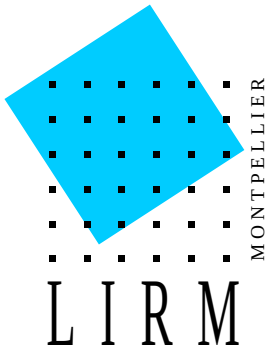

LABORATOIRE D'INFORMATIQUE,
DE ROBOTIQUE ET DE MICROÉLECTRONIQUE
DE MONTPELLIER

Unité Mixte CNRS – Université Montpellier II
C 55060

RAPPORT DE RECHERCHE

Compilation séparée de l'envoi de message dans les langages à typage statique

Roland Ducournau

4 décembre 2001, révisé 11 juillet 2002 R.R.LIRMM 2001–014

161, Rue Ada – 34392 Montpellier Cedex 5 – France
Tél: (33) 04 67 41 85 85 – Fax: (33) 04 67 41 85 00

Résumé

Dans les langages objets à typage statique, l'envoi de message, ou liaison tardive, s'implémente en général par des tables, appelées tables de fonctions virtuelles en C++, qui permettent de réduire l'envoi de message à un simple appel de fonction, modulo un nombre limité d'indirections supplémentaires. Suivant que l'héritage et le sous-typage sont simples ou multiples, ces tables sont plus ou moins complexes et le surcoût qu'elles apportent est plus ou moins prononcé.

Dans ce travail, nous examinons les différents schémas d'implémentation existants et suggérons quelques pistes pour les améliorer.

La compilation globale ayant manifestement de nombreux atouts en terme d'efficacité du code obtenu, nous proposons divers compromis entre la compilation séparée à la C++ et la compilation globale à la SMALL EIFFEL.

Mots-clés : langage à objets, héritage, compilation, envoi de message, typage statique, généricité, contravariance.

Table des matières

1 Introduction **4**
- 1.1 Les mécanismes considérés . . . 4
- 1.2 Efficacité spatiale . . . 5
- 1.3 Efficacité temporelle et architecture des processeurs . . . 6
- 1.4 Plan . . . 6

2 En héritage et sous-typage simples **8**
- 2.1 Le principe . . . 8
- 2.2 Le *casting* . . . 9
- 2.3 Appel à `super` . . . 12
- 2.4 Pointeurs sur `null` . . . 13
- 2.5 Variables de classes et héritage non monotone . . . 13
- 2.6 Evaluation . . . 14

3 En héritage multiple **16**
- 3.1 Le problème de l'héritage multiple . . . 16
- 3.2 Principe d'implémentation . . . 16
- 3.3 Le *casting* . . . 18
- 3.4 Redéfinition de types . . . 21
- 3.5 Appel à `super` et `call-next-method` . . . 25
- 3.6 Pointeurs sur `null` . . . 26
- 3.7 Variables de classes et héritage non monotone . . . 26
- 3.8 Evaluation . . . 26

4 Alternatives pour l'héritage multiple **28**
- 4.1 Variante avec partage des tables . . . 28
- 4.2 L'héritage multiple non virtuel de C++ . . . 31
- 4.3 Réduire les indirections . . . 33
- 4.4 Compilation des attributs par des accesseurs . . . 34
- 4.5 Comparaisons . . . 36

5 En héritage simple et sous-typage multiple **37**
- 5.1 Les spécifications . . . 37
- 5.2 Variante de l'héritage multiple . . . 37
- 5.3 Variante du sous-typage simple . . . 42
- 5.4 Application à JAVA. . . . 45
- 5.5 Application à l'héritage multiple : les *mixins* . . . 46
- 5.6 Evaluation . . . 47

6 La généricité **49**
- 6.1 En sous-typage simple et généricité bornée . . . 49
- 6.2 En héritage simple et sous-typage multiple . . . 49
- 6.3 En héritage multiple . . . 51

7 Comparaison avec une compilation globale **53**
7.1 Compilation globale 53
7.2 Heuristique de coloration 58
7.3 Intermédiaires entre compilations globale et séparée 63

8 Conclusion **70**

A Éléments d'implémentation **72**
A.1 Structure d'indexation dynamique des super-classes 72
A.2 Ordonner et numéroter les classes 72
A.3 Codage d'ordre 73
A.4 *Garbage collector* 73

B Analyse de types séparée **76**

Table des figures

1.1 Les paramètres de l'efficacité 5
1.2 Héritage et sous-typage 7
1.3 Les quatre catégories de langages considérés 7

2.1 Structure des objets et des tables de méthodes en héritage simple 9

3.1 Tables des attributs et des méthodes en héritage multiple 17
3.2 Tables des méthodes pour l'exemple 18
3.3 Les types en jeu sur le receveur 19
3.4 *Casting* ascendant, descendant et latéral 20
3.5 Tables de *casting* pour la classe (τ_d) D 21
3.6 Redéfinition covariante de la valeur de retour 22
3.7 Passage de paramètres covariants 23

4.1 Tables de méthodes avec partage 29
4.2 Tables des attributs et des méthodes en héritage multiple « non virtuel » 31
4.3 Tables des méthodes, avec partage de tables ou en héritage non virtuel 33

5.1 Sous-typage multiple et héritage simple : variante de l'héritage multiple 38
5.2 Les différents cas de *casting* en héritage simple et sous-typage multiple 39
5.3 Table des méthodes en THETA pour l'exemple 40
5.4 La technique de THETA en héritage multiple 41
5.5 Sous-typage multiple et héritage simple : variante du sous-typage simple, avec tables de conversion (variante 1). 42
5.6 Sous-typage multiple et héritage simple : variante du sous-typage simple, avec pointeurs dans la table de méthodes (variante 4). 43
5.7 Sous-typage multiple et héritage simple : variante du sous-typage simple, avec cache en ligne. 44

6.1 Table d'indirection des méthodes en sous-typage multiple pour les types paramétrés 50
6.2 Spécialisation de types paramétrés 51

7.1 L'heuristique de coloration appliqué à l'exemple 58
7.2 Tables des méthodes nécessaires pour la coloration unidirectionnelle. 60
7.3 Schémas de compilation 64
7.4 Comparaisons des diverses techniques 68
7.5 Comparaisons : nombre de cycles et d'instructions 69

A.1 Schéma d'implémentation uniforme des objets, avec *garbage collector*. 74

Chapitre 1

Introduction

Dans les langages objets à typage statique, l'envoi de message, ou liaison tardive, s'implémente en général par des tables, appelées tables de fonctions virtuelles en C++, qui permettent de réduire l'envoi de message à un simple appel de fonction, modulo un nombre limité d'indirections supplémentaires. Suivant que l'héritage et le sous-typage sont simples ou multiples, ces tables sont plus ou moins complexes et le surcoût qu'elles apportent est plus ou moins prononcé. L'objectif de cet article est de décrire et d'évaluer les principales techniques utilisées.

L'efficacité de l'implémentation d'un langage s'évalue par deux catégories de critères. L'efficacité temporelle peut être jugée en moyenne, mais l'idéal est que les différents mécanismes conceptuellement unitaires se fassent en un temps constant faible. L'efficacité spatiale s'évalue par l'occupation mémoire globale nécessaire à l'implémentation. Enfin, il n'est pas inutile de penser à l'efficacité du compilateur lui-même : les optimisations NP-difficiles sont à éviter.

L'implémentation d'un langage se juge aussi dans le contexte « politique » de sa chaîne d'exécution, interprétation ou compilation — nous ne nous intéresserons ici qu'à cette dernière —, compilation globale ou séparée, édition de liens statique ou dynamique.

1.1 Les mécanismes considérés

Dans le contexte particulier des langages objets, sont concernés les deux mécanismes fondamentaux
- d'accès aux attributs d'un objet, en lecture et en écriture ;
- d'envoi de message, c'est-à-dire de sélection et d'appel de la méthode correspondant au type dynamique du receveur (liaison tardive).

A ces deux mécanismes fondamentaux, s'ajoutent divers mécanismes secondaires qui sont tous indispensables dans n'importe quel langage à objets et dont l'implémentation n'est pas toujours aussi triviale qu'il n'y paraît :
- le polymorphisme (ou *casting* ascendant) doit être pris en compte, de telle sorte que toute entité de type statique donné pointe sur la « bonne » valeur : `self`[1] n'est pas forcément invariant, sa valeur peut dépendre de son type statique ;
- la vérification dynamique de type, nécessaire pour les constructions comme le *casting* descendant ou `typecase`, doit être implémentée efficacement, idéalement en temps constant ;
- l'implémentation doit en particulier gérer la redéfinition des types dans les sous-classes, qu'il s'agisse du type de retour, des paramètres ou des attributs, et que cette redéfinition soit sûre du point de vue des types, ou pas ;
- des variables (attributs) de classes, accessibles à partir du type dynamique des objets, comme le mot-clé `:allocation` de CLOS, dont les variables `static` de C++ et JAVA donnent une pauvre

[1] `Self` désigne classiquement le receveur du message : c'est le mot réservé en SMALLTALK pour cet usage, l'équivalent de `this` en C++ ou JAVA et de `current` en EIFFEL. On appelle `self` le receveur du message, mais c'est déjà du passé : lorsqu'il apparaît, c'est déjà lui l'envoyeur. Le type statique d'une occurrence de `self` est la classe courante, c'est-à-dire la classe dans une méthode de laquelle figure cette occurrence.

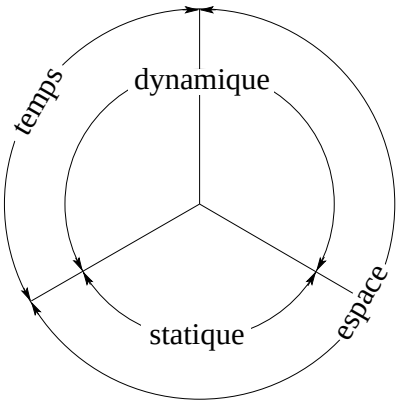

FIG. 1.1 – Les paramètres de l'efficacité

simulation ;
- l'appel à `super` qui permet à une méthode d'appeler celle qu'elle redéfinit : ce mécanisme permet en particulier d'implémenter des pseudo-méthodes comme les constructeurs et destructeurs de C++ ;
- le traitement de la valeur `null` pour les variables ou attributs non instanciés ;
- les classes paramétrées doivent enfin être compilées comme une extension efficace de l'implémentation des classes.

Seule la sélection sur un receveur unique est prise en compte ici : la sélection multiple — telle qu'elle est réalisée en CLOS, CECIL, DYLAN ou CLAIRE, théorisée par [Castagna, 1997], ou généralisée par [Ernst *et al.*, 1998] dans la sélection par prédicat (*predicate dispatch*) — relève de techniques assez similaires mais la combinatoire provoquée par la multiplicité (produit cartésien des types) est telle que l'évaluation des différentes techniques doit être reprise à zéro.

On se place, pour l'essentiel, dans un cadre de typage orthodoxe, respectant la règle de contravariance mais on examinera, au passage, les contraintes supplémentaires résultant d'une approche moins orthodoxe [Castagna, 1995 ; Ducournau, 2001b]. Le problème de la vérification, par le compilateur, de la correction du typage ne sera pas abordé ici. En revanche, la question d'une vérification dynamique de type, cruciale avec le *casting* descendant, aussi bien qu'avec la politique de covariance d'EIFFEL sera examinée.

Par ailleurs, on ne s'occupe pas ici, à proprement parler, de savoir quelle méthode sélectionner mais uniquement de l'implémentation efficace de cette sélection : ainsi, la surcharge statique [Ducournau, 2001b] est censée avoir été résolue, de même que la question de l'héritage [Ducournau *et al.*, 1995].

Nous ne considérerons pas plus les différents mécanismes d'encapsulation, de protection ou d'exportation, tels qu'ils s'expriment en C++, JAVA ou EIFFEL, car ils ont un effet marginal sur l'implémentation elle-même, puisqu'on peut les réduire à un droit d'accès à une implémentation existante. Notre seule exception concernera l'encapsulation à la SMALLTALK, qui réserve à `self` le droit d'accès aux attributs, ce qui peut avoir un effet non négligeable sur leur implémentation [Goldberg et Robson, 1983].

Ne sera pas considérée, non plus, la question de l'implémentation des types primitifs, en particulier le problème — s'il existe en typage statique — de l'envoi de message à une entité dont le type peut être aussi bien primitif que construit.

1.2 Efficacité spatiale

Sur le plan spatial, on peut distinguer trois catégories d'occupation mémoire qu'il faut prendre en compte :
- une part complètement dynamique consiste en l'implémentation des objets eux-mêmes, en général comme une table de leurs attributs, pointant sur les structures de données associées à leur classe ;
- une part statique, représentée par des structures de données associées aux classes, en lecture seule ;
- la part du code proprement dit, dans lequel la mise en œuvre d'un mécanisme peut occuper une séquence d'instructions plus ou moins longue.

Dans le même ordre d'idées, il faut aussi s'intéresser à la gestion automatique de la mémoire : l'implémentation des objets doit rester compatible avec un *garbage collector* efficace.

L'efficacité temporelle et l'efficacité spatiale varient, en général, en sens inverse : un critère unique est donc impossible et le choix résultera toujours d'un compromis qui pourra dépendre du contexte. Si une analyse de la mémoire statique a été menée dans le cadre des techniques globales qui nécessitent un compactage de tables [Ducournau, 1997], l'efficacité spatiale a été relativement peu analysée. Quant à la mémoire dynamique, elle ne semblait pas poser beaucoup de problèmes : les techniques standard (cf. section 3) nécessitent un surcoût faible et la coloration (cf. section 7.2) est la seule technique « classique » qui puisse provoquer un surcoût non négligeable.

Mais l'attrait de l'efficacité temporelle pousse régulièrement à sacrifier l'efficacité spatiale : il est clair qu'une façon d'améliorer significativement l'efficacité temporelle (sans considération du reste) consiste à implémenter tout ou partie des structures de classes dans les instances, ce qui réduit d'autant les indirections [Eckel et Gil, 2000]. Nous laisserons ces techniques de côté, en nous donnant comme objectif de conserver une occupation mémoire dynamique minimale, ou plutôt un « surcoût » minime.

