodiquement sur ces blocs, durant la phase de balayage.

3.6 Optimisations spécifiques à Eiffel

La définition même du langage Eiffel, comme de tout autre langage, peut être prise en compte afin d'exploiter certaines de ses particularités pour effectuer diverses optimisations spécifiques au langage.

3.6.1 L'objet racine

L'*objet racine*, le premier objet créé, sur lequel la routine racine du système est appelée, est vivant pendant toute l'exécution du programme. Il ne peut donc jamais être collecté et reste marqué en permanence. Cependant, comme ses attributs peuvent changer lors de l'exécution, leur marquage reste nécessaire.

De plus, comme l'objet racine est fréquemment le seul objet de son type dans le système, il n'est pas alloué dans un bloc normal mais à part, pour éviter une utilisation de mémoire supérieure.

3.6.2 Le résultat des fonctions `once`

Les routines *once* [Meyer, 1992] sont spécifiques à Eiffel. Le corps d'une telle routine n'est exécuté qu'une seule fois dans le vie du programme, la première fois qu'elle est appelée. Les appels successifs retournent sans exécuter le corps de la routine.

Quand la routine *once* est une fonction, son résultat est calculé lors du premier appel, mémorisé et retourné pour tous les appels suivants. En conséquence, les objets retournés par des fonctions *once*, ou objets *once*, sont vivants depuis leur allocation jusqu'à la fin du programme. Ainsi, exactement comme l'objet racine, un objet *once* doit toujours être considéré marqué par le collecteur et ne peut être récupéré. La gestion des objets *once* peut donc être optimisée en se basant sur un marquage spécifique et en évitant complètement le balayage.

Pour certaines fonctions *once*, il est possible lors de la compilation de connaître de façon non ambigüe le type du résultat. Ce dernier peut donc être pré-calculé, c'est à dire créé au tout début du programme [Zendra *et al.*, 1997]. Des optimisations supplémentaires, comme le fait de ne pas vérifier si l'objet est `NULL`, peuvent être intégrées au ramasse-miettes pour la gestion de ces objets.

3.6.3 Les chaînes manifestes

Une *chaîne manifeste* est une chaîne dont la valeur apparaît directement dans le code source. Une chaîne manifeste n'est pas une chaîne constante; il s'agit d'une référence sur un tableau de caractères, modifiable et redimensionnable. En effet, en Eiffel, le développeur n'a normalement pas accès directement à l'objet redimensionnable, mais à un objet de taille fixe qui encapsule le comportement de l'objet redimensionnable d'une façon portable et qui cache à l'utilisateur son implantation réelle.

Une chaîne manifeste peut donc être considérée comme un type particulier de fonction *once* dont la valeur est pré-calculable lors de la compilation. En conséquence, toutes les chaînes manifestes sont allouées dans des zones mémoire spécifiques, ne sont pas sujettes au balayage et ont un processus de marquage spécialisé et optimisé comme décrit pour les fonctions *once* pré-calculables.

3.7 Résultats expérimentaux

Afin d'asseoir la validité de notre approche pour la conception d'un ramasse-miettes spécialisé généré par le compilateur, ainsi que la validité de son implantation dans le compilateur SmallEiffel, nous avons effectué un nombre important d'expérimentations. Avant tout, nous avons tenté de couvrir un ensemble de comportements de programmes vis-à-vis de la mémoire qui soit aussi exhaustif que possible. En effet, un algorithme donné peut très bien être très efficace pour un certain schéma d'exécution et se comporter très mal sur un autre.

Nous avons donc testé le comportement de SmallEiffel sur trois types de bancs d'essais. Nous avons tout d'abord conçu un ensemble de programmes de tests mettant en relief différents schémas d'exécution synthétiques, de façon à couvrir un ensemble de comportements aussi vaste que possible. Ces programmes ainsi que les résultats qu'ils permettent d'obtenir sont présentés dans les sections 3.7.2 et 3.7.3.

Évidemment, il est possible d'objecter que bien qu'étant nécessaires, ces schémas synthétiques ne sont pas suffisants, car ils représentent seulement des instantanés de la vie d'un programme et non pas de véritables applications réellement utilisées. Nous avons donc aussi mesuré le comportement de l'implantation de notre méthode sur de véritables programmes, en l'occurrence divers jeux d'Othello réalisés par des groupes d'étudiants. Ces programmes et les résultats s'y rapportant sont détaillés dans la section 3.7.4.

Enfin, nous avons également considéré une application bien réelle et de taille plus importante: le compilateur SmallEiffel lui-même. Ces résultats apparaissent en section 3.7.5.

3.7.1 La plate-forme expérimentale

Nous avons comparé le ramasse-miettes généré par SmallEiffel, qui est spécialisé et semi-conservateur (car connaissant avec précision le type des objets pour la partie mémoire gérée en tas), avec le ramasse-miettes Boehm-Demers-Weiser (BDW) [Boehm et Weiser, 1988]. Ce dernier est un ramasse-miettes renommé et de qualité, entièrement conservateur, qui a été l'objet de nombreux travaux de recherche [Boehm, 1993; Zorn, 1993; Detlefs *et al.*, 1994]. Il a également été développé depuis de nombreuses années, notamment sur les systèmes UNIX, et est donc tout à fait mature. Implanté dans divers systèmes de type industriel, "le BDW", comme on l'appelle, est un ramasse-miettes rapide, robuste et compact, grâce à des algorithmes fortement optimisés et des structures de données très efficaces, tant du point de vue conceptuel qu'au niveau du codage. Le BDW représente donc un système de référence tout à fait solide et reconnu, qui a été porté sur de nombreux systèmes.

De plus, il a été utilisé avec succès en conjonction avec SmallEiffel, depuis longtemps, notamment car SmallEiffel n'a pas fourni de ramasse-miettes intégré avant la version `-0.81`. Il aurait également pu être intéressant de comparer notre ramasse-miettes, qui connaît la structure du tas mémoire, avec un BDW "typé". En effet, dans [Boehm et Shao, 1993], Boehm et Shao montrent que des améliorations des performances par rapport au BDW "classique", totalement conservateur, semblent possibles. Cependant, comme ils l'indiquent, il n'est pas certain que de telles améliorations seraient toujours valables pour des programmes de très grande taille. De plus, dans notre étude, nous voulions comparer notre implantation d'un ramasse-miettes *partiellement conservateur* à celle d'un ramasse-miettes *totalement conservateur*. Nous avons donc uniquement considéré le ramasse-miettes BDW classique.

Depuis sa version `-0.81`, SmallEiffel est capable de produire le code C adapté à une application avec ou sans génération du code C correspondant au ramasse-miettes de l'application, selon que l'option `-no_gc` a été sélectionnée ou non. Quand le ramasse-miettes est aussi généré, les instructions d'instanciation s'appuient sur l'ensemble du ramasse-miettes décrit dans les sections 3.4 à 3.6. Au contraire, avec `-no_gc`, la routine chargée de l'allocation d'un nouvel objet (par ex. `newTRIANGLE`) appelle simplement la fonction `malloc` de la bibliothèque C standard. Ceci rend l'intégration d'une bibliothèque implantant un ramasse-miettes par redéfinition des fonctions `malloc` et `free`, comme le BDW, extrêmement facile. Il est aussi possible de n'utiliser aucun ramasse-miettes, lorsque le développeur veut impérativement gérer sa mémoire lui-même, ou lorsque l'application est considérée sans fuite car conçue sans aucune désallocation mémoire.

Afin de montrer que notre méthode de génération d'un ramasse-miettes spécialisé est générique, et non pas pas liée à une plate-forme ou un type de plate-forme spécifique[31], les différentes expérimentations que nous avons faites l'ont été sur diverses architectures et divers systèmes d'exploitation: MacOS, des variantes d'UNIX (Linux, Solaris, HP-UX, DEC OSF) et de Windows (95 et NT). Les résultats étant globalement cohérents entre ces différentes plates-formes, nous nous limitons dans les sections suivantes aux résultats sous deux d'entre elles: Solaris et Windows NT.

La plate-forme Solaris, représentant une configuration haut de gamme, consistait en une station SUN Enterprise 450 avec processeur Ultra SPARC II et 512 MO de RAM, fonctionnant sous Solaris version

31. Cependant, comme expliqué page 67, il est nécessaire d'avoir accès au contenu des registres du processeur. Sur certaines plates-formes, comme Solaris SPARC, ceci ne peut être fait en C ANSI, mais seulement en assembleur, ce qui impose d'avoir un très court fichier dépendant de la plate-forme.

2.6. La grande quantité de mémoire dont disposait cette machine permettait à tous les programmes de test, même les plus gourmands en mémoire, de s'exécuter entièrement en RAM sans être déportés en mémoire virtuelle sur le disque (*swapped out*) et donc d'avoir des mesures non biaisées.

La plate-forme NT était une configuration beaucoup plus modeste, à savoir un PC Dell GXPro équipé d'un Pentium Pro 200 MHz avec 256 KO de cache, doté de 64 MO de RAM, tournant sous Windows NT 4.0 Workstation. Toujours pour d'éviter que certains tests soient déportés en mémoire virtuelle disque, nous avons recalibré ceux d'entre eux qui consommaient le plus de mémoire.

Les expérimentations ont été réalisées avec SmallEiffel `-0.81`, sauf celles pour les programmes Othello, qui ont eu lieu un peu après et ont utilisé la version suivante, `-0.80`. Les fonctionnalités du ramasse-miettes de ces deux versions sont toutefois identiques, seules quelques corrections de bogues ayant été effectuées. SmallEiffel a été utilisé avec les options d'optimisation `-boost` et `-no_split` pour la génération de code C. Bien entendu, `-no_gc` a été ajoutée lors de la production du code C destiné à intégrer la bibliothèque BDW 4.12 (dans laquelle l'option `-Dall_interior_pointers` avait été inhibée). Sauf mention contraire, ce code C a ensuite été compilé avec `egcs-1.0.1` au niveau d'optimisation `-O6`. Ces options correspondent aux niveaux d'optimisations maximum tant pour la version utilisant le ramasse-miettes de SmallEiffel que celle utilisant le BDW.

Notre méthodologie de mesure de performances a été la suivante. Chaque exécution de programme mesurée (que ce soit au niveau mémoire ou du point de vue du temps) l'a été 5 fois immédiatement consécutives. La première mesure n'a pas été considérée, de façon à éliminer l'influence potentielle du chargement depuis le disque, de la non-présence en cache, etc. Les 4 mesures suivantes ont servi a établir une moyenne, qui est le résultat que nous fournissons. Ce processus a été répété pour tous les programmes à mesurer. Tous les tests ont été faits à charge aussi constante que possible. Enfin, l'ensemble des tests a été refait par la suite, de façon à vérifier la stabilité des résultats.

3.7.2 Développement de schémas d'exécution synthétiques

Nous décrivons ici les schémas d'exécution que nous avons conçus et utilisés pour mesurer les performances de notre ramasse-miettes.

Bien que cette section puisse être sautée en première lecture, elle est toutefois nécessaire pour bien comprendre les résultats de la section 3.7.3 débutant page 77. Notons que le code de ces programmes de tests est fourni en standard dans la distribution de SmallEiffel et que chacun peut donc les utiliser.

3.7.2.1 Schémas d'exécution comprenant de nombreux objets vivants

MFSO Le programme de test *Many Fixed-Size Objects* (Nombreux Objets de Taille Fixe) illustre un schéma d'exécution avec une très forte augmentation de la mémoire allouée, sans jamais la moindre libération mémoire. Il consiste en un grand tableau initialement vide qu'une boucle remplit ensuite avec des objets feuilles (sans référence vers des fournisseurs) de taille fixe. Son principal objectif est de vérifier les performances de l'allocation mémoire d'objets de taille fixe (voir section 3.4.1).

MRO Le test *Many Resizable Objects* (Nombreux Objets Redimensionnables) présente un schéma d'exécution dans lequel de nombreux objet redimensionnables sont créés et restent vivants jusqu'à la fin de l'exécution. Ceci est implanté à l'aide d'une longue boucle qui crée des tableaux d'entiers et les conserve dans une liste. Ce programme de tests a pour but de mettre en évidence le comportement du ramasse-miettes face à un accroissement très fort de la mémoire allouée dû à l'allocation massive d'objets redimensionnables (voir section 3.5.1).

3.7.2.2 Schémas d'exécution avec des objets contenants

CNoLfO Dans le banc d'essai *Container of Non-Leaf Objects* (Container d'Objets Non Feuilles), après une forte augmentation initiale, la mémoire reste stable jusqu'à la fin de l'exécution. Ce test consiste à créer et conserver une référence sur un grand tableau, en le remplissant de façon répétée de nouveaux objets de taille fixe ayant des références vers des fournisseurs, afin de déclencher périodiquement le

ramasse-miettes. Ceci est destiné à évaluer le marquage des containers d'objets de taille fixe non feuilles (voir section 3.5.2).

CLfO *Container of Leaf Objects* (Container d'Objets Feuilles) consiste en un schéma d'exécution où la mémoire allouée reste stable tout le long de de la vie du programme. Ce test alloue puis abandonne de longues chaînes de caractères. Les chaînes vivantes doivent donc être marquées à chaque fois que le ramasse-miettes est déclenché. L'objectif est ici de contrôler le marquage d'un container "plat", contenant seulement des objets feuilles, les caractères (voir section 3.5.2).

3.7.2.3 Schémas d'exécution avec fuites de mémoire

LkFSO *Leak on Fixed-Size Objects* (Fuite d'Objets de Taille Fixe) offre un schéma d'exécution qui impose un travail important au ramasse-miettes, car l'exécution ne cesse d'allouer et d'abandonner des objets feuilles ou non feuilles. Il comprend une très longue boucle qui alloue des objets de taille fixe sans garder de référence sur eux. Son but est de tester l'ensemble du processus de recyclage des objets de taille fixe, qu'ils soient des feuilles ou non (voir section 3.4.3).

LkROSm Dans *Leak on Resizable Objects of Small size* (Fuite d'Objets Redimensionnables de Petite Taille), de nombreux petits tableaux redimensionnables sont alloués sans qu'aucune référence ne soit gardée sur eux. La quantité de mémoire réellement utilisée est donc très faible. Le but de ce banc d'essai est de montrer l'efficacité du recyclage des objets redimensionnables de petite taille, qui sont typiques de l'utilisation des chaînes de caractères (voir section 3.5.3).

LkROInc *Leak on Resizable Objects of Increasing size* (Fuite d'Objets Redimensionnables de Taille Croissante) représente un schéma d'exécution avec de nombreux objets feuilles redimensionnables ayant une durée de vie courte. Ce banc d'essai contient une boucle qui crée des tableaux d'entiers de taille croissante. Il est conçu pour mettre en évidence l'efficacité de la récupération et la fusion des objets redimensionnables (voir section 3.5.3).

LkRODec *Leak on Resizable Objects of Decreasing size* (Fuite d'Objets Redimensionnables de Taille Décroissante) est une variante du précédente programme de test créant des objets de taille décroissante. Son objectif principal est de montrer les performances du recyclage et du découpage des objets redimensionnables (voir section 3.5.3).

LkRORnd Finalement, la variante *Leak on Resizable Objects of Random size* (Fuite d'Objets Redimensionnables de Taille Aléatoire) offre un schéma d'exécution avec de nombreux objets feuilles redimensionnables de courte durée de vie et ayant diverses tailles. Son but est de vérifier l'efficacité du recyclage, du découpage et de la fusion des objets redimensionnables (voir section 3.5.3).

3.7.2.4 Schémas d'exécution impliquant le polymorphisme

PMO *Polymorphism on Many Objects* (Polymorphisme sur Nombreux Objets) présente un schéma d'exécution où, après un accroissement important, la mémoire allouée est rapidement libérée puis remonte très rapidement. Ce programme consiste en un grand tableau qui est tout d'abord rempli avec un type d'objets feuille de taille fixe, puis vidé et de nouveau rempli, avec un autre type type d'objets feuilles de taille fixe. Son but est de tester le marquage polymorphique et le recyclage de la mémoire d'un type d'objets de taille fixe à un autre (voir sections 3.4.2 et 3.4.3).

PLk *Polymorphism and Leaks* (Polymorphisme avec Fuites) représente un schéma d'exécution avec de nombreux objets feuilles de taille fixe de différents types, ayant une durée de vie courte. Ce test consiste en deux boucles successives qui allouent deux types d'objets sans conserver de référence sur eux. Son but est de vérifier que les besoins en mémoire sont maintenus à un faible niveau grâce au bon recyclage des objets morts récemment, quel que soit leur type.

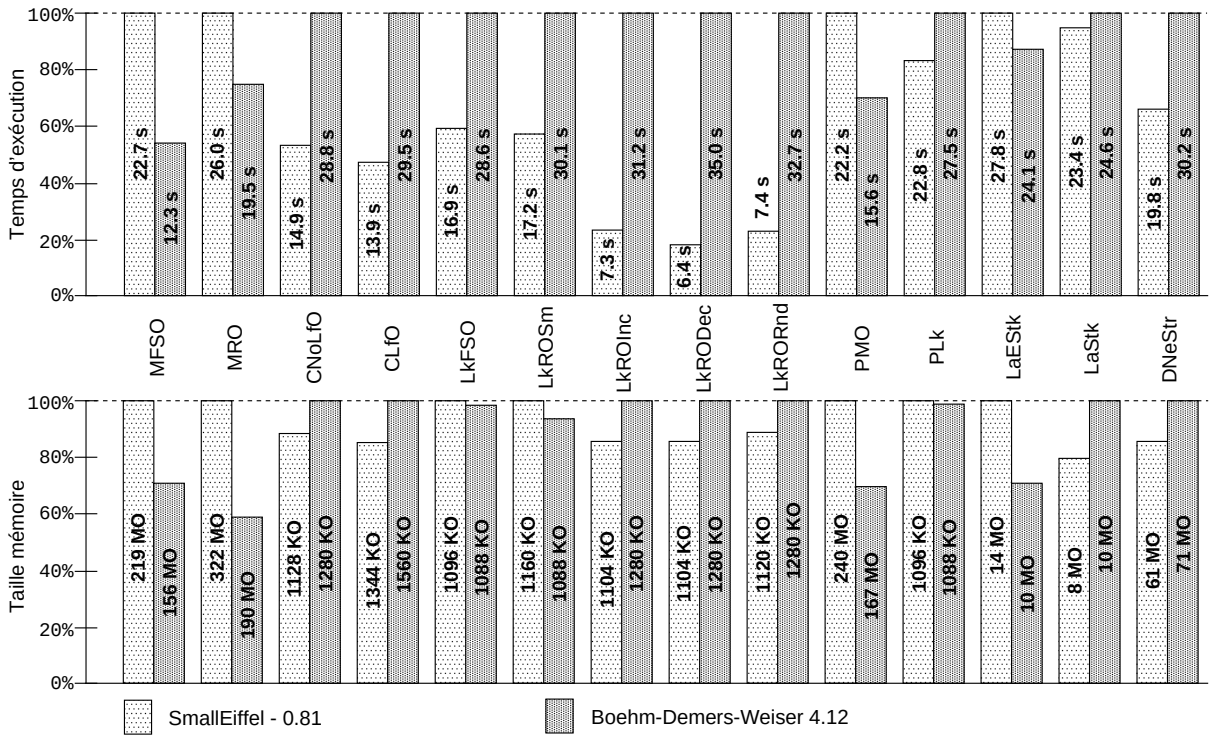


FIG. 3.10 – *Comparaisons des temps d'exécution et taille mémoire utilisée sous UNIX.*

3.7.2.5 Schémas d'exécution avec pile d'exécution de grande taille

LaEStk *Large Empty Stack* (Grande Pile Vide) est un schéma d'exécution comprenant une pile d'exécution de très grande taille, contenant très peu d'objets vivants. Ce banc d'essai consiste en une routine récursive qui est appelée un grand nombre de fois et dont l'appel imbriqué déclenche le ramasse-miettes en créant de nombreux objets qui sont immédiatement abandonnés. Son objectif est de mesurer l'efficacité de la recherche des racines (voir section 3.4.2.1) au sein d'une pile de grande taille très creuse.

LaStk *Large Stack* (Grande Pile) présente une variante du schéma d'exécution précédent où un objet feuille de taille fixe est alloué à chaque appel récursif. Comme cet objet est stocké dans une variable locale de la routine récursive, il se retrouvera dans la pile d'exécution.

3.7.2.6 Autres schémas d'exécution

DNeStr *Deeply Nested Structure* (Structure à Fort Niveau d'Imbrication) comprend l'allocation d'une structure à très fort niveau d'imbrication. Ce test crée une très longue liste chaînée, avec laquelle un code de marquage purement récursif causerait un débordement de pile. Son but est bien sûr de montrer le comportement du processus de marquage, notamment le suivi des références internes aux objets (cf. section 3.4.2.2) et la suppression du marquage récursif (cf. section 3.4.2.3), sur une structure à fort degré d'imbrication.

