Étude de l'impact des weak references sur la gestion mémoire d'un langage à objets de haut niveau

Frederic Merizen

Université Henri Poincaré – Nancy I

Étude de l'impact des *weak references* sur la gestion mémoire d'un langage à objets de haut niveau.

MÉMOIRE

soutenu le 4 septembre 2003

pour l'obtention du

DEA de l'Université Henri Poincaré – Nancy I

(Spécialité Informatique)

par

Frederic Merizen

Composition du jury

	Dominique Méry	Professeur UHP, Responsable formation DEA
	Didier Galmiche	Professeur UHP
	Noëlle Carbonell	Professeur UHP
Encadrant :	Olivier Zendra	Chargé de recherche INRIA-Lorraine

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

Mis en page avec la classe thloria.

Table des matières

Introduction

Ce mémoire présente le résultat des travaux que j'ai effectués au LORIA au sein de l'équipe DESIGN de début avril à fin août 2003. Ces travaux ont porté sur les références faibles dans les langages à objets. J'ai dans un premier temps étudié la notion de référence faible et analysé l'existant dans d'autres langages. J'ai ensuite travaillé sur les concepts précédamment développés en vue de les intégrer au langage Eiffel. Enfin, j'ai validé mes travaux par leur implantation dans le ramasse-miettes de SmartEiffel, le compilateur GNU Eiffel.

SmartEiffel a la particularité de *générer* un ramasse-miettes *adapté* au programme précis qui est compilé. De plus, comme SmartEiffel est écrit en Eiffel, il est son propre compilateur, et ce ramasse-miettes contrôle donc aussi la mémoire de SmartEiffel lui-même.

Dans le chapitre 1, nous découvrirons les bases de la gestion automatique de la mémoire, ainsi que le concept de référence faible.

Dans le chapitre 2, nous montrerons plusieurs applications réelles qui justifient l'emploi de référence faible.

Dans le chapitre 3, nous passerons en revue les interfaces sous lesquelles les références faibles sont exposées en pratique par plusieurs langages courants, en montrant comment celles-ci rendent possibles les applications vues précédemment.

Dans le chapitre 4, nous élaborerons un modèle de référence faible pour le langage Eiffel, en confrontant les besoins et principes vus précédemments au particularités d'Eiffel.

Enfins, dans le chapitre 5, nous verrons comment j'ai mis en pratique mes travaux dans le compilateur SmartEiffel.

Chapitre 1

Gestion mémoire automatique

1.1 Limites du modèle `free`/`malloc`

Les langages fortement typés, dont Eiffel fait partie, gèrent généralement la mémoire de façon automatique, au moyen de ce qu'on appelle un *ramasse-miettes* . En effet, la gestion manuelle de la mémoire par le couple `free`/`malloc` est la source de deux classes d'erreurs :

Les fuites de mémoire se produisent quand une application perd la dernière référence vers un objet alloué sans libérer la mémoire associée.

Les références errantes sont causées par une application qui libère de la mémoire tout en gardant des références vers celle-ci, puis qui utilise ces références pour accéder à la mémoire.

Dans un système à gestion automatique de la mémoire, le ramasse-miettes garantit qu'une portion de mémoire ne sera libérée que si celle-ci devient inaccessible [1]. Parmi les nombreux algorithmes pour détecter les objets inatteignables, deux des plus connus sont décrits succinctement en annexe A (page 20).

1.2 Rôle d'un ramasse-miettes

La rôle d'un ramasse-miettes est de libérer de la mémoire inutilisée tout en garantissant la propriété essentielle suivante :

Tant qu'un objet est référencé, la mémoire associée n'est pas libérée.

C'est cette propriété qui nous assure qu'une erreur de référence errante ne peut pas se produire. En revanche, il est difficile de formuler une garantie concernant les fuites de mémoire : les références circulaires (cf. annexe A.1), le fait que certains ramasse-miettes sont conservatifs (cf. annexe A.3), et simplement celui que la plupart ne sont pas en permanence à la recherche de mémoire à libérer font que certains objets inutilisés peuvent très bien rester alloués jusqu'à la fin de l'exécution du programme.

[1] En revanche, il n'est pas garanti que la mémoire sera libérée *dès* qu'elle devient inaccessible.

1.3 Concept de référence faible

Par contraste avec les références ordinaires, ou fortes, les références faibles permettent d'accéder aux objets qu'elles référencent sans empêcher le ramasse-miettes de libérer la mémoire. Pour que cela soit possible sans créer un risque de références errantes, le ramasse-miettes offre pour les références faibles une garantie symétrique de celle offerte pour les références fortes :

Un objet reste faiblement référencé tant que la mémoire associée à cet objet n'est pas libérée.

Cela veut dire que le ramasse-miettes peut libérer un objet si celui-ci n'est référencé que par des références faibles, mais que quand il le fait, il met à jour toutes les références vers cet objet pour indiquer qu'elles ne sont plus valides.

Chapitre 2

Contexte : un besoin de références plus faibles

Nous avons vu au chapitre précédent que les ramasse-miettes protègent le développeur contre les références errantes en garantissant que la mémoire d'un objet n'est pas libérée tant que celui-ci est accessible. Étudions maintenant quelques cas où on souhaiterait pouvoir accéder à des objets tant qu'il existent sans empêcher le ramasse-miettes de libérer la mémoire associée. Les références faibles répondent à ce besoin sans créer de risque de références errantes.

2.1 Cache de données

Un programme manipule parfois des données qui ont été obtenues à un certain coût, mais que l'on peut déterminer à nouveau si elles sont perdues. On peut penser par exemple à la liste des fichiers d'un répertoire ou au contenu d'un fichier, qui peuvent être relus, ou encore à la connexion à une base de données, qui peut être rouverte si elle vient à être fermée.

Il est cependant intéressant de conserver ce genre de données entre deux utilisations pour des raisons de performances. Néanmoins, si le programme conserve toutes les données susceptibles d'être réutilisées, il finira par venir à bout de la mémoire disponible. Au mieux, les performances seront dégradées par des accès au fichier d'échange sur disque ; au pire, l'exécution du programme ne pourra pas se poursuivre. Il est donc souhaitable de libérer certains des objets gardés en mémoire quand celle-ci devient rare : il s'agit d'implanter un *cache mémoire*. Puisque c'est le ramasse-miettes qui libère les objets inutilisés, il paraît logique de lui confier aussi la libération des objets du cache. Pour réaliser cela, le cache sera constitué d'objets faiblement référencés.