1.3 Efficacité temporelle et architecture des processeurs

Traditionnellement, l'une des mesures de l'efficacité temporelle d'un mécanisme est constituée par la longueur de la séquence d'instructions machine qui l'implémentent. Les processeurs modernes ont rendu cette mesure obsolète car ils offrent à la fois une dose limitée de paralléllisme (processeurs super-scalaires), une architecture de *pipe-line* et des capacités variables de prédiction sur les branchements. En contrepartie, les latences nécessaires pour les accès mémoires ou les branchements sont de plusieurs cycles. La longueur de la séquence d'instructions n'est donc plus une mesure temporelle fiable : elle ne sert guère qu'à mesurer l'espace mémoire occupé par un appel. Le coût de la composition des mécanismes suivrait donc plutôt une loi du maximum que de la somme.

L'implémentation de l'envoi de message par des tables assurant un accès direct a longtemps été considéré comme optimale. Les processeurs avec prédiction de branchements conditionnels ont mis aux premiers rangs la technique, pourtant assez rudimentaire, du cache en ligne, qui est basée sur une comparaison entre le type du receveur et un type attendu : un tel test, statistiquement bien prédictible pour les programmes et bien prédit par le processeur, rend cette technique très efficace [Driesen *et al.*, 1995 ; Zendra *et al.*, 1997]. Les techniques de tables pouvaient donc être considérées comme dépassées. Mais, après la prédiction des branchements conditionnels, les processeurs se mettent actuellement à la prédiction des branchements indirects [Driesen, 1999]. Il y a donc tout lieu de penser que le handicap des tables sera bientôt comblé.

En tout état de cause, notre appréciation de l'efficacité des différentes techniques ne dépassera pas un niveau très intuitif, basé éventuellement sur un pseudo-code. Le lecteur pourra se reporter à [Driesen, 1999] pour considérer l'influence de l'architecture du processeur.

1.4 Plan

Cet article vise à faire une synthèse des implémentations existantes, qu'elles soient implémentées ou simplement décrites dans la littérature. Une difficulté se présente dès le départ : de nombreuses implémentations ne sont pas décrites — pour des raisons de confidentialité ou par manque supposé d'originalité —, et il est nécessaire d'extrapoler à partir de ce qui en est dit dans la littérature, ou de la propre expérience de l'auteur. Inversement, de nombreux schémas d'implémentation, voire des principes de compilation, sont décrits dans la littérature sans avoir probablement vu la moindre implémentation. Les spécifications des langages permettraient bien d'écarter certaines hypothèses d'implémentations, qui seraient incompatibles. Malheureusement, les spécifications décrites ne sont pas toujours implémentées en totalité. Les schémas que nous décrivons sont donc plus probables que véritables, mais le principe des implémentations effectives ne doit pas s'en éloigner beaucoup. Il n'est pas exclu, quoique peu probable, que certaines solutions soient, au moins marginalement, originales.

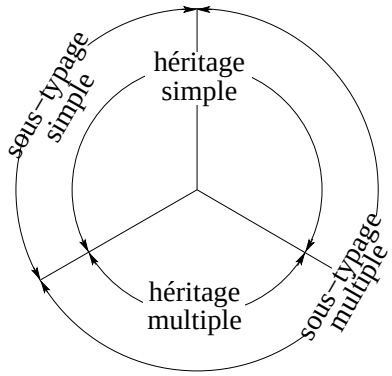

FIG. 1.2 – Héritage et sous-typage

héritage	sous-typage simple	sous-typage multiple
simple	SIMULA, C++ (hs)	JAVA, THETA, EIFFEL#
arborescent	—	C++ (non virtuel)
multiple	—	C++ (virtuel), EIFFEL

FIG. 1.3 – Les quatre catégories de langages considérés

On peut classer les langages à typage statique suivant que l'héritage et le sous-typage y sont simples ou multiples et, dans ce dernier cas, suivant que l'héritage est arborescent ou quelconque (figure 1.3). Les quatre catégories résultantes font appel à des techniques différentes.

Les 2 sections suivantes présentent successivement le principe de l'implémentation standard dans les deux cas extrêmes : en sous-typage simple puis en héritage multiple. Les différentes notions sont introduites à l'occasion de ce premier exposé. Dans la section 4, nous étudions ensuite quelques alternatives à l'implémentation standard de l'héritage multiple, dont l'héritage « non virtuel » de C++. La section suivante présente alors le cas intermédiaire, illustré par JAVA, où l'héritage est simple mais le sous-typage multiple : des applications à l'héritage multiple sont examinées, en particulier pour l'implémentation des *mixins*. La section 6 présente le principe de l'implémentation des classes paramétrées. Les 5 sections précédentes considéraient le problème du point de vue d'une compilation purement séparée : la section 7 propose alors une comparaison avec les techniques utilisées en compilation globale, principalement dans un cadre de typage dynamique, et propose diverses stratégies combinant des phases séparées et globales. La coloration est décrite dans une variante bidirectionnelle, et son application aux classes, méthodes et attributs est considérée : nous proposons de s'en servir à l'édition de liens pour pouvoir l'utiliser dans un cadre de compilation séparée. L'article se termine par une conclusion ouvrant quelques perspectives de recherche.

Des annexes complètent l'article. La première décrit quelques éléments d'implémentation liés à la structure d'ordre des hiérarchies de classes et de types, ainsi qu'à la gestion mémoire. La suivante récapitule les différentes implémentations à la lumière de l'architecture des processeurs. La troisième annexe donne le principaux résultats concernant la coloration. Une dernière annexe présente différentes statistiques (cette annexe est repoussée dans un second rapport de recherches [Ducournau, 2001a].

Cet article a pour objet l'implémentation de langages : il ne prétend surtout pas justifier, par leur implémentation ou leur implémentabilité, les différents traits de ces langages qui seront examinés.

Chapitre 2

En héritage et sous-typage simples

2.1 Le principe

En héritage et sous-typage simples, un objet est implémenté comme une table de ses attributs, auxquels s'ajoute un pointeur sur une table de méthodes commune à toutes les instances d'une même classe (figure 2.1). La première table contient, pour chaque objet, la valeur de ses attributs. La seconde table contient les adresses des méthodes. L'implémentation des objets et de l'envoi de message se caractérise par les invariants suivants.

Invariant 2.1 *La valeur d'une référence — paramètre, variable, attribut ou valeur de retour d'un appel fonctionnel — sur un objet est invariante relativement à son type statique.*

Invariant 2.2 *Chaque attribut ou méthode a un indice non ambigu et invariant par héritage, donc indépendant du type statique du receveur.*

Le sous-typage simple se caractérise ainsi par une invariance absolue vis-à-vis des types statiques, qui n'ont aucun rôle à l'exécution. L'implémentation se conforme donc à la sémantique fondamentale de l'approche objet, pour laquelle le type dynamique exprime l'essence de l'objet, le type statique étant purement contingent. On notera par la suite τ_s et τ_d les types statique et dynamique d'une entité, avec $\tau_d \leq \tau_s$: le type statique est celui qui annote explicitement une entité d'un programme[1], alors que le type dynamique est la classe dont l'instanciation a créé l'objet qui value l'entité considérée.

L'envoi de message se compile alors par une séquence de trois instructions[2] :

```
load [object + #tableOffset], table
load [table + #selectorOffset], method                    2L + B
call method
```

et l'accès à un attribut est immédiat :

```
load [object + #attributeOffset], attribute                    L
```

Le calcul des tables de méthodes (pour la classe) et d'attributs (pour les instances) est un cas particulier de l'heuristique de coloration décrite dans [Ducournau, 1997] (cf. section 7.2). L'algorithme est le suivant : pour chaque classe dont la super-classe a déjà été compilée, on numérote les attributs (resp. méthodes) introduits[3] dans la classe, en partant de l'indice maximum des attributs (resp. méthodes) de la super-classe. L'absence d'héritage multiple et de surcharge (au sens de l'article précité, cf. note 11, page 58) garantit la correction du résultat (c'est-à-dire qu'il n'y a jamais aucun problème avec les indices des méthodes héritées et qu'il n'est jamais nécessaire de vérifier que l'indice que l'on veut affecter n'est pas déjà occupé).

[1] Ou qui peut s'en déduire, assez directement, par exemple, le type de retour d'une méthode, en cas de redéfinition covariante.

[2] Le pseudo-code qui illustre les différentes techniques est emprunté à [Driesen, 1999] : s'il est représentatif du nombre d'instructions nécessaires, il ne l'est pas du nombre de cycles de processeur. Chaque séquence est accompagnée d'une estimation du nombre de cycles nécessaires pour son exécution : L représente la latence de chargement et vaut 2 ou 3, et B la latence de branchement indirect et peut valoir de 3 à 15 (cf. annexe **??**).

[3] On utilisera le verbe « introduire » pour désigner la définition, dans une classe, d'une propriété (attribut ou méthode) non définie dans ses super-classes.

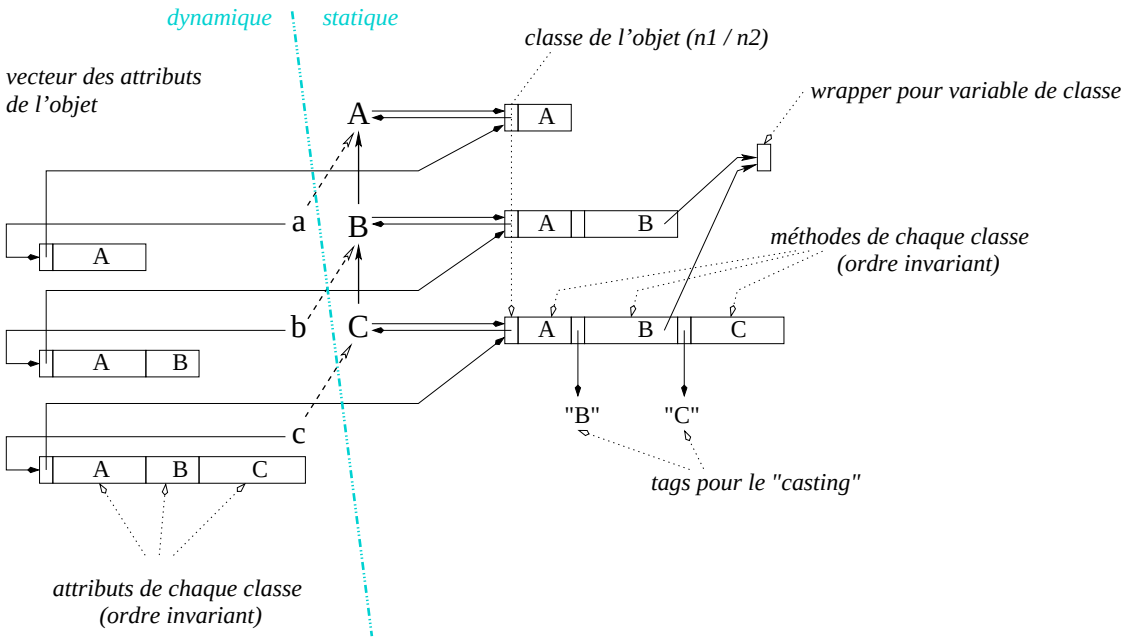

FIG. 2.1 – Structure des objets et des tables de méthodes en héritage simple : 3 classes A, B et C et leurs instances respectives, a, b et c.

C'est l'implémentation de base de la plupart des langages à objets, aussi bien celle de C++, tant qu'on reste en héritage simple et « non virtuel » (sans usage du mot-clé `virtual` pour l'héritage, cf. section 4.2), que celle de JAVA en ne se servant pas des interfaces. La figure 2.1 résume cette implémentation.

2.2 Le *casting*

2.2.1 La notion de *casting*

Le terme de *casting* est utilisé — essentiellement en C et C++, puis, par mimétisme de façon plus générale, en programmation par objets — pour désigner différentes opérations, plus ou moins bien définies conceptuellement, qui ont toutes un rapport à la conversion d'une valeur d'un type (la source) dans un autre (la cible).

Parmi les nombreuses opérations qui peuvent être concernées par ce terme de *casting*, on peut distinguer, de façon probablement non exhaustive :

- la réinterprétation d'une zone mémoire par une structure d'un autre type que celui qui l'a allouée ou intialisée : ce style de programmation — courant dans les langages, comme C ou PL/1, qui permettent l'arithmétique de pointeurs ou dont les pointeurs ne sont pas typés — enfreint tous les canons de la programmation évoluée et du typage orthodoxe et n'a rien à voir avec la programmation objet ;
- la conversion ou coercition entre types voisins, qui agit par copie : typiquement, la conversion entre représentations différentes des nombres (simple ou double précision, etc.) ; la levée du caractère constant d'une donnée (`const_cast` en C++) est du même ordre[4] ;
- la migration d'instance, telle qu'elle est spécifiée en CLOS par la fonction générique `change-class`, qui agit comme un effet de bord sur l'objet concerné qui devient instance d'une autre classe, tout en conservant son « identité » ;
- la classification d'instances est un cas particulier du précédent, dans lequel la classe cible est contrainte

[4] Si l'on considère comme absolu le caractère constant. S'il n'est que relatif et que l'on s'autorise à l'enfreindre localement, le *casting* ne constitue qu'une indication au compilateur d'autoriser un usage qui serait, sinon, interdit.

à être une sous-classe de la classe source : l'objet reste donc instance de la classe source, mais indirectement ; bien que cette fonctionnalité soit étrangère au monde de la programmation, [Ducournau et Pavillet, 2001] montre qu'elle n'est pas incompatible ;
- les *state classes* de [Drossopoulou *et al.*, 2001] et les *predicate classes* de [Chambers, 1993] sont des variantes ou des cas particuliers des deux précédents ;
- le *casting* ascendant, souvent qualifié d'implicite parce qu'il ne nécessite aucun mécanisme syntaxique, correspond à l'affectation (ou au passage de paramètre) polymorphe, lorsque l'on affecte à une entité x de type statique X la valeur d'une entité y d'un type statique Y sous-type de X (noté $Y \prec X$) : ce soit-disant *casting* ascendant n'a aucune existence conceptuelle — c'est l'essence même du polymorphisme (dit d'inclusion) — mais on verra qu'il peut néanmoins nécessiter une implémentation non triviale ;
- le *casting* descendant (ou *downcast*) nécessite une vérification dynamique de type — donc une levée potentielle d'exception — et revient à faire l'hypothèse qu'une entité de type statique X est en fait d'un sous-type Y : ce *casting* peut être effectué au travers d'une affectation ou d'un passage de paramètre, ou par une construction comme `typecase`[5] dans le langage THETA [Liskov *et al.*, 1995] ou les tentatives d'affectation (*assignment attempts*) en EIFFEL [Meyer, 1992 ; Meyer, 1997] ; l'usage du *casting* descendant se justifie souvent par le fait que les modèles naturellement covariants sont implémentés dans des langages contravariants [Ducournau, 2001b].