3.7.3 Résultats des schémas d'exécution synthétiques

3.7.3.1 Résultats sous UNIX

Comme indiqué dans la description de la plate-forme expérimentale, section 3.7.1, nous disposions d'une machine avec 512 MO de RAM. Ceci nous a permis de faire tourner sans biais certains schémas d'exécution extrêmement "gourmands" en mémoire. Ainsi, par exemple, le schéma MRO génère 14 *millions*

d'objets (des tableaux d'entiers) vivants en même temps. La taille mémoire minimale théorique (i.e., incluant uniquement les données du mutateur proprement dit) de ce test MRO sur cette plate-forme est donc très importante: 187 MO.

La figure 3.10 montre que les performances globales du ramasse-miettes généré selon notre méthode sont très bonnes, tout particulièrement quand on les compare avec le très efficace BDW. Les différences les plus importantes apparaissent dans les temps d'exécution où SmallEiffel a généralement un avantage marqué sur le BDW. Ce dernier est néanmoins supérieur à SmallEiffel en ce qui concerne l'utilisation mémoire. Ces résultats globaux confirment nos attentes, puisque notre méthode consistant à générer un ramasse-miettes spécialisé avait essentiellement pour but d'accélérer de façon sensible le processus de gestion mémoire, alors que la réduction de la taille mémoire occupée par le programme généré n'était pas notre objectif principal.

Occupation mémoire Globalement, l'occupation mémoire obtenue avec notre méthode pour ces 14 programmes de test est plutôt similaire à celle obtenue avec le BDW. Toutefois cette dernière est généralement plus faible et peut être jusqu'à 41 % inférieure à la première.

Avec le BDW, MFSO ((Many Fixed-Size Objects) prend environ 28 % moins de mémoire qu'avec SmallEiffel. Ceci est dû à la structure très simple que nous avons choisie pour l'entête des objets de taille fixe dédiée au ramasse-miettes. En effet, pour des raisons de simplicité d'implantation de cette première version de ramasse-miettes pour SmallEiffel, le drapeau de marquage associé à chaque objet de taille fixe est un entier, alors qu'un seul bit, comme dans le BDW, suffit.

Le surcoût mémoire causé par le ramasse-miettes généré par SmallEiffel peut donc être très important sur les petits objets de taille fixe et apparaît clairement dans les tests qui se caractérisent par une occupation mémoire importante due à la présence de nombreux objets de taille fixe vivants, comme MFSO et PMO (Polymorphism on Many Objects). Ceci est sans aucun doute un problème à considérer dans de futures versions, dans l'optique de l'implantation d'un ramasse-miettes optimal en termes de mémoire.

Ce désavantage de l'implantation du ramasse-miettes de SmallEiffel pour les objets de taille fixe est aussi apparent dans le test MRO (Many Resizable Objects). En effet, en Eiffel, l'utilisateur n'a normalement pas accès directement à un objet redimensionnable, mais à un objet de taille fixe qui encapsule (le comportement de) l'objet redimensionnable d'une façon portable et en cachant son implantation à l'utilisateur. Par exemple, bien qu'un tableau redimensionnable en Eiffel est implanté à l'aide de la classe idxatarrayARRAY, ce dernier est en fait un objet de taille fixe contenant des attributs qui décrivent le tableau (bornes `lower` et `upper`, ...) et un attribut privé représentant la zone de stockage d'une façon dépendant du compilateur[32].

En conséquence, les développeurs doivent utiliser "l'enveloppe" — totalement portable — de taille fixe pour accéder à un objet redimensionnable — intrinsèquement non portable. Il n'est donc pas possible d'avoir plus d'objets redimensionnables que d'objets de taille fixe dans un programme Eiffel portable. Notre banc d'essai MRO alloue donc un grand nombre d'objets, dont la moitié sont des objets de taille fixe. Ceci explique pourquoi BDW prend 41 % de mémoire en moins que SmallEiffel pour MRO.

Temps d'exécution Le critère de comparaison pour lequel apparaissent les différences les plus importantes entre SmallEiffel et BDW est le temps d'exécution. Sur la majorité des programmes de test (10 sur 14), notre méthode produit un ramasse-miettes qui est plus rapide que le BDW et dans la plupart des cas (7 sur 10) cet avantage en termes de vitesse est de plus de 40 %. Le BDW, en revanche, dépasse SmallEiffel dans 4 cas sur 14 dont un seul avec un avantage de plus de 40 %.

Ce programme est MFSO (Many Fixed-Size Objects), pour lequel l'exécution est 46 % plus rapide avec BDW. Un tel avantage, bien que plus modéré, apparaît aussi sur PMO (Polymorphism on Many Objects). Nous pensons que les temps d'exécution accrus avec SmallEiffel sur ces deux programmes sont la conséquence directe de notre implantation grossière du bit de marquage des objets de taille fixe par un entier. En effet, ce surcoût implique l'allocation de plus de blocs mémoire et peut également diminuer la localité de l'algorithme de marquage-balayage. Des analyses supplémentaire et plus fines sont toutefois encore nécessaires pour bien comprendre ce problème.

32. Son type est NATIVE_ARRAY dans SmallEiffel et SPECIAL dans les compilateurs commerciaux iss-base et ISE.

Au contraire, sur les tests basés sur les objets de taille fixe, comme LkFSO — Leak on Fixed-Size Objects — et DNeStr — Deeply Nested Structure —, où la consommation mémoire, bien que très importante (61 MO dans DNeStr), n'est pas déraisonnable, notre méthode se comporte très bien. Ces deux programmes testent le marquage, le balayage, le recyclage et la (ré)allocation des objets de taille fixe.

Nos affirmations sur l'efficacité du ramasse-miettes implanté dans SmallEiffel, grâce à la spécialisation du code, sont renforcées par les deux schémas conçus pour mesurer les performances sur des containers d'objets, CNoLfO (Container of Non-Leaf Objects) et CLfO (Container of Leaf Objects). Sur ces deux bancs d'essais, l'exécution avec le ramasse-miettes de SmallEiffel est environ deux fois plus rapide que celle avec BDW. Ceci confirme que notre implantation du marquage est très efficace sur les objets containers redimensionnables, qu'ils contiennent des objets feuilles ou non.

Le recyclage des objets redimensionnables normaux (c'est à dire dont la taille est inférieure à 4 pages dans notre implantation actuelle) est testé avec quatre différents schémas d'exécution: LkROSm, LkROInc, LkRODec and LkRORnd. Nous considérons LkROSm comme particulièrement représentatif de l'utilisation typique des tableaux dans les programmes à objets. LkROSm représente l'utilisation de petits tableaux de diverses tailles, ce qui correspond à l'utilisation de chaînes de caractères dans un nombre significatif de programmes pour, par exemple, les messages à l'utilisateur, des propriétés d'un système, les noms des identificateurs dans un compilateur, etc. Chacun de ces trois bancs d'essai indique clairement que le ramasse-miettes de SmallEiffel a des performances qui dépassent nettement celles du BDW pour les objets redimensionnables de taille normale. En effet, pour les petits tableaux de LkROSm, notre méthode est 43 % plus rapide que BDW; sur de grands tableaux, cet avantage atteint jusqu'à 82 % (LkRODec). Nous pensons que de telles performances viennent probablement de notre implantation des allocations des tableaux dans des BALs (cf. page 64) ainsi que des routines de marquage des objets containers spécialisées (décrites en section 3.5).

Il apparaît donc que le ramasse-miettes spécialisé de SmallEiffel est très rapide — plus que le BDW — sur tous les types d'objets et dans la quasi totalité des schémas d'exécution. Ceci confirme que notre but initial, un ramasse-miettes optimisé pour la vitesse, a été atteint. En revanche, notre ramasse-miettes spécialisé se comporte moins bien que BDW lorsqu'il a affaire à l'allocation de très nombreux objets, qui restent vivants. Il est possible que la relative jeunesse de notre implantation du ramasse-miettes joue négativement sur ce point, et que sa maturation, ainsi que le développement d'heuristiques pour des cas particuliers, puisse résoudre les quelques problèmes de montée en charge restants.

3.7.3.2 Résultats sous Windows NT

En plus de comparer l'implantation de notre méthode pour le ramasse-miettes de SmallEiffel à celle du BDW, nous avons pu, sous Windows NT, la comparer à celle fournie par un compilateur Eiffel utilisé dans l'industrie, le produit iss-base d'Halstenbach[33] version 1.6.

Les compilations Eiffel réalisées avec iss-base 1.6 ont été faites en mode de "finalisation", non incrémental, fortement optimisé, en partant de zéro, sans bibliothèque pré-compilée, de façon à générer le meilleur code C possible en termes de vitesse d'exécution[34].

Les compilations d'Eiffel vers C faites par SmallEiffel utilisaient les options normales décrites en section 3.7.1.

En revanche, le code C généré soit par SmallEiffel soit par iss-base a été compilé non pas avec `egcs`, comme d'habitude pour SmallEiffel, mais avec Microsoft Visual C++ (version 5.0), qui est le compilateur C recommandé pour iss-base par son fournisseur, Halstenbach. Toutes les compilations C ont été réalisées avec les mêmes options, celles d'iss-base, qui utilise le niveau d'optimisation `-O2`. Sur deux des programmes de test, LaEStk et LaStk, nous avons dû compiler sans `-O2`, car, sur le code généré par SmallEiffel, le compilateur C arrivait à supprimer la récursivité terminale et la remplaçait par une boucle bien plus efficace. Ceci aurait donné un avantage indu à SmallEiffel et à BDW par rapport à iss-base, amenant à des temps d'exécution anormalement bons, et aurait également enlevé tout sens à ces programmes de tests, dont le but est précisément de tester de très grandes piles d'exécution.

33. Ce compilateur est basé sur celui d'ISE (Interactive Software Engineering).

34. Ces compilations ont utilisé l'ensemble des optimisations pour la vitesse d'exécution, à savoir: `assertion (no)`, `dead_code_removal(yes)`, `array_optimization(yes)`, `inlining(yes)`, `exception_trace(no)` et `inlining_size("20")`. Pour ce dernier paramètre, nous avons testé diverses valeurs, sans trouver de différence significative.

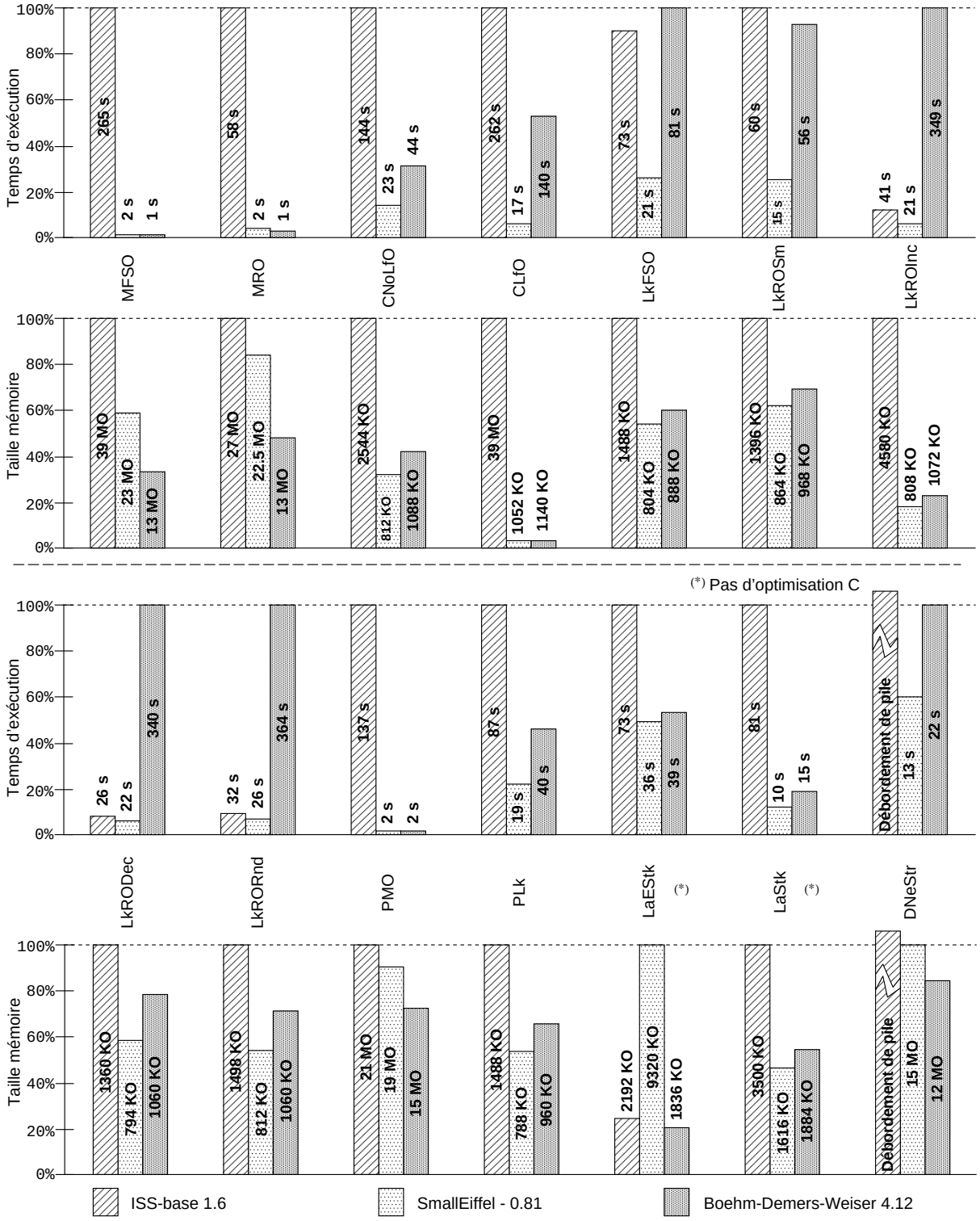


FIG. 3.11 – *Comparaisons des temps d'exécution et taille mémoire sous Windows NT.*

Nous avons pu constater que les résultats globaux du ramasse-miettes de SmallEiffel comparé au BDW sont très similaires sur Windows NT et sur UNIX.

Taille mémoire La comparaison de l'occupation mémoire faite sous UNIX entre SmallEiffel et BDW, à la page 78, apparaît toujours valide sous Windows. La seule exception se produit sur LaEStk (Large Empty Stack), où BDW est 80 % plus économe en mémoire que SmallEiffel. Ceci est probablement dû au fait que le critère de déclenchement du ramasse-miettes de SmallEiffel est peu sensible jusqu'à 10 MO.

Comparé à celui d'iss-base, le ramasse-miettes de SmallEiffel se comporte très bien, générant une occupation mémoire plus grande sur un seul programme, LaEStk. Sur ce dernier, iss-base utilise 76 % de mémoire en moins que SmallEiffel. Celui-ci a cependant un avantage sur iss-base qui dépasse 40 % pour 9 schémas d'exécution, et atteint même 98 % sur l'un d'entre eux, CLfO (Containers of Leaf Objects).

L'un des programmes de tests, DNeStr (Deeply Nested Structure), produit un dépassement de pile à l'exécution lorsqu'il est compilé avec iss-base. Ce problème vient probablement d'un marquage récursif, comme nous l'avons expliqué en section 3.4.2.

Tous ces résultats montrent que les performances de SmallEiffel en termes d'occupation mémoire sont aussi bonnes sous Windows que sous UNIX, quand on les compare avec celle de BDW, et sont bien meilleures que celle d'un important compilateur Eiffel commercial.

Temps d'exécution La comparaison des temps d'exécution faite entre BDW et SmallEiffel sous UNIX, page 78, semble elle aussi toujours valide sous Windows. De plus, les résultats en termes de vitesse d'exécution obtenus par notre méthode implantée dans SmallEiffel sont encore meilleurs que ceux sous UNIX. En effet, le ramasse-miettes de SmallEiffel dépasse maintenant le BDW sur 11 schémas d'exécution, dont 8 montrent un avantage en vitesse entre 40 % et 80 % et 4 un avantage de plus de 80 %. Nous estimons que cette augmentation des performances du ramasse-miettes de SmallEiffel par rapport à BDW vient du fait que BDW a initialement été développé sous UNIX et pourrait donc ne pas être aussi mature et optimisé sur la plate-forme "Wintel".

Lorsqu'on les compare avec celles d'iss-base, les performances du ramasse-miettes de SmallEiffel apparaissent remarquables. Sur tous les programmes de test, SmallEiffel est plus rapide qu'iss-base d'au moins 10 %. Cinq de ces programmes montrent un avantage en faveur de SmallEiffel compris entre 40 et 80 %, alors que dans 7 cas cet avantage dépasse 80 %.

Ceci montre que notre méthode basée sur la génération d'un ramasse-miettes spécialisé ne dépend pas d'une plate-forme particulière, puisqu'elle est efficace non seulement sous UNIX mais également sous Windows. Ses performances varient cependant d'une plate-forme à l'autre, les résultats sous Windows étant encore meilleurs que ceux sous UNIX.

On peut également remarquer que les temps de compilation, que nous n'avons pas détaillés ici, ont été nettement meilleurs avec SmallEiffel qu'avec iss-base (que ce soit la compilation d'Eiffel vers C ou la compilation de C vers l'exécutable). En effet, SmallEiffel s'avère en général plus rapide d'au moins un facteur 10.

3.7.4 Résultats expérimentaux avec des variantes d'un même programme

En plus des schémas d'exécution synthétiques présentés dans les précédentes sections, nous avons testé le comportement de l'implantation de notre méthode pour le ramasse-miettes de SmallEiffel avec des programmes de jeu Othello (ou Réversi).

Ces derniers avaient été conçus par 24 groupes d'étudiants utilisant une version précédente de SmallEiffel, qui ne comportait pas de ramasse-miettes. Ceci nous a permis de comparer de vrais programmes réalisant le même type de tâche avec différents algorithmes et *styles de programmation*, résultant en différents comportements à l'exécution, notamment par rapport à la mémoire.

Deux de ces programmes Othello étaient incorrects (échouaient suite à des violations d'assertions) et n'ont donc pas pu être utilisés. Les 22 programmes restant pouvaient être divisés en deux catégories: les programmes sans fuite, où la mémoire avait été gérée avec parcimonie et les programmes avec fuites où se trouvaient de nombreux objets ayant une durée de vie courte.

Par souci de concision, nous ne présentons ici que les résultats obtenus sur une plate-forme UNIX. Les résultats sur les autres plates-formes sont similaires et conduisent aux mêmes interprétations et conclusions.

A cause des différences de complexité entre les divers algorithmes de jeu mis en oeuvre par les différents groupes et à cause des différences majeures dans l'efficacité des implantations de ces algorithmes, nous avons dû mesurer les performances de ces programmes avec différentes tailles de plateau de jeu, afin d'obtenir les résultats les plus significatifs. Bien entendu, chaque programme a été testé avec la même taille de plateau dans ses 3 différentes versions: sans ramasse-miettes, avec le BDW et avec le ramasse-miettes de SmallEiffel. Pour chacun de ces tests, nos résultats ont été obtenus en lançant le programme 4 fois consécutives, sous une charge machine sensiblement constante, et en prenant la moyenne des 3 dernières exécutions — la première n'étant pas prise en compte afin d'éviter d'intégrer à la mesure des phénomènes parasites comme le temps de chargement du programme en mémoire. Ces tests ont été répétés, de façon à en vérifier la stabilité des résultats.

3.7.4.1 Taille des exécutables

Comme le BDW est une bibliothèque autonome très compacte, son surcoût sur la taille de l'exécutable est constant et d'environ 45 KO. Le ramasse-miettes de SmallEiffel, au contraire, génère du code spécialisé additionnel pour chaque type vivant. Évidemment, on peut se demander si cet inconvénient intrinsèque de notre méthode n'est pas un problème pour les programmes qui comportent un très grand nombre de types vivants. Sur le compilateur SmallEiffel version `-0.80` lui-même, qui comporte environ 300 types vivants, l'augmentation de taille induite par le ramasse-miettes est d'environ 440 KO, ce qui représente 45 %. Il faut néanmoins être conscient que, sur SmallEiffel comme sur de nombreux autres programmes sur lesquels nous avons fait cette mesure, ce surcoût représente seulement environ 1,5 KO par type vivant, ce qui semble tout à fait raisonnable.

3.7.4.2 Les programmes sans fuite

Quatorze des programmes Othello étaient sans fuite, comportant des instanciations d'objets assez parcimonieuses et réutilisant les objets autant que possible. Ceci s'explique par le fait qu'ils avaient été développés à une époque où SmallEiffel ne fournissait pas son propre système de ramasse-miettes intégré. La figure 3.12 montre les résultats obtenus sur ces programmes.

Taille mémoire Comme on pouvait s'y attendre, pour tous ces programmes sans fuite, la taille mémoire maximale s'est révélée sensiblement la même sans ramasse-miettes et avec le BDW ou le ramasse-miettes de SmallEiffel. L'un des 14 programmes Othello nécessitait environ 2 MO de mémoire, alors que les autres utilisaient environ 1 MO seulement.