2.2 Finalisation par référence

Il est parfois nécessaire d'effectuer des actions de nettoyage avant la destruction d'un objet : le mécanisme permettant de le faire s'appelle la *finalisation*. C'est par exemple un moyen de libérer des ressources externes, telles que de la mémoire non gérée par le ramasse-miettes, un

fichier ouvert ou un *handle* de fenêtre [2].

Dans une mise en œuvre du mécanisme, appelée *finalisation par objet*, chaque objet peut posséder une routine particulière qui est automatiquement appelée au moment de la destruction de l'objet. Cette façon de faire peut introduire un phénomène connu sous le nom de *résurrection* : si la routine de finalisation communique à d'autres objets une référence sur l'objet en cours de destruction, celui-ci peut redevenir vivant. On sait bien gérer la résurrection, mais cela a un coût important en temps d'exécution et/ou en consommation mémoire. C'est probablement pourquoi la résurrection est fortement déconseillée en Java [GJS96] et en Eiffel.

La mauvaise réputation de la résurrection explique peut-être que certains langages offrent uniquement une autre forme de *finalisation*, dite *par référence*. On citera par exemple le compilateur Glasgow Haskell [JME99]. La finalisation n'est alors pas effectuée par l'objet sur le point de disparaître, mais par un objet qui surveille le premier. Évidemment, cette surveillance ne peut se faire qu'à travers une référence faible puisque une référence classique maintiendrait l'objet surveillé en vie. Comme la finalisation a lieu après la destruction de l'objet, le problème de la résurrection est éliminé. Par ailleurs, le fait que la destruction de l'objet et la finalisation soient rendues asynchrones est avantageux dans le cas de la programmation parallèle. En revanche, comme la routine qui réalise la finalisation n'a pas accès à l'objet finalisé, on est souvent obligé de dupliquer des données. Nous verrons les conséquences de ce point en discutant des ephemerons au paragraphe 3.5.

Certains langages ont choisi la finalisation par référence comme complément plutôt que comme substitut à la finalisation par objet. Elle permet en effet à un objet d'être informé de la mort d'un autre objet sans que celui-ci en soit affecté. En particulier, si on veut surveiller des objets d'une classe fournie par une tierce partie, il serait gênant de devoir créer un descendant de cette classe simplement pour redéfinir la routine de finalisation. Nous verrons au paragraphe 2.4 que la finalisation par référence peut servir à mesurer la durée de vie d'objets quelconques.

2.3 Attachement d'informations à un objet

On a parfois besoin d'associer des données à un objet sans pouvoir stocker celles-ci directement dans l'objet. L'alternative consiste alors à utiliser une paire de références, l'un référençant l'objet et l'autre les données. C'est en particulier ainsi que sont implantées les tables de hachage. Souvent, on souhaite conserver l'objet et les données, et des références ordinaires sont alors adaptées.

Si, au contraire, il faut simplement exprimer qu'un lien existe entre l'objet et les données, *tant que* les deux existent, c'est deux références faibles qu'il faut employer. On aura d'ailleurs intérêt à y associer une finalisation par référence, pour supprimer le couple de références de la table dès qu'un des deux objets disparaît. Ce type de table peut par exemple constituer un cache de données pour un dictionnaire, avec des mots pour clefs et leurs définitions pour valeurs.

Les cas intermédiaires, où seule une des deux références est faible, sont également utiles. Un mécanisme de finalisation attaché à la référence faible servira alors à détruire la paire quand l'objet faiblement référencé disparaît. Parfois, la finalisation est le seul but de ce type d'association, et l'objet fortement référencé comporte alors les informations nécessaires à la finalisation

[2] Ceux qui découvrent la finalisation s'appuient parfois trop sur ce mécanisme — quand la libération de ressources dépend trop fortement de la gestion mémoire le programme devient compliqué et des bugs apparaissent.

de l'objet faiblement référencé. Par exemple, si un objet accède à un fichier qui doit être fermé si l'objet est détruit, la paire de références comportera une référence faible vers cet objet et une référence forte vers le descripteur du fichier. Si l'objet est détruit, le mécanisme de finalisation sera activé et fermera le fichier grâce à la référence forte. Ensuite, il supprimera la paire de références, ce qui aura pour effet de rendre le descripteur de fichier candidat à la libération s'il n'est pas référencé par ailleurs.

2.4 Instrumentation du ramasse-miettes

Ole Agesen et Alex Garthwaite présentent dans [AG00] un ramasse-miettes qui s'adapte dynamiquement au programme géré, et ce au moyen de mesures comportant en particulier la durée de vie des objets en fonction de leur classe et de leur site de création. Pour ne pas détériorer les performances du logiciel, les mesures ne sont en fait effectuées que sur un échantillon de tous les objets créés. Or, la liste des objets à surveiller ne peut pas comporter de références ordinaires sur les objets : les références fortes fausseraient les mesures au point de les rendre sans objet puisque aucun objet surveillé ne pourrait être libéré.

Les auteurs emploient donc des références qui n'empêchent pas le ramasse-miettes de récupérer la mémoire. Puisque des informations comme la durée de vie des objets leur sont attachées, on peut considérer cette application comme un cas particulier de l'attachement de données à des objets, vu à la section précédente. Par ailleurs, le mécanisme de finalisation par références est utilisé pour mesurer la durée de vie des objets. L'adaptation dynamique du ramasse-miettes proposée n'a de sens que si celui-ci est *à générations*. En revanche, le concept d'instrumentation peut s'appliquer à n'importe quel ramasse-miettes.

2.5 Recyclage de la mémoire

Les situations compliquées, où des objets sont référencés par plusieurs objets à la fois et où les références sont détruites dans un ordre imprévisible sont le domaine de prédilection des ramasse-miettes : ils épargnent au développeur des bugs difficiles à cerner et se révèlent souvent plus efficaces que les solutions développées au cas par cas pour libérer la mémoire.