On voit donc que le même terme s'applique, plus ou moins bien, à des notions très différentes, conceptuellement aussi bien qu'opérationnellement, puisqu'elles mettent en jeu des réinterprétations de zone mémoire, des copies, des effets de bord ou de simples déplacements de pointeurs, avec ou sans levée d'exceptions.

Dans cet article, nous ne considérerons le terme de *casting* que sous ses deux variantes ascendantes et descendantes, la première avec une justification purement implémentatoire. L'annexe A.4 fera juste un bref examen de l'implémentation de la migration d'instances.

A ces deux directions et sémantiques différentes du *casting*, on peut enfin rajouter une distinction suivant que le type cible est connu statiquement ou pas. On pourra donc parler aussi de *casting statique* et de *casting dynamique*[6]. Tel qu'il est pratiqué explicitement par un programmeur, le *casting* est toujours statique, mais on verra que la redéfinition de type, peut se réaliser, au choix, par des *castings* statique ou dynamique. Le *casting* dynamique impose que l'identifiant du type cible soit accessible à partir des objets considérés.

2.2.2 Implémentation en sous-typage simple

Casting ascendant

En héritage et sous-typage simples, comme les références à l'objet ne dépendent pas de leur type statique, la notion de *casting* ascendant ne se justifie pas plus du point de vue implémentatoire que du point de vue conceptuel.

Casting descendant

Le *casting* descendant, qui se réduit à une vérification dynamique de type, peut s'implémenter en mettant dans la table des méthodes des étiquettes du nom des super-classes successives : une instance de type statique A est une instance de type dynamique B, si l'objet possède, au bon endroit — par exemple à un indice Δ_B correspondant au nombre de méthodes des super-classes de B — une étiquette "B"[7]. De fait, ces Δ ne dépendent pas du type statique de départ (A) : ils peuvent donc être donnés relativement à l'origine de l'objet. Cette technique est simple, utilisable en compilation complètement séparée et efficace en temps, mais elle l'est moins en espace. C'est encore un cas particulier de l'heuristique de coloration (cf. section 7.2) et il semble, d'après [Raynaud et Thierry, 2001], qu'elle ait été décrite par [Cohen, 1991].

[5] Qui sélectionne la première branche dont l'étiquette est un super-type de τ_d.

[6] Sans que cela ait le moindre rapport avec les mots-clés `static_cast` et `dynamic_cast` de C++, qui sont tous les deux *statiques* (cf. section 4.2).

[7] Dans le cas peu probable où le *tag* pourait être confondu avec le contenu de la table de méthodes, il faudrait en faire une table séparée, ou, les mettre dans les indices négatifs des tables. Tout cela est discuté plus en détail en section 7.2.

La technique optimale, qui est tout aussi simple mais a le défaut de ne pas être généralisable à l'héritage multiple, consiste en une double numérotation des classes, notée n_1 et n_2, obtenue par un parcours en profondeur en incrémentant un compteur à la descente : n_1 est la valeur du compteur affectée à la classe à la descente, après l'incrément du compteur, et n_2 est la valeur du compteur, affectée quand on quitte la classe à la remontée. Cette numérotation est souvent appelée *Schubert's numbering* ou *relative numbering* [Raynaud et Thierry, 2001]. n_2 peut être défini par

$$n_2(C) = \max_{D \preceq C}(n_1(D)) \tag{2.1}$$

Alors :

$$D \prec C \Leftrightarrow n_1(C) < n_1(D) \leq n_2(C) \tag{2.2}$$

Seuls deux petits entiers sont nécessaires, dont l'un (n_1) peut servir à identifier la classe. La numérotation est globale — elle doit donc se faire à l'édition de liens — et n'est pas incrémentale ce qui rend problématique un chargement ou une édition de liens dynamiques[8] (cf. section 7.3.2). On notera que, lors du test (2.2), $n_1(D)$ est dynamique (D est en fait le type dynamique τ_d), alors que $n_1(C)$ et $n_2(C)$ sont statiques et peuvent être compilés comme des constantes de l'exécutable. Pour un surcoût très faible, on pourra mettre l'identifiant de classe n_1 dans l'objet lui-même, ce qui évite une indirection. Le pseudo-code serait le suivant :

```
load [object + #n1Offset], classid
comp classid, #n1
blt #fail                                   L + 2
comp classid, #n2
bgt #fail
// casting réussi
```

2.2.3 Redéfinition de types

Redéfinition covariante du type de retour

Le typage sûr autorise une redéfinition covariante des types de retour des méthodes. Cette redéfinition ne nécessite donc aucune vérification dynamique de type et l'invariance des pointeurs en rend l'implémentation transparente.

Passage de paramètres non invariants

La sûreté du typage impose une redéfinition contravariante des types des paramètres. Le fait est qu'une redéfinition strictement contravariante n'a aucun intérêt puisque les modèles que l'on souhaite implémenter relèvent de la covariance [Ducournau, 2001b]. Cela explique que les langages comme C++ ou JAVA interdisent la redéfinition du type des paramètres (autrement que sous la forme d'une surcharge statique).

Cela étant, la question d'une redéfinition non invariante se pose, ne serait-ce que pour les langages covariants comme EIFFEL : le passage de paramètres se ramène alors à un *casting*, ascendant ou descendant, suivant que la redéfinition est contravariante ou covariante.

En sous-typage et héritage simples, les références à l'objet sont invariantes relativement au type statique : la redéfinition contravariante ne demande donc aucun traitement particulier et la redéfinition covariante nécessite juste une vérification de type. Cette vérification peut se faire, dans la méthode appelée, par la technique décrite précédemment, avec l'inconvénient qu'elle est alors systématique : il n'est plus possible de l'éviter lorsque le type statique de l'argument est un sous-type du type du paramètre dans la méthode du type dynamique du receveur.

Dans un contexte un peu différent (cf. section 5.2.3), [Myers, 1995] propose une technique qui s'applique ici aussi : au lieu d'affecter un indice à chaque sélecteur de méthode, on l'affecte à chaque signature, un peu comme si on voulait implémenter la surcharge statique (qui est bien entendu incompatible avec la redéfinition). Chaque appel de méthode fait appel à l'indice correspondant aux types statiques des arguments de l'appel : si l'entrée correspondant contient l'adresse d'une méthode dont les types de paramètres sont

[8] Encore que l'algorithme soit suffisamment rapide pour autoriser un recalcul global : mais cela annulerait la remarque suivante.

plus spécialisés, l'adresse de la méthode est remplacée dans la table par l'adresse d'un petit bout de code qui fait les vérifications de type et saute à l'adresse de la méthode. Il s'agit là de la technique des *thunks*[9] bien connue en héritage multiple (cf. chapitre 3). Cela revient très exactement à faire autant d'entrées à la méthode qu'il y a de vérification de types à réaliser. La connaissance du type statique de l'argument n'est pas nécessaire[10]. Pour un type dynamique τ_d donné, il y a au plus autant de *thunks* que de types statiques τ_s définissant la méthode[11].

Redéfinition covariante du type des attributs

Le typage sûr est incompatible avec la redéfinition du type des attributs, car il faudrait qu'elle soit covariante en lecture et contravariante en écriture : mais une redéfinition covariante est en général souhaitable [Ducournau, 2001b]. L'accès en lecture s'effectue de façon transparente, mais l'accès en écriture nécessite une vérification de type. Il y a deux façons de réaliser cette vérification : soit l'on recourt systématiquement à des accesseurs en écriture (cf. section 4.4) et l'on traite le problème comme décrit prédédemment, soit l'on fait la vérification de type dans la méthode appelante : c'est alors un *casting* dynamique, puisque le type dépend de τ_d. Il faut donc stocker l'identifiant du type des attributs dans la table des méthodes et le pseudo-code est le suivant :

```
load [object + #tableOffset], table
load [val + #n1Offset], n1source
load [table + #attributen1Offset], n1cible
load [table + #attributen2Offset], n2cible
comp n1cible, n1source
blt #fail
comp n2cible, n1source
bgt #fail
store val, [object + #attributeOffset]
```

Les deux méthodes ont leurs inconvénients : la première impose un appel de méthode pour chaque écriture, mais la vérification n'est faite que si elle est statiquement nécessaire, alors que la seconde impose une vérification systématique. La seconde est sans doute préférable, mais cela peut dépendre très précisément des processeurs : le surcoût du *casting* dynamique est bien inférieur à celui d'un appel de méthode et la deuxième méthode est donc meilleure s'il y a des redéfinitions.

Inférence de types et types ancrés

Lorsque le compilateur peut démontrer que certaines expressions sont sûres, par exemple avec les types ancrés d'EIFFEL, la vérification de type est inutile et le *casting* descendant aussi. De même, tous les attributs qui se partagent la même ancre peuvent se partager l'indice de l'ancre dans la table des méthodes.

2.3 Appel à `super`

Il consiste en l'appel d'une méthode de la super-classe, sur le receveur courant (`self`). Bien que conçue essentiellement pour permettre d'appeler la méthode de même nom qui a été redéfinie dans la classe courante, ce mécanisme permet, autant en SMALLTALK qu'en JAVA, de faire appel à une méthode de n'importe quel nom, ce qui revient à court-circuiter le mécanisme d'envoi de message[12]. En héritage simple, aussi bien `self` que la méthode à appeler sont invariants par rapport au type dynamique du receveur : l'appel à `super` s'effectue donc de façon parfaitement immédiate. L'opérateur :: de C++ est encore plus

[9] L'étymologie de mot est obscure : il ne figure dans aucun de mes dictionnaires. [Myers, 1995] utilise le terme *trampoline* : on pourrait traduire par *tremplin*.

[10] Ce ne serait pas modulaire : précisons que chaque classe n'a un indice que pour les signatures des méthodes définies dans la classe ou ses super-classes.

[11] Au prix d'une plus grande combinatoire, on peut indicer, non pas les signatures des méthodes effectivement définies, mais les combinaisons des types des paramètres de ces méthodes. Dans ce dernier cas, un *thunk* ne vérifie que les paramètres qu'il faut vérifier, alors que dans le premier, il les vérifie tous dès qu'il faut en vérifier un.

[12] En JAVA, `super` est une pseudo variable (comme `self`) dont le type statique est la super-classe de la classe courante.

général, puisqu'il permet une désignation explicite de la méthode d'une classe précise : il s'implémenterait de façon analogue, si C++ était en héritage simple.

Les constructeurs et destructeurs s'implémentent sans problèmes comme des méthodes chaînées par appels successifs à `super`. Le constructeur[13] — qui mérite mal son nom puisque c'est un initialiseur, l'équivalent du `initialize-instance` de CLOS — est appelé sur un objet fraîchement alloué (par un équivalent de `allocate-instance`) et dont seul le pointeur sur la table de méthodes est initialisé.

En BETA, l'appel à `super` est remplacé par l'appel à `inner`, mais c'est toute la sémantique de l'appel de méthode qui est transformée : la sélection de la méthode primaire est statique, seul l'appel de `inner` nécessite une liaison tardive. C'est une façon élégante de supprimer le mot-clé `virtual` de C++ (pour les méthodes), mais la motivation est la même : la sémantique est encore à la remorque de l'efficacité.

2.4 Pointeurs sur `null`

Toute entité typée par un objet nécessite d'être initialisée : cette initialisation n'a rien de trivial pour les variables locales et les attributs. Pour les variables locales, une analyse du flot intraprocédural permet de s'assurer de cette initialisation, mais il n'en est pas de même pour les attributs, dont l'initialisation doit souvent être différée longtemps après la création de l'objet. Une initialisation par un pointeur `null` évite les aléas liés à l'état courant de la mémoire. Bien entendu, il est hors de question de tester le pointeur sur un objet à chaque accès.

[Alpern *et al.*, 1999] décrit une solution qui repose sur le matériel et le système d'exploitation : le pointeur `null` a la valeur 0 et les indices des champs sont tous négatifs. Le système d'exploitation (AIX) génère une interruption lors d'un accès à ces indices négatifs.

Une solution plus logicielle consiste à avoir un objet `null` par type, dont tous les attributs sont initialisés à `null`, et dont la table de méthodes contient des adresses à une même procédure génératrice d'exception. Cet objet peut être alloué dans la zone code, présumée en lecture seule, pour que toute tentative d'affectation déclenche une exception. Le problème des accès en lecture reste entier : `null` se propage de lecture en lecture, mais la première exception levée ne permet pas de savoir où se situe le pointeur `null` initial.

Dans les cas les plus favorables, on peut envisager d'avoir un objet `null` localement invariant par héritage, avec une table de méthodes contenant le maximum de méthodes des classes concernées, voire une unique valeur `null`. Le partage dépend du contenu des tables : adresses de méthodes, identifiants de types d'attributs, etc.

2.5 Variables de classes et héritage non monotone

Nous examinons ici deux traits de langages, dont le premier est rarement spécifié et implémenté correctement, bien qu'il soit aussi sûr que conceptuellement bien défini. Quant au second, il est plus hétérodoxe.

2.5.1 Variables de classes et `:allocation`

La notion de « variable de classes » est délicate car ce terme vague recouvre au moins trois notions distinctes :

- les variables `static` de C++ ou JAVA qui, une fois n'est pas coutume, méritent bien leur nom : elles ne relèvent pas d'un modèle objet, car il est impossible, en cas de redéfinition, d'accéder à la variable associée au type dynamique de l'objet (τ_d) ;
- les variables d'instances des classes, considérées comme des instances de méta-classes dans un modèle réflexif, à la SMALLTALK [Goldberg et Robson, 1983], CLOS [Steele, 1990] ou OBJVLISP [Cointe, 1987] : cette notion n'a pas d'équivalent direct dans les langages que nous considérons ici ;
- le fait qu'une variable d'instance « normale » puisse être allouée dans la classe, et non dans chaque instance, parce que sa valeur est partagée par toutes les instances de la classe.