Dans tous les cas, l'exécution du programme Othello sans ramasse-miettes a conduit à la plus petite utilisation mémoire. En effet, avec des programmes sans fuite, un ramasse-miettes collecte peu — ou pas — d'objets, alors qu'il occupe de la place puisqu'il implique du code et des structures de données supplémentaires.

Sur 6 des Othellos, SmallEiffel autorise une taille mémoire plus faible que le BDW, la différence étant de 1 à 7 %, alors que le BDW a un avantage de 1 à 9 % pour 7 programmes. Tous deux requièrent en général environ 25 % de mémoire de plus que la version `-no_gc`, ce qui représente environ 300 KO sur nos programmes de tests. Les performances atteintes par le BDW et le ramasse-miettes de SmallEiffel sont donc très similaires, ce qui confirme la validité de notre approche.

Temps d'exécution Sur ces programmes sans fuite, les temps d'exécution sans ramasse-miettes ou avec n'importe lequel des deux sont généralement semblables.

Globalement, la version `-no_gc` est la plus rapide, dépassant les deux autres ramasse-miettes dans 6 cas sur 14, avec un avantage atteignant les 8 % par rapport au plus rapide des deux. Dans un cas seulement, la version la plus rapide est celle avec le BDW alors que dans 4 cas il s'agit de celle avec le ramasse-miettes de SmallEiffel.

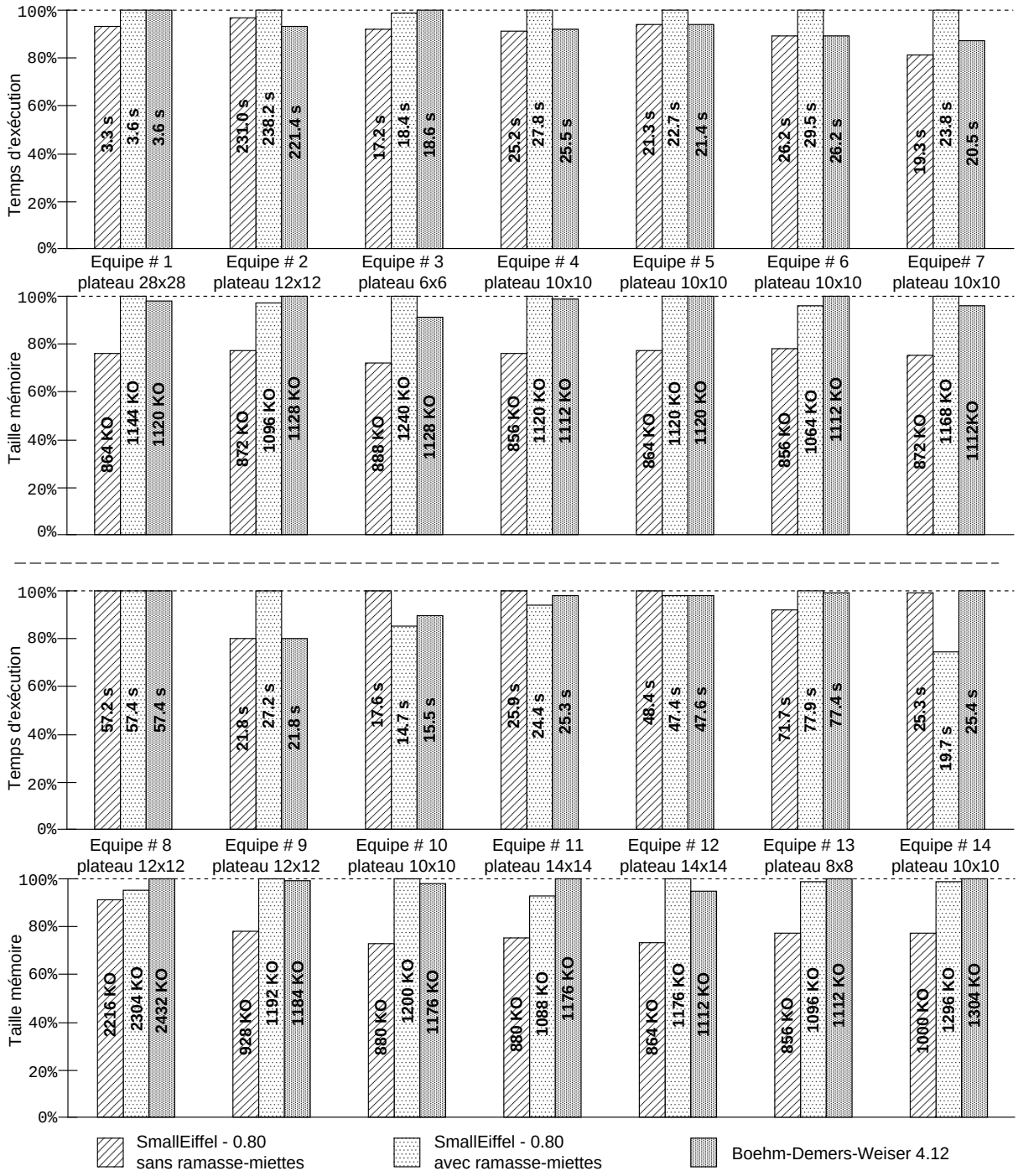


FIG. 3.12 – *Comparaisons des temps d'exécution et tailles mémoire des programmes sans fuites sous UNIX.*

Ces résultats confirment que, pour les programmes sans fuite, il est préférable de ne pas utiliser de ramasse-miettes du tout et que l'utilisation de celui de SmallEiffel tout comme du BDW résulte en un ralentissement — toutefois limité — du programme.

Quand seuls le BDW et SmallEiffel sont considérés, le premier se révèle plus rapide dans 7 cas, jusqu'à 20 % (programme n°9). En revanche, SmallEiffel offre un avantage en terme de vitesse d'exécution dans

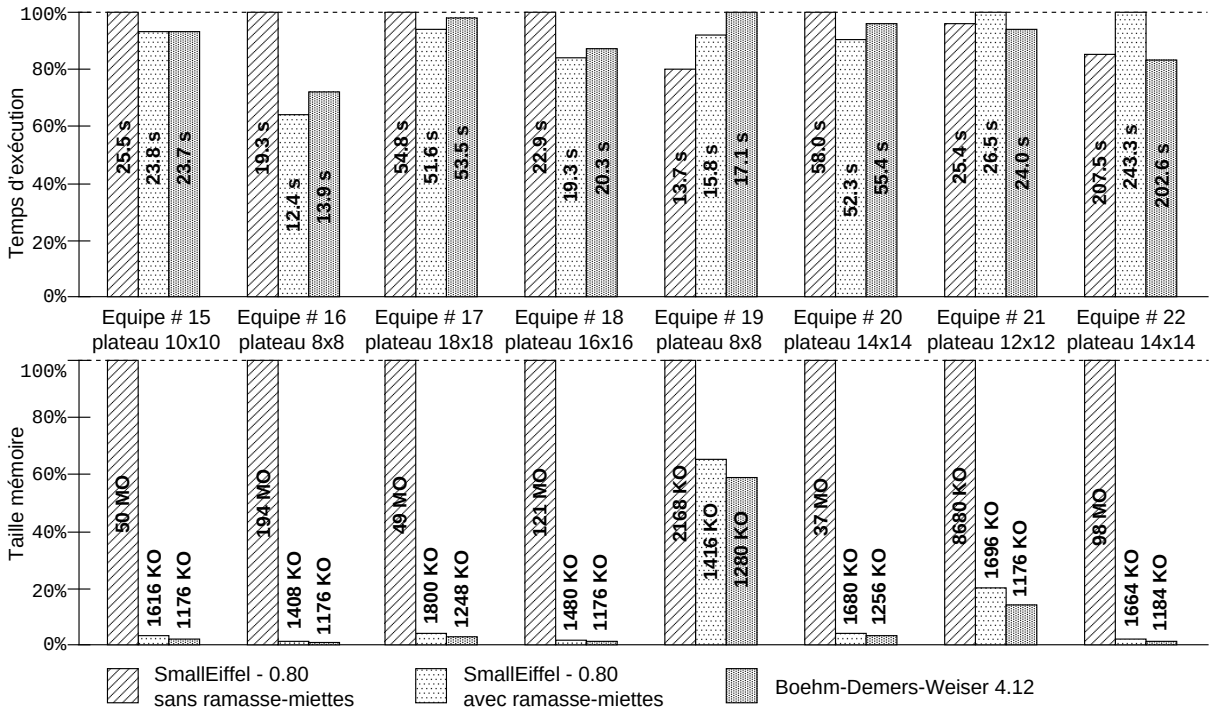


FIG. 3.13 – *Comparaisons des temps d'exécution et tailles mémoire des programmes avec fuites sous UNIX.*

5 cas, avec un écart qui atteint jusqu'à 20 % (programme n°14). Ceci montre que le surcoût induit par le ramasse-miettes sur des programmes sans fuite tend à être plus faible avec le BDW qu'avec SmallEiffel. Ceci peut s'expliquer par le fait que l'implantation du ramasse-miettes de SmallEiffel est encore très récent et peut encore être optimisé.

3.7.4.3 Les programmes avec fuites

Nous avons bien entendu également mesuré les performances de 8 programmes Othello ayant des fuites de mémoire, parmi lesquels une gestion très insuffisante des allocations et désallocations mémoire causait des fuites très importantes — de 37 à 193 MO — dans 6 cas. Nous pensons que ces programmes constituent un banc d'essai plus représentatif de l'utilisation typique de la mémoire quand le développeur se fie à un ramasse-miettes automatique. La figure 3.13 montre les résultats que nous avons obtenus.

Taille mémoire L'utilité d'un ramasse-miettes apparaît clairement pour ces programmes ayant des fuites de mémoire. Bien que les versions `-no_gc` prennent entre 2 et 194 MO de mémoire, tous les programmes utilisant soit le BDW soit le ramasse-miettes de SmallEiffel ont une consommation mémoire très raisonnable, de 1 à 2 MO, similaire à celle des programmes sans fuites. Ceci montre que les deux collecteurs considérés font effectivement le travail qu'on en attendait.

Lorsque seules les versions BDW et SmallEiffel sont considérées, leur performances sur le plan mémoire sont équivalentes. Néanmoins, le BDW a en général un avantage de 136 à 552 KO, ce qui sur ces programmes ayant une faible consommation mémoire optimale se traduit par une différence de 10 à 31 %.

Temps d'exécution Globalement (5 cas sur 8), les versions avec l'un ou l'autre des ramasse-miettes sont plus rapides que la version `-no_gc`. Ceci s'explique par le fait que moins de mémoire doit être allouée par le système d'exploitation, grâce au recyclage des objets morts par le ramasse-miettes du programme. Le seul cas où la version `-no_gc` est la plus rapide — le programme n°19 — confirme également cette

Équipe		#15	#16	#17	#18	#19	#20	#21	#22
Nombre d'appels au ramasse-miettes	BDW	473	1636	276	1386	12	352	60	820
	SE	97	454	57	286	4	74	10	119
Temps moyen par cycle de marquage-balayage (ms)	SE	1.4	1.1	2.5	1.0	0.9	1.4	3.3	2.9
Temps total passé dans le ramasse-miettes (ms)	SE	795	3466	930	1811	22	549	223	1935
Temps total passé dans le ramasse-miettes (%)	SE	3.2	20.6	1.7	8.8	0.1	0.9	0.8	0.7

TAB. 3.1 – *Comportement du ramasse-miettes de SmallEiffel sur des programmes avec fuites mémoires, sous UNIX.*

explication. En effet, ce programme est de très loin celui où la fuite mémoire est la plus petite, moins d'1 MO, ce qui limite donc le gain qu'un ramasse-miettes peut apporter. Sur la plupart des programmes, ce gain est d'environ 10 % pour les deux versions avec ramasse-miettes, et il atteint même près de 30 % sur le programme gérant le plus mal ses objets (le programme n°16, avec une fuite de 193 MO).

Ainsi, sur les programmes qui présentent des fuites mémoire, le ramasse-miettes de SmallEiffel se comporte bien par rapport au BDW. Ce dernier est meilleur que SmallEiffel dans 3 cas, avec un avantage sur le temps d'exécution qui atteint 17 % (sur le programme n°22), alors que SmallEiffel est jusqu'à 11 % plus rapide (programme n°16) dans 5 cas.

Le tableau 3.1 nous permet de montrer plus précisément le comportement du ramasse-miettes de SmallEiffel (SE). Le nombre d'appels au ramasse-miettes pour BDW est donné pour mémoire, pour mieux illustrer l'activité mémoire du programme considéré. Comme on peut le voir, le BDW est appelé de 3 à 7 fois plus souvent que le ramasse-miettes de SmallEiffel. Ce dernier, n'étant pas incrémental, est donc susceptible de causer des pauses plus longues dans l'exécution du programme. Le temps moyen par appel au ramasse-miettes (cycle de marquage-balayage), qui va de 0,9 à 3,3 millisecondes, apparaît raisonnable pour la plupart des programmes, exceptés les programmes temps réel à contraintes strictes.

Globalement, le temps total passé dans le ramasse-miettes (incluant les allocations et les cycles de marquage-balayage, mais excluant l'initialisation des structures du ramasse-miettes[35]) peut être aussi faible que 0,1 % et aussi élevé que 20,6 % du temps total d'exécution du programme. Cette importante différence entre les deux extrêmes s'explique facilement compte tenu du comportement des programmes mesurés, comme le montre la figure 3.13. En effet, le programme n°16 comporte énormément de fuites de mémoire puisqu'il produit environ 193 MO d'objets morts, bien plus de 150 fois la taille mémoire maximum dont il a besoin, ce qui impose une charge de travail considérable au ramasse-miettes (20,6 %). Au contraire, le programme n°19 produit environ 1 MO d'objets morts, ce qui ne requiert pas beaucoup de travail de la part du ramasse-miettes (0,1 %). Ceci est confirmé par des résultats similaires obtenus sur les programmes sans fuite. Notons également que cette variabilité correspond aux valeurs moyennes indiquées par [Jones et Lins, 1996], qui sont de quelques pour cent à 20 %.

Sur l'ensemble des programmes avec fuites, le ramasse-miettes de SmallEiffel prend en moyenne 4,6 % du temps d'exécution, ce qui constitue un excellent résultat, puisque les spécialistes du domaine considèrent actuellement que 10 % constitue une valeur moyenne raisonnable pour un système bien conçu [Wilson, 1994].

En conséquence, bien qu'il n'ait pas la maturité du BDW, le ramasse-miettes de SmallEiffel apparaît comme très prometteur et est déjà capable de se mesurer avec ce qui constitue un système de gestion automatique de la mémoire très efficace et très renommé.

35. Ces initialisations, ainsi qu'une localité diminuée, peuvent aussi expliquer une partie des différences de performances entre la version `-no_gc` et celle avec le ramasse-miettes de SmallEiffel.

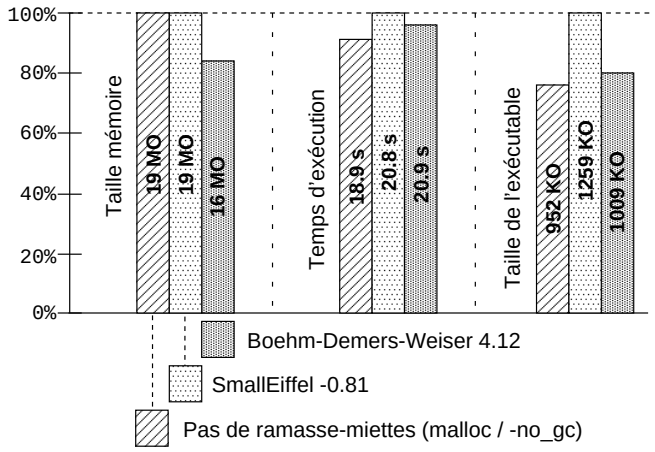


FIG. 3.14 – *Auto-compilation de SmallEiffel: le test* `compile_to_c`.

3.7.5 Résultats expérimentaux avec le banc d'essai SmallEiffel

Comme nous l'avons indiqué au chapitre 2, le compilateur SmallEiffel (commande `compile_to_c`) est le plus gros programme de tests Eiffel dont nous disposons. Plus de 99 % du compilateur est écrit en Eiffel pur: analyse lexicale, analyse syntaxique, analyse sémantique, prédiction de type, génération de code (y compris la génération du ramasse-miettes spécialisé). Auto-compilé en Eiffel, cette application réelle et complète comprend donc environ 70000 lignes d'Eiffel pour 300 classes vivantes[36].

Comme SmallEiffel a tout d'abord été conçu sans ramasse-miettes, il s'agit d'un programme qui contient très peu de fuites de mémoire, où les objets alloués deviennent très rarement des objets morts. En conséquence, SmallEiffel suit un schéma d'exécution où l'utilisation mémoire croît de façon continue durant une première étape, puis reste en gros constante jusqu'à la fin de l'exécution. Ceci correspond à un schéma d'exécution en "plateau", selon la définition de [Wilson *et al.*, 1995].

Bien entendu, un tel programme n'est pas le plus approprié pour tester le comportement d'un ramasse-miettes, puisque très peu de recyclage se produit. Il est toutefois très significatif en ce qui concerne les phases d'allocation mémoire, de marquage et de balayage. Tester le ramasse-miettes sur SmallEiffel lui-même revient donc à révéler si le surcoût du ramasse-miettes est important sur des schémas d'exécution sans fuites de mémoire. En fait, on peut considérer qu'un programme comme SmallEiffel ne se fie pas au ramasse-miettes, mais est *gêné* par ce dernier. De même, le ramasse-miettes est mis dans une situation défavorable avec un tel programme, puisque le recyclage des objets (la raison d'être du ramasse-miettes) sera très faible.

La figure 3.14 montre les résultats que nous avons obtenus en compilant `compile_to_c` avec lui-même. La plate-forme et le compilateur C utilisés sont ceux décrit pour la plate-forme UNIX en section 3.7.1. Comme ce banc d'essai comporte très peu de fuites de mémoire, nous avons aussi inclus les résultats sans aucun ramasse-miettes (une allocation système `malloc` pour chaque objet).

La comparaison des occupations mémoire produites montre que notre ramasse-miettes spécialisé n'introduit pas de surcoût sensible en terme de mémoire occupée par le programme à l'exécution, celle-ci étant quasiment identique entre le mode normal (avec ramasse-miettes de SmallEiffel) et le mode `-no_gc`. L'utilisation mémoire la plus faible est obtenue avec BDW grâce à la compacité des informations rajoutées pour la gestion du ramasse-miettes (i.e. en fait un bit de marquage par objet plus quelques structures globales). Les temps d'exécution sont très similaires mais l'exécution sans ramasse-miettes est la plus rapide, ce qui n'est pas surprenant car le mutateur considéré comporte très peu de fuites de mémoire.

La taille des exécutables montre que le ramasse-miettes spécialisé cause un accroissement de taille des exécutables de 24 %. Cette taille dépend du nombre de types concrets vivants et croît linéairement avec le nombre de type. Bien que cet accroissement ne puisse être négligé, il reste tout de même acceptable. De plus nous pensons qu'il est encore possible de factoriser certaines parties du code, et donc de gagner

36. Chiffres de la version `-0.81` avec laquelle la plupart des mesures sur le ramasse-miettes ont été faites.

en taille, sans pour autant diminuer les performances. Par exemple, le code pour les conteneurs d'objets feuilles (conteneurs de caractères, d'entiers...) peut être factorisé. C'est aussi le cas lorsque deux types d'objets feuilles de taille fixe ont exactement la même taille. Enfin, le code de liaison dynamique au sein de certaines fonctions de marquage (voir section 3.4.2) peut être optimisé de la même manière que celle décrite en section 2.6 pour la liaison dynamique des routines utilisateur.

3.8 Autres travaux liés à la spécialisation des ramasse-miettes

Un certain nombre de travaux ont été menés sur les ramasse-miettes spécialisés, notamment depuis 1990. La plupart de ces travaux sont basés sur des informations fournies explicitement par le développeur.

Les recherches menées par Detlefs *et al.* [Detlefs *et al.*, 1994] tendent à montrer que les performances des ramasse-miettes conservateurs se comparent favorablement avec la désallocation explicite de la mémoire.

Mais bien que les collecteurs conservateurs classiques se comportent correctement, l'adjonction d'un peu d'informations supplémentaires à propos des schémas mémoire spécifiques à l'application peut probablement améliorer leurs résultats. Ces informations peuvent provenir de différentes sources: du développeur, d'un algorithme de prédiction de type, de statistiques d'exécution (*profiling*), ...