En revanche, dans les cas simples où un objet est créé, existe pendant un certain temps puis cesse indiscutablement d'être utilisé, par exemple parce que la référence initiale est supprimée et qu'aucune autre référence sur lui n'a été créée, les ramasse-miettes sont moins efficaces que le serait une gestion manuelle de la mémoire. Pour réduire le nombre d'activations du ramasse-miettes, on peut *recycler* les objets en fin de vie en les stockant dans une liste d'objets inutilisés. Avant de créer un nouvel objet du même type, on vérifier alors si cette liste contient un objet réutilisable, et on ne crée un nouvel objet que si cette liste est vide. La liste peut contenir des références ordinaires sur les objets recyclables, au prix d'un double inconvénient :

– avant d'insérer un objet dans la liste de recyclage, il faut annuler toutes les références qu'il contient pour ne pas risquer une rétention mémoire importante. Cela veut dire que le centre de recyclage de chaque type d'objets doit comporter une méthode de nettoyage des références spécifique à ce type.

– les références fortes empêchent les objets inutilisés d'être libérés et de redevenir éventuellement de la mémoire à usage généraliste. C'est particulièrement fâcheux si un certain type d'objets connaît un pic d'utilisation important suivi d'une phase où peu d'objets de ce type existent.

Ces deux inconvénients disparaissent en remplaçant les références fortes par des références faibles. N'oublions pas, cependant, qu'un développeur qui recyclerait à tort un objet encore en utilisation redécouvrirait les délices des bugs incompréhensibles de la gestion manuelle de la mémoire. Le mot d'ordre en matière de recyclage est donc : dans le doute, s'abstenir[3].

2.6 Rupture de références circulaires

Dans les langages employant un ramasse-miettes par comptage de références, les cycles d'objets inutilisés ne sont pas libérés (cf. annexe A.1 p.20). On essaye donc de détruire un des liens constituant le cycle juste avant d'abandonner la dernière référence vers ce cycle. Pourtant, dans l'esprit de la gestion automatique de la mémoire, le développeur ne devrait pas avoir à se soucier du moment précis où un objet cesse de pouvoir être atteint.

On a donc proposé d'éviter de constituer des cycles, en remplaçant certaines références judicieusement choisies par une variante qui ne serait pas comptabilisée dans le décompte des références. Par exemple, dans une liste doublement chaînée, on pourrait implanter les liens vers l'avant comme des références ordinaires, et ceux vers l'arrière comme des références non décomptées. On peut étendre le concept, par exemple à des arbres dont chaque noeud contiendrait un lien vers le parent.

Nous avons cité cette application parce qu'elle semble avoir constitué la motivation première pour l'implantation de références faibles dans de nombreux langages où la mémoire est gérée par comptage de références. Pourtant, ce n'est à nos yeux qu'un pis-aller : il vaut mieux implanter un ramasse-miettes capable de gérer les cycles de références que de confronter le développeur à des détails de gestion de la mémoire.

[3] Cette remarque n'a pas pour ambition de s'appliquer en dehors du cadre de la gestion mémoire !

Chapitre 3

État de l'art : Références faibles dans différents langages

Ce chapitre fait un tour de l'état de l'art des référence faible telles qu'elles sont intégrées aux langages existants.

3.1 Le type entier, référence faible du pauvre

Visual Basic, jusqu'à la version 6, gérait la mémoire par comptage de références, et ne disposait pas de références faibles. La technique suivante a été développée dans [vB03] pour éviter le problème des références circulaires : certaines références sont remplacées par des entiers.

Pour contourner le système de typage de Visual Basic et lire la valeur d'une référence sans incrémenter le compteur de l'objet, une copie mémoire brute est utilisée :

```
CopyMemory wref, obj, 4
```

De la même façon, pour transférer le contenu d'un entier dans une référence, une copie mémoire est utilisée. Celle-ci est suivie d'une incrémentation manuelle du compteur de références de l'objet.

```
CopyMemory obj, wref, 4
CopyMemory RefCount, ByVal (ObjPtr(obj)) + 4, 4
CopyMemory ByVal (ObjPtr(obj)) + 4, (RefCount + 1), 4
```

Cette astuce ne constitue pas une véritable référence faible, car le ramasse-miettes ne peut pas signaler au programme qu'un objet a été libéré, et de plus, la technique ne fonctionnerait pas si le ramasse-miettes était conservatif [4] et identifiait les entiers comme des références valides (voir l'annexe A.3 en page 21 à propos des ramasse-miettes conservatifs). En somme, bien que nous ne soyons pas en présence d'un outil généraliste convenable, il s'agit cependant d'une solution *ad hoc* au problème précis des références circulaires.

[4] Le comptage de références n'est pas conservatif.

3.2 *Soft, weak* et *phantom* : trois forces de références faibles

Java [GJS96], depuis la version 1.2 du langage, offre les références faibles au développeur sous la forme des descendants de la classe `java.lang.ref.Reference`. Sa particularité est d'offrir non pas une, mais trois références faibles, selon l'usage auquel elles sont destinées. De plus, chaque référence faible peut être associée à une queue, dans laquelle elle est ajoutée quand l'objet référencé est libéré. Cette queue est une aide à la finalisation, surtout dans un contexte parallèle, où il est possible à un fil (ou *thread*) de finalisation de faire une attente passive sur la queue. Ces trois types de références faibles sont :

Les *soft-references* indiquent au ramasse-miettes que l'objet désigné ne doit pas être détruit *dans la mesure du possible*. Cependant, tous les objets *soft-référencés*, c'est-à-dire désignés par une ou plusieurs *soft-references* mais aucune référence forte, seront détruits avant qu'une erreur de manque de mémoire ne soit provoquée. Le ramasse-miettes cherchera typiquement à libérer les objets inutilisés depuis longtemps en premier (avec des algorithmes de type LRU ou *Least Recently Used*). Ce type de référence est par exemple utile pour les caches de données (cf. paragraphe 2.1).

Les *weak-references*, elles, *ne maintiennent pas l'objet référencé en vie* : si le ramasse-miettes découvre qu'un objet n'est plus atteignable que par des *weak-references*, celui-ci sera détruit. Ces références peuvent servie à attacher des données à d'autres objets (cf. paragraphe 2.3). Le ramasse-miettes de Java est capable de gérer les cycles de références, mais c'est également ce type de référence qui permet de les rompre (cf. paragraphe 2.6) si le ramasse-miettes ne sait pas les gérer.