[13] Ce sont les termes utilisés en C++. En EIFFEL, le terme utilisé est « méthode de création », qui n'est pas plus heureux.

Si la première catégorie sort du modèle objet et s'implémente sans difficulté particulière, les 2 autres catégories diffèrent par leur sémantique — on peut les illustrer respectivement, par la liste des instances d'une classe, qui est une propriété de la classe, et par le nombre de côtés des quadrilatères, qui vaut 4, mais qui est une propriété des instances — et par leur implémentation, puisque qu'une classe ne partage pas ses variables d'instance avec ses sous-classes, alors qu'elle partage ses variables allouées dans la classe, sauf en cas de redéfinition dans les sous-classes.

Au-delà de ces différences, l'implémentation est analogue. Dans le cadre de langages non réflexifs comme ceux que nous étudions ici, `:allocation :class` sera implémenté en allouant un indice pour cet attribut dans la table des méthodes de la classe. Le partage entre la classe et ses sous-classes sera assuré par une indirection supplémentaire (*wrapper*), qui est inutile lorsqu'il s'agit d'une constante, c'est-à-dire d'un attribut immutable. Curieusement, ce mécanisme très simple n'existe dans aucun des trois langages principaux que nous examinons : C++ et JAVA n'ont que les variables `static`[14] et EIFFEL rien du tout[15]. Pour tous ces langages, l'efficacité n'est pas une excuse valable : des variables de classe à sélection dynamique, avec une syntaxe $a.p$, pourra aussi être sélectionnées statiquement, avec la syntaxe $A.p$ (ou $A :: p$).

La situation se complique avec le mot-clé `:allocation` de CLOS [Steele, 1990], qui gouverne l'implémentation d'un attribut dans chaque instance ou dans la classe, suivant qu'il est propre à chaque instance ou partagé par toutes. En effet, ce mot-clé peut prendre 2 valeurs, `:instance` et `:class`, et il est redéfinissable à volonté dans les sous-classes, ce qui viole de façon caractérisée l'invariant de position de l'attribut. L'utilisation d'accesseurs (cf. section 4.4) permettrait d'en exprimer la sémantique sans obtenir l'économie de place que l'on attend du partage.

2.5.2 Héritage non monotone

Cette implémentation pourrait enfin permettre de ne pas hériter certains attributs, ce qui serait possible sans erreurs de types avec un minimum de précautions. On définirait `Carré` comme sous-classe de `Rectangle`, en rajoutant un attribut `côté` et en n'héritant pas des attributs `largeur` et `longueur`. Pour qu'il n'y ait pas d'erreurs de type, il faut bien sûr que ces attributs non hérités soient parfaitement encapsulés à la SMALLTALK (accessibles que sur `self`) et que toutes les méthodes de `Rectangle` qui les utilisent soient redéfinies dans `Carré` (ce qui pose un problème de compilation séparée). Avec ces précautions, l'implémentation se fait sans problème : il suffit de considérer que les indices des attributs non hérités sont libres et peuvent être attribués à de nouveaux attributs, de la classe considérée ou des sous-classes ultérieures.

Dans le même ordre d'idée, EIFFEL permet de redéfinir une méthode sans argument par un attribut, ce qui est parfaitement indépendant de l'implémentation : il suffit que, dans les redéfinitions, les accès en lecture soient systématiquement encapsulés dans des accesseurs générés par le compilateur (cf. section 4.4).

Le non héritage de certaines méthodes est plus difficile dès lors qu'elles sont publiques. SMALLTALK, dans un contexte dynamique donc, offre une exception prédéfinie (`ShouldNotImplement`) pour interdire l'usage d'une méthode dans une sous-classe : une telle fonctionnalité ne s'étend pas au typage statique sans introduire l'équivalent d'erreurs de types.

2.6 Evaluation

L'efficacité temporelle est relativement optimale, puisque tout se fait par une simple indirection dans une table obtenue par une simple indirection dans l'objet. Même le *casting* se fait en temps constant. D'un point de vue spatial :

[14] La seule façon de l'implémenter consiste alors à définir deux méthodes de lecture et d'écriture (des accesseurs, cf. section 4.4) accédant à une variable statique : cette variable et ses accesseurs doivent être redéfinis, à la main, dans chaque sous-classe de la classe considérée pour laquelle on souhaite redéfinir l'attribut de classe. En SMALLTALK, la définition des variables de classes peut être aussi bien interprétée à la C++ ou à la CLOS : il n'est pas possible de les redéfinir.

[15] Les méthodes `once` permettent de simuler des attributs en lecture seule.

– les objets occupent une place optimale : un « champ » par attribut, plus deux pointeurs vers la table des méthodes et l'identifiant de classe ;
– les tables de méthodes ne dépendent que du type dynamique des objets, ce qui fait que chaque classe a une table unique ; elles occupent globalement une place égale au nombre de couples classe-sélecteur valides, ce qui correspond à l'optimum de compactage des tables en typage dynamique et héritage multiple [Ducournau, 1997] ; si M_C désigne le nombre de méthodes connues (introduites ou héritées) par une classe C, la place occupée totale est en $\Sigma_C(M_C)$;
– la vérification dynamique de type (*casting* descendant) se fait en temps constant et ne nécessite que deux petits entiers par classe.

Une seule fonctionnalité ne s'implémente pas avec cette technique : la redéfinition du mot-clé `:allocation` de CLOS qui gouverne l'implémentation d'un attribut, au choix, dans la classe ou dans l'instance : le mot-clé est implémentable, mais sans redéfinition.

En revanche, la transparence à peu près complète du *casting* — réduit à un simple test numérique pour la vérification de type nécessaire au *casting* descendant — enlève tout surcoût à la redéfinition covariante.

Cette implémentation à peu près optimale de l'héritage simple est la référence par laquelle on peut mesurer le « surcoût » de l'héritage ou du sous-typage multiple, en ce qui concerne l'efficacité temporelle (nombres d'indirections) ou spatiale (occupation mémoire statique des tables de méthodes et dynamique des objets).

Chapitre 3

En héritage multiple

3.1 Le problème de l'héritage multiple

En héritage multiple, le problème se complique considérablement comme le montre [Ellis et Stroustrup, 1990, chapitre 10] dans le cas de C++. La complication est d'autant plus importante que C++ offre des traits de langage — en l'occurrence le double rôle du mot-clé `virtual` — qui ne relèvent pas d'un bon usage de l'approche objet (cf. section 4.2). Nous nous placerons, dans cette section, dans un cadre plus orthodoxe qui reviendrait, en C++, à utiliser systématiquement `virtual` :

- toutes les fonctions sont *virtuelles*, au sens où elles sont toutes sélectionnées par liaison tardive,
- tous les héritages sont *virtuels*, au sens où chaque super-classe n'est utilisée qu'une seule fois (cf. section 4.2).

Ces précautions liminaires sont inutiles pour un langage objet normalement constitué comme EIFFEL [Meyer, 1992 ; Meyer, 1997].

Le problème causé par l'héritage multiple s'énonce simplement : l'indice d'une méthode ou d'un attribut ne peut plus être invariant par héritage (invariant 2.2), en tout cas tant que ces indices sont calculés de façon séparée, en cherchant à les minimiser (cf. section 7.2). La raison en est la suivante : étant donné deux classes incomparables B et C, qui occupent les mêmes indices, il est toujours possible d'en définir une sous-classe commune D. On se trouve donc en situation de conflit puisque deux attributs ou deux méthodes sont en compétition pour le même indice.

Ce premier constat a une conséquence décisive : si l'on veut que l'envoi de message s'effectue par une indirection dans une table, un pointeur sur un objet n'est plus invariant suivant son type statique (invariant 2.1).

On verra plus loin comment, et à quel prix, cette conséquence peut être inversée : l'invariance de pointeur donne à l'envoi de message un coût non constant (cf. section 5.5) ou nécessite une compilation globale, ou au moins une phase globale (cf. section 7.2).

3.2 Principe d'implémentation

On est donc conduit à relâcher l'invariance de l'indice des attributs et méthodes comme suit :

Invariant 3.1 *Chaque attribut a un indice non ambigu et invariant dans le contexte du type statique qui l'introduit, donc indépendamment du type dynamique.*

Invariant 3.2 *Chaque méthode a un indice non ambigu et invariant dans le contexte d'un type statique qui connait la méthode, donc indépendamment du type dynamique.*

Mais ces indices ne sont plus invariants par héritage, c'est-à-dire entre deux types statiques liés par une relation de spécialisation, pas plus que l'objet lui-même : la valeur de `self` et de tout pointeur sur un objet dépend de son type statique. Tout se passe comme si les tables des attributs (l'objet) et des méthodes étaient constituées de sous-tables (ou sous-objets), une par super-classe. Et l'invariant principal est alors le suivant (figure 3.1) :

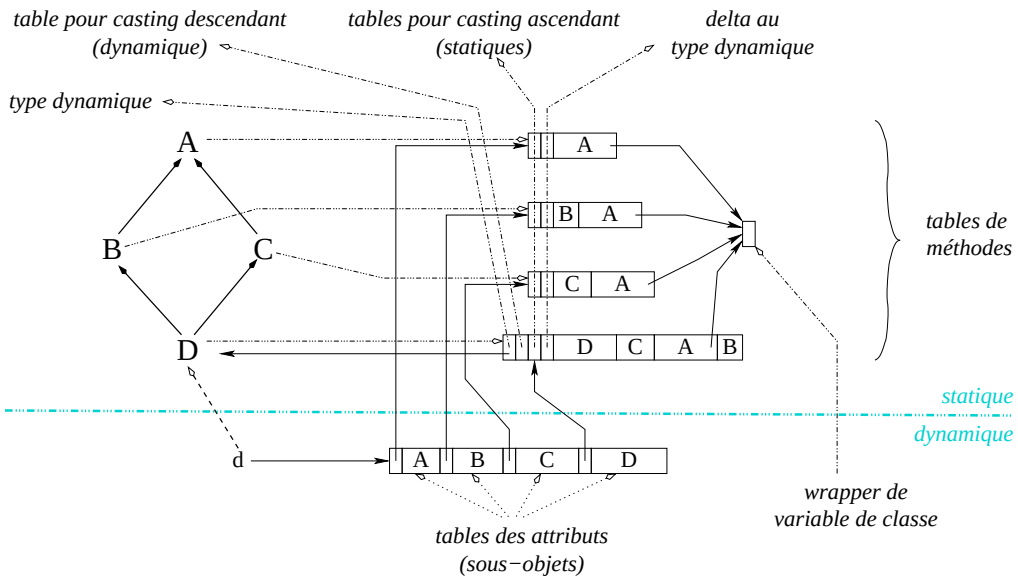

FIG. 3.1 – Tables des attributs et des méthodes en héritage multiple : contrairement à la figure 2.1, une seule instance est décrite.

Invariant 3.3 *Toute entité de type statique T est liée au sous-objet correspondant à T, muni de sa propre table de méthodes.*

Cet invariant est trivialement vérifié par l'implémentation de l'héritage simple (invariant 2.1). En héritage multiple, il est nécessaire de rajouter une propriété de non trivialité, qui est la source du surcoût de cette implémentation :

Invariant 3.4 *Deux sous-objets de types statiques différents sont distincts.*

L'unique exception a lieu lorsqu'une classe F spécialise une classe E, en héritage simple, sans rajouter de nouveaux attributs : le sous-objet de type F peut disparaître dans le sous-objet de type E muni de la table de méthodes de F[1].

Contrairement au sous-typage simple, l'implémentation de l'héritage multiple se caractérise donc par une dépendance absolue vis-à-vis des types statiques, à tel point qu'il n'est jamais évident que le comportement des programmes respecte bien la sémantique invariante de rigueur. Pour la théorie des types, un type n'a qu'un but dans la vie : se faire éliminer (*erasure*) par le compilateur, qui prouve qu'il peut le faire sans risque d'erreur de type à l'exécution. Avec cette implémentation, c'est manifestement raté.

Chaque sous-objet ne contient que les attributs *introduits* par son type statique. Chaque table de méthodes d'un type statique contient toutes les méthodes connues par ce type, mais avec des valeurs (adresses) correspondant aux méthodes effectivement héritées par le type dynamique. Ainsi, deux instances directes de classes différentes ne partagent pas les tables de méthodes de leurs super-classes communes : ces tables ont la même structure, mais pas le même contenu (figure 3.2).

Pour un type statique donné, l'ordre de ces méthodes est *a priori* quelconque, mais il est raisonnable de les regrouper par classe (ce qui est fait dans la figure) et d'assurer une certaine invariance par spécialisation, lorsque c'est possible (cas d'héritage simple). Mais cette organisation n'a aucun effet sur l'efficacité de l'implémentation.

L'invariant 3.3 impose de recalculer la valeur de `self` à chaque envoi de message : lorsque le receveur est une entité de type statique τ_s, et que la méthode sélectionnée a été définie dans la classe v, il faut savoir de combien il faut incrémenter ou décrémenter `self` pour obtenir, à partir du sous-objet de type τ_s un

[1] [Zendra, 2000] suggère de partager les pointeurs de type B et D, ainsi que leurs tables de méthodes, ce qui réduirait ainsi le surcoût. Cette interprétation est manifestement erronée : il suffit de considérer une deuxième sous-classe E de B et C. Comment pourrait-on implémenter une sous-classe commune à D et E ? En fait, cette suggestion suppose, pour être correcte, que B et D ne soient pas « virtuelles » (cf. section 4.2).

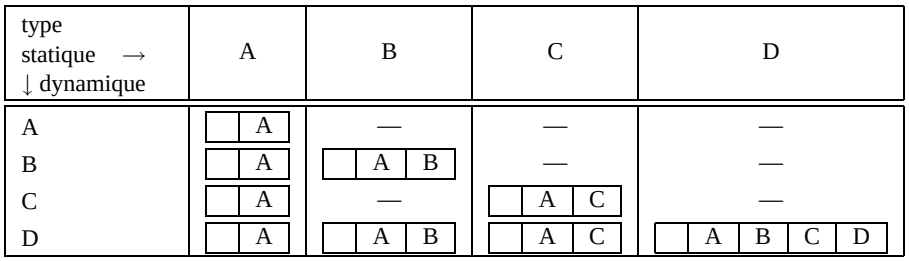

FIG. 3.2 – Tables des méthodes pour l'exemple de la figure 3.1, suivant les types statiques et dynamiques. Pour un même type statique (verticalement), les tables sont isomorphes mais diffèrent par leurs contenus (adresses des méthodes) ; en revanche, pour un même type dynamique (horizontalement), les morceaux isomorphes contiennent les mêmes adresses mais pas les mêmes décalages.

sous-objet de type v, ce que l'on notera $\Delta_{\tau_s,v}$[2]. La table des méthodes est donc double : elle contient, pour chaque méthode, l'adresse de la méthode ainsi que la valeur de cet incrément (figures 3.3 et 4.1, en haut).