Grunwald et Zorn décrivent dans [Grunwald et Zorn, 1993] leur système CustoMalloc. Après avoir fait un profil d'exécution d'un programme, CustoMalloc produit un allocateur mémoire (`malloc`) et un désallocateur (`free`) spécialisés qui sont à même de gérer plus efficacement les tailles d'objets les plus fréquentes. Leurs travaux montrent en effet sur divers programmes d'usage courant que quelques classes de tailles d'objets — généralement les petites tailles — constituent la quasi totalité de la mémoire allouée. Contrairement à CustoMalloc, SmallEiffel génère non seulement un allocateur et un désallocateur spécialisés, mais bien un système de ramasse-miettes *complet* spécialisé. De plus, comme dans notre système seule l'analyse statique est utilisée pour fournir l'information nécessaire à la spécialisation, aucune pré-exécution n'est nécessaire. Il semble néanmoins possible de faire bénéficier SmallEiffel d'informations tirées de profils d'exécution, notamment pour prédire les tailles des objets redimensionnables et les classes d'objets de taille fixe les plus utilisées.

Dans [Boehm et Shao, 1993], Boehm et Shao ont étudié les performances d'une version améliorée du BDW basée sur de l'inférence de type lors des phases de collection, grâce à un `malloc` typé par le développeur. Leur technique consiste à observer pendant l'exécution les premiers objets alloués pour chaque type et à ainsi inférer une "carte des types". Cette information supplémentaire permet un marquage plus efficace de la mémoire et accélère la collection dans certains cas.

Le ramasse-miettes *Mostly Copying collector* de Bartlett [Bartlett, 1988] est un hybride de collecteur copieur et conservateur. Il ne présuppose aucune connaissance des registres du processeur ou de l'organisation de la pile d'exécution, mais suppose que tous les pointeurs dans la zone mémoire allouée en tas peuvent être trouvés avec précision grâce à l'enregistrement par le développeur de toutes les racines internes [Bartlett, 1990].

D'autres expérimentations autour de la spécialisation des ramasse-miettes ont été menées dans [Attardi et Flagella, 1994] et [Attardi *et al.*, 1995]. Leur gestion mémoire adaptable — *Customizable Memory Management (CMM)* — autorise la spécialisation de la gestion mémoire par les développeurs, qui indiquent pour chaque allocation d'objet quelle stratégie doit être adoptée pour son stockage. Ainsi, pour chaque type, une routine de parcours optimale est fournie. Le principal intérêt de ce dernier système de spécialisation de la gestion mémoire explicite, "à la main", est qu'il permet une gestion tenant compte du type exact des objets (*type-accurate*).

Dans le ramasse-miettes pour C++ d'Edelson [Edelson, 1992; Edelson, 1993], les routines de marquage sont automatiquement produites par un préprocesseur qui génère un appel à une fonction de marquage pour chaque pointeur dans la classe. Cette substitution syntaxique réduit les erreurs d'identification sur les pointeurs internes et accélère le processus de marquage.

Dans [Branquart et Lewi, 1971] , Branquart et Lewi décrivent une méthode qui utilise des informations de type disponibles lors de la compilation pour générer automatiquement des tables mettant en correspondance des positions dans la pile et les routines de collection appropriées, dans une implantation Algol-68.

Diwan *et al.* [Diwan *et al.*, 1992], ainsi qu'Agesen et Detlefs [Agesen et Detlefs, 1997], décrivent des types similaires et améliorés de ramasse-miettes avec support du compilateur, qui permettent une collection *exacte*. Afin de pouvoir trouver les pointeurs dans la pile et les registres du processeur à l'exécution, leurs compilateurs génèrent statiquement des tables qui encodent la position de ces pointeurs pour tous les points où une collection mémoire pourrait avoir lieu. Lors de ces collections, les adresses de retour des contextes d'exécution de la pile donnent accès à ces tables.

Les travaux de Goldberg [Goldberg, 1991] étudient également une recherche efficace des racines dans la pile d'exécution grâce à des routines spécifiques. Bien que reliée à la méthode de Branquart et Lewi, celle de Goldberg en diffère de façon importante car elle évite l'utilisation de tables pour cartographier la pile. Des routines spécifiques peuvent être générées pour chaque fonction afin de tracer les pointeurs locaux dans chaque contexte d'activation (*frame*). En suivant les adresses de retour stockées dans la pile, il est possible de déterminer tous les contextes d'activation à un instant donné et d'appeler pour chacune des fonctions correspondantes la routine de marquage appropriée.

Une différence importante entre SmallEiffel et la plupart des recherches précédemment décrites est que SmallEiffel génère *automatiquement* des routines de gestion mémoire spécialisées. En effet, grâce à la prédiction de type réalisée par SmallEiffel lors de la compilation, aucune information additionnelle — qu'elle soit fournie par le développeur ou par la collection de statistiques sur des pré-exécutions — n'est nécessaire.

3.9 Conclusions et perspectives

Nous avons décrit dans ce chapitre l'implantation dans un compilateur Eiffel d'une technique de génération d'un ramasse-miettes spécialisé avec support du compilateur, basée sur un algorithme classique de type marquage-balayage. Contrairement à de nombreux travaux précédents, cette spécialisation est entièrement et automatiquement réalisée par le compilateur, sans aucune intervention du développeur. La majeure partie de cette technique n'est pas spécifique à notre système et peut probablement être adaptée à d'autres langages de classes et à d'autres algorithmes de ramasse-miettes, même distribués.

Nous avons testé dans la section 3.7 l'implantation de notre méthode pour le ramasse-miettes de SmallEiffel de façon très complète, avec de nombreux programmes, de taille variable, qui présentaient différents *styles de programmation*, et qui nous ont permis d'évaluer les performances de notre ramasse-miettes face à différents types d'utilisation de la mémoire sur diverses plates-formes. Les résultats obtenus sur ces nombreuses exécutions montrent clairement la validité de notre approche de la génération d'un ramasse-miettes spécialisé grâce à l'analyse globale du programme, en ce qui concerne tant la taille mémoire des programmes générés que leur temps d'exécution.

Bien que le ramasse-miettes généré par SmallEiffel se comporte fort bien, il pourrait bénéficier de l'adjonction de certaines capacités. L'incrémentalité des appels au ramasse-miettes, par exemple, peut être très utile dans certaines situations (temps réel notamment) et semble donc un point important à explorer.

Nos travaux futurs pourraient se focaliser sur l'amélioration des performances de notre système de ramasse-miettes. L'addition à SmallEiffel d'une analyse de flot ou d'une analyse de profils d'exécution pourrait fournir des informations supplémentaires au ramasse-miettes à propos des besoins en mémoire, comme par exemple les tailles d'objets les plus fréquentes. Ceci aiderait à mieux régler le ramasse-miettes et permettrait un degré accru de spécialisation des routines de gestion mémoire. Un balayage et une fusion des blocs différés sont également de nature à améliorer le comportement du ramasse-miettes tel qu'il est perçu par l'utilisateur d'un programme interactif, en retardant des opérations qui ne sont pas immédiatement nécessaires.

Enfin, le passage à une version du ramasse-miettes non conservatrice, c'est à dire complètement *type-accurate*, permettrait de générer un code qui serait totalement portable sur diverses architectures. L'impact d'une telle modification sur les performances du ramasse-miettes n'est a priori pas évident et devrait donc faire l'objet d'études supplémentaires.

Chapitre 4

Impact de l'aliasing dans un compilateur Eiffel

4.1 Introduction

Ce chapitre présente les travaux que nous avons effectués autour de *l'aliasing* dans le cadre d'un compilateur et du langage Eiffel, plus précisément avec le compilateur SmallEiffel. Ces travaux ont été publiés dans [Zendra et Colnet, 1999b; Zendra et Colnet, 2000b; Zendra et Colnet, 2000a].

Ce chapitre se caractérise par une double originalité, par rapport à cette thèse et par rapport aux travaux généralement menés autour de l'aliasing. Son originalité au sein de cette thèse vient du fait qu'il constitue pour nous une direction de travail différente de celles que nous avons détaillées dans les deux précédents chapitres. En effet, ceux-ci ont plutôt pour sujet des techniques de *génération de code* optimisé, alors que ce chapitre présente une technique de *codage* optimisé. L'originalité de nos travaux par rapport à ceux du domaine de l'aliasing tient quant à elle à l'approche "positive" envers l'aliasing que nous avons adoptée, et qui a notre connaissance a rarement été suivie.

En effet, l'aliasing est connu depuis déjà longtemps (voir par exemple [Meyer, 1988]), et est en général considéré comme quelque chose de dangereux qui devrait être évité autant que possible dans les programmes. Beaucoup de travaux se sont donc concentrés sur la détection, la prévention et les moyens d'éviter l'aliasing [Hogg, 1991; Hogg *et al.*, 1992; Minsky, 1996]. Il semble cependant très difficile, si ce n'est impossible, d'éviter l'aliasing dans les langages à objets et des techniques ont donc également été recherchées pour le contrôler.

Nous pensons au contraire que l'aliasing est inhérent à la programmation à objets et ne peut être supprimé sans que l'on perde une bonne partie de la facilité de développement que les objets permettent. Nous sommes également convaincus que l'aliasing offre des avantages certains et significatifs en termes de performances et qu'il peut donc être utilisé avec profit.

Pour ces raisons, notre compilateur Eiffel, SmallEiffel, *The GNU Eiffel Compiler*, fait un usage intensif mais soigneusement conçu de l'aliasing, essayant de réconcilier performance et sûreté à l'aide de techniques pour la plupart déjà connues. Dans ce chapitre, qui rapporte notre expérience avec SmallEiffel, nous montrons donc comment certains choix de conception simples peuvent permettre un usage plus sûr de l'aliasing, même lorsque le langage, comme Eiffel, n'intègre pas lui-même le support de l'aliasing.

Ce chapitre est organisé comme suit. La section 4.2 rappelle tout d'abord brièvement ce qu'est l'aliasing et la section 4.3 présente l'état des recherches dans le domaine. Puis dans la section 4.4, nous expliquons pour quelles raisons nous considérons que l'aliasing est très utile dans un compilateur. La section 4.5 introduit ensuite le concept de *fournisseur d'alias* et détaille comment celui-ci peut être mis en oeuvre en Eiffel en utilisant le modèle de conception "singleton" [Gamma *et al.*, 1995; Jézéquel *et al.*, 1999]. Puis nous décrivons dans la section 4.6 comment le *design by contract* et les assertions Eiffel peuvent être utilisés pour rendre l'emploi de l'aliasing plus sûr. La section 4.7 considère le problème de l'aliasing des objets modifiables. Enfin, les bénéfices dûs à l'utilisation de l'aliasing sont discutés en section 4.8 et la section 4.9 conclut.

4.2 L'aliasing: définition

On peut dire de façon humoristique que l'aliasing est a de nombreux informaticiens ce que la prose est à Monsieur Jourdain. En effet, l'aliasing est le fait de *référencer une même donnée via deux variables différentes*, ou plus généralement deux chemins différents.

La donnée en question peut être aussi simple qu'un booléen, ou aussi évoluée qu'une grande structure de données ou un objet complexe. L'aliasing se produit dans tous les types de langages, pas nécessairement à objets.

Bien entendu, l'aliasing est extrêmement courant dans de nombreux types de programmes. On comprend aisément le danger qu'il peut représenter. En effet, lorsque le développeur manipule un objet aliasé[37], il n'est pas certain que lui seul contrôle les modifications qui peuvent survenir sur cet objet, puisqu'un autre alias peut être utilisé pour changer son état. L'aliasing brise donc, en quelque sorte, la séquentialité et l'atomicité apparentes de certains morceaux de code. Ce problème est bien entendu exacerbé lorsqu'on est dans un contexte d'exécution parallèle, par exemple dans un programme multi-threads.

En revanche, lorsque le développeur est conscient des aliasings qui sont présents dans le programme sur lequel il travaille, il peut les gérer, et en tirer parti.

Il est à noter que, comme le montre notre revue des travaux du domaine ci-dessous, l'aliasing est la plupart du temps considéré comme un *phénomène* néfaste. Nous avons adopté dans nos recherches une approche qui consiste plutôt à le considérer comme une *technique* utile, comme nous le détaillerons à partir de la section 4.4

4.3 Travaux du domaine

Dans cette section, nous présentons brièvement certains des travaux qui ont été réalisés depuis déjà pas mal de temps sur l'aliasing et l'analyse de pointeurs. Comme l'aliasing peut dans certains cas être à l'origine d'importants problèmes, voir de cauchemars de déverminage, ces travaux se sont souvent focalisés sur la prévention de l'aliasing et les moyens de l'éviter. La détection des alias a aussi été un sujet de recherche actif, notamment en relation avec les techniques d'optimisation des programmes.

Hogg décrit dans [Hogg, 1991] une méthode basée sur des modes d'accès permettant de limiter l'aliasing et de créer des *îlots* (*islands*) d'objets. Un *îlot* est un ensemble d'objets qui peuvent se référencer les uns les autres librement, mais pour lesquels il n'existe qu'une porte d'accès vers et depuis le monde extérieur, chargée de contrôler et de restreindre l'aliasing. Ainsi, l'aliasing *intra îlot* est autorisé, ce qui permet d'utiliser des modèles de conception utilisant l'aliasing, tandis que l'aliasing *inter-îlots* est restreint, ce qui permet aux *îlots* d'éviter les problèmes potentiellement créés par l'aliasing.

Les *types ballons* (*balloon types*) [Almeida, 1997] présentent certaines similarités avec les *îlots*. Cependant, les *types ballons* sont plus puissants, car il s'agit d'extensions aux langages de programmation qui sont intégrées au système de type. Ils permettent un contrôle plus complet vis-à-vis du partage de l'état interne d'un type de donnée (encapuslation).

Un certain nombre d'analyses d'alias ou de pointeurs ont été réalisées et divers algorithmes développés par les chercheurs depuis une dizaine d'années. [Wilson et Lam, 1995] détaille une analyse de pointeurs sensible au contexte basée sur des fonctions de transfert partielles (*partial transfer functions*) chargées de "résumer" les effets d'une routine et montre des résultats encourageants sur des programmes C. Néanmoins, [Ruf, 1995] indique que la prise en compte du contexte n'améliore pas toujours l'analyse et que dans les cas qu'ils ont étudiés, l'analyse d'alias sans prise en compte du contexte peut être à la fois "étonnamment précise" (*"surprisingly precise"*) et efficace.

Une technique pour éviter les alias est décrite dans [Minsky, 1996], basée sur des *objets non partageables* (*unshareable objects*) référencés par des *pointeurs uniques* qui peuvent être déplacés mais pas copiés. Quand une affectation implique des pointeurs uniques, la source de l'affectation est nullifiée, garantissant ainsi que la destination de l'affectation sera la seule (unique) référence restant sur l'objet. Ceci permet l'implantation, avec un certain coût à l'exécution, d'objets qui ne sont jamais sujets à l'aliasing, en plus des objets normaux qui eux peuvent être aliasés.

37. Nous utilisons dans ce mémoire ce néologisme assez malheureux, traduction de l'anglais *aliased*, à défaut d'avoir une expression française raisonnablement courte ayant précisément la même sémantique.

L'analyse d'alias présentée dans [Debray *et al.*, 1998] est effectuée à bas niveau, directement sur le code exécutable. L'avantage principal de cette approche est que les aspects bas niveau liés aux pointeurs (arithmétique de pointeurs, ...) peuvent être pris en compte, alors qu'ils ne le sont pas dans les analyses de haut niveau (langage).

Ghiya et Hendren montrent dans [Ghiya et Hendren, 1998] comment deux analyses de pointeurs existantes — la *points-to analysis* et la *connection analysis* — peuvent être comparées et comment elles peuvent être utilisées pour étendre et améliorer certaines optimisations classiques dans un compilateur, comme l'extraction d'invariants de boucles, l'élimination des sous-expressions communes et les tests de dépendance sur les tableaux (*array dependence testing*).

Une protection d'alias flexible (*flexible alias protection*) est proposée dans [Noble *et al.*, 1998]. Grâce à l'addition d'une information explicite à chaque déclaration afin de spécifier le *mode d'aliasing* des objets, elle permet de contrôler le degré d'aliasing et les modifications autorisées depuis le code client. Cette technique impose une extension des langages à objets actuels, mais n'implique aucun surcoût à l'exécution et permet d'éviter les problèmes potentiellement causés par l'aliasing tout en gardant un style de programmation à objets relativement naturel.

Un système de type formel pour une partie du travail précédent a été conçu dans [Clarke *et al.*, 1998]. Il se base sur des *ownership types* pour contrôler la visibilité des références et les chemins d'accès aux objets, permettant de garder la représentation interne des objets confinée (*representation containment*).

Des extensions à Java sont proposées dans [Bokowski et Vitek, 1999] pour implanter des *types confinés* (*confined types*). Ces derniers sont des types dont les instances sont confinées dans (i.e. ne peuvent être référencées depuis l'extérieur du) module, ou *package*, auquel ils appartiennent. Les *types confinés* n'impliquent aucun surcoût à l'exécution et permettent de déceler statiquement, à la compilation, les erreurs d'aliasing et de dissémination.

Trois types d'analyses d'alias sont comparés dans [Diwan *et al.*, 1999]. La première est basée sur les seules informations de compatibilité de types. La seconde étend la précédente en ajoutant des informations de haut niveau à propos des attributs. Enfin, la troisième étends la seconde avec une analyse ne prenant pas en compte le flot de données (*flow insensitive*). Des évaluation statiques et dynamiques montrent que la prise en compte de la compatibilité des types est d'un faible coût, mais fournit des informations d'aliasing imprécises, qui peuvent sensiblement être améliorées par les deux autres analyses. L'analyse basée sur les types apparaît très efficace et permet des optimisations supplémentaires du programme (élimination des affectations redondantes).

Une nouvelle analyse de pointeurs inter-procédurale, prenant en compte le contexte et le flot d'exécution (*context-sensitive* et *flow sensitive*), est présentée dans [Rugina et Rinard, 1999] pour des programmes *multithreadés* où les pointeurs partagés peuvent être mis à jour de façon concurrente. Cet algorithme supporte un large ensemble de constructions, est correct et s'exécute en temps polynomial. Des résultats expérimentaux montrent que cet algorithme est assez précis et a de bonnes propriétés de convergence sur les bancs d'essais testés.

Yong *et al.* décrivent tout d'abord dans [Yong *et al.*, 1999] le difficultés supplémentaires introduites par les conversions de types (*type casting*), puis présentent un schéma (*framework*) d'analyse de pointeurs réglable permettant de gérer les structures en présence de conversions de types. Des résultats sont donnés pour 4 implantations de ce schéma. Ils montrent qu'il est important de distinguer les différents champs des structures et confirment qu'il est possible de faire à faible coût des hypothèses conservatrices afin de préserver la portabilité.

Comme nous l'avons précédemment mentionné, ces travaux et les nôtres ont des approches assez différentes. La plupart d'entre eux se focalisent sur la prévention et la limitation de l'aliasing, alors que nous avons choisi, lors de la conception de SmallEiffel, d'essayer de *nous baser sur* l'aliasing, d'une façon disciplinée, afin d'obtenir les meilleures performances possibles sans sacrifier la sécurité.

4.4 L'aliasing dans un compilateur: pour quoi faire?

Nos travaux, portant sur la compilation des langages à classes, ont tout naturellement étudié l'aliasing dans ce cadre, et notamment dans le compilateur SmallEiffel. Mais l'aliasing n'est bien entendu pas un

phénomène que l'on trouve uniquement dans le code source des compilateurs. Ceux-ci semblent cependant favoriser de façon importante son utilisation.

Une bonne illustration en est l'emploi de l'aliasing lorsqu'un identificateur est rencontré. En effet, une des tâches les plus communes d'un compilateur consiste à faire des comparaisons de noms, pour par exemple vérifier si une variable est déclarée, si est locale, globale, ou si elle est un argument de la routine englobante. Il est donc très fréquent dans un compilateur d'avoir besoin de savoir si deux symboles (ou, plus exactement, deux noms d'entité, rencontrés à différents endroits du code source) sont les mêmes ou non. Pour effectuer ce travail efficacement, il est capital d'éviter de coûteuses comparaisons sur les *contenus* de chaînes de caractères.

L'idée de base de l'aliasing est ici d'utiliser exactement la même chaîne de caractères quand plusieurs symboles ont le même nom. Ainsi, il est possible de comparer deux (noms de) symboles avec une simple comparaison de références, au lieu de devoir examiner l'ensemble des caractères qui composent le nom. Ceci évite les duplicata d'une même chaîne, ce qui économise une importante quantité de mémoire. Il semble également y avoir un gain possible en termes de vitesse. Cependant, le coût de la gestion d'un tel système d'aliasing des chaînes représentant les noms de symboles peut gommer une partie de ses avantages et doit donc être évalué plus précisément.

Bien entendu, les chaînes de caractères ne sont pas les seuls objets "aliasables" dans un compilateur, mais comme ce sont des objets communs et bien compris, nous les utiliserons comme exemple de base tout au long de ce chapitre. De nombreux autres types d'objets plus complexes sont aussi de bons candidats à l'aliasing, comme par exemple les différents objets utilisés pour représenter des noeuds de l'arbre abstrait [Aho et Ullman, 1977].