Les *phantom-references* ne maintiennent pas les objets référencés en vie, et *ne sont déréférençables que par le mécanisme de finalisation*. Elles sont obligatoirement associées à des queues de références, et servent donc à la finalisation.

À ma connaissance, aucun langage sauf Java n'implante de *soft-references*. Dans le cas des langages qui n'ont pas de *soft-references*, si le ramasse-miettes n'est pas activé trop souvent, on peut leur substituer des *weak-references* pour constituer des caches de données. En revanche, dans le cas des langages qui s'appuient sur le comptage de références, les objets sont libérés dès que la dernière référence forte est perdue, ce qui réduit fortement l'intérêt d'un cache qui emploierait des *weak-references*. Or, on voit mal comment implanter des *soft-references* à l'aide d'un comptage de références, et on en est donc réduit pour créer d'éventuels caches à utiliser des références fortes et à gérer manuellement la libération des objets les plus anciens.

3.3 Conteneurs faibles

On constate que les applications typiques des références faibles concernent plutôt des groupes d'objets que des objets individuels. Partant de là, beaucoup de langages, dont Java et de nombreux dialectes Smalltalk [GR83] offrent un ou plusieurs des conteneurs faibles suivants.

Les vecteurs faibles sont l'équivalent d'un vecteur ou d'un tableau, à ceci près que les éléments du tableau peuvent éventuellement être détruits par le ramasse-miettes.

Les tableaux associatifs à clefs faibles contiennent des couples clef-valeur qui référencent faiblement les clefs, et qui sont supprimés du tableau dès que la clef est libérée par le ramasse-miettes.

Les tableaux associatifs à valeurs faibles contiennent également des couples clef-valeur, mais ici ce sont les valeurs qui sont faiblement référencées et qui peuvent provoquer la suppression de la paire du tableau.

Les tableaux associatifs doublement faibles référencent faiblement à la fois les clefs et les valeurs, et pour chaque paire il suffit qu'un des deux éléments soit libéré pour que la paire soit retirée

Certains langages, dont Guile [5] n'offrent les références faibles que sous la forme de ces conteneurs.

Les vecteurs faibles peuvent servir à implanter le recyclage d'objets du chapitre 2.5 ; les tableaux associatifs faibles permettent d'attacher de données à un objet (paragraphe 2.3).

3.4 Objets mandataires (*proxy*)

Toute référence faible peut être vue comme le mandataire d'une référence forte. Dans le langage Python, en plus des références faibles, il existe des objets dits *mandataires* qui référencent faiblement leur objet cible. Au lieu de fournir une référence forte sur l'objet cible, ils adoptent eux-mêmes l'interface publique de celui-ci. L'appel d'une routine de l'objet mandataire est résolu en l'appel de la routine équivalente de l'objet cible. Si celui-ci a été détruit, une exception est déclenchée.

Les objets mandataires Python n'apportent aucune fonctionnalité additionnelle par rapport aux références faibles : ils allègent simplement la syntaxe en cachant un niveau d'indirection. En revanche, ils sont un peu plus lents à l'utilisation car le mandataire doit vérifier *à chaque opération* que l'objet référencé est encore vivant. On peut supposer que les objets mandataires ont été créés pour rendre transparente la suppression des cycles de références.

3.5 Ephemerons

Nous avons vu au paragraphe 2.3 sur l'association de données à un objet comment créer des conteneurs associatifs faibles par rapport à la clef au moyen d'un mécanisme de finalisation. Mais que se passe-t-il si la valeur référence la clef ? Citons l'exemple donné par [JME99], ou les clefs sont des noms de personnes et les valeurs des fiches sur ces personnes : les fiches peuvent légitimement référencer le nom. Si aucune autre partie du programme ne référence un nom donné, nous voudrions que la paire nom/fiche soit identifiée comme inatteignable et éliminée. Or, ce n'est pas le cas : du point de vue du ramasse-miettes, le nom est atteignable par une chaîne de références fortes allant au conteneur associatif, puis du conteneur à la valeur, et enfin de la valeur à la clef. Pour que la table se comporte comme prévu, il faudrait en fait que la force des références partant de la valeur soit conditionnée par le fait que la clef soit atteignable ou non.

[5] Guile est l'implantation GNU du dialecte Scheme du langage Lisp.

Dans [Hay97], Barry Hayes identifie ce problème dans le cas de la finalisation par référence, puis le généralise aux conteneurs associatifs semi-faibles, et propose la solution suivante, qu'il baptise *ephemeron* :

Un ephemeron est une paire constituée d'une référence faible et d'une référence forte. La référence forte n'est considérée comme atteignable, et ne conduit au marquage récursif de l'objet qu'elle référence, que si l'ephemeron et l'objet faiblement référencé sont tous les deux atteignables.

On se convainc facilement qu'en utilisant des ephemerons pour représenter les paires clef/valeur, les valeurs ne peuvent plus maintenir les clefs en vie indûment. En revanche, l'algorithme de ramassage de miettes se trouve compliqué par la gestion des ephemerons. Il est en effet nécessaire de parcourir tous les ephemerons atteignables après l'examen des autres objets, pour vérifier si les objets faiblement référencés sont atteignables ou non, et donc s'il faut marquer récursivement la partie fortement référencée ou non. De plus, ce marquage pouvant à son tour rendre atteignables des ephemerons ou des objets faiblement référencés par un ephemeron, il est nécessaire de reparcourir la liste des ephemerons jusqu'à aboutir à un point fixe. Malgré cette difficulté, les ephemerons ont été implantés dans le compilateur Dolphin Smalltalk [dol].

Notons qu'une solution similaire, baptisée *pointeur faible clef-valeur*, a été découverte par Jones, Marlow et Elliott indépendamment de Hayes, et décrite dans [JME99].

Un pointeur faible clef-valeur est une paire constituée d'une référence faible et d'une référence forte. La référence forte n'est considérée comme atteignable, et ne conduit au marquage récursif de l'objet qu'elle référence, que si l'objet faiblement référencé est atteignable.

Les pointeurs faibles clef-valeur résolvent des types de problèmes et présentent des difficultés comparables aux ephemerons. Ils ont été développés pour et implantés dans le compilateur Glasgow Haskell [JME99]. Notons enfin que les concepteurs de Java sont conscients du problème des conteneurs associatifs semi-faibles, mais se sont contentés de le documenter [jav], estimant probablement qu'il ne se produit pas suffisamment souvent en pratique pour justifier une solution aussi élaborée que les ephemerons.