Au total, l'envoi de message se compile donc par une séquence de cinq instructions[3] :

```
load [object + #tableOffset], table
load [table + #deltaOffset], delta
load [table + #selectorOffset], method                    2L + B + 1
add object, delta, object
call method
```

Une technique alternative consiste à définir une petite fonction intermédiaire qui fait le décalage (technique dénommée *thunks* provenant d'ALGOL, mais pas utilisable sur tous les processeurs, d'après [Ellis et Stroustrup, 1990]) : ces fonctions peuvent être partagées par tous les appels de la même méthode avec le même décalage[4]. En cas d'implémentation par *thunk*, la séquence d'instructions est la même qu'en héritage simple, mais elle provoque un branchement au code suivant :

```
add object, #delta, object
jump #method
```

On économise ainsi une indirection dans une table, au prix d'un saut constant[5].

Chaque table de méthodes doit aussi contenir un décalage vers le type dynamique (noté $\Delta_{\Downarrow}^{\tau_s} = \Delta_{\tau_s,\tau_d}$). L'identifiant du type dynamique n'est nécessaire que dans la table du type dynamique : il peut servir pour vérifier l'égalité du type dynamique, pour un *casting* descendant simplifié. On peut mettre cet identifiant dans la table de méthodes de chaque type statique pour un surcoût négligeable, mais il n'en est pas de même si on le met dans l'objet lui-même. Quant aux références au type statique, elles sont statiques.

3.3 Le *casting*

Si le *casting* est inexistant en héritage et sous-typage simples, il prend en héritage multiple, si ce n'est tout son sens, du moins toute sa réalité, avec cette implémentation par sous-objets. [Rossie et Friedman, 1995] le définit ainsi comme un changement de sous-objet. Le *casting* peut alors s'implémenter par deux tables supplémentaires.

Notons d'abord deux propriétés fondamentales des Δ : pour toute sous-classe commune à t, u et v,

$$\Delta_{t,v} = \Delta_{t,u} + \Delta_{u,v} \tag{3.1}$$

$$\Delta_{t,t} = 0 \tag{3.2}$$

[2] Précisons que la notation $\Delta_{t,u}$, ainsi que toutes les notations Δ qui vont suivre, sous-entend un type dynamique τ_d donné, dont l'explicitation alourdirait trop la notation.

[3] En italiques, les instructions supplémentaires par rapport à l'héritage simple (cf. page 8).

[4] Dans le pire des cas, il y a un *thunk* par triplet de classes τ_s, τ_d, v, où v est la classe où est définie la méthode sélectionnée pour τ_d, avec $\tau_d \preceq v$, pour un décalage $\Delta_{\tau_s,v}$. v étant déterminé par τ_d, le nombre de *thunks* est quadratique. Bien entendu, le nombre effectif de *thunks* peut être réduit, les *thunks* de même décalage pouvant être factorisés. De plus, le *thunk* de décalage 0 (uniquement quand $\tau_s = v$) s'identifie à la méthode elle-même.

[5] Saut qui peut être parfaitement anticipé par le processeur, sauf que la séquence précédent le saut est ici un peu courte.

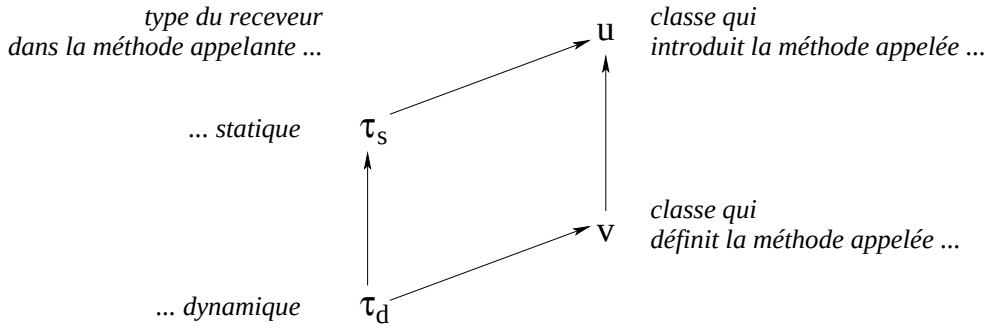

FIG. 3.3 – Les types en jeu sur le receveur, dans un appel de méthode : un décalage $\Delta_{\tau_s,v}$ est nécessaire

3.3.1 *Casting* ascendant.

Pour passer du type statique τ_s à un super-type (statique) t, il est nécessaire de connaître le $\Delta_{\tau_s,t}$, qui peut varier suivant le type dynamique τ_d : une table supplémentaire, notée $\Delta^{\uparrow}_{\tau_s}$, est donc nécessaire dans chaque table de méthodes. L'indice de t dans $\Delta^{\uparrow}_{\tau_s}$ est invariant relativement au type dynamique et est donc connu statiquement.

Invariant 3.5 *Chaque classe a un indice non ambigu et invariant dans le contexte statique de ses sous-classes.*

On notera $i_{\tau_s}(t)$ cet indice : $\Delta^{\uparrow}_{\tau_s}(t)$ est alors donné par $\Delta^{\uparrow}_{\tau_s}[i_{\tau_s}(t)]$.

Au lieu d'être implémentée comme une table indépendante, cette table peut être implémentée dans les tables de méthodes. En toute généralité, le *casting* ascendant peut être traité comme les méthodes, comme si chaque classe définissait la méthode de *casting* ascendant vers elle-même. Il est donc possible et plus efficace de se servir de la table des méthodes, qui contiendrait le décalage au lieu de l'adresse d'une procédure : cela fait gagner une indirection et beaucoup sont prêts à pire pour cela [Eckel et Gil, 2000] (cf. section 4.3). Mais la description par une table supplémentaire est plus claire[6].

3.3.2 Accès aux attributs.

Cette table $\Delta^{\uparrow}_{\tau_s}$ sert aussi à l'accès aux attributs lorsqu'ils sont introduits dans une super-classe de τ_s. Si $\delta(p,u)$ représente l'indice d'un attribut p dans le sous-objet de type u — ce qu'il faut connaître pour accéder à un attribut d'une entité de type statique u — et que δ_p est l'indice de l'attribut p dans le type t_p qui l'introduit (invariant 3.1), on obtient la position de l'attribut par :

$$\delta(p,\tau_s) = \Delta_{\tau_s,t_p} + \delta(p,t_p) = \Delta^{\uparrow}_{\tau_s}(t_p) + \delta_p \tag{3.3}$$

Dans le cas général, c'est-à-dire lorsque l'attribut n'a pas été introduit dans le type statique de l'objet ($t_p \neq \tau_s$), l'accès à un attribut sera donc sensiblement plus compliqué qu'en héritage simple :

```
load [object + #tableOffset], table
load [table + #castOffset], delta
add object, delta, object                          3L + 1
load [object + #attributeOffset], attribute
```

Bien entendu, sur toute affectation $a.x = b.y$ il faut composer le *casting* ascendant entre les types de y et de x, avec les *castings* ascendants sur a et b. Ces castings se parallélisent[7] en partie — les deux objets en parallèle, mais l'affectation en séquence —, et la séquence résultante est 5 fois plus longue qu'en héritage simple :

[6] Le *casting* descendant ne se prête pas à un tel traitement, puisque la classe cible n'est alors pas encore connue au moment de la définition de la classe source.

[7] Mais au détriment de ce qui aurait pu être exécuté en parallèle.

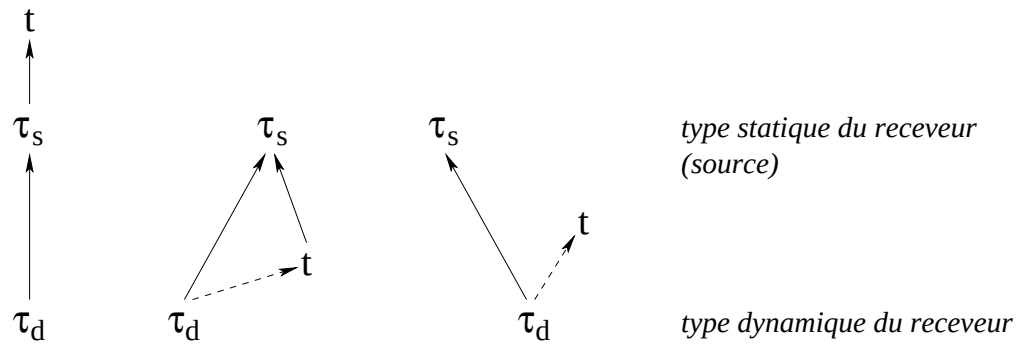

FIG. 3.4 – *Casting* ascendant (gauche), descendant (milieu) et latéral (droite), vers un type cible t : la relation de sous-typage en pointillé doit être vérifiée dynamiquement.

```
load [object1 + #tableOffset], table1
load [object2 + #tableOffset], table2
load [table1 + #cast1Offset], delta1
load [table2 + #cast2Offset], delta2
add object1, delta1, object1
add object2, delta2, object2                                    6L + 2
load [object1 + #attribute1Offset], attribute
load [attribute + #tableOffset], table
load [table + #castOffset], delta
add attribute, delta, attribute
store attribute, [object2 + #attribute2Offset]
```

3.3.3 *Casting* descendant.

Pour passer du type statique τ_s à un sous-type (statique) t, il faut à la fois une vérification de type et le $\Delta_{\tau_s,t}$. La difficulté réside dans l'impossibilité (au moins en compilation séparée, cf. section 7.2) d'un accès direct et une recherche séquentielle est nécessaire : le plus simple consiste sans doute à associer à chaque classe τ_d une table d'association $t \mapsto \Delta_{\tau_d,t}$, pour toutes les super-classes t de τ_d (cf. annexe A.1) : on notera cette table $\Delta^{\Uparrow}$. Pour faire un *casting* descendant du type statique τ_s à un type (statique) t dans un objet de type dynamique τ_d, il faut alors retourner au type dynamique par $\Delta^{\tau_s}_{\Downarrow} = \Delta_{\tau_s,\tau_d}$ et rechercher t dans la table d'association : si on ne le trouve pas, il y a une erreur de type, et sinon on retourne $\Delta_{\tau_d,t}$. On obtient alors le résultat par

$$\Delta_{\tau_s,t} = \Delta_{\tau_s,\tau_d} + \Delta_{\tau_d,t} = \Delta^{\tau_s}_{\Downarrow} + \Delta^{\Uparrow}(t) \tag{3.4}$$

La table $\Delta^{\Uparrow}$, de même que le type dynamique lui-même, n'ont besoin d'être référencés que par la table du type dynamique.

Il est important de noter que ces deux tables, $\Delta^{\updownarrow}_{\tau_s}$ et $\Delta^{\Uparrow}$, contiennent la même information[8], structurée différemment et à utiliser dans des contextes différents : dans la première, τ_s est connu statiquement, alors que dans la deuxième, τ_d ne l'est pas. Leur contenu est encore identique aux décalages qui sont inclus dans les tables de méthodes.

Cette implémentation de la vérification de type est coûteuse et pas très efficace. Un codage d'ordre, notablement plus difficile en héritage multiple (cf. sections 2.2.2 et A.3), ne convient pas puisque le problème n'est plus booléen : les décalages sont nécessaires. La solution naïve serait une matrice d'incidence de la fermeture transitive, basée une simple numérotation des classes (notée $n(C)$) et contenant les $\Delta_{t,u}$: elle coûterait N^2 octets[9], si N est le nombre de classes. C'est raisonnable pour 100 classes, moins pour

[8] Ou plutôt que $\Delta^{\Uparrow}$ contient l'information contenue dans $\Delta^{\updownarrow}_{\tau_s}$, à Δ_{τ_s,τ_d} près, pour chacune des super-classes τ_s de τ_d (cf. figure 3.5).

[9] A la condition que cette numérotation des classes soit une extension linéaire (cf. annexe A.2), ce qui permet de considérer une matrice triangulaire. Il faut sinon compter le double.

τ_s	$\Delta_{\Downarrow}$	$\Delta^{\uparrow}$	$\Delta_{\downarrow}$	$\Delta^{\Uparrow}$
A	Δ_{AD}	[ ]	$[(A,0)(B,\Delta_{AB})(C,\Delta_{AC})(D,\Delta_{AD})]$	
B	Δ_{BD}	$[\Delta_{BA}]$	$[(B,0)(D,\Delta_{DB})]$	
C	Δ_{CD}	$[\Delta_{CA}]$	$[(C,0)(D,\Delta_{CD})]$	
D	0	$[\Delta_{DA},\Delta_{DB},\Delta_{DC}]$	$[(D,0)]$	$[(A,\Delta_{DA})(B,\Delta_{DB})(C,\Delta_{DC})(D,0)]$

FIG. 3.5 – Tables de *casting* pour la classe (τ_d) D de la figure 3.1 : $\Delta_{\Downarrow}$ est un scalaire, $\Delta^{\uparrow}$ une table à accès direct, $\Delta_{\downarrow}$ et $\Delta^{\Uparrow}$ des structures d'association.

1000, et impossible pour 10000[10]. Une solution plus compacte est donnée par l'heuristique de coloration (cf. section 7.2).

3.3.4 *Casting* latéral

Ce traitement du *casting* descendant permet aussi bien un *casting* latéral, la cible t devant être une super-classe de τ_d mais pas forcément une sous-classe de τ_s (figure 3.4). Ce *casting* latéral peut être nécessaire lorsque la hiérarchie de types n'est pas un treillis, ce qui est (toujours, jamais[11]) le cas, lorsque les types sont réduits aux classes : la vérification dynamique du type t suffit pour garantir que l'objet est bien du type de l'un des minorants maximaux[12] de t et τ_s.