De plus, l'utilisation de l'aliasing n'est pas limitée aux compilateurs, mais peut en fait être appliquée à la plupart des types de logiciels, tels que les moteurs de recherche, les interfaces graphiques, etc.

4.5 Utilisation d'un fournisseur d'alias

Les noms de symboles partagés sont apparentés aux *interned atoms* offerts par Lisp ou Smalltalk, ou aux *interned strings* de Java. Contrairement à ces langages, Eiffel ne propose pas un tel mécanisme qui doit donc être émulé explicitement par le développeur Eiffel.

Dans SmallEiffel, nous implantons ce type d'aliasing à l'aide d'un *fournisseur d'alias* ou "*aliaseur*". En effet, bien que la gestion locale des objets "aliasés" soit très facile et intuitive dans une routine ou un petit algorithme, ce n'est plus le cas lorsque l'aliasing doit être géré à travers l'ensemble du système.

Dans un compilateur, par exemple, bien que les noms de symboles soient seulement créés dans l'analyseur syntaxique, ils sont manipulés par un objet spécial chargé de fournir un même et unique objet partagé pour chaque nom. Cet objet spécial est appelé dans SmallEiffel et dans ce mémoire le STRING_ALIASER. Il présente certaines similarités avec un pont (*bridge*) donnant accès à des îlots (*islands*) d'objets [Hogg, 1991]. Cependant, contrairement à un objet "pont", notre fournisseur d'alias n'a pas pour but de fournir un accès seulement temporaire aux objets qu'il contient (aliasing dynamique [Hogg *et al.*, 1992]) mais également un accès contrôlé et à long terme (aliasing statique).

Dans cette optique, il est de la plus haute importance que l'objet STRING_ALIASER soit unique dans l'ensemble du système. Le modèle de conception "singleton" [Gamma *et al.*, 1995] est ici parfaitement adéquat.

4.5.1 Mise en oeuvre du modèle de conception "singleton" en Eiffel

La raison d'être du modèle de conception "singleton" est, comme son nom l'indique, de garantir qu'une classe n'a qu'une seule instance dans un système et de fournir un accès global à cette instance.

L'implantation C++ proposée dans [Gamma *et al.*, 1995] garantit que la classe Singleton est instanciée une seule fois grâce à un test explicite lors de l'exécution.

En Java il est possible de garantir sans aucun coût à l'exécution qu'une seule instance d'une classe est jamais créée. Ceci se fait en rendant le constructeur privé, en créant une seule instance statique dans le bloc d'initialisation statique et en redéfinissant la méthode `clone` afin qu'elle retourne une référence vers cette instance statique.

En Eiffel, un singleton peut être implanté en se basant sur les mécanismes d'assertions et de routines *once* de la manière suivante:

```
class A_SINGLETON

-- D'autres primitives ici...

feature {NONE}  -- Partie Singleton

   singleton_memory: A_SINGLETON is
      once
         Result := Current;
      end;

invariant

   is_actually_singleton: Current = singleton_memory

end  -- A_SINGLETON
```

`singleton_memory` est une fonction Eiffel `once` [Meyer, 1992]. Une telle fonction est exécutée une seule fois, la première fois qu'elle est appelée. Les appels consécutifs retournent exactement le même résultat qui est automatiquement mémorisé pour toutes les instances de la même classe. Comme `singleton_memory` est sélectivement exportée à NONE, elle est privée et peut être appelée uniquement depuis l'objet `Current`, de type A_SINGLETON ou un de ses sous-types. Ainsi, la fonction `singleton_memory` mémorise (la référence vers) le premier objet qui l'appelle.

Les invariants de classes en Eiffel sont des assertions qui sont déclenchées à chaque fois qu'un client appelle une routine sur une instance de la classe englobant l'invariant. Ces invariants sont hérités par les descendants des classes où ils apparaissent. L'invariant `is_actually_singleton` garantit qu'un seul objet de type A_SINGLETON (ou de l'un de ses sous-types) est jamais créé dans le système. En effet, il vérifie que la référence sur l'objet `Current` est exactement celle stockée par `singleton_memory`. Si plus d'un objet de type A_SINGLETON était créé, cet invariant serait faux et déclencherait une exception sur le second objet, car `Current` ferait référence à ce second objet alors que `singleton_memory` référencerait le premier.

Il est important de noter que comme la solution Java mentionnée précédemment, et contrairement à l'implantation référence en C++, l'implantation Eiffel d'un singleton n'implique absolument aucun surcoût en mode optimisé. En effet, les invariants, comme les autres assertions, sont conçus pour être utilisés durant la phase de développement du code, mais ne sont plus activés dans le code final livré aux utilisateurs.

Bien entendu, à cause de la sémantique des routines `once` d'Eiffel, chaque classe qui doit être un singleton doit implanter la fonction `singleton_memory` et son invariant `is_actually_singleton`. Simplement hériter de A_SINGLETON n'est pas une solution viable car ainsi il serait impossible d'avoir plus d'un singleton dans le système, puisque comme nous l'avons indiqué précédemment, le singleton est commun au type dans lequel il apparaît ainsi que tous ses sous-types.

4.5.2 Implantation d'un fournisseur d'alias

Grâce à l'implantation pour les singletons proposée ci-dessus, la spécification de notre fournisseur d'alias devient très simple. En plus du code pour le modèle de conception "singleton", notre STRING_ALIASER offre simplement une fonction, `item`, chargée de fournir une référence vers la chaîne partagée correspondant au modèle spécifié. En voici l'interface:

```
class interface  STRING_ALIASER

   item (model: STRING): STRING
```

```
      require
         model /= Void

      ensure
         Result.is_equal(model)

invariant
   is_actually_singleton: Current = singleton_memory
```

La première fois qu'`item` est appelée avec un certain nom comme argument, elle le retourne et mémorise la chaîne `model` dans un DICTIONARY – une sorte de table de hachage – privé. Les appels suivants avec soit la même chaîne modèle soit une chaîne modèle différente ayant le même contenu retourneront tous (la référence vers) le tout premier objet chaîne mémorisé.

Encore une fois, le fait de vérifier que l'aliasing est bien fait n'induit aucun surcoût à l'exécution, puisque les préconditions (`require`) et les postconditions (`ensure`) ne sont pas générées en mode optimisé.

4.6 Promouvoir et contrôler l'usage du fournisseur d'alias

Nous avons décrit dans les sections précédentes comment l'unicité du fournisseur d'alias pouvait être garantie. Il est bien sûr également très important que dans tout le système tous les objets utilisent ce fournisseur d'alias et qu'aucun ne garde ses proches duplicata des chaînes "aliasées". De cette façon, une seule et unique version de la chaîne est créée et est gérée par le fournisseur d'alias.

Toutes les classes du système doivent donc respecter la "règle du jeu" et utiliser le fournisseur d'alias. Elles doivent également prendre part à la sécurité de l'ensemble du système en vérifiant que leurs propres fournisseurs eux aussi utilisent bien le fournisseur d'alias. Ceci peut facilement être implanté en Eiffel à l'aide des pré- et postconditions des routines.

L'exemple suivant montre LOCAL_NAME, une classe utilisée par SmallEiffel pour représenter un identificateur correspondant au nom d'une variable locale. La procédure de création `make` prend deux arguments: un pour enregistrer la position dans le code source, l'autre pour la chaîne qui contient de nom de l'identificateur.

```
class interface LOCAL_NAME

  --
  -- Un nom local dans une liste de déclarations
  --

creation

   make (sp: POSITION; n: STRING)
      require
         sp /= Void;
         n = string_aliaser.item(n)  -- (1)
      ensure
         start_position = sp;
         to_string = n               -- (2)
```

Dans l'extrait de code ci-dessus, l'assertion *(1)* est une précondition qui vérifie que l'appelant passe un argument qui est bien une chaîne *aliasée* gérée par le STRING_ALIASER. En effet si le contenu de `n` est le même que celui d'une chaîne déjà rencontrée et mémorisée par l'aliaseur de chaînes, `string_aliaser.item` retourne (une référence vers) l'objet mémorisé. Donc si la chaîne `n` a été fournie par le STRING_ALIASER, l'objet mémorisé correspondant au contenu de `n` est `n` elle-même.

Similairement, la postcondition *(2)* garantit aussi le comportement correct de la classe englobante, c'est à dire que la chaîne aliasée a bien été mémorisée par l'objet LOCAL_NAME courant, qui est donc un nouvel *alias* vers l'objet partagé.

Ceci s'applique de même à d'autres genres de noms dans le compilateur, comme CLASS_NAME ou ARGUMENT_NAME par exemple.

4.7 Aliasing pour les objets modifiables

Bien entendu, promouvoir l'utilisation du fournisseur d'alias n'est pas suffisant pour travailler avec l'aliasing en toute sécurité. Il est également nécessaire de contrôler strictement qui modifie les objets aliasés et où.

Les chaînes Eiffel sont des objets modifiables: leur contenu peut être changé, étendu, raccourci, etc. Dans le contexte du compilateur SmallEiffel, néanmoins, une chaîne aliasée ne devrait jamais être modifiée. S'il en était autrement, il en résulterait le chaos le plus total dans le processus de compilation.

Puisqu'une classe comme STRING, issue de la bibliothèque standard, contient de nombreuses routines de modification publiquement accessibles, la meilleure façon de contrôler leur utilisation serait d'encapsuler l'objet STRING dans, par exemple, une IMMUTABLE_STRING qui comporterait les accesseurs vers l'objet STRING mais aucun modificateur, ou au moins des modificateurs sélectivement exportés. Les sous-classes de STRING — qui n'existent pas actuellement dans le contexte du compilateur SmallEiffel — pourraient aussi être encapsulées de la même façon, par IMMUTABLE_STRING ou l'une de ses sous-classes.

Cette solution, bien que très satisfaisante du point de vue conceptuel, n'a pas été mise en oeuvre dans SmallEiffel car elle aurait impliqué un coût important tant en termes de mémoire (objets IMMUTABLE_STRING supplémentaires) qu'en vitesse d'exécution (niveau d'indirection supplémentaire).

Il est néanmoins possible de prendre des mesures pour s'assurer que les chaînes aliasées sont en effet non modifiées. Pour ce faire, des assertions peuvent être ajoutées au sein du STRING_ALIASER. Ceci requiert dans STRING_ALIASER un second DICTIONARY, dans lequel sont conservées des copies (pas des alias) des chaînes aliasées. Puis à chaque fois qu'`item` est appelée sur STRING_ALIASER, une assertion doit être déclenchée, vérifiant que le contenu des deux dictionnaires sont égaux. Comme les copies conservées dans le second dictionnaire ne sont pas accessibles de l'extérieur, une violation d'assertion à ce point indiquerait que l'un des objets aliasés a été modifié, signalant ainsi un comportement incorrect de l'un des alias. Ceci n'est cependant pas parfait, car ces vérifications sont réalisées de façon asynchrone, c'est à dire seulement lorsqu'`item` est appelée.

En revanche, aliaser des objets modifiables (et effectivement modifiés) pose moins de problèmes lorsqu'on a affaire à des objets qui ont étés *conçus pour être aliasés*.

De tels objets sont par exemple les instances du type BASE_CLASS, qui représentent les informations disponibles sur les classes après leur analyse syntaxique. Ces objets sont bien plus complexes que des chaînes de caractères car ils contiennent toutes les informations concernant la classe qu'ils représentent: nom, ancêtres, attributs, routines, etc. Si une classe C est présente dans le système, définie dans un fichier `c.e`, une seule instance de BASE_CLASS y correspondant est jamais créée, quel que soit le nombre d'occurrences de C dans le code client. Évidemment, aliaser ce type d'objets est indispensable. Faire autrement, si c'est possible, ne pourrait résulter qu'en la création d'un nombre considérable d'instances de BASE_CLASS et serait — au mieux — totalement inefficace.

Le mécanisme utilisé ici est très semblable à celui précédemment détaillé pour les chaînes de caractères. Le fournisseur d'alias pour les objets de type BASE_CLASS est le singleton EIFFEL_PARSER. Son rôle est de garantir que le fichier source de chaque classe est analysé une seule fois, sur demande, pour d'évidentes raisons de performances, et de fournir des alias sur les objets de type BASE_CLASS. En effet, EIFFEL_PARSER est le seul objet autorisé à créer des instances de BASE_CLASS. Grâce au mécanisme d'export sélectif d'Eiffel, cette propriété peut être *contrôlée statiquement*. L'extrait de code suivant montre la clause d'exportation de la seule méthode de création de BASE_CLASS:

```
class BASE_CLASS
...
creation {EIFFEL_PARSER} make
...
```

Ainsi, à chaque fois qu'un objet de type BASE_CLASS apparaît dans le code client, on sait statiquement qu'il s'agit d'un objet aliasé qui vient — directement ou non — de EIFFEL_PARSER. Les vérifications appropriées sont donc effectuées à la compilation et aucun surcoût n'apparaît à l'exécution, que ce soit en mode optimisé ou lors du développement. Ceci contraste avec les objets STRING que nous avons précédemment détaillés, qui peuvent aussi être créés hors du STRING_ALIASER et donc nécessitent du code supplémentaire dans le client pour contrôler leur origine lors de l'exécution.

Ce mécanisme d'export sélectif est aussi extrêmement utile pour obtenir un contrôle fin sur les modifications des objets de type BASE_CLASS. Contrairement à ce qui se produit pour les objets STRING, aucune "enveloppe" supplémentaire n'est nécessaire et aucun surcoût n'apparaît lors de l'exécution, ni au niveau mémoire, ni au niveau vitesse. Les routines modificatrices peuvent n'être exportées que vers le type des objets qui ont besoin d'y accéder, de la façon suivante:

```
class BASE_CLASS
...
feature {TYPE}
   smallest_ancestor(type, other: TYPE): TYPE is
...
feature {EIFFEL_PARSER}
   add_index_clause(index_clause: INDEX_CLAUSE) is
...
feature {RUN_CLASS, PARENT_LIST}
   collect_invariant(rc: RUN_CLASS) is
...
```

Il est intéressant de remarquer que ce mécanisme du langage Eiffel — l'export sélectif — concilie avec succès le passage élégant du niveau conceptuel à l'implantation et la recherche de performances maximales. Il n'y a à notre connaissance aucun équivalent direct dans d'autres langages à objets communs tels que C++ ou Java.

Avec l'aliasing de BASE_CLASS, il est également capital de faciliter la réalisation sur nécessité de certaines opérations et d'éviter de les faire plusieurs fois. La remontée de l'arbre d'héritage pour trouver l'origine d'une primitive héritée est une opération très fréquente. Comme elle implique des calculs coûteux qui se propagent aux ancêtres de la BASE_CLASS, il est beaucoup plus efficace de ne la déclencher qu'une seule fois et de modifier la liste des parents en conservant ses résultats intermédiaires pour les réutiliser par la suite. Grâce à l'aliasing, toutes les occurrences d'un certain nom de classe dans le code partagent le même objet BASE_CLASS et donc la même liste d'ancêtres. Ainsi, une fois qu'un contrôle sémantique est effectué sur l'une de ces occurrences, l'information correspondante devient immédiatement — et pratiquement sans aucun coût supplémentaire — disponible pour toutes les autres occurrences.

On peut remarquer que de cette façon, les alias ne reçoivent pas de notifications des modifications de l'objet aliasé. En effet, il n'y avait aucun besoin de notification dans notre application. Il est cependant extrêmement facile de rajouter un tel système de notification en cas de besoin, grâce à l'EIFFEL_PARSER qui gère toutes les demandes d'alias.

SmallEiffel comprend également d'autres fournisseurs d'alias basés sur les mêmes principes, par exemple pour les noeuds de l'arbre abstrait, les identificateurs de types, les routines spécialisées [Zendra *et al.*, 1997], etc.

4.8 Résultats expérimentaux

Comme nous l'avons montré ci-dessus, l'aliasing est très présent dans l'ensemble du compilateur SmallEiffel et concerne de nombreux types d'objets. Afin de quantifier précisément l'impact de l'aliasing, les performances de deux versions du compilateur devraient être comparées: une avec aliasing (partout où c'est possible) et une autre sans aucun aliasing. Cette dernière aurait impliqué une réécriture majeure et très longue du compilateur SmallEiffel. De plus, comme indiqué précédemment, il semble même impossible d'écrire un compilateur sans avoir au moins certaines formes d'aliasing, par exemple pour BASE_CLASS.

En conséquence, nous avons dû restreindre nos mesures de performances au seul aliasing des chaînes de caractères.

Pour quantifier l'impact de l'aliasing des chaînes durant le processus de compilation, nous avons donc développé une seconde version de SmallEiffel sans l'aliasing des chaînes. Il est intéressant de noter que ceci fut grandement facilité par le fait que l'ensemble du processus d'aliasing était concentré dans le STRING_ALIASER. Tous les tests d'égalité référentielle entre deux STRINGs ont été remplacés par la comparaison structurelle faite par `is_equal`.

Nous avons mesuré les performances des deux versions du compilateur lors du *bootstrap* (auto-compilation) de SmallEiffel version `-0.78`[38]. Comme indiqué dans les chapitres précédents, SmallEiffel constitue un banc d'essai significatif, car il s'agit d'une véritable application, complètement écrite en Eiffel et représentant plus de 70000 lignes de code pour 300 classes vivantes.

La taille mémoire semble pouvoir bénéficier de l'aliasing des chaînes. Le nombre total de chaînes (mots) rencontrées durant l'auto-compilation (bootstrap) est précisément de 79838. Tous ces mots contiennent 637014 octets, ce qui correspond à une longueur de mot moyenne de 8 octets.

Sachant qu'un objet de type STRING a une occupation mémoire de 3 mots plus le contenu de la chaîne lui-même (les caractères), on peut facilement calculer la taille mémoire totale théorique de tous les objets de type STRING non aliasés sur une machine 32 bits:

$$\begin{array}{rcrcl} \text{nb\_chaînes} & \times & (\text{taille\_objet\_string} & + & \text{longueur\_moyenne}) & = \\ 79838 & \times & (12 & + & 8) & = \\ & & 1596760 \text{ octets} & = & 1559 \text{ KO} & \end{array}$$

Ces 79838 occurrences de chaînes correspondent à seulement 4265 chaînes différentes, dont la taille mémoire peut être estimée à:

$$4265 \times (12 + 8) = 85300 \text{ octets} = 83 \text{ KO}$$

Donc lorsque l'aliasing est utilisé, la taille mémoire théorique des objets de type STRING dans le compilateur est réduite de 94,6 % par rapport à ce qu'elle est sans l'aliasing des chaînes.

Ces valeurs théoriques ne prennent pas en compte l'espace supplémentaire nécessaire pour la gestion mémoire, comme les entêtes des `malloc` C ou les données pour la gestion du ramasse-miettes [Colnet *et al.*, 1998a]. Nos résultats expérimentaux (figure 4.1) montrent qu'en fait la quantité de mémoire économisée grâce à l'aliasing est de 2456 KO.

Comme la taille mémoire maximale de l'ensemble du compilateur (lorsqu'il compile SmallEiffel) est d'environ 14708 KO, le bénéfice *venant du seul aliasing des chaînes de caractères* représente une diminution de plus de 14 % et est donc très significatif. Ceci est d'autant plus vrai si l'on se rappelle que SmallEiffel n'est pas seulement un projet de recherche, mais également un produit gratuit utilisé dans le monde entier par — entre autres — de nombreuses personnes sur de petits ordinateurs individuels pour lesquels une mémoire vive de 32 MO n'est pas si rare (ou même 16 MO, à l'époque où le projet SmallEiffel a débuté).

Cette réduction d'occupation mémoire est aussi un point positif en termes de vitesses d'exécution, car diminuer la taille mémoire du programme a également pour conséquence d'alléger le fardeau du ramasse-miettes (cf. chapitre 3) et de diminuer les échanges avec le disque (*swapping*).

En pratique, les effets de l'aliasing des chaînes sur le temps d'exécution sont bien plus que des conséquences indirectes. En effet, comme nous l'avons écrit dans la section 4.4, une des raisons principales à l'utilisation de l'aliasing des chaînes dans le compilateur SmallEiffel est de pouvoir faire des comparaisons de chaînes rapides dans le code client, en comparant simplement deux références avec l'opérateur `=` au lieu de comparer (les contenus de) deux objets avec la fonction `is_equal`. Cependant, l'inconvénient de l'aliasing des chaînes est que la gestion du STRING_ALIASER induit un certain coût supplémentaire. En effet, à chaque fois qu'une nouvelle chaîne est rencontrée, il est nécessaire de vérifier si elle est déjà dans le fournisseur d'alias et, dans le cas contraire, le nouvel objet doit être créé puis mémorisé par l'aliaseur.

38. Les résultats ont été obtenus sur un PC Pentium II 333 MHz avec 128 MO de RAM, sous Linux, noyau 2.0.36, et avec le compilateur C `egcs-1.1.1`.