Chapitre 4

Références faibles : conception

4.1 Modèle de référence faible pour Eiffel

Des cas étudiés précédemment se dégagent les besoins suivants :

- Une référence faible ne doit pas empêcher la libération de la mémoire associée à un objet.
- Aucune référence errante ne doit apparaître, donc il ne doit pas être possible d'atteindre un objet à travers une référence faible une fois qu'il a été détruit.
- Un objet atteignable à travers une référence faible doit le rester tant que l'objet n'a pas été détruit.
- Le ramasse-miettes doit informer le programme quand un objet faiblement référencé est détruit.

Dans le modèle de référence faible que nous avons conçu pour Eiffel, le ramasse-miettes libère la mémoire d'un objet s'il découvre que celui-ci n'est plus fortement référencé. Cette référence est donc similaire aux *weak-references* de Java. Il s'agit à notre avis de la référence la plus généraliste. En effet, elle n'altère pas la durée de vie de l'objet référencé, ce qui est très désirable voire indispensable pour la gestion semi-automatique de la mémoire et l'instrumentation du ramasse-miettes. Par ailleurs, si le ramasse-miettes n'est pas invoqué trop souvent, elle permet de réaliser des caches de données intéressants. Enfin, cette référence nous a semblé une base saine au dessus de laquelle on pourrait implanter, si le besoin s'en faisait sentir, des perfectionnements tels que les *soft-references* ou les ephemerons.

Le développeur utilisera notre référence faible par deux routines :

- `set_item` pour indiquer quel objet est faiblement référencé
- `item` pour obtenir une référence forte sur l'objet faiblement référencé, ou `Void` si celui-ci a été détruit

On accède à un objet faiblement référencé par les étapes suivantes :

1. Obtenir de la référence faible une référence ordinaire sur l'objet.
2. Vérifier que cette référence n'est pas `Void`.
3. Travailler avec la référence comme d'ordinaire.
4. S'assurer que la référence forte ne maintient pas l'objet en vie inutilement une fois qu'on n'en a plus besoin. Si la référence est une variable locale, elle est automatiquement perdue à la fin de la routine, sinon il peut être souhaitable de la mettre à `Void`.

4.2 Construction du langage ou objet de bibliothèque ?

En choisissant d'ajouter le support des références faibles au compilateur SmartEiffel, j'ai été amené à choisir entre les présenter comme une extension au langage Eiffel en créant un nouveau mot clef, ou bien rendre cette fonctionnalité accessible au moyen d'une classe de la bibliothèque standard.

La philosophie du développement d'Eiffel, exposée dans [Mey92], est de garder un langage de petite taille pour minimiser les interactions arbitraires entre ses différentes composantes. De plus, le choix d'utiliser une classe plutôt qu'un mot-clef m'évitait de devoir intervenir dans l'analyseur syntaxique du compilateur, ce qui n'était pas l'objet de mon travail. Un classe constitue aussi un terrain d'expérimentation plus facile pour des développements futurs. J'ai donc conçu la référence faible sous la forme d'une classe. Quand suffisament d'expérience de terrain aura été collectée, il sera temps de reconsidérer la question de l'emploi d'un mot clef.

4.3 Références faibles expansées ou référencées ?

Les références faibles étant de simples objets et non des constructions particulières du langage, il faut alors choisir si elles sont de type référence, expansé, ou si les deux sont admis (La notion d'objet expansé est décrite en annexe B p. 22). Nous avons pour cela considéré des critères d'efficacité à l'exécution, de correction du programme et de facilité d'emploi par le développeur.

Efficacité Nous nous intéressons ici à la surconsommation mémoire entraînée par des références sur des références faibles, par opposition à des référence faible expansées. Le surcoût d'une référence, soit un mot machine, peut *a priori* sembler négligeable, mais la représentation en mémoire d'une référence faible peut se limiter à un pointeur sur l'objet référencé. Si on y ajoute l'entête du ramasse-miettes, une référence faible n'occupe donc que deux mots machine en mémoire, et le surcoût lié à une référence faible référencée plutôt qu'expansée est donc de 50 %, et l'encombrement mémoire est donc un argument sérieux en faveur du modèle expansé.

Correction Il est bien sûr primordial que notre implantation des références faibles soit correcte, et en particulier qu'elle ne permette pas de créer de références errantes. Or, les variables locales de types expansés ne sont pas allouées dans le tas mais déclarées comme des variables locales C et stockées dans la pile ou un registre du processeur. En conséquence, elles seront sujettes à la partie conservative du marquage des références.

Pour comprendre comment se passera ce marquage, il nous faut maintenant imaginer comment la référence faible désigne l'objet référencé. Deux cas de figure s'offrent à notre choix :

- La référence faible désigne l'objet indirectement, par exemple par un indice dans un tableau de pointeurs.
- La référence faible désigne directement l'objet.

Le premier cas est identique à celui d'une référence faible référencée plutôt qu'expansée et ne nous intéresse pas ici, nous nous plaçons donc dans le deuxième cas.

Supposons pour l'instant que dans un souci de cohérence, la référence faible contient un pointeur vers l'objet référencé, tout comme une référence ordinaire est en fait un pointeur.

Dans sa phase conservative, le ramasse-miettes traite comme une référence forte tout ce qui y ressemble ; en particulier, le pointeur contenu dans la référence faible ne pourra pas être distingué d'une référence ordinaire valide, et provoquera donc le marquage récursif de l'objet faiblement référencé. En d'autres termes, une référence faible expansée dans une variable locale se comporte comme une référence ordinaire. Cela n'aura pas pour conséquence une erreur du programme, mais simplement une surconsommation de mémoire, ce qui est moins grave mais néanmoins indésirable.