Si le *casting* descendant peut apparaître comme un typage dynamique borné par un type statique, on voit en fait qu'il n'en est rien, ni d'un point de vue conceptuel, ni même du point de vue de l'implémentation : la connaissance du type statique n'apporte aucune information supplémentaire. Pour tirer parti du typage statique, il faudrait associer à chaque type statique une nouvelle table[13] $\Delta_{\downarrow}^{\tau_s}$, restriction de $\Delta^{\Uparrow}$ aux classes comprises entre τ_s et τ_d. Ces nouvelles tables évitent le décalage en deux temps, puisque

$$\Delta_{\tau_s,t} = \Delta_{\downarrow}^{\tau_s}(t) \tag{3.5}$$

mais le gain temporel à en attendre ne justifie sans doute pas le surcoût spatial. Le *casting* latéral serait alors impossible avec les seules $\Delta_{\downarrow}^{\tau_s}$.

Enfin, un test d'égalité physique entre deux références — référencent-elles le même objet ? — s'effectue, en toute généralité, en ramenant ces 2 références au type dynamique par $\Delta_{\Downarrow}$.

3.4 Redéfinition de types

Si, en héritage simple, la redéfinition de types s'implémentait de façon à peu près transparente, ce n'est plus le cas en héritage multiple. C++ interdit toute redéfinition, à part celle du type de retour[14] : ce n'est peut-être pas simplement à cause des erreurs de type.

3.4.1 Redéfinition covariante du type de retour

En héritage multiple, la redéfinition du type de retour va nécessiter un *casting* ascendant dans la méthode appelante du type effectivement retourné au type statique de la cible. C'est un *casting* ascendant,

Cette matrice triangulaire revient à associer à chaque τ_d la table des $\Delta_{\tau_d,t}$ pour tous les t tels que $n(t) < n(\tau_d)$, et non pas simplement pour les super-classes de τ_d. Comme en héritage simple, cela suppose une numérotation globale des classes, à faire à l'édition de liens, mais cette numérotation-ci est incrémentale et donc compatible avec une édition de liens dynamique.

[10] Notons que l'on n'a que peu d'informations sur le nombre de classes en typage statique et compilation séparée, contrairement au typage dynamique pour lequel on dispose de quelques statistiques [Ducournau, 1997]. Quelques centaines semblent courant — le compilateur SMALL EIFFEL par exemple [Zendra *et al.*, 1997] — mais quelques milliers ? La littérature regorge bien de *benchmarks*, [Vitek *et al.*, 1997 ; Eckel et Gil, 2000] par exemple, mais il représentent plus des bibliothèques de classes que des applications.

[11] Parle-t-on de la proposition positive, qui est niée, ou de sa négation, qui est affirmée ?

[12] Le fait qu'il puisse y en avoir plusieurs n'est pas un problème puisque la cible reste *t*.

[13] La mnémotechnie est la suivante : $\uparrow$ désigne le *casting* ascendant, $\downarrow$ le descendant, $\Downarrow$ le *casting* descendant vers le type dynamique et $\Uparrow$ le *casting* descendant obtenu par des moyens ascendants à partir du type dynamique.

[14] Sur ce point, la position de C++ n'est pas claire : [Ellis et Stroustrup, 1990, page 210 *sq.* et page 421] affirme à la fois que cette possibilité fait partie du standard ANSI mais que son implémentation nécessiterait une complication du mécanisme d'appel et de retour de méthodes. De fait, le *casting* à la C ne l'autorisait que marginalement, alors que `dynamic_cast` l'autorise dans tous les cas où ce n'est pas intrinsèquement ambigu (cf. section 4.2).

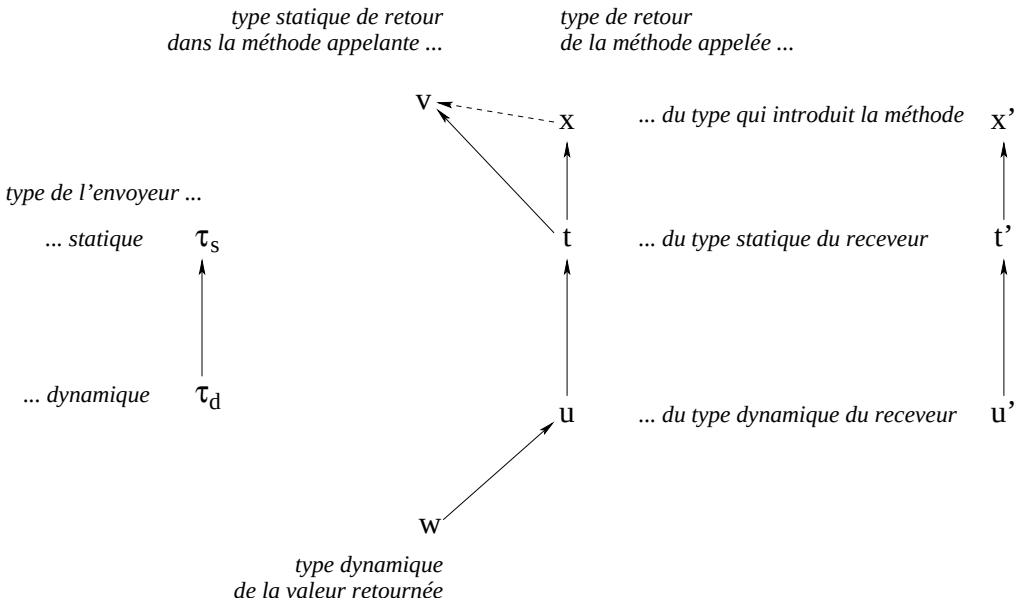

FIG. 3.6 – Redéfinition covariante de la valeur de retour. 10 types sont en jeu, pour 3 valeurs : l'envoyeur, le receveur et la valeur de retour. Le *casting* ascendant et statique de t à v est suffisant en l'absence de redéfinition (cas où $x = t = u$), la méthode appelée assurant le *casting* de w à t. En cas de redéfinition, u n'est plus connu statiquement : le *casting* nécessaire, de u à v, est toujours ascendant mais n'est plus statique. Dans les cas favorables, le *casting* de x à t, toujours statique, peut être ascendant.

puisque la source est un sous-type de la cible, mais il est dynamique car le type source n'est pas connu statiquement. Il n'est donc pas possible de se servir des tables $\Delta^{\uparrow}$ et il faudrait avoir recours, dynamiquement, à la table $\Delta^{\Uparrow}$. La seule différence avec un *casting* descendant serait alors que la vérification de type est inutile car le *casting* réussit toujours.

Une solution entièrement statique est pourtant possible (figure 3.6). Le problème, en effet, est que la méthode appelée ne sait pas que l'appelante attend un type t, et l'appelante ne sait pas qu'elle va recevoir un type $u \prec t$: l'indice de t (ou de v) dans $\Delta^{\uparrow}_{u}$ est donc inconnu. Il est possible de stocker cet indice dans la table des méthodes de τ_s, le type statique de l'envoyeur : son type dynamique τ_d connaissant u saura quelle valeur y mettre.

```
// self est l'envoyeur du message
// return est la valeur retournée, de type statique u
load [self + #tableOffset], table1
load [return + #tableOffset], table2
load [table1 + #cast1Offset], offset
add table2, offset, table2
load table2, delta1
add return, delta1, return                          // casting de u vers t
load [return + #tableOffset], table3
load [table3 + #cast2Offset], delta2
add return, delta2, return                          // casting de t vers v
```

Comme toute méthode est *a priori* spécialisable, il est nécessaire d'ajouter dans la table de chaque type statique τ_s, pour toute méthode appelée par le type statique, l'indice du décalage pour ajuster la valeur de retour. Les tables de méthodes vont donc augmenter dans des proportions qu'il est difficile de mesurer dans la mesure où cela dépend des méthodes appelées par le type statique et non pas définies par lui, et la séquence d'envoi de message va doubler de longueur et de durée. Principal inconvénient de la technique : elle est aveugle et s'applique à tous les envois de message, car il n'est pas possible, statiquement, de déterminer que $u = t$. Elle pose de plus un petit problème de modularité : les sous-classes vont devoir maintenir des données pour des appels qu'elles ne connaissent pas et qu'elles n'effectuent qu'indirectement.

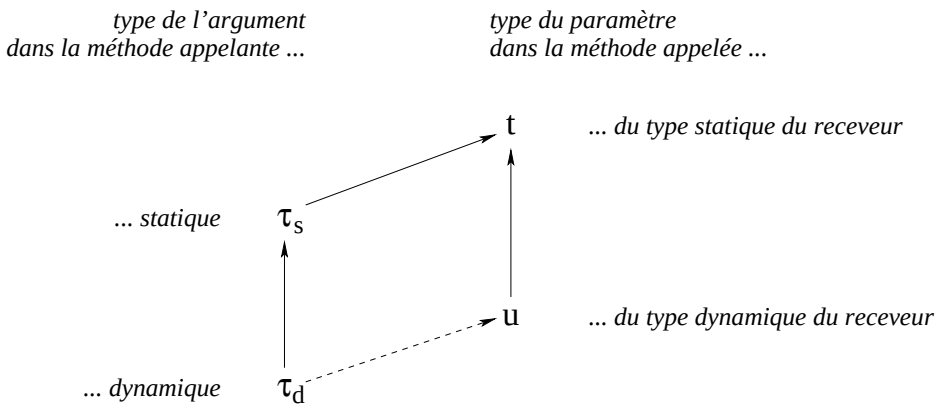

FIG. 3.7 – Passage de paramètres covariants : le *casting* latéral de τ_s à u est nécessaire.

Une deuxième solution serait que la méthode appelée retourne systématiquement le type de retour dans la classe x d'introduction de la méthode (seul type invariant pour l'appelée). Si c'est un sous-type de v, la méthode appelante effectue un *casting* ascendant. Si ce n'est pas le cas, on est ramené au cas général, avec un *casting* descendant sûr.

Enfin, la troisième solution consistera à traiter le problème par un *thunk*, qui effectuera le décalage $\Delta_u^{\uparrow}(t)$, pour chaque paire (t', u').

```
add object, #delta, object
call #method
load [return + #tableOffset], table
load [table + #castOffset], delta
add return, delta, return                          // casting de u vers t
```

Le deuxième *casting* est alors à faire dans la méthode appelante. Principal inconvénient : l'appel de la méthode par le *thunk* n'est plus terminal. Principaux avantages : il y a moins d'indirections, cela ne rajoute rien à la combinatoire des *thunks* (cf. note 4, page 18) et l'on ne s'en sert qu'en cas de besoin, c'est-à-dire pour les *thunks* de paire (t', u') tels que $t \neq u$.

3.4.2 Passage de paramètres non invariants

Lorsque le type du paramètre est invariant, le passage nécessite juste un *casting* ascendant entre le type statique de l'argument et celui du paramètre : le type cible est connu statiquement et le décalage est fourni par la table $\Delta_{\tau_s}^{\uparrow}$.

Pour une redéfinition strictement contravariante, le *casting* est toujours ascendant, mais le type cible n'est plus connu statiquement, puisqu'il dépend du type dynamique du receveur. L'extension de la table des méthodes aux Δ des paramètres serait possible, mais très coûteuse en place : la méthode des *thunks* est donc sans doute la plus adaptée et cela ne rajouterait rien à leur combinatoire.

Pour la redéfinition covariante, la situation est décrite par la figure 3.7 et 6 types sont en jeu :

- τ_s et τ_d sont les types statiques et dynamiques de l'argument dans la méthode appelante ;
- t et u sont les types formels du paramètre dans les méthodes respectives des types statique et dynamique du receveur.

Par construction, on dispose de 3 relations de sous-typage :

$$\tau_d \preceq \tau_s \qquad \text{polymorphisme} \tag{3.6}$$

$$\tau_s \preceq t \qquad \text{vérification statique à la compilation} \tag{3.7}$$

$$u \preceq t \qquad \text{redéfinition covariante} \tag{3.8}$$

L'objectif est de vérifier que $\tau_d \preceq u$, en faisant le décalage $\Delta_{\tau_s,u}$, qui peut-être aussi bien ascendant,

descendant que latéral, mais u n'est pas connu statiquement. Deux démarches sont possibles :

$$\Delta_{\tau_s,u} = \Delta_{\tau_s,t} + \Delta_{t,u} = \Delta^{\uparrow}_{\tau_s}(t) + \Delta^{t}_{\downarrow}(u) \quad (3.9)$$

$$\Delta_{\tau_s,u} = \Delta_{\tau_s,\tau_d} + \Delta_{\tau_d,u} = \Delta^{\tau_s}_{\Downarrow} + \Delta^{\Uparrow}(u) \quad (3.10)$$

Dans les deux cas, on commence par un premier décalage statique dans la méthode appelante. Le décalage dynamique et la vérification de type qui suivent se réalisent, dans le premier cas (3.9), par un *thunk*, dans le deuxième cas (3.10), par la méthode appelée elle-même. La première méthode a l'avantage de n'imposer un *casting* qu'en cas de nécessité. Notons que les *thunks* rendent inutile la technique d'indiçage des signatures de [Meyer, 1995].

3.4.3 Bilan sur les *thunks*

Au total, dans une implémentation par *thunks*, pour chaque méthode introduit dans un type u, il y a exactement un *thunk* par couple (τ_s, τ_d) avec $\tau_d \prec \tau_s \preceq u$. Chaque *thunk* effectue :

- le *casting* ascendant sur le receveur (figure 3.3), entre τ_s et le type v qui définit la méthode héritée par τ_d ;
- les éventuels *casting* descendant sur les paramètres dont le type statique dans v est un sous-type du type dans u (figure 3.7),
- l'appel de la méthode,
- l'éventuel *casting* ascendant du type de retour (figure 3.6).

Si $\tau_s = v$, le *thunk* est la méthode elle-même : ni décalage, ni redéfinition.

Il faut enfin ajouter à cela, en dehors des *thunks*, le *casting* ascendant :

- du type des arguments au type du paramètre de la méthode du type statique du receveur (de τ_s à t dans la figure 3.7) ;
- du type de retour de cette même méthode au type statique de l'expression qui consomme cette valeur de retour (de t à v dans la figure 3.6).