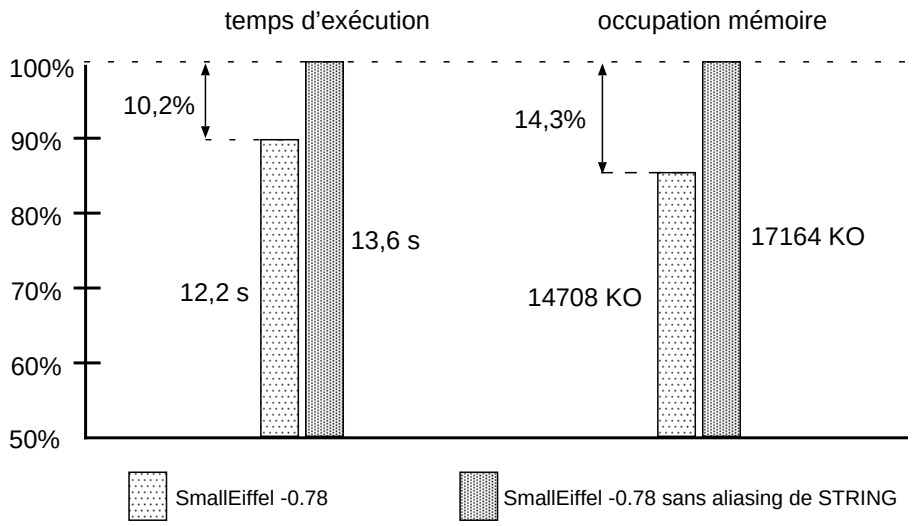


FIG. 4.1 – *Comparaison de performances: SmallEiffel avec et sans l'aliasing des chaînes de caractères.*

Il était donc raisonnable de se demander si le coût pour aliaser les chaînes n'allait pas annuler l'avantage en temps d'exécution procuré par les comparaisons de références.

Les expérimentations que nous avons menées montrent (figure 4.1) que *le seul aliasing des chaînes* conduit en fait à une amélioration de plus de 10 % du temps de compilation total, ce qui est tout à fait considérable.

Encore une fois, il est important de souligner que cet accroissement sensible des performances vient uniquement de l'aliasing des chaînes. Il n'est pas déraisonnable de penser que les autres types d'aliasing (sur les BASE_CLASSes, LOCAL_NAMEs, etc.) amènent eux aussi des gains en vitesse substantiels. L'aliasing sous diverses formes semble donc jouer un rôle important dans les très bonnes performances (voir [Collin *et al.*, 1997] et section 2.7) du compilateur SmallEiffel.

4.9 Conclusions et perspectives

Bien que l'aliasing présente des inconvénients importants et soit souvent considéré — au mieux — avec méfiance, il offre également un certain nombre d'avantages qui peuvent être utiles au développeurs.

Nous avons souligné ces avantages dans le domaine de la compilation et décrit comment nous avons mis en oeuvre l'aliasing dans SmallEiffel, *The GNU Eiffel Compiler*. Nous avons montré comment des choix de conception raisonnables nous permettaient de faire cela d'une façon plus sûre et comment certains idiomes du langage Eiffel rendaient un aliasing plus sûr facile à implanter, bien que l'aliasing ne soit pas directement supporté au niveau du langage. Dans ce cadre, la conception par contrat (*design by contract*), implanté à l'aide des assertions d'Eiffel, ainsi que les règles d'export sélectif, se sont révélés des outils puissants et utiles. Le modèle de conception "singleton", pour lequel nous avons proposé une implantation Eiffel novatrice, retenue par Jézéquel *et al.* pour leur ouvrage récent consacré aux modèles de conception en Eiffel [Jézéquel *et al.*, 1999], a également été crucial.

Nous avons réalisé un important travail expérimental. Les résultats en termes d'utilisation mémoire et de temps d'exécution ont ainsi pu être précisément caractérisés et se sont avérés très positifs. En utilisant l'aliasing sur les seuls objets de type STRING, nous avons pu réduire la taille mémoire de ces objets d'un facteur 20. Sur l'ensemble de la taille mémoire du programme, ceci revient à une diminution de plus de 14 %. Similairement, la vitesse d'exécution a été améliorée de plus de 10 %.

Les très bonnes performances atteintes par SmallEiffel en termes de vitesse et de mémoire semblent ainsi venir en bonne partie de l'aliasing. Nous sommes donc convaincus que l'aliasing est une *technique* tout à fait *utile* et fondamentale pour la programmation à objets. Considérer l'aliasing comme un phénomène nuisible serait à notre avis une erreur. Nous pensons également que les informations que nous fournissons

dans ce chapitre peuvent être très utiles non seulement aux développeurs de compilateurs mais aussi, de façon plus générale, à tout développement avec les langages à objets. Des travaux complémentaires sont néanmoins nécessaires pour pouvoir plus précisément évaluer l'impact de l'aliasing sur différents types de programmes, liés à la compilation ou non. Enfin, pour avoir dû développer des techniques d'usage de l'aliasing dans un langage ne le supportant pas, nous estimons qu'il est extrêmement souhaitable que les langages à objets intègrent eux-mêmes dès leur conception la notion d'aliasing.

Chapitre 5

Conclusion

5.1 Bilan et apports de la thèse

Nous résumons ici nos travaux et leurs résultats. De façon synthétique, on peut les considérer selon plusieurs angles d'approche: l'analyse statique globale, la compilation sur nécessité, la spécialisation de code généré, la liaison dynamique et l'aliasing. Une dernière sous-partie présente également plus en détails l'application de ces résultats dans le cadre du compilateur SmallEiffel, le compilateur GNU Eiffel.

5.1.1 Analyse statique globale et prédiction de type

Afin de pouvoir améliorer le processus de compilation et le code généré de façon portable et à haut niveau, nous nous sommes tout d'abord intéressés à l'analyse statique de l'ensemble du code source à compiler. Le but global de cette analyse est l'extraction du maximum d'informations du système compilé.

Une des premières étapes conceptuelles de cette analyse statique est l'exécution d'un algorithme de prédiction de type. Alors que beaucoup de travaux de recherche ont porté sur l'inférence de type avec d'intéressants résultats, les compilateurs couramment utilisés en production ne semblent guère s'appuyer sur les avantages de puissants algorithmes de prédiction de type. En effet, les implantations de tels algorithmes ne sont souvent pas assez rapides pour être utilisées dans le cadre du développement incrémental. La prédiction de type a néanmoins été incorporée dans nos travaux (section 2.3), afin de rassembler des informations permettant d'optimiser tant le processus de compilation que le code généré.

De nombreux algorithmes d'inférence de type existent dans la littérature, avec des résultats variables selon la précision des informations qu'ils donnent et le coût qu'ils représentent pour le processus de compilation. L'algorithme qui a fait l'objet de nos travaux se limite à une analyse du programme source purement statique, *context-sensitive* mais *flow insensitive*. Il n'est donc pas optimal en terme de complétude de l'information récupérée, même s'il en est proche. En revanche, il est très peu coûteux et ne ralentit quasiment pas la compilation. Ceci représente un bon compromis qui en permet l'incorporation dans les compilateurs utilisés en production logicielle.

Il faut également noter que, en analysant l'ensemble du système compilé, cet algorithme peut permettre de déceler un certains nombre d'erreurs plus sûrement et plus tôt, facilitant ainsi le travail du développeur d'applications. Ceci constitue la première implantation existante (même partielle) du *system-level validity checking* (vérification de cohérence globale) du langage Eiffel [Meyer, 1992].

5.1.2 Compilation sur nécessité

Les informations de typage et d'instanciations d'objets rassemblées par l'analyse statique globale permettent entre autres de connaître le code qui est vivant, c'est à dire potentiellement exécuté.

Il est selon nous important de prendre en compte ce type d'information *dès les premières étapes de la compilation*. En effet, le code mort, qui ne pourra jamais être exécuté, n'a pas besoin d'être analysé complètement, et peut ne pas donner lieu à génération de code. Il est ainsi possible d'alléger de manière non négligeable la tâche du compilateur, en diminuant la quantité de code à compiler réellement. Ceci

permet donc de dégager des ressources, notamment en temps d'exécution, qui peuvent affectées à des phases ultérieures de la compilation (optimisation du code généré) ou qui servent simplement à améliorer les temps de réponse du processus de compilation lui-même. Ceci contraste avec les compilateurs commerciaux, où il arrive de voir une étape supplémentaire (et donc coûteuse en temps) à la fin de la compilation, dite de "suppression de code mort".

Notre compilation sur nécessité permet donc en général un gain important en vitesse d'exécution de SmallEiffel, d'où un gain en temps de développement total pour l'utilisateur du compilateur.

5.1.3 Spécialisation du code généré

Un autre usage, que nous considérons comme absolument primordial, des informations collectées lors de l'analyse statique, est de servir à générer un code compilé qui soit le plus optimisé possible.

Une méthode qui permet à ces informations et cette optimisation de s'exprimer au mieux est la duplication suivie d'une spécialisation du code généré (voir section 2.4). Très brièvement, cette technique consiste, pour toute primitive présente dans une classe, à recopier un exemplaire de la primitive dans chacune des sous-classes de celle où elle est définie. Les versions recopiées dans les sous-classes ne sont pas identiques à la version originale, mais elles sont au contraire spécialisées, c'est à dire que leur code est adapté et optimisé par rapport au contexte propre à chacune des sous-classes. Ainsi, au lieu d'avoir une version commune complexe, traitant tous les cas possibles, avec de nombreux branchements conditionnels, la duplication-spécialisation génère de nombreux clones simples, adaptés chacun à un sous-ensemble de cas, ayant un code très linéaire. De même, mais en général dans une moindre mesure, les structures de données peuvent être spécialisées en fonction du contexte calculé par l'analyse statique. Ces versions dupliquées et spécialisées semblent être très prédictibles et exécutées très efficacement par les microprocesseurs actuels.

Ces optimisations de haut niveau, et donc portables, permettent ainsi d'améliorer sensiblement les performances du système compilé, notamment en temps d'exécution. La duplication-spécialisation a également un impact très positif, sur le code généré car, en simplifiant le code des primitives, elle facilite grandement un certain nombre d'optimisations classiques de plus bas niveau, telles que les évaluations partielles par calcul statique des expressions et les *inlinings*, comme nous l'avons montré dans nos travaux (voir en particulier les sections 2.4 et 2.7.2.3).

Ces optimisations portent non pas sur les points faciles à optimiser mais sur ceux qui sont liés aux concepts objets et ont donc un fort impact. Elles éliminent ainsi le surcoût potentiel dû à l'utilisation de ces concepts, ce qui permet réellement aux développeurs de se concentrer sur une programmation à objets de qualité.

Il est important de souligner encore une fois qu'étant tirées de la sémantique du système compilé, les optimisations de haut niveau mentionnées ci-dessus ne dépendent pas de l'architecture cible.

Notons enfin que nous avons appliqué cette technique d'optimisation par spécialisation non seulement au code du programme compilé, mais également au code du ramasse-miettes automatiquement généré par SmallEiffel (voir chapitre 3). Là encore, les résultats obtenus ont été particulièrement positifs.

5.1.4 Une technique novatrice d'implantation de la liaison dynamique

Nos travaux nous ont amenés à une réflexion approfondie sur la liaison dynamique, résultant en la conception d'une technique d'implantation nouvelle et performante, détaillée dans le chapitre 2 de ce mémoire, et plus particulièrement dans sa section 2.6.

Cette technique consiste, pour chaque site d'appel de méthode polymorphique, à générer une séquence de code qui remplace la table d'indirection classique par un arbre de branchement binaire spécialisé, dont les branches représentent les différents types dynamiques possibles pour le receveur à ce site d'appel. Grâce aux informations contextuelles fournies par l'analyse statique globale et l'algorithme de prédiction de type, les types dynamiques réellement possibles à chaque site sont connus et limités, ce qui permet d'avoir des arbres de branchement en général très réduits. Le cheminement dans ces arbres de branchement est déterminé en examinant l'identifiant de type dynamique codé dans l'objet receveur.

Ainsi, au lieu d'avoir un même code de liaison dynamique à chaque site d'appel polymorphique, comme dans les méthodes classiquement utilisées de nos jours, on a avec la méthode développée dans le cadre de

cette thèse des sites d'appels spécialisés, dont le code n'est capable de traiter que la liaison dynamique propre au site considéré, mais cela de façon très efficace.

Il faut noter que tout le processus de liaison dynamique se trouve ainsi dans le code de l'application et non pas dans ses données (mis à part le type dynamique associé au receveur), ce qui garantit de bonnes propriétés de localité, permettant d'obtenir des performances accrues.

Les mesures de performances effectuées et présentées en section 2.7 montrent que cette nouvelle méthode est très nettement supérieure aux méthodes classiques à base de VFT lorsque le nombre de types dynamiques possibles est faible, ce qui représente la majorité des cas. Dans le cas où de nombreux types dynamiques sont possibles (sites *mégamorphiques*, rares), notre méthode semble globalement équivalente aux méthodes à base de VFT. Cette nouvelle technique apparaît donc comme relativement simple à mettre en oeuvre, très efficace et supportant bien la "montée en charge".

Ici encore, notre implantation et notre optimisation de la liaison dynamique permet aux développeurs de s'appuyer sans réserve sur ce mécanisme, sans crainte d'aucun surcoût gênant, ce qui facilite une bonne programmation à objets.

5.1.5 Impact de l'aliasing

Enfin, nous avons dans le cadre de cette thèse mené certaines études sur l'impact de l'aliasing sur le processus de compilation.

L'aliasing existe et est connu dans les systèmes à objets (ou même plus classiques) depuis longtemps. Il est généralement considéré comme un phénomène néfaste, puisqu'il favorise diverses erreurs et rend le déverminage plus difficile. De nombreuses recherches ont donc eu pour objectif la suppression ou au mieux la prévention de l'aliasing.

Il existe pourtant peu de travaux quantifiant *l'impact* de l'aliasing sur un système en termes de performances. Comme l'aliasing permet de toute évidence un certain nombre d'optimisations dans le codage d'une application (comparaisons de références à la place de comparaisons de contenu, notamment), nous avons eu l'idée de considérer l'aliasing non pas comme un phénomène, néfaste qui plus est, mais bien comme une technique, potentiellement très utile. Notre démarche, détaillée dans le chapitre 4 de ce mémoire, a donc consisté à prendre le contre-pied des travaux habituels sur l'aliasing et à caractériser l'impact de l'aliasing dans un compilateur écrit en Eiffel, pour comprendre comment mieux l'utiliser.

L'aliasing s'est révélé délicat à singulariser, dans la mesure où il est entrelacé avec l'ensemble du code du compilateur. Il a cependant été possible d'extraire l'aliasing des chaînes de caractères du reste du compilateur. Cet aliasing apparaît comme ayant un impact positif et non négligeable en termes d'accélération de l'exécution du compilateur, ainsi qu'en ce qui concerne la mémoire consommée.

Il a également été possible de dégager certaines règles de codage (*patterns*) qui permettent une utilisation plus sûre de cette technique utile mais potentiellement dangereuse.

5.1.6 Application: SmallEiffel, The GNU Eiffel Compiler

Le résultat principal de nos recherches, en sus des publications qu'elles ont permises, a été le développement et la diffusion d'un compilateur pour le langage Eiffel nommé SmallEiffel.

En effet, les buts éminemment pratiques de nos travaux nous ont dès le début amenés à nous poser la question de la validation expérimentale de nos idées. Il nous est fort naturellement apparu qu'une des meilleures façons de le faire consistait à les implanter dans un compilateur. Afin de pouvoir implanter nos idées efficacement, en nous concentrant autant que possible sur les algorithmes eux-mêmes sans perdre de temps à les implanter, nous avons choisi d'utiliser un langage à objets de haut niveau, offrant des constructions puissantes et ayant un fort pouvoir expressif: le langage Eiffel. Comme aucun compilateur Eiffel de qualité suffisante n'était disponible, et encore moins le code source correspondant, nous avons très rapidement décidé de créer *ex nihilo* notre propre compilateur, SmallEiffel, dont les deux objectifs principaux étaient qu'il soit capable de *compiler vite*, tout *en générant du code efficace*.

Afin de mettre nos travaux à disposition du plus grand nombre, nous avons choisi de diffuser SmallEiffel comme logiciel libre[39]. Notre souci de vulgarisation s'est également manifesté par la publication de

39. `http://SmallEiffel.loria.fr`.

nombreux articles techniques [Colnet *et al.*, 1998b; Zendra *et al.*, 1998b; Zendra *et al.*, 1998a; Coucaud *et al.*, 1999a; Coucaud *et al.*, 1999b; Coucaud *et al.*, 1999c; Zendra *et al.*, 1999a; Zendra *et al.*, 1999b]. Cette diffusion de nos travaux auprès du public hors secteur académique nous a permi de disposer, en plus de la validation de la communauté scientifiques, d'une validation la plus large possible et également la plus "pratique" possible.

SmallEiffel nous a ainsi servi d'outil expérimental dans lequel nous avons pu implanter nos idées et valider nos hypothèses, ainsi que de vecteur de diffusion auprès de la communauté scientifique et du public. Mais il a aussi constitué un sujet d'observation, un banc d'essai significatif, extrêmement intéressant et plein d'enseignements, facilitant ainsi l'émergence de nouvelles idées. En effet, rappelons que SmallEiffel est complètement auto-compilé (*bootstrapped*) en Eiffel et représente dans sa version `-0.76Beta5` d'octobre 2000 environ 90000 lignes d'Eiffel pour environ 300 classes vivantes, ce à quoi il convient d'ajouter l'ensemble de la série de programmes utilisés pour valider le compilateur, soit environ 2500 classes représentant 210000 lignes d'Eiffel. A part quelques briques de base très localisées, l'ensemble des constituants du compilateur est entièrement écrit en Eiffel: les analyses lexicale, syntaxique et sémantique, la prédiction de type, la génération et l'optimisation de code, etc... La bibliothèque standard distribuée avec SmallEiffel, qui a également été l'objet d'efforts importants de notre part (voir [Colnet *et al.*, 1999; Zendra et Colnet, 1999a] et `http://SmallEiffel.loria.fr/libraries/classes.html`), est évidemment celle utilisée pour le développement du compilateur lui-même. On peut y vérifier que tout est écrit en Eiffel, même des classes très basiques, comme STRING ou ARRAY.

L'intérêt majeur que le codage "tout Eiffel" présente mérite d'être souligné. En effet, le fait d'avoir très peu de briques de base écrites en C, et donc l'immense majorité du code en Eiffel portable, permet bien évidemment de simplifier considérablement, voire d'éliminer, les problèmes de portabilité. La conception s'en trouve aussi facilitée, puisqu'il est ainsi possible de rester la plupart du temps au niveau d'un langage de haut niveau, Eiffel, dans lequel les algorithmes et concepts s'expriment nettement mieux qu'en C, langage bas niveau que nous considérons plutôt comme un excellent macro-assembleur relativement portable. Cela aide également fortement le déverminage, grâce notamment aux assertions pour la conception par contrat intégrées au langage Eiffel, qui diminuent sensiblement le besoin d'un débogueur extérieur. Cela représente aussi une masse de code Eiffel qui va elle-même servir de test et de validation expérimentale pour l'ensemble du compilateur et des algorithmes qui y sont implantés. Enfin, le simple fait de ne pas avoir à développer deux fois, une fois en Eiffel, une fois en C, les mêmes algorithmes et/ou structures de données représente un gain d'efforts de développement qu'il est important de ne pas sous-estimer.

Intégrant les techniques d'optimisation décrites dans ce mémoire, SmallEiffel produit un code d'excellente qualité, dont les performances sont en général supérieures voire très supérieures à celles des autres compilateurs Eiffel disponibles dans le commerce. Il se compare également très favorablement aux compilateurs C++, pour des programmes aux fonctionnalités équivalentes.

La très bonne qualité du code généré par SmallEiffel permet l'obtention d'exécutables de très petite taille, grâce notamment à la compilation sur nécessité et la spécialisation des primitives. De même, cette spécialisation est indéniablement à l'origine de la rapidité des exécutables produits. SmallEiffel étant écrit en Eiffel, ses exécutables bénéficient eux aussi des optimisations appliquées aux programmes compilés. Ses propres performances, en terme de mémoire et surtout de temps de compilation sont elles aussi excellentes, comparées aux autres compilateurs Eiffel ou à des compilateurs C++.

Dans un souci d'uniformité et d'efficacité, les techniques développées précédemment et appliquées au code généré l'ont été également au niveau du ramasse-miettes. Ainsi, de façon relativement originale, SmallEiffel n'intègre pas un même ramasse-miettes à tous les programmes qu'il compile, mais produit automatiquement un système de gestion mémoire adapté à chaque application générée, basé sur un ramasse-miettes à marquage-balayage. Ce dernier est donc différent pour chaque application, étant spécialisé en tenant compte des informations fournies par l'analyse statique. Seules les routines et les structures de données réellement nécessaires à la gestion mémoire des objets du programme compilé sont produites.