Revenons alors sur notre hypothèse et supposons que la référence faible contient en fait un pointeur altéré pour ne plus être reconnu par le ramasse-miettes, tout en continuant à pouvoir facilement être transformé en référence ordinaire [6]. Le marquage conservatif ne reconnaissant plus les pointeurs des références faibles, le problème de rétention mémoire est éliminé. Par conséquent, si un objet est détruit pendant qu'une référence faible de la pile le désigne, il faut la mettre à jour pour qu'elle fournisse la référence `Void` plutôt qu'une référence sur un objet détruit. Or, il n'existe pas de moyen certain de modifier exactement les références faibles de la pile. Si le ramasse-miettes met à jour tous les objets qui ressemblent à des références faibles, un entier avec une valeur malheureuse serait également mis à jour, ce qui constituerait une faute grave. L'autre alternative, consistant à ne pas mettre à jour les références faibles en pile, serait également inacceptable puisqu'elle entraînerait des risques de références errantes. Nous sommes donc obligés d'abandonner l'hypothèse de l'emploi de pointeurs déguisés.

En conséquence, il est possible d'autoriser les références faibles à être de typé expansé, mais au prix d'une augmentation potentielle de la rétention mémoire du ramasse-miettes.

Facilité d'emploi Les références faibles sont plus fréquemment employées dans des tableaux qu'individuellement — on pensera par exemple au cas des caches. Il faut donc qu'elles soient faciles à utiliser dans ce contexte. Or, si `tab` est un tableau de références expansées, l'accès à un élément de celui-ci, par `tab.item(i)` produira une *copie* de la référence en ième position du tableau. L'instruction à laquelle on pense naturellement pour changer l'objet référencé par un élément du tableau, `tab.item(i).set_item(obj)` ne fonctionne donc pas : elle crée une copie de la référence faible, et applique `set_item` à celle-ci, sans modifier l'élément du tableau. La solution consiste à utiliser une référence faible auxiliaire `aux`, à la positionner à la bonne valeur et à l'insérer dans le tableau :

```
aux.set_item(obj)
tab.put(aux, i)
```

Par ailleurs, si on cherche à contourner la question en autorisant à la fois les références faibles expansées et référencées, une autre lourdeur apparaît. En effet, dans la révision 2 (qui est l'actuelle) du langage Eiffel [Mey92], une classe expansée ne peut avoir qu'aucune ou une seule routine de création (ou constructeur), laquelle ne doit pas prendre de paramètre. Pratiquement, cela signifie que l'instruction de création de la référence faible `wref` avec valeur initiale `create wref.set_item(obj)`, n'est permise que si les références faibles expansées sont interdites. Si elles sont autorisées, il faut recourir à la forme plus longue suivante :

```
create wref
wref.set_item(obj)
```

[6] Un ou exclusif bit à bit avec un motif binaire judicieusement choisi convient.

Si les références faibles expansées sont autorisées, la syntaxe intuitive ne produira pas l'effet attendu dans le premier cas et sera interdite dans le deuxième. Dans les deux cas, la syntaxe correcte serait plus lourde et ferait obstacle à l'acceptation des références faibles en Eiffel. Ceci constitue un argument fort en défaveur des références faibles expansées.

Choix retenu Nous avons laissé l'ergonomie primer sur la performance en implantant des références faibles référencées. Cependant, il ne serait pas nécessaire d'apporter beaucoup de changements au compilateur pour supporter les références faibles expansées. Quand SmartEiffel implantera la construction `default_create` proposée dans la future troisième révision du langage Eiffel, il sera possible d'offrir au développeur le choix entre des références faibles expansées et référencées sans interdire la forme `create wref.set_item(obj)`.

4.4 Des références faibles d'objets expansés ?

Faut-il autoriser l'utilisateur à créer des références faibles sur des objets de type expansé ? Je ne leur ai pas trouvé d'application pratique, ni même de sémantique raisonnable. Plutôt que d'encombrer les références faibles avec une non-fonctionnalité, j'ai donc choisi de créer une situation claire en interdisant les références faibles vers des objets expansés. Je me sens conforté dans ce choix par l'évolution du langage Eiffel.

En effet, si dans la révision actuelle du langage Eiffel, il est permis à une référence ordinaire de désigner un objet expansé, ce n'est plus le cas dans le brouillon de la future troisième révision : quand on tente d'attacher un objet expansé à une référence, c'est en fait une copie référencée de l'objet qui est attachée. L'auteur justifie cela par le fait que l'ancien modèle complique significativement le modèle dynamique, et l'implantation, en particulier celle du ramasse-miettes, sans apporter d'amélioration clairement utile au pouvoir expressif.

Chapitre 5

Implantation dans SmartEiffel

Après avoir développé les concepts liés aux références faibles dans Eiffel, je me suis attaché à les appliquer dans le compilateur SmartEiffel.

Ce chapitre fait référence au fonctionnement du ramasse-miettes de SmartEiffel, qui est du type marquage-balayage. On trouvera le principe de fonctionnement de ce type de ramasse-miettes en annexe A.2 (p.20), et le détail du ramasse-miettes et de l'organisation de la mémoire de SmartEiffel dans [CCZ98].

5.1 La classe `weak_ref`

Les références faibles sont mises à la disposition du développeur par l'intermédiare d'une classe particulièrement simple, puisqu'elle ne comporte que la variable `item` (que le développeur voit comme une fonction sans argument) et la procédure `set_item`. Cette classe est paramétrée par un type G, qui est le type des objets faiblement référencés, ce qui permet de rester fortement typé.

Il faut bien comprendre que ce n'est pas une classe factice, écrite pour présenter de manière synthétique le fonctionnement des références faibles en Eiffel, mais d'une véritable classe qui est compilée par SmartEiffel et qui détermine la représentation en mémoire de la référence faible.

```
class WEAK_REF[G]
--
-- Weak reference to an object.
-- This kind of reference does not prevent the object from being
-- reclaimed by the garbaged collector (in which case item returns Void).
-- Item makes it possible to get (a strong reference to) the object.
-- Inheriting from this class is prohibited.
-- This class works with compile_to_c, but NOT with compile_to_jvm.
--

creation
   set_item

feature

   item: G;
 -- Return a (strong) reference to the object

   set_item(element: like item) is
```

```
 -- Set the object to be weak referenced
      do
 item := element
      ensure
 item = element
      end

end -- WEAK_REF
```

5.2 Affaiblissement de la référence

La classe, telle qu'elle est présentée, référence fortement `item`, c'est-à-dire qu'elle empêche l'objet référencé d'être libéré. Puisque le ramasse-miettes de SmartEiffel est généré pour chaque programme compilé, je suis intervenu au niveau de la génération de la fonction de marquage. Celle-ci vérifie maintenant si elle est en train de générer la fonction de marquage d'un type `WEAK_REF[G]` et, si c'est le cas, elle ne génère pas l'instruction de marquage de `item`.