3.4.4 Redéfinition covariante des attributs

La redéfinition covariante des attributs est une synthèse des redéfinitions covariante du type de retour et du type des paramètres. Nous avons vu que la redéfinition covariante des paramètres ne posait pas d'autres problèmes que la nécessité d'une vérification dynamique de type par un *casting* descendant. On a donc — ce qui n'est pas rien — le risque d'erreur de types prévu par la théorie, ainsi qu'un certain surcoût. Mais l'implémentation elle-même ne pose pas de problèmes particuliers.

Pour les attributs, la redéfinition pose les mêmes problèmes, plus un qui pourrait être plus grave : l'implémentation ne peut pas satisfaire l'invariant 3.1, qui s'applique à toute « entité » typée des programmes, y compris les attributs. Si le type statique d'un attribut change suivant le contexte statique, le pointeur de cet attribut sur un objet ne pourrait pas rester invariant.

Il est donc nécessaire d'interpréter l'invariant de la façon suivante :

Invariant 3.6 *L'attribut pointe sur le sous-objet correspondant au type de l'attribut dans le type dynamique de l'objet qui contient l'attribut*[15].

Tout accès en lecture est suivi d'un *casting* ascendant sur le type statique de l'attribut, de même que tout accès en écriture est précédé d'un *casting* descendant sur le type de l'attribut dans le type dynamique de l'objet, pour assurer la première règle. Ces deux *castings* sont identiques à ceux qui sont faits en cas de redéfinition covariante, respectivement, du type de retour ou du type d'un paramètre.

Bien entendu, si l'on implémente les accès aux attributs par des accesseurs, ces problèmes deviennents transparents, l'attribut ayant toujours le type statique de ses accesseurs (cf. section 4.4) : mais cela n'économiserait pas les *castings*.

[15] Prendre comme invariant le type dynamique de la valeur aurait l'inconvénient de ne pas être compatible avec l'implémentation sans redéfinition. En revanche, le type de l'attribut dans la classe d'introduction conviendrait très bien, surtout si la solution 2 est adoptée pour la redéfinition du type de retour (cf. section 3.4.1).

Du point de vue de l'efficacité de l'accès en lecture, il n'est pas évident de comparer la technique 1, qui est systématique mais évite tout appel de fonction, à la technique 3, qui oblige à un appel de fonction mais n'impose un *casting*, statique, qu'en cas de besoin.

3.4.5 Note sur les types ancrés

En ce qui concerne EIFFEL, les types ancrés peuvent simplifier localement les implémentations, en particulier en permettant une certaine factorisation, sans changer le principe. Toute entité de type ancré `like` p doit vérifier l'invariant 3.6, même les paramètres ou les variables locales : l'information de type est celle de p. En fait, un type ancré est son seul sous-type : une entité de type `like` p ne peut avoir reçu une valeur que par p[16], par une autre entité de type `like` p, ou par l'instanciation du type de p. Toute autre façon de faire enfreindrait la règle de sous-typage d'EIFFEL et nécessiterait un *casting* descendant (en EIFFEL, une tentative d'affectation).

Mais, contrairement à l'implémentation en sous-typage simple, même si le compilateur peut prouver qu'un *casting* descendant est sans risque d'erreur de type, il faut quand même implémenter le décalage.

3.5 Appel à `super` et `call-next-method`

En héritage multiple, l'appel à `super` n'existe pas en tant que tel, dans le cas général : en effet, « la » méthode de la super-classe n'est pas uniquement déterminée. Il existe plusieurs variantes du mécanisme : le `precursor` d'EIFFEL, la désignation explicite (:: de C++) et le `call-next-method` de CLOS, à base de linéarisation [Steele, 1990 ; Ducournau *et al.*, 1995].

Le `precursor` d'EIFFEL diffère du `super` de JAVA ou SMALLTALK sur deux points : le mécanisme s'applique en héritage multiple mais uniquement en l'absence de conflit et il ne permet pas de changer le nom de la méthode, corrigeant ainsi le défaut de `super`. L'implémentation de `precursor` se ramène donc à celle de la désignation explicite de C++ : elle ne pose pas plus de problèmes que `super` en héritage simple, si ce n'est qu'elle impose un *casting* ascendant sur le receveur. En revanche, la désignation explicite pose, dans le cas général, un problème de double évaluation, analogue à l'héritage répété des attributs (cf. section 4.2), qui la rend délicate à utiliser : si la méthode `m` de D fait appel à `B` :: `m` et à `C` :: `m`, et que chacune de ces dernières fait appel à `A` :: `m`, alors `A` :: `m` sera exécuté deux fois lors de l'appel de `D` :: `m`.

Le `call-next-method` de CLOS consiste à appeler la méthode suivante dans l'ordre de la linéarisation des super-classes du type dynamique du receveur. Cette méthode suivante va donc dépendre du type dynamique (τ_d) et pas seulement du type statique (τ_s). L'implémentation ne peut donc pas se ramener statiquement à une désignation explicite. CLOS est un langage au typage dynamique, qui sort du cadre de cet article, et, à notre connaissance, ce mécanisme de `call-next-method` n'existe dans aucun langage à typage statique. Mais ce serait parfaitement envisageable, d'autant que 1) ce mécanisme est une bonne solution au problème de double évaluation causé par la désignation explicite, 2) l'appel des constructeurs et destructeurs repose déjà, en C++, sur une linéarisation [Huchard, 2000].

L'implémentation la plus simple consistera à allouer, dans la table des méthodes, un indice pour le `call-next-method` de chaque méthode. Cette nouvelle partie des tables sera remplie suivant un procédé différent : alors que, pour une méthode normale (« primaire », en CLOS), les tables associées aux différents types statiques contiennent la même valeur (figure 3.2), elles contiendront, pour un `call-next-method`, des valeurs toutes différentes. Comme les indices des méthodes sont propres à chaque type statique, les entrées des `call-next-method` n'auront besoin de figurer dans les tables que pour les méthodes qui y font explicitement appel : ce nouveau mécanisme ne coûterait que si l'on s'en sert.

Ce serait une bonne façon d'implémenter les constructeurs et destructeurs, comme des méthodes chaînées par appels successifs à `call-next-method`. Le constructeur serait dans ce cas appelé sur un objet fraîchement alloué et dont seuls la structure et les pointeurs sur les tables de méthodes seraient initialisés. L'implémentation effective de C++ est différente, puisqu'il est patent que les constructeurs n'ont pas accès à un objet définitivement typé : un appel de méthode depuis un constructeur semble se faire sans liaison tardive[17].

[16] Est-ce encore sûr ? C'est raisonnable, mais cela introduit une certaine dissymétrie.

[17] En EIFFEL, les méthodes de création n'ont pas ce défaut. De plus, comme ce sont des méthodes normales, leur appel ne repose

3.6 Pointeurs sur `null`

L'implémentation de l'héritage multiple permet de faire un compromis entre les deux implémentations proposées pour l'héritage simple : un `null` global ou un par type. On peut définir une classe $\perp$ absurde, sous-classe de toutes les classes, dont l'unique instance est initialisée à elle-même, modulo le *casting* ascendant convenable.

3.7 Variables de classes et héritage non monotone

Pour les variables de classes (`:allocation :class`), l'existence de plusieurs tables de méthodes pour une même classe ne complique pas le problème, puisque l'implémentation en héritage simple imposait déja un partage, par un *wrapper*, entre les tables des classes et des sous-classes.

En ce qui concerne l'héritage non monotone, il n'est plus possible de franchir la barrière des sous-objets pour mettre un attribut d'une classe dans le trou laissé par un attribut non hérité d'une super-classe. Il est donc toujours possible de ne pas hériter certains attributs, mais le gain de place ne sera effectif que s'il s'agit des derniers attributs d'un sous-objet.

3.8 Evaluation

Le surcoût de l'héritage multiple est à la fois évident et potentiellement important :

- le surcoût spatial dynamique dans chaque objet est égal au nombre de super-classes de la classe de l'objet ;
- la taille des tables de méthodes n'est plus linéaire dans le nombre de classes, mais potentiellement quadratique : le nombre de tables est lui-même quadratique[18] ; de plus, il n'y a pas de possibilité immédiate de partage puisque les Δ ne sont jamais invariants ; dans un contexte d'héritage massivement « non virtuel », [Driesen et Hölzle, 1995 ; Driesen, 1999] rapporte des facteurs supérieurs à 3, semble-t-il sur le nombre d'entrées des tables ;
- toute affectation ou passage de paramètres qui n'est pas à type statique constant exige un *casting* ;
- l'efficacité de l'accès aux attributs est significativement réduite puisqu'il faut passer par un décalage stocké dans la table de méthodes ;
- l'efficacité de l'appel de méthode lui-même est plus discutable parce que le décalage, qui représente un surcoût effectif, peut s'effectuer dans les latences du processeur (*pipe-line*) ou en parallèle (processeur super-scalaire) [Driesen, 1999][19] ; de même que les décalages nécessaires sur les paramètres ; mais la remarque qui suit sur les *thunks* laisse à penser que le surcoût est sérieux ;
- les décalages on un effet spatial non négligeable, sur le code ou sur les tables statiques, de l'ordre du doublement ;
- le *casting* descendant ne se fait en temps constant qu'au prix de très grandes tables de fermeture transitive.

Les expérimentations de [Driesen, 1999] semblent donner un léger avantage à la technique des *thunks* d'un point de vue temporel. Mais cette conclusion vaut pour des programmes C++ classiques, qui pratiquent un héritage majoritairement « non virtuel » (cf. section 4.2), dans lesquels les décalages entre types différents peuvent souvent être nuls. Ce sont donc ces décalages nuls, qui disparaissent en pratique, qui rendent les *thunks* plus efficaces, ce qui tend à prouver que dans la technique sans *thunk* (cf. page 18), le décalage a un surcoût sensible[20]. Ces expérimentations ne permettent donc pas de conclure pour l'implé-

pas sur une linéarisation. JAVA a les mêmes constructeurs que C++, sans le défaut de typage, et l'héritage simple rend la linéarisation triviale.

[18] Si l'on admet une répartition uniforme des méthodes dans les classes, la taille des tables est même cubique dans le nombre de classes, au lieu d'être quadratique en héritage simple. Notons qu'en sous-typage simple, l'indiçage des signatures proposé par [Myers, 1995] a l'inconvénient de rajouter, au moins marginalement, ce facteur quadratique.

[19] Bien entendu, il reste toujours possible d'imaginer insérer, en héritage simple, des instructions supplémentaires dans ces temps morts du processeur.

[20] Il est probable que ce raisonnement peut s'étendre à d'autres mécanismes : on ne peut tabler sur la parallélisation que lorsque l'on est sûr qu'aucune séquence ne s'exécute déjà en parallèle : une telle hypothèse est difficile à soutenir, lorsque l'on ne considère qu'une séquence isolée.

mentation standard[21]. D'un point de vue spatial, les deux techniques sont à peu près à égalité : il y moins de *thunks* que d'entrées dans les tables, mais ils occupent 2 mots au lieu d'un. De plus, la séquence de l'envoi de message a deux instructions de moins.

Cette implémentation présente plusieurs inconvénients notables. D'abord, son surcoût est important, même lorsque l'on ne se sert pas de l'héritage multiple : la compilation séparée ne permet pas de savoir si une classe sera spécialisée en héritage multiple ou seulement en héritage simple. C'est là le défaut primordial.

Ensuite, le surcoût de l'implémentation est très notablement augmenté — à peu près doublé — dès que l'on accepte des redéfinitions covariantes, même dans le cas parfaitement sûr du type de retour. Cela étant, tout le traitement de la redéfinition covariante des méthodes peut être intégré dans les *thunks*, sans en compliquer la combinatoire — un par couple (τ_s, τ_d) —, avec l'avantage de n'imposer le surcoût d'un *casting* qu'en cas de besoin, c'est-à-dire pour les redéfinitions effectives entre τ_s et τ_d (comme en héritage simple avec la technique de [Myers, 1995], mais sans avoir besoin de complexifier l'indiçage). Dans ce cas particulier, la technique des *thunks* semble s'imposer. En revanche, pour les attributs, les *thunks* sont à éviter pour éviter une implémentation par de vrais accesseurs et un appel de méthode supplémentaire (cf. section 4.4). En contrepartie, les *castings* doivent être faits systématiquement.

Enfin, cette implémentation de l'héritage multiple augmente le coût de la gestion mémoire : un *garbage collector* classique ne permet en général pas de pointer au milieu d'un objet (cf. annexe A.4).

[21] Ce que l'on pourrait appeler l'héritage multiple « virtuel », mais ce que l'on ne fera pas pour ne pas s'incliner devant C++, qui oblige en permanence à sur-spécifier les notions naturelles et à considérer les aberrations comme normales.

Chapitre 4

Alternatives pour l'héritage multiple

Cette complexité de l'implémentation de l'héritage multiple explique à la fois certains mauvais côtés de C++, le fait que de nombreux langages, comme JAVA, aient renoncé à l'héritage multiple, ainsi que la recherche de techniques alternatives, dont la plus radicale est celle du compilateur SMALL EIFFEL — que nous examinerons plus loin (cf. section 7.1) — qui évite l'usage de tables de méthodes, au prix d'une compilation globale [Zendra *et al.*, 1997].

Les alternatives que nous considérons ici consistent soit à privilégier un type d'efficacité (temporelle, spatiale statique ou dynamique) par rapport à un autre (sections 4.1 et 4.3), soit à chercher une implémentation qui s'applique sans surcoût à l'héritage simple, au prix de la généralité ou de la sémantique de l'héritage multiple (section 4.2), soit enfin à réduire, sans l'annuler, la question de l'invariance des indices des méthodes et des attributs (sections 4.4 et 7.1.2).

4.1 Variante avec partage des tables

La présence des décalages fait qu'aucun partage de table n'est possible, dans l'implémentation proposée jusqu'ici. Il existe néanmoins un moyen de partager les tables, en faisant le décalage $\Delta_{\tau_s,t}$ en deux temps, comme pour les attributs (3.3) ou pour le passage de paramètres non invariants (3.10) :

$$\Delta_{\tau_s,t} = \Delta_{\tau_s,\tau_d} + \Delta_{\tau_d,t} = \Delta^{\tau_s}_{\Downarrow} + \Delta_{\tau_d,t} \tag{4.1}$$

Le premier décalage ramène au type dynamique et le second ne dépend plus du type statique.