Ce ramasse-miettes, portable, a lui aussi fait preuve d'excellentes performances, notamment quand on le compare à un ramasse-miettes beaucoup plus mûr, ayant fait l'objet de beaucoup plus d'optimisations et de réglages que lui et qui a pu être intégré à SmallEiffel à des fins de comparaison.

Enfin, il faut noter que, étant écrit dans un langage portable (Eiffel), générant du code portable (C ANSI ou *bytecode* Java) et n'appliquant que des techniques d'optimisation de haut niveau, SmallEiffel est très facile à porter sur de nouvelles architectures, et ne nécessite dans la plupart des cas qu'une simple

recompilation.

Ces qualités montrent donc de façon probante l'efficacité et l'applicabilité des techniques sur lesquelles nous avons focalisé ce travail de thèse. Elles montrent également que les objectifs initiaux (processus de compilation plus efficace, code généré plus efficace, portabilité) ont été atteints de façon fort satisfaisante.

5.2 Perspectives

Ces bons résultats ne doivent bien sûr pas être considérés comme ultimes et ne pouvant être améliorés. Au contraire, ils nous incitent à poursuivre les recherches présentées dans ce mémoire.

A court ou moyen terme, nous souhaiterions donc étudier les extensions possibles aux techniques présentées ci-dessus, notamment en améliorant l'analyse statique et en la couplant avec des techniques d'analyse dynamique. Nous pensons également qu'il peut être intéressant de généraliser nos travaux à des systèmes non figés, en étudiant tout particulièrement l'impact d'une modification ponctuelle sur le système. Des extensions à d'autres langages modernes et couramment utilisés comme Java seraient bien entendu bienvenues. Enfin, afin de maîtriser plus effectivement les techniques d'optimisations, il nous semble judicieux d'étudier et de bien comprendre l'impact des architectures matérielles.

Tout ceci doit se faire en gardant à l'esprit la vision à long terme qui sous-tend le travail présenté dans ce mémoire, à savoir la production de compilateurs plus "intelligents" et le passage, en ce qui concerne les optimisations, du stade actuel un peu "artisanal" à un stade plus scientifique et industriel, où les optimisations sont mieux comprises, maîtrisées et appliquées, avec pour but final l'obtention de programmes toujours plus rapides.

Les sections ci-dessous évoquent plus en détails ces perspectives.

5.2.1 Amélioration de l'analyse statique

Bien qu'il donne d'excellents résultats actuellement, l'algorithme d'analyse globale et de prédiction de type peut encore être amélioré. Des informations supplémentaires peuvent être rassemblées en augmentant la précision de l'analyse statique.

Une voie qui se présente naturellement à l'esprit consiste à prendre en compte le flot d'exécution, de façon à quasiment éliminer les parties du code qui sont considérées vivantes par notre algorithme actuel alors qu'il s'agit en fait de code mort. Ceci permettrait également d'affiner encore la précision de la prédiction de type, d'où une amélioration potentielle des performances de la liaison dynamique. Cependant, il est important d'évaluer le coût à la compilation d'une telle analyse *flow sensitive*, qui pourrait s'avérer incompatible avec la compilation rapide requise pour des environnements de développement incrémental.

Il est également possible d'augmenter encore la spécialisation du code généré en l'adaptant non seulement au type du receveur, mais aussi aux types dynamiques des paramètres des méthodes. C'est ce que fait l'algorithme du produit cartésien (*Cartesian Product Algorithm — CPA*, [Agesen, 1995]) proposé par Agesen. Ceci impose cependant la génération de très nombreuses versions d'une même méthode, qui devraient être plus efficaces car plus spécifiques, mais qui risquent de provoquer un accroissement important de la taille du code généré et du temps de compilation. Il semble donc intéressant d'étudier plus en détail les différents degrés de spécialisation possibles et l'adéquation des compromis qu'ils représentent.

5.2.2 Couplage avec des techniques d'analyse dynamique

Les travaux présentés dans ce mémoire se sont pour l'instant focalisés sur les bénéfices de l'analyse et l'optimisation *statiques*, faites lors de la compilation. Un domaine de recherche connexe et très actif existe, qui porte sur les techniques d'analyse et d'optimisations *dynamiques*, c'est à dire effectuées pendant l'exécution du programme compilé. Nous avons d'ailleurs mentionné quelques-unes de ces techniques dans le cadre de la liaison dynamique (section 2.5.2).

Il nous semblerait judicieux d'aborder aussi ce domaine très actuel, qui génère une littérature importante. Nous souhaitons étudier ces techniques dynamiques et leurs apports au processus de compilation, notamment au niveau du couplage possible entre l'analyse statique et l'analyse dynamique. Nous pensons en effet qu'il est possible de bénéficier de synergies importantes entre ces deux mondes et que les

informations statistiques, de profils, rassemblées au cours de l'analyse dynamique devraient permettre d'améliorer encore les performances des programmes compilés à l'aide de notre système actuel.

Ces techniques dynamiques appliquées à la liaison dynamique (basées généralement sur divers types de caches à l'exécution), par opposition aux techniques statiques (basées sur des informations et structures définies à la compilation) comme celles sur lesquelles nous avons travaillé jusqu'à présent, nous apparaissent comme particulièrement intéressantes. Elles présentent à notre avis un fort potentiel, notamment en complément de techniques statiquement calculées. Comme nous l'avons déjà souligné, la liaison dynamique et son implantation optimale sont en effet pour nous des clés importantes vers des programmes à objets performants.

5.2.3 Généralisation à des systèmes non figés

Pour l'instant, nos travaux ont porté sur des systèmes complets et figés (sans chargement ni création dynamique de classe).

Dans l'optique de rendre les techniques que nous avons étudiées encore plus largement utilisables, il nous semble souhaitable d'étendre leur capacités aux cas de systèmes incomplets. Des extensions relativement simples sont possibles pour utiliser ces techniques dans de tels contextes, comme la compilation séparée d'applications ou de bibliothèques, mais elles semblent de nature à dégrader fortement les performances du code compilé dans ces conditions. Il nous parait donc indispensable d'étudier des solutions permettant d'éviter cette dégradation. Ceci pourrait impliquer de fortes modifications des algorithmes actuellement utilisés dans notre système SmallEiffel.

Dans le même ordre d'idée, nous souhaitons vivement étendre nos recherches de façon à y inclure la compilation de langages plus dynamiques, tout particulièrement Java. Ce type de langages offre des possibilités différentes de celles des langages de classes plus classiques, statiques, mais impose des contraintes plus importantes au compilateur. Ceci devrait nous amener très rapidement à nous intéresser à la compilation dynamique (en ligne) proprement dite, qui, elle, permet de s'adapter à un environnement et un source changeant. L'intégration dans ce cadre de compilation dynamique des techniques présentées dans ce mémoire représente donc une direction de travail importante pour nous.

5.2.4 Impact d'une modification

Fort naturellement, la compilation dynamique conduit à s'intéresser au problème de l'impact d'une modification du source disponible (par chargement dynamique de classe par exemple) sur le reste du système, afin de mettre à jour le code compilé.

Plus généralement, ce problème présente également un intérêt considérable même dans le cas de la compilation statique de systèmes figés. En effet, l'incrémentalité des recompilations est l'une des voies possibles pour offrir au développeur un environnement de compilation lui permettant des cycles d'édition-compilation-corrections les plus rapides possibles. Une meilleure compréhension de l'impact d'une modification ponctuelle du source par le développeur est ainsi utile non seulement au compilateur mais également au développeur car elle peut permettre à l'interface utilisateur de l'environnement de mieux mettre en valeur l'ensemble du code affecté, ce qui nous semble constituer une aide au développement intéressante.

5.2.5 Généralisation et extension à d'autres langages

Comme nous l'avons indiqué à plusieurs reprises, il nous parait indispensable de ne pas limiter nos recherches à un seul langage. En plus d'Eiffel, qui nous a servi pour implanter nos idées tout au long de cette thèse, il serait donc intéressant de pouvoir disposer d'autres implantations. Nous estimons qu'il devrait être assez facile d'appliquer ce travail à d'autres langages de classes statiques sans chargement dynamique de classes, comme C++. Il est par contre probable que le passage nécessaire vers des langages couramment utilisés et plus dynamiques, comme Java, demande plus de travail. Cependant, cela représenterait également une opportunité pour développer de nouvelles techniques adaptées à ces nouveaux contextes.

Il faut remarquer que les résultats que nous avons obtenus ont porté sur un langage, Eiffel, où la liaison dynamique s'effectue sur le receveur. Il nous semble également intéressant et relativement aisé de s'attacher à les généraliser aux cas de langages offrant des multiméthodes et du polymorphisme paramétrique.

5.2.6 Caractérisation et prise en compte de l'impact des architectures

Un dernier point qui nous parait pouvoir faire l'objet de travaux futurs très enrichissants est la prise en compte de l'impact de l'architecture cible tant sur le processus de compilation que sur le système compilé.

Cet impact est fort probablement dépendant de divers facteurs, comme les unités d'exécution du processeurs, les caches et la bande passante des transmissions d'informations dans la machine. Comme nous l'avons indiqué dès l'introduction de ce mémoire, nous ne souhaitons pas travailler à très bas niveau (matériel) et donc de façon non portable.

Il s'agit donc pour nous de chercher des moyens *portables* de caractériser les effets de l'architecture sur les optimisations sémantiques et de voir dans quelle mesure ces effets contribuent ou nuisent à l'efficacité de ces optimisations, tout particulièrement au niveau de la liaison dynamique. Cette caractérisation devrait se faire (au moins en partie) grâce au développement de modèles et de simulateurs de ces artefacts architecturaux.

Bibliographie

[Agesen *et al.*, 1993] Ole Agesen, Jens Palsberg et Michael I. Schwartzbach. Type Inference of SELF. Analysis of Objects with Dynamic and Multiple Inheritance. Dans *7th European Conference on Object-Oriented Programming (ECOOP'93)*, volume 707 de *Lecture Notes in Computer Sciences*, pages 247–267. Springer-Verlag, 1993.

[Agesen et Detlefs, 1997] Ole Agesen et David Detlefs. Finding References in Java Stacks. Dans *OOPSLA'97 Workshop on Garbage Collection and Memory Management*, 1997.

[Agesen et Hölzle, 1995] Ole Agesen et Urs Hölzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. Dans *10th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, volume 30 de *SIGPLAN Notices*, pages 91–107. ACM Press, Octobre 1995. Austin, TX, USA.

[Agesen, 1995] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. Dans *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 de *Lecture Notes in Computer Sciences*, pages 2–26. Springer-Verlag, 1995.

[Agesen, 1996] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Department of Computer Science of Stanford University, Published by Sun Microsystem Laboratories (SMLI TR-96-52), 1996.

[Aho *et al.*, 1988] Alfred V. Aho, Brian W. Kernighan et Peter J. Weinberg. *The AWK programming Language.* Addison-Wesley, Reading, Massachusetts, 1988.

[Aho et Ullman, 1977] Alfred V. Aho et Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Massachusetts, 1977.

[Aigner et Hölzle, 1996] Gerald Aigner et Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. Dans *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 de *Lecture Notes in Computer Sciences*, pages 142–166. Springer-Verlag, 1996.

[Almeida, 1997] Paulo Sérgio Almeida. Ballon Types: Controlling Sharing of State in Data Types. Dans *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 de *Lecture Notes in Computer Sciences*, pages 32–59, Juin 1997.

[André et Royer, 1992] Pascal André et Jean-Claude Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. Dans *7th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*, volume 27 de *SIGPLAN Notices*, pages 110–126. ACM Press, Octobre 1992. Vancouver, BC, Canada.

[Appel, 1992] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992. Chapitre 16, pages 205–214.

[Appleby *et al.*, 1988] Karen Appleby, Mats Carlsson, Seif Haridi et Dan Sahlin. Garbage Collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.

[Attardi *et al.*, 1995] Giuseppe Attardi, Tito Flagella et Pietro Iglio. Performance-Tuning in a Customizable Collector. Dans *International Workshop on Memory Management (IWMM'95)*, volume 986 de *Lecture Notes in Computer Sciences*, pages 179–198, 1995.

[Attardi et Flagella, 1994] Giuseppe Attardi et Tito Flagella. Customising Object Allocation. Dans *8th European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821, pages 320–343, 1994.

[Bacon et Sweeney, 1996] David F. Bacon et Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. Dans *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 324–341. ACM Press, 1996.

[Barett et Zorn, 1993] David A. Barett et Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. Dans *1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, volume 28 de *SIGPLAN Notices*, pages 187–196. ACM Press, 1993.

[Bartlett, 1988] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Rapport technique, 88/2. DEC Western Research Laboratory., 1988.

[Bartlett, 1990] Joel F. Bartlett. A generationnal, compacting collector for C++. Dans *ECOOP/OOPSLA'90 Workshop on Garbage Collection in Object-Oriented Systems*, 1990.

[Birrell *et al.*, 1994] Andrew Birrell, Greg Nelson, Susan Owicki et Edward Wobber. Network Objects. Rapport technique, RR-115, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, USA, Février 1994. Révisé en Décembre 1995.

[Boehm et Shao, 1993] Hans-Juergen Boehm et Zhong Shao. Inferring type maps during garbage collection. Dans *ECOOP/OOPSLA'93 Workshop on Garbage Collection in Object-Oriented Systems*, 1993.

[Boehm et Weiser, 1988] Hans-Juergen Boehm et Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[Boehm, 1993] Hans-Jurgen Boehm. Space-efficient conservative garbage collection. Dans *1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, volume 28 de *SIGPLAN Notices*, pages 197–206. ACM Press, 1993.

[Bokowski et Vitek, 1999] Boris Bokowski et Jan Vitek. Confined Types. Dans *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 33 de *SIGPLAN Notices*, pages 82–96. ACM Press, Octobre 1999.

[Branquart et Lewi, 1971] P. Branquart et J. Lewi. A Scheme of Storage Allocation and Garbage Collection for Algol-68. Dans *Algol-68 Implementation*, rédacteur J. E. L. Peck, pages 198–238. North-Holland, Amsterdam, 1971.

[Calder et Grunwald, 1994] Brad Calder et Dirk Grunwald. Reducing Indirect Function Call Overhead In C++ Programs. Dans *21st Annual ACM Symposium on the Principles of Programming Languages (POPL'94)*, pages 397–408. ACM Press, Janvier 1994.

[Cann *et al.*, 1992] D. C. Cann, J. T. Feo, A. D. W. Bohoem et Rod R. Oldehoeft. SISAL Reference Manual, Language Version 2.0, 1992.

[Cartwright et Fagan, 1991] Robert Cartwright et Mike Fagan. Soft Typing. Dans *1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 278–292. ACM Press, 1991.

[Chailloux, 1992] Emmanuel Chailloux. Conservative Garbage Collector with Ambiguous Roots, for Static Type Checking Languages. Dans *International Workshop on Memory Management (IWMM'92)*, volume 637 de *Lecture Notes in Computer Sciences*, pages 218–229, 1992.

[Chambers et Chen, 1999] Craig Chambers et Weimin Chen. Efficient Multiple and Predicate Dispatching. Dans *Proceedings of 14th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 33 de *SIGPLAN Notices*, pages 82–96. ACM Press, Octobre 1999.

[Chambers et Ungar, 1989] Craig Chambers et David M. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Language. Dans *1989 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'89)*, volume 24 de *SIGPLAN Notices*, pages 146–160. ACM Press, Juillet 1989.

[Chambers et Ungar, 1990] Craig Chambers et David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. Dans *1990 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, volume 25 de *SIGPLAN Notices*, pages 150–164. ACM Press, 1990.

[Chambers, 1992] Craig Chambers. Object-Oriented Multi-Methods in Cecil. Dans *6th European Confe-*

rence on Object-Oriented Programming (ECOOP'92), volume 615 de *Lecture Notes in Computer Sciences*, pages 33–56. Springer-Verlag, Juin 1992.

[Chambers, 1993] Craig Chambers. The Cecil Language: Specification and Rationale. Rapport technique, UW-CSE-93-03-05, Department of Computer Science and Engineering, University of Washington, 1993. Révisé en Mars 1997.

[Clarke *et al.*, 1998] David G. Clarke, John M. Potter et James Noble. Ownership Types for Flexible Alias Protection. Dans *Proceedings of 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, volume 33 de *SIGPLAN Notices*, pages 48–64. ACM Press, Octobre 1998.

[Collin *et al.*, 1996] Suzanne Collin, Dominique Colnet et Olivier Zendra. Compiling Late Binding in Object-Oriented Languages with a Copy-and-Adapt Technique. Application to the Eiffel Language. Rapport Technique 96-R-068, LORIA - UMR 7503, 1996.

[Collin *et al.*, 1997] Suzanne Collin, Dominique Colnet et Olivier Zendra. Type Inference for Late Binding. The SmallEiffel Compiler. Dans *Joint Modular Languages Conference (JMLC'97)*, volume 1204 de *Lecture Notes in Computer Sciences*, pages 67–81. Springer-Verlag, 1997.

[Collins, 1960] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, Décembre 1960.

[Colnet *et al.*, 1998a] Dominique Colnet, Philippe Coucaud et Olivier Zendra. Compiler Support to Customize the Mark and Sweep Algorithm. Dans *ACM SIGPLAN International Symposium on Memory Management (ISMM'98)*, pages 154–165, Octobre 1998. Intégré à la 13th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98).

[Colnet *et al.*, 1998b] Dominique Colnet, Philippe Coucaud et Olivier Zendra. Eiffel: la puissance de la recherche au service de vos programmes. *Programmez!*, (2):82–83, Juillet 1998.

[Colnet *et al.*, 1998c] Dominique Colnet, Olivier Zendra et Philippe Coucaud. Using Type Inference to Customize the Garbage Collector in an Object-Oriented Language. The SmallEiffel Compiler. Rapport Technique 98-R-196, LORIA - UMR 7503, 1998.

[Colnet *et al.*, 1999] Dominique Colnet, Emmanuel Stapf et Olivier Zendra. NICE Eiffel Library Kernel Standard 2000 proposal (ELKS'2000). Rapport Technique 99-R-359, LORIA - UMR 7503, 1999.

[Colnet et Zendra, 1999] Dominique Colnet et Olivier Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. Dans *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, pages 341–350. IEEE Computer Society, Juin 1999.

[Colnet et Zendra, 2000] Dominique Colnet et Olivier Zendra. Targeting the Java Virtual Machine with Genericity, Multiple Inheritance, Assertions and Expanded Types. Rapport Technique A00-R-137, LORIA - UMR 7503, Septembre 2000.

[Corney et Gough, 1994] Diane Corney et John Gough. Type Test Elimination using Typeflow Analysis. Dans *PLSA 1994 International Conference*, volume 782 de *Lecture Notes in Computer Sciences*, pages 137–150. Springer-Verlag, 1994. Zurich, Suisse.

[Coucaud *et al.*, 1999a] Philippe Coucaud, Olivier Zendra et Dominique Colnet. Gestion mémoire: manuelle ou automatique? *Programmez!*, (7):54–57, Février 1999.

[Coucaud *et al.*, 1999b] Philippe Coucaud, Olivier Zendra et Dominique Colnet. La programmation à objets. Application au langage Eiffel. Partie 1. *Linux Magazine*, (8):63–66, Juillet 1999.

[Coucaud *et al.*, 1999c] Philippe Coucaud, Olivier Zendra et Dominique Colnet. La programmation à objets. Application au langage Eiffel. Partie 2. *Linux Magazine*, (9):52–55, Septembre 1999.

[Cox, 1983] B. J. Cox. Object-Oriented Programming in C. *UNIX Review*, pages 67–70, Octobre/Novembre 1983.

[Dean *et al.*, 1995] Jeffrey Dean, David Grove et Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. Dans *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 de *Lecture Notes in Computer Sciences*, pages 77–101. Springer-Verlag, 1995.

[Dean *et al.*, 1996] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov et Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. Dans *11th Annual ACM Conference*

on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96), pages 83–100. ACM Press, 1996.

[Debray *et al.*, 1998] Saumya Debray, Robert Muth et Matthew Weippert. Alias analysis of Executable Code. Dans *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 12–24. ACM Press, 1998.

[Detlefs *et al.*, 1994] David Detlefs, Al Dosser et Benjamin Zorn. Memory Allocation Costs in large C++ Programs. *Software: Practice and Experience*, 24(6), 1994.

[DeTreville, 1990] John DeTreville. Experience with Concurrent Garbage Collectors for Modula-2+. Rapport technique, TR-64. DEC Systems Research Center, Palo Alto, CA, USA, Août 1990.

[Deutsch et Bobrow, 1976] Peter L. Deutsch et Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, Septembre 1976.

[Deutsch et Schiffman, 1984] Peter L. Deutsch et Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. Dans *11th Annual ACM Symposium on the Principles of Programming Languages (POPL'84)*. ACM Press, 1984.

[Dhamdhere, 1991] D.M. Dhamdhere. Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):291–294, 1991.