5.3 Éventuelle mise à `Void` de la référence

Il reste à mettre `item` à `Void` si l'objet référencé disparaît. Cela se fait logiquement au balayage, une fois que le marquage a identifié tous les objets atteignables. Je suis intervenu dans la routine qui génère la fonction de balayage, détectant également si le type pour lequel on est en train de générer la fonction est un `WEAK_REF[G]`. Deux points ont rendu cette intervention plus délicate que la précédente.

Détermination de l'entête d'objet

Dans l'organisation de la mémoire de SmartEiffel, l'entête d'un objet, qui contient l'information de marquage, est situé après cet objet et non avant [7]. Or, `item` contient un pointeur vers l'objet et non vers l'entête. Pour accéder à celui-ci, la routine de balayage a donc besoin de connaître la taille de l'objet.

Si un type n'a pas de descendant, la taille de l'objet pointé, et donc la position de l'entête par rapport au début de l'objet sont connues à la compilation. Ma modification à la routine de génération de la fonction de balayage exploite cette information et génère un code simple pour accéder à l'entête.

En revanche, si un type a des descendants (de taille variable), la taille de l'objet référencé par `item` n'est pas connue à la compilation. De tels objets comportent toujours un identifiant de type pour résoudre les appels polymorphes, comme décrit dans [ZCC97], et cet identifiant est toujours le premier champ de l'objet. Je génère donc une fonction de balayage qui lit cet identifiant et qui trouve la position de l'entête au moyen d'une table qui relie l'identifiant d'un type à sa taille. Cette table était déjà générée et exploitée par le débugger sedb de SmartEiffel mais je l'ai complétée et rendu sa génération plus efficace pour qu'elle puisse être utilisée dans du code de production.

[7] Sauf dans le cas des objets redimensionnables, qui sont expansés et ne nous concernent donc pas.

Détermination du statut d'un objet

S'il est possible d'accéder à la marque de l'objet, la signification de celle-ci n'est pas immédiate. Il faut en effet savoir qu'elle partage sa mémoire avec un pointeur servant à chaîner les emplacements mémoires libres. Ce champ peut donc soit contenir un pointeur, soit une des deux valeurs spéciales `FSOH_MARKED` et `FSOH_UNMARKED`, choisies pour ne pas représenter de pointeur valide, et dont la signification est la suivante à l'issue de la phase de marquage :

`FSOH_MARKED` signale un objet qui a été déterminé comme accessible

`FSOH_UNMARKED` signale un objet qui était accessible, et qui a été détecté comme inaccessible par la phase de marquage qui vient de se terminer.

Or, au cours du balayage, le ramasse-miettes accomplit deux tâches différentes :
- Il insère dans des listes d'objets libres les objets qui étaient déjà libres avant le ramassage de miettes, dont la marque contient donc un pointeur, et ceux nouvellement libérés, dont la marque contient donc la valeur spéciale `FSOH_UNMARKED`. Après le balayage, la marque d'un objet libre contient donc un pointeur, constitutif de la liste d'emplacements mémoire libres.
- Il positionne à `FSOH_UNMARKED` les objets toujours accessibles, pour préparer le travail du cycle de ramassage suivant.

En résumant, si un entête contient un pointeur, l'objet est forcément inaccessible, et s'il contient `FSOH_MARKED`, il est accessible, mais si l'entête contient `FSOH_UNMARKED`, il faut savoir si l'objet a déjà été balayé, auquel cas il est accessible, ou s'il ne l'a pas encore été et est donc inaccessible. Or, il n'est pas possible d'assurer que les références faibles seront balayées avant, ou après tous les autres, à moins de procéder à deux balayages et de dégrader les performances. En revanche, une modification subtile du ramasse-miettes à permis de garantir que les objets sont balayés par adresses croissantes. Pour savoir si l'objet désigné par `item` a déjà été balayé, il suffit alors à la fonction de balayage de comparer l'adresse de l'objet pointé par `item` à celle de la référence faible en cours d'examen. Munie de cette information et du marquage de l'objet, elle peut alors déterminer si celui-ci est en cours de destruction, et donc si `item` doit être positionné à `Void`.

5.4 Autres interventions

J'ai ajouté des contrôles de sûreté dans le compilateur pour produire un message d'erreur si le développeur essaye d'hériter de `WEAK_REF[G]` ou de déclarer des référence faible d'objets expansés

Chapitre 6

Conclusion et travaux futurs

J'ai apprécié ce stage qui m'a permis d'approfondir mes connaissance théoriques en compilation et en de gestion automatique de la mémoire. Le fait que j'aie pu pratiquer mes recherches dans le cadre du langage à objets moderne qu'est Eiffel, et que l'implantation se soit effectuée dans le compilateur libre SmartEiffel n'ont pas non plus été pour me déplaire. Enfin, j'aimerais remercier le LORIA, et plus particulièrement l'équipe DESIGN qui m'a accueilli, pour l'ambiance dans laquelle j'ai effectué ce stage.

Ce travail ouvre sur trois axes de développement. Il sera intéressant d'apporter l'équivalent des *soft-references* Java au langage Eiffel pour rendre moins aléatoire le fonctionnement des caches mémoire. Il faudra également trouver un moyen de rendre les références faibles aussi économiques en mémoire que si elles étaient expansées, sans sarifier leur souplesse d'utilisation. Cette solution passera peut-être par le troisème axe de travail, à savoir l'intégration des références faibles dans le langage Eiffel sous la forme d'un mot-clef.

Annexe A

Quelques types de ramasse-miettes

Depuis l'implantation en 1960 du premier ramasse-miettes par McCarthy, de nombreux algorithmes ou variantes ont été découverts, et il existe sur le sujet une littérature abondante. Je conseille au lecteur intéressé l'ouvrage de Richard Jones [Jon96] qui fait une synthèse du savoir dans ce domaine.

A.1 Comptage de références

Dans un système à comptage de références, chaque objet possède un champ caché, à savoir un entier qui compte le nombre de références vers l'objet. Quand une nouvelle référence vers cet objet est créée le compteur est incrémenté, et quand une référence vers l'objet disparaît, le compteur est décrémenté. Quand le compteur arrive à zéro, l'objet est inatteignable et la mémoire peut être libérée.

Cette technique présente l'avantage de libérer la mémoire au plus tôt, de répartir le surcoût liée à la gestion de la mémoire, et d'être facile à implanter. En revanche, comme chaque changement de référence nécessite la manipulation d'un, et souvent de deux compteurs, le surcoût global en temps et en mémoire est important. De plus, cette technique est incapable de gérer les références circulaires : on peut facilement vérifier que, quand la dernière référence vers un cycle d'objets se référençant est détruite, tous les membres du cycle gardent leur compteur de références à 1, et sont donc considérés comme atteignables.

A.2 Marquage-balayage

Historiquement, la première technique à résoudre le problème des cycles de références fut l'algorithme dit de marquage-balayage. Ce type de ramasse-miettes laisse les objets inaccessibles s'accumuler pendant un certain temps. Quand il détermine qu'un nettoyage est nécessaire — typiquement, si la mémoire totale consommée par le programme dépasse un certain seuil — le ramasse-miettes démarre une phase de *marquage récursif* de tous les objets atteignables. Il part pour cela de certains objets que l'on sait être atteignables, tels que l'objet racine du programme et les variables locales des routines en cours, puis marque les objets référencés par ceux-ci et ainsi de suite. Une fois le marquage terminé commence la phase de *balayage* : tous les objets en mémoire sont examinés tour à tour ; ceux qui ne sont pas marqués sont libérés.

Cette technique gère correctement les cycles et elle consomme moins de temps que le comptage de références. En revanche, le ralentissement n'est pas distribué tout au long de l'exécution, mais au contraire concentré dans les phases de ramassage de miettes où le programme ne répond pas, caractéristique indésirable par exemple dans le cas de systèmes en temps réel. Ce point a motivé le développement des ramasse-miettes incrémentaux et des ramasse-miettes à execution concurrente (voir [Jon96]).

Le fait que la mémoire d'un objet ne soit pas libérée dès que celui-ci devient inaccessible n'est généralement pas gênant ; c'est même cette particularité qui permet, comme on le voit au paragraphe 3.2 (p.9), de substituer dans une certaine mesure des *weak-references* à des *soft-references*.

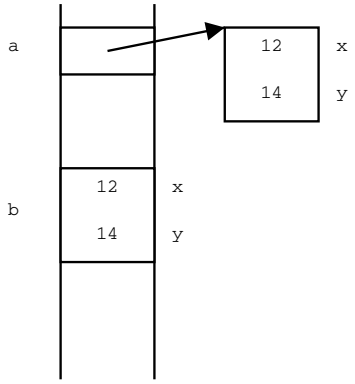
A.3 Marquage conservatif ou non

Dans la discussion qui précède, nous avons éludé la question de la détection des références que l'on va suivre lors du marquage. Dans un système au typage suffisamment fort, ce point ne pose pas problème : le type de chaque objet est exactement connu, et on sait donc à quelles positions dans l'objet se trouvent les références. On parle alors de *marquage non conservatif* (ou *type-accurate*). Dans d'autres cas, il n'est pas possible de dire avec certitude si un certain emplacement mémoire contient réellement une référence ou, par exemple, un entier. Le ramasse-miettes fait alors l'examen de la valeur de la référence potentielle dans le but de nier l'hypothèse qu'il s'agit effectivement d'une référence. S'il n'y parvient pas, il doit alors traiter effectivement la valeur comme une référence et marquer comme atteignable l'objet désigné. Cette technique, qui porte le nom de *marquage conservatif*, ne libère jamais un objet à tort ; en revanche, elle peut conserver des objets inutilement.

Le typage du langage Eiffel est suffisamment fort pour permettre un marquage non conservatif. Cependant, le compilateur SmartEiffel s'appuie sur un compilateur C sous-jacent pour la gestion des variables locales, et il ne lui est pas possible de savoir quel type de variable est stocké dans un emplacement de la pile ou dans un registre du processeur donné. C'est pourquoi le marquage contient une phase conservative à partir des valeurs trouvées en pile et dans les registres du processeur. En revanche, le marquage à partir des objets du tas se fait de manière non conservative. On parle donc de *marquage semi-conservatif*.

Annexe B

Objets *expanded* en Eiffel



La plupart des objets Eiffel sont gérés par référence, ce qui n'est pas très remarquable : en Java, par exemple, tous les objets sont gérés de cette manière.

En revanche, l'autre type d'objet, l'objet expansé, mérite une explication. Le schéma ci-dessus illustre la différence entre deux objets a et b, tous les deux stockés en pile.

A est une référence d'objet : il contient un pointeur vers les données effectives, ici un objet avec deux champs entiers x et y qui se trouvent avoir les valeurs 12 et 14.

B est un objet expansé : aucun pointeur n'intervient ici, la pile contient directement les champs de l'objet, à savoir les deux entiers x et y.

De la même variable, un objet peut être expansé à l'interieur d'un autre objet, ou bien l'autre objet peut détenir une référence vers le premier.

Bibliographie

[AG00] Ole Agesen and Alex Garthwaite. Efficient object sampling via weak references. In *ACM SIGPLAN International Symposium on Memory Management*, pages 121–126, 2000.

[CCZ98] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler suppoert to customize the mark and sweep algorithm. In *ACM SIGPLAN International Symposium on Memory Management*, pages 154–165, October 1998.

[dol] Object arts. http ://www.object-arts.com/.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[Hay97] Barry Hayes. Ephemerons : a new finalization mechanism. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, pages 176–183, 1997.

[jav] Class weakhashmap. http ://java.sun.com/j2se/1.4.2/docs/api/java/util/WeakHashMap.html.

[JME99] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager : Weak pointers and stable names in haskell. In *Implementation of Functional Languages*, pages 37–58, 1999.

[Jon96] Richard Jones. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.

[Mey92] Bertrand Meyer. *Eiffel : the language*. Prentice Hall, New York, NY, 1 edition, 1992.

[vB03] Harry von Borstel. Avoiding circular references : Weakreference in vb-classic. http ://www.programmersheaven.com/articles/harry/article1.htm, January 2003.

[ZCC97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables. the smalleiffel compiler. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, pages 125–141, October 1997.