[Diwan *et al.*, 1992] Amer Diwan, J. Eliot B. Moss et Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. Dans *1992 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, volume 27 de *SIGPLAN Notices*, pages 273–282. ACM Press, 1992.

[Diwan *et al.*, 1996] Amer Diwan, J. Eliot B. Moss et Kathryn S. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. Dans *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 292–305. ACM Press, 1996.

[Diwan *et al.*, 1999] Amer Diwan, Kathryn S. McMkinley et J. Eliot B. Moss. Type-Based Alias Analysis. Dans *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, volume 33, pages 106–117. ACM Press, Juin 1999.

[Dixon *et al.*, 1989] R. Dixon, T. McKee, Paul Schweitzer et M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. Dans *4th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 de *SIGPLAN Notices*, pages 211–214. ACM Press, Octobre 1989. New Orleans, LA, USA.

[Driesen *et al.*, 1995] Karel Driesen, Urs Hölzle et Jan Vitek. Message Dispatch on Pipelined Processors. Dans *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 de *Lecture Notes in Computer Sciences*, pages 253–282. Springer-Verlag, 1995.

[Driesen et Hölzle, 1995] Karel Driesen et Urs Hölzle. Minimizing Row Displacement Dispatch Tables. Dans *10th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, volume 30 de *SIGPLAN Notices*, pages 141–155. ACM Press, Octobre 1995. Austin, TX, USA.

[Driesen et Hölzle, 1996] Karel Driesen et Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. Dans *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 306–323. ACM Press, 1996.

[Driesen, 1993] Karel Driesen. Method Lookup Strategies in Dynamically-Typed Object-Oriented Languages. Master's thesis, Vrije Universiteit Brussel, 1993.

[Driesen, 1999] Karel Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, Computer Science Department, University of California, Santa Barbara, USA, Juin 1999. Technical Report TRCS99-24.

[Edelson, 1992] Daniel R. Edelson. Precompiling C++ for Garbage Collection. Dans *International Workshop on Memory Management (IWMM'92)*, volume 637 de *Lecture Notes in Computer Sciences*, pages 299–314. Springer-Verlag, 1992.

[Edelson, 1993] Daniel R. Edelson. Type Specific Storage Management. Rapport technique, UCSC-CRL-93-26. Baskin Center for Computer Engineering & Informations Sciences, University of California, Santa Cruz, 1993.

[Eifrig *et al.*, 1995] Jonathan Eifrig, Scott Smith et Valery Trifonov. Sound Polymorphic Type Inference for Objects. Dans *10th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, volume 30 de *SIGPLAN Notices*, pages 169–184. ACM Press, Octobre 1995. Austin, TX, USA.

[Ellis et Stroustrup, 1990] Margaret A. Ellis et Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[Foderaro et Fateman, 1981] John K. Foderaro et Richard J. Fateman. Characterization of VAX Macsyma. Dans *1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 14–19. ACM Press, 1981. Berkeley, CA, USA.

[Gabriel, 1985] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press Series in Computer Science. MIT Press, Cambridge, MA, USA, 1985.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gelernter *et al.*, 1960] H. Gelernter, J. R. Hansen et C. L. Gerberich. A Fortran-Compiled List Processing Language. *Journal of the ACM*, 7(2):87–101, Avril 1960.

[Ghiya et Hendren, 1998] Rakesh Ghiya et Laurie J. Hendren. Putting Pointer Analysis to Work. Dans *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 121–133. ACM Press, 1998.

[Goldberg et Robson, 1983] Adele Goldberg et David Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Goldberg, 1991] Benjamin Goldberg. Tag-Free Garbage Collection for Strongly Typed Programming Languages. Dans *1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, volume 26 de *SIGPLAN Notices*, pages 165–176. ACM Press, 1991.

[Gosling *et al.*, 1996] James Gosling, Bill Joy et Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Graver et Johnson, 1990] J. Graver et Ralph E. Johnson. A Type System for Smalltalk. Dans *17st Annual ACM Symposium on the Principles of Programming Languages (POPL'90)*, pages 139–150. ACM Press, 1990.

[Grunwald et Zorn, 1993] Dirk Grunwald et Benjamin Zorn. CustoMalloc: Efficient Synthesized Memory Allocators. *Software: Practice and Experience*, 23(8):851–869, August 1993.

[Hayes, 1991] Barry Hayes. Using key object opportunism to collect old objects. Dans *6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, volume 26, pages 33–46. ACM Press, Novembre 1991.

[Hayes, 1992] Barry Hayes. Finalization in the Collector Interface. Dans *International Workshop on Memory Management (IWMM'92)*, volume 637 de *Lecture Notes in Computer Sciences*, pages 277–298, Septembre 1992.

[Hennessy et Patterson, 1996] John L. Hennessy et David A. Patterson. *Computer Architecture: A Quantitative Approach. Second edition.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[Hogg *et al.*, 1992] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux et Richard Holt. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3(2), Avril 1992.

[Hogg, 1991] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. Dans *6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, volume 26, pages 271–285. ACM Press, Novembre 1991.

[Hölzle *et al.*, 1991] Urs Hölzle, Craig Chambers et David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. Dans *5th European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 de *Lecture Notes in Computer Sciences*, pages 21–38. Springer-Verlag, 1991.

[Hölzle et Ungar, 1994] Urs Hölzle et David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. Dans *1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, volume 29 de *SIGPLAN Notices*, pages 326–335. ACM Press, 1994.

[Hölzle et Ungar, 1995] Urs Hölzle et David Ungar. Do Object-Oriented Languages Need Special Hardware Support? Dans *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 de *Lecture Notes in Computer Sciences*, pages 283–302. Springer-Verlag, 1995.

[ISE, 1999] Garbage Collection for ISE Eiffel. Rapport technique, ISE TR-EI-56/GC, version 3.3.9, Interactive Software Engineering, Inc., 1999.

[Jones et Lins, 1996] Richard Jones et Rafael Lins. *Garbage Collection*. Wiley, 1996.

[Jézéquel *et al.*, 1999] Jean-Marc Jézéquel, Michel Train et Christine Mingins. *Design Patterns with Contracts*. Addison-Wesley, Reading, Massachusetts, 1999.

[Kernighan et Ritchie, 1978] Brian W. Kernighan et Denis M. Ritchie. *The C Programming Language*. Prentice Hall Inc., 1978.

[Knoop *et al.*, 1994] Jens Knoop, Oliver Ruthing et Bernhard Steffen. Partial Dead Code Elimination. Dans *1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pages 147–. ACM Press, 1994.

[Krasner, 1983] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, Massachusetts, 1983.

[Kurihara *et al.*, 1990] Satoshi Kurihara, Mikio Inari, Norihisa Doi, Kazuki Yasumatsu et Takemi Yamazaki. SPiCE Collector: The run-time garbage collector for Smalltalk-80 programs translated into C. Dans *ECOOP/OOPSLA'90 Workshop on Garbage Collection in Object-Oriented Systems*, 1990.

[Lippman, 1993] Stanley B. Lippman. *C++ primer — 2nd edition*. Addison-Wesley, 1993.

[Masini *et al.*, 1989] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard et Karl Tombre. *Les langages à objets*. InterEditions, Paris, 1989.

[McBeth, 1963] J. Harold McBeth. On the Reference Counter Method. *Communications of the ACM*, 6(9):575–, Septembre 1963.

[McCarthy, 1960] John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Communications of the ACM*, 3:184–195, Septembre 1960.

[Meyer, 1988] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall Inc., 1988.

[Meyer, 1992] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall Inc., 1992.

[Milner, 1978] R. Milner. A Theory of Type Polymorphism in Programming. Dans *Journal of Computer and System Sciences*, pages 348–375, 1978.

[Minsky, 1963] Marvin L. Minsky. A Lisp Garbage Collector Algorithm Using Serial Secondary Storage. Rapport technique, TR Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, USA, Décembre 1963.

[Minsky, 1996] Naftaly Minsky. Towards Alias-Free Pointers. Dans *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 de *Lecture Notes in Computer Sciences*, pages 189–209, 1996.

[Mozilla.org, 1998] Mozilla.org. Modularization Techniques. Reference Counting Basics. `http://www.mozilla.org/docs/modunote.htm#RefCount`, 1998.

[Noble *et al.*, 1998] James Noble, Jan Vitek et John M. Potter. Flexible Alias Protection. Dans *12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 de *Lecture Notes in Computer Sciences*, pages 158–185, 1998.

[Palsberg et Schwartzbach, 1991] Jens Palsberg et Michael I. Schwartzbach. Object-Oriented Type Inference. Dans *6th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91)*, pages 146–161. ACM Press, 1991.

[Palsberg et Schwartzbach, 1992] Jens Palsberg et Michael I. Schwartzbach. Safety Analysis Versus Type Inference for partial Types. *Information Processing Letters*, pages 175–180, 1992.

[Plainfossé et Shapiro, 1995] David Plainfossé et Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. Dans *International Workshop on Memory Management (IWMM'95)*, volume 986 de *Lecture Notes in Computer Sciences*, pages 211–249, 1995.

[Plevyak et Chien, 1994] John Plevyak et Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented languages. Dans *9th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'94)*, pages 324–340. ACM Press, 1994.

[Rose, 1988] John R. Rose. Fast Dispatch Mechanisms for Stock Hardware. Dans *3rd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88)*, pages 27–35. ACM Press, 1988.

[Ruf, 1995] Erik Ruf. Context-Insensitive Alias Analysis Reconsidered. Dans *1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, volume 30, pages 13–22. ACM Press, Juin 1995.

[Rugina et Rinard, 1999] Radu Rugina et Martin Rinard. Pointer Analysis for Multithreaded Programs. Dans *1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, volume 34, pages 77–90. ACM Press, Mai 1999.

[Sansom et Jones, 1993] Patrick M. Sansom et Simon L. Peyton Jones. Generational Garbage Collection for Haskell. Dans *1993 Conference on Functional Programming and Computer Architecture (FPCA'93)*, volume 523 de *Lecture Notes in Computer Sciences*. Springer-Verlag, Juin 1993.

[Schmucker, 1986] K. J. Schmucker. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, Hasbrouck Heights, New Jersey, 1986.

[Steele, 1975] Guy L. Steele. Multiprocessing Compactifying Garbage Collection. *Communications of the ACM*, 18(9):495–508, Septembre 1975.

[Stroustrup, 1986] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Series in Computer Science, 1986.

[Sun, 2000a] Java Remote Method Invocation. Garbage Collection of Remote Objects. `http://java.sun.com/products/jdk/1.3/docs/guide/rmi/spec/rmi-arch4.html`, 2000. Sun Microsystems, Inc.

[Sun, 2000b] The Java HotSpot Performance Engine Architecture. A White Paper About Sun's Second Generation Performance Technology. `http://java.sun.com/products/hotspot/whitepaper.html`, 2000. Sun Microsystems, Inc.

[Sun, 2000c] The Java Tutorial. Learning the Java Language. Object and Data Basics. Cleaning Up Unused Objects. `http://java.sun.com/docs/books/tutorial/java/data/garbagecollection.html`, 2000. Sun Microsystems, Inc.

[Suzuki et Terada, 1984] N. Suzuki et M. Terada. Creating Efficient System for Object-Oriented Languages. Dans *11th Annual ACM Symposium on the Principles of Programming Languages (POPL'84)*, pages 290–296. ACM Press, 1984.

[Suzuki, 1981] N. Suzuki. Inferring Types in Smalltalk. Dans *8th Annual ACM Symposium on the Principles of Programming Languages (POPL'81)*, pages 187–199. ACM Press, 1981.

[Turner, 1985] David A. Turner. Miranda — A Non-Strict Functional Language with Polymorphic Types. Dans *1985 Conference on Functional Programming and Computer Architecture (FPCA'85)*, volume 201 de *Lecture Notes in Computer Sciences*, pages 1–16. Springer-Verlag, Septembre 1985.

[Ungar et Patterson, 1987] David M. Ungar et David A. Patterson. What Price Smalltalk? *IEEE Computer Society*, 20(1), Janvier 1987.

[Ungar et Smith, 1987] David M. Ungar et Randall B. Smith. Self: The Power of Simplicity. Dans *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 227–241. ACM Press, 1987.

[Ungar, 1984] David M. Ungar. Generation Scavenging: a Non-Disruptive High-Performance Storage Reclamation Algorithm. Dans *ACM SIGSOFT/SIGPLAN Sotware Engineering Symposium on Practical Sofware Development Environments*, volume 19 de *SIGPLAN Notices*, pages 157–167. ACM Press, Avril 1984.

[Ungar, 1987] David M. Ungar. *The Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, Cambridge, MA, USA, 1987.

[Vitek *et al.*, 1992] Jan Vitek, R. Nigel Horspool et James S. Uhl. Compile-Time Analysis of Object-Oriented Programs. Dans *International Conference on Compiler Construction*, volume 641 de *Lecture Notes in Computer Sciences*, pages 237–250. Springer-Verlag, 1992.

[Wall et Schwartz, 1991] Larry Wall et Randal L. Schwartz. *Programming Perl.* O'Reilly and Associates, Inc, 1991.

[Weis *et al.*, 1989] P. Weis, M.V. Aponte, A. Laville, M. Mauny et A. Suárez. The CAML Reference Manual. Rapport technique, RT 121, INRIA, 1989.

[Wilson *et al.*, 1995] Paul R. Wilson, Mark S. Johnston, Michael Neely et David Boles. Dynamic Storage Allocation: A Survey and Critical Review. Dans *International Workshop on Memory Management (IWMM'95)*, volume 986 de *Lecture Notes in Computer Sciences*, pages 1–116, 1995.

[Wilson et Lam, 1995] Robert P. Wilson et Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. Dans *1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, volume 30, pages 1–12. ACM Press, Juin 1995.

[Wilson, 1992] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. Dans *International Workshop on Memory Management (IWMM'92)*, volume 637 de *Lecture Notes in Computer Sciences*, pages 1–42. Springer-Verlag, 1992.

[Wilson, 1994] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. Rapport technique, University of Texas, USA, Janvier 1994. Version étendue de l'article d'IWMM'92, à paraître dans ACM Computing Surveys. Accessible à `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`.

[Wirth, 1971] N. Wirth. The Programming language PASCAL. *Acta Informatica*, 1(1):25–68, 1971.

[Wolz, 1997] Dietmar Wolz. Comparaison de performances entre compilateurs Eiffel et C++. Communication personelle, 1997.

[Yong *et al.*, 1999] Suan Hsi Yong, Susan Horwitz et Thomas Reps. Pointer Analysis for Programs with Structures and Casting. Dans *1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, volume 34, pages 91–103. ACM Press, Mai 1999.

[Zendra *et al.*, 1997] Olivier Zendra, Dominique Colnet et Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. Dans *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, pages 125–141. ACM Press, Octobre 1997.

[Zendra *et al.*, 1998a] Olivier Zendra, Dominique Colnet et Philippe Coucaud. SmallEiffel: l'Eiffel à Très Grande Vitesse. *Programmez!*, (3):64–67, Octobre 1998.

[Zendra *et al.*, 1998b] Olivier Zendra, Dominique Colnet et Philippe Coucaud. With SmallEiffel, The GNU Eiffel Compiler, Eiffel joins the Free Software community. *GNU Bulletin*, (25), 1998. A paraître.

[Zendra *et al.*, 1999a] Olivier Zendra, Dominique Colnet et Philippe Coucaud. La programmation à objets. Application au langage Eiffel. Partie 3. *Linux Magazine*, (10):60–63, Octobre 1999.

[Zendra *et al.*, 1999b] Olivier Zendra, Dominique Colnet et Philippe Coucaud. La programmation à objets. Application au langage Eiffel. Partie 4. *Linux Magazine*, (11):56–59, Novembre 1999.

[Zendra et Colnet, 1999a] Olivier Zendra et Dominique Colnet. Adding external iterators to an existing Eiffel class library. Dans *32th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99)*, pages 188–199. IEEE Computer Society, Novembre 1999.

[Zendra et Colnet, 1999b] Olivier Zendra et Dominique Colnet. Towards safer aliasing with the Eiffel language. Dans *Intercontinental Workshop on Aliasing in Object-Oriented Systems (IWAOOS'99) - ECOOP'99 Workshop Reader*, volume 1743 de *Lecture Notes in Computer Sciences*, pages 153–154. Springer-Verlag, Juin 1999. Intégré à la 13th European Conference on Object-Oriented Programming (ECOOP'99).

[Zendra et Colnet, 2000a] Olivier Zendra et Dominique Colnet. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software: Practice and Experience*, 2000. A paraître.

[Zendra et Colnet, 2000b] Olivier Zendra et Dominique Colnet. Vers un usage plus sûr de l'aliasing avec Eiffel. Dans *5ème Colloque Langages et Modèles à Objets (LMO'2000), Mont St-Hilaire, Québec*, pages 183–194. Hermes, Janvier 2000.

[Zorn, 1989] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California, Berkeley, Rapport Technique UCB/CSD 89/544, Mars 1989.

[Zorn, 1991] Benjamin G. Zorn. The Effect of Garbage Collection on Cache Performance. Rapport technique, CU-CS-528-91. University of Colorado, Boulder, CO, USA, 1991.

[Zorn, 1993] Benjamin G. Zorn. The measured cost of conservative Garbage Collection. *Software: Practice and Experience*, 23:733–756, 1993.

Index

Abstract

This work falls within the scope of research pertaining to the compilation of class-based languages — especially Eiffel — and more generally about the compilation of statically-typed object-oriented languages. In a nutshell, it can be said the aim of this thesis is to try to answer a fundamental question: *how is it possible to better compile objet-oriented languages, that is, how is it possible to have faster and safer programs?*

This research work is mainly based on static analysis, with a two-pronged approach. The first axis consists in being able to do program validity and coherency checks, not only on finished programs but also from the beginning of the development, so as to be able to assist developers as much as possible during the design and implementation phase. The second axis — the very core of this thesis — considers the use of information coming from the system's static analysis to improve the generated code quality. Indeed, this kind of information offers important optimizations opportunities on the generated code, be it algorithm-based optimizations or data structures optimizations.

We propose and experiment an approach based on code duplication and customization thanks to system-wide analysis, so as to efficiently implement the compiled program data structures and code, especially what pertains to late binding. We thus introduce a novel late binding method based on direct branching trees whose performance are greater than or equal to those of current classical systems relying on indirection tables.

This approach is also extended to the generation by the compiler of a garbage collector which is automatically customized to the compiled application.

We also carry some studies to evaluate the optimizations allowed by massive use of aliasing in a compiler written in a class-based language, as well as means to better handle this technique.

This work is validated by, among other things, the development of an Eiffel compiler named SmallEiffel and its libraries, which, widely distributed and used, have become *The GNU Eiffel Compiler*.

Keywords: class-based language, object-oriented language, compilation, static analysis, global analysis, type prediction, late binding, dynamic dispatch, optimization, customization

Résumé

Ce travail s'inscrit dans le cadre des recherches menées autour de la compilation des langages de classes, notamment Eiffel, et plus généralement des langages à objets à typage statique. Très brièvement, on peut dire que le but de cette thèse revient à tenter de répondre à une question fondamentale: *comment mieux compiler les langages à objets, c'est à dire comment avoir des programmes plus rapides et plus sûrs ?*

Ce travail de recherche est basé en grande partie sur l'analyse statique, abordée via deux axes principaux. Le premier consiste à pouvoir effectuer des contrôles de validité et de cohérence du programme, et ce non seulement sur les programmes finis, mais bien dès le début du développement, de façon à pouvoir assister au maximum les développeurs durant la phase de conception et d'implantation. Le second axe, qui est la substance même de cette thèse, considère l'utilisation des informations apportées par l'analyse statique du système pour améliorer la qualité du code généré. En effet, ces informations offrent des possibilités importantes en terme d'optimisation du code généré, aussi bien par des optimisations liées aux algorithmes que par des optimisations sur les structures de données.

Nous proposons et expérimentons une approche basée sur la duplication et la spécialisation du code par analyse globale du système, afin d'implanter de façon efficace les structures de données et le code du programme compilé, notamment en ce qui concerne la liaison dynamique. Nous introduisons ainsi une nouvelle méthode de liaison dynamique, basée sur des arbres de branchement directs, dont les performances sont supérieures ou égales à celles des systèmes actuels classiques à base de tables d'indirection.

Cette approche est également étendue à la génération par le compilateur d'un ramasse-miettes automatiquement adapté à l'application compilée.

Nous menons aussi certaines études pour évaluer les optimisations permises par l'utilisation massive de l'aliasing dans un compilateur écrit dans un langage de classes, ainsi que des moyens de mieux maîtriser cette technique.

Ces travaux sont validés entre autres par le développement d'un compilateur Eiffel nommé SmallEiffel et de ses bibliothèques, qui, très largement diffusés et utilisés, sont devenus *The GNU Eiffel Compiler*.

Mots-clés: langage de classes, langage à objets, compilation, analyse statique, analyse globale, prédiction de type, liaison dynamique, optimisation, spécialisation