Supposons que la méthode qui doit être utilisée pour le type dynamique τ_d ait été définie dans la classe t. Pour l'implémentation classique, dans les tables associées aux différents types statiques τ_s, l'adresse de cette méthode est constante mais associée à des $\Delta_{\tau_s,t}$ qui sont tous différents. Avec le partage, cette adresse sera associée dans les tables à l'unique décalage $\Delta_{\tau_d,t}$, quel que soit le type statique τ_s considéré.

Mais le décalage $\Delta^{\tau_s}_{\Downarrow}$ doit lui-même être stocké dans une structure propre à τ_s : le partage impose donc une indirection supplémentaire (figure 4.1). Les tables de méthodes ne contiennent plus qu'un décalage depuis τ_d : l'implémentation par des *thunks* ne nécessite donc plus qu'un *thunk* par classe, sauf en cas de redéfinition covariante (cf. note 4, page 18 et section 3.4). Deux variantes sont alors possibles.

4.1.1 Partage : variante 1

Dans la première variante (figure 4.1, milieu), la table intermédiaire pointe sur le début d'une table de méthodes et le partage repose sur le fait qu'à chaque classe est associé un ordre sur ses super-classes, ordre utilisé aussi bien pour les tables de méthodes, qui sont ordonnées par blocs correspondant à chacune des super-classes, que pour la table $\Delta^{\uparrow}_{\tau_s}$ (cf. annexe A.2).

Invariant 4.1 *2 super-classes de τ_d se partagent les tables de méthodes ssi l'ordre des super-classes de l'une est section commençante de l'ordre des super-classes de l'autre.*

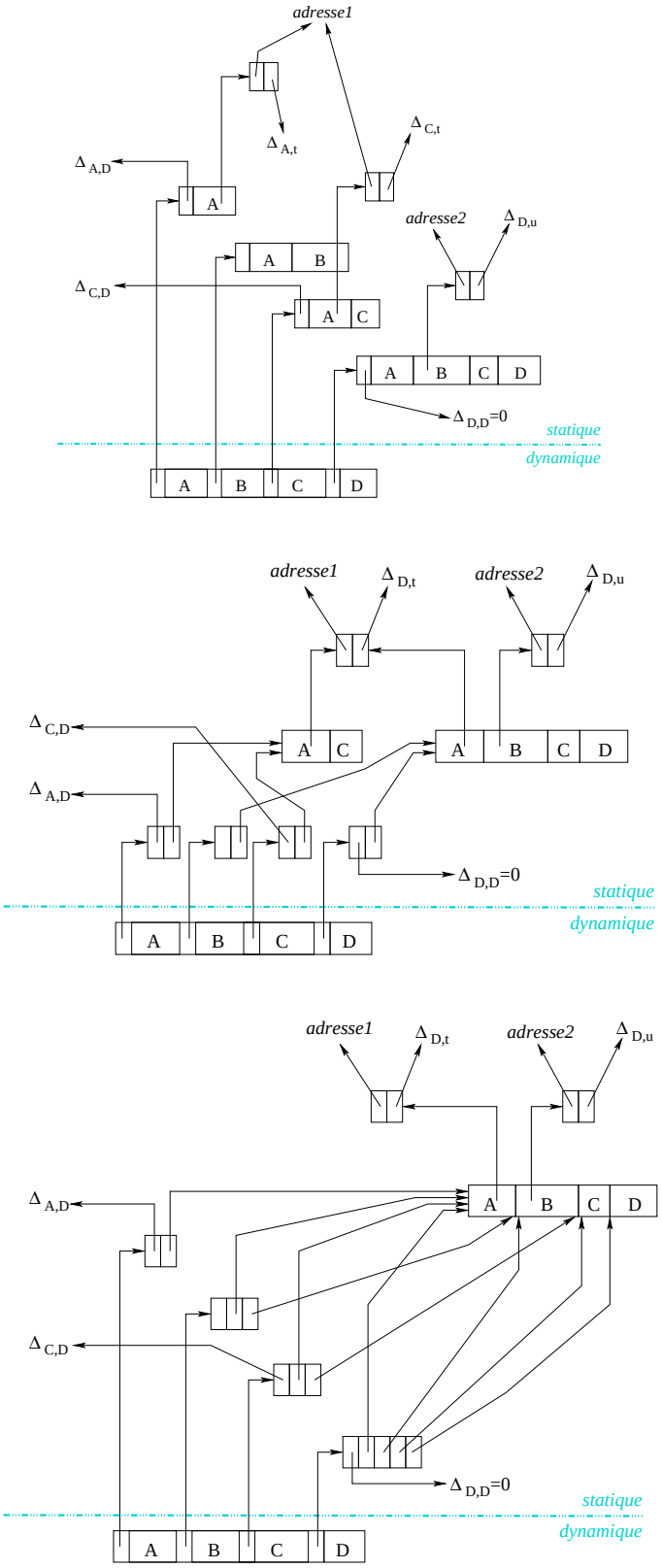

FIG. 4.1 – Tables de méthodes avec partage (milieu et bas) ou sans (haut) : $\Delta_{C,t}$ (haut) $= \Delta_{C,D} + \Delta_{D,t}$ (bas).

Les tables de *casting* $\Delta^{\uparrow}_{\tau_s}$ peuvent, elles aussi, être partagées[1], à condition qu'elles contiennent des décalages relatifs à τ_d et que le *casting* se fasse en deux temps.

Toutes les paires adresse-Δ ou table-Δ de la figure peuvent être implémentées directement dans les tables qui les référencent, en évitant ainsi une indirection. Mais ce serait contradictoire de le faire pour les paires table-Δ : elles sont référencées directement par les objets et intégrer les Δ dans les objets reviendrait à augmenter très significativement la taille dynamique, alors que l'on cherche à réduire la taille statique[2]. Par ailleurs, les entrées des `call-next-method` ne peuvent pas forcément être partagées : il faut les stocker dans les tables intermédiaires qui ne se réduisent donc pas à la paire table-Δ.

4.1.2 Partage : variante 2

Dans la seconde variante (figure 4.1, bas), il n'y a plus qu'une table de méthodes, où les méthodes sont regroupées par blocs correspondant aux classes qui les introduisent. La table intermédiaire ne pointe plus sur le début de la table, mais sur chacun des blocs, chaque super-classe de τ_s ayant un indice indépendant de τ_d, comme pour les tables $\Delta^{\uparrow}$. L'invariant des méthodes est alors le même que celui des attributs. Le compactage est encore plus efficace pour les tables de méthodes, mais les $\Delta^{\uparrow}$ ne peuvent plus être partagés.

4.1.3 Evaluation

Dans les deux variantes, l'envoi de message se compile par la même séquence de huit instructions, la différence portant sur la valeur de `#table2Offset` :

```
load [object + #table1Offset], table1
load [table1 + #table2Offset], table2
load [table1 + #delta1Offset], delta1
load [table2 + #delta2Offset], delta2
add object, delta1, object                          3L + B + 1
load [table2 + #selectorOffset], method
add object, delta2, object
call method
```

Quant à l'accès aux attributs, si les tables de *casting* sont partagées, il se fera comme suit :

```
load [object + #table1Offset], table1
load [table1 + #table2Offset], table2
load [table1 + #delta1Offset], delta1
load [table2 + #castOffset], delta2                 4L + 1
add object, delta1, object
add object, delta2, object
load [object + #attributeOffset], attribute
```

Un partage des tables est donc possible : il nécessite une indirection et un décalage supplémentaires à l'exécution, ce qui entraîne fatalement une moindre efficacité temporelle[3] et une augmentation de la place prise par le code généré pour chaque appel de méthode, ce qui peut contredire l'objectif. Pour la variante 1, le nombre et la taille des tables ainsi obtenues est du même ordre qu'en héritage « non virtuel » (cf. section 4.2 et figure 4.3), alors qu'il égal à celui de l'héritage simple pour la variante2, au prix d'une légère augmentation des tables intermédiaires. Le surcoût statique lié au nombre et à la taille des tables est réduit, mais le surcoût dynamique, lié au nombre de tables référencées par chaque objet, demeure. Enfin, la redéfinition covariante nécessitera de faire appel à la technique de [Myers, 1995] utilisée en sous-typage simple si l'on veut éviter des vérifications de type trop systématiques.

[1] Le partage de ces tables est en fait indépendant du partage des tables de méthodes : il peut donc se faire aussi bien sur l'implémentation de base.

[2] Les implémentations qui n'ont pas ces scrupules de bonne économie de la mémoire dynamique (cf. section 4.3) ne sont pas concernées par ces petites économies de mémoire statique.

[3] Le double décalage n'entraîne pas de surcoût car il peut se faire en parallèle ou dans la latence du processeur. En revanche, l'indirection supplémentaire va coûter un temps de latence supplémentaire, soit 2-3 cycles.

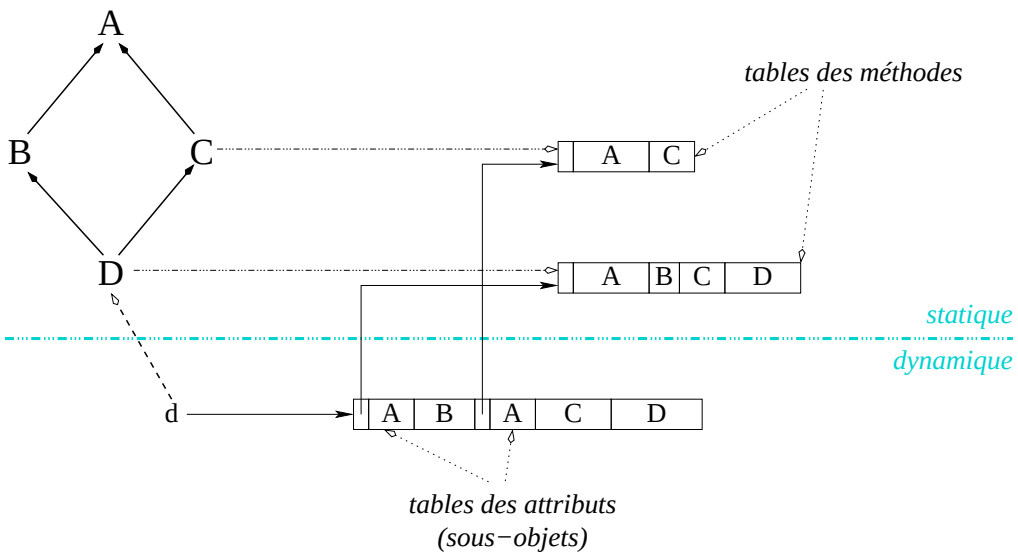

FIG. 4.2 – Tables des attributs et des méthodes en héritage multiple « non virtuel »

4.2 L'héritage multiple non virtuel de C++

L'implémentation de C++ telle qu'elle est décrite dans [Ellis et Stroustrup, 1990] diffère de celle que nous avons présentée ici sur deux points :

- elle est plus complexe à exposer et à implémenter car elle traite aussi bien l'héritage « virtuel » que « non virtuel », suivant l'usage qui est fait du mot-clé `virtual` ;
- son surcoût est moindre dès qu'il n'est pas fait usage de ce mot-clé `virtual` ;
- le mélange d'héritage virtuel et non virtuel, ajouté aux protections d'héritage, donne une complexité telle à l'héritage qu'il est fortement déconseillé d'en exploiter toute la combinatoire.

Le principe d'implémentation de l'héritage non virtuel est le suivant[4] :

- une classe sans super-classe s'implémente exactement comme en héritage simple ;
- en cas d'héritage, qu'il soit simple ou multiple, l'implémentation des instances de la sous-classe est obtenue par concaténation des implémentations des instances des super-classes directes[5], prolongée par les attributs introduits par la sous-classe : la nouvelle classe a autant de tables de méthodes que ses super-classes directes réunies, les méthodes qu'elle introduit prolongeant la table de la première de ses super-classes.

Invariant 4.2 *Le sous-objet associé à τ_d est concaténé au sous-objet de l'une de ses super-classes directes non virtuelles et leurs tables de méthodes sont communes.*

En cas d'héritage simple, l'implémentation est donc exactement celle du sous-typage simple, modulo la présence des décalages, bien qu'ils soient tous nuls : la technique des *thunks* semble donc plus adaptée, puisqu'elle fait alors disparaître, en pratique, les décalages. En effet, alors qu'en héritage multiple standard, l'unique *thunk* de décalage zéro correspond au cas où la méthode est définie dans τ_s (cf. note 4, page 18), dans l'héritage multiple non virtuel, le cas est beaucoup plus fréquent et c'est toujours vrai pour les cas d'héritage simple.

En cas d'héritage multiple effectif, la compilation des envois de messages ou des accès aux attributs est exactement la même qu'en héritage multiple : la seule différence est que le recours effectif aux décalages est moins fréquent pour les attributs.

[4] L'auteur s'excuse de décrire une fonctionnalité par son implémentation, mais il s'agit ici d'un cas typique où l'organe a créé la fonction.

[5] Dans le cas d'un mélange d'héritage virtuel et non virtuel, les super-classes indirectes virtuelles sont exclues de cette concaténation et rajoutées à la fin.

4.2.1 Héritage répété

L'absence du mot-clé `virtual` a un effet très positif sur le coût, mais très négatif sur la sémantique de l'héritage multiple et sur la réutilisabilité. Dans l'exemple de la figure 3.1, lorsque l'héritage de la classe A par B et C n'est pas virtuel, la classe A est dupliquée dans l'objet, c'est-à-dire que deux sous-objets A sont incorporés physiquement dans les sous-objets B et C : on parle alors d'*héritage répété*[6]. Dans le cas contraire, lorsque le graphe d'héritage d'une classe est une arborescence, on parlera d'*héritage arborescent*[7].

Dans tous les cas, il n'est alors nécessaire de créer des tables de méthodes supplémentaires qu'en cas d'héritage multiple : lorsqu'une classe à k super-classes directes, il faut $k - 1$ tables de méthodes supplémentaires (figure 4.2). Si l'héritage est arborescent, le nombre total de tables est égal au nombre de classes sans super-classes.

4.2.2 *Casting*

Lorsqu'il n'y a pas d'ambigüité, c'est-à-dire en cas d'héritage arborescent, le *casting* ascendant s'effectue statiquement et le *casting* descendant s'implémente comme en héritage simple. En revanche, la présence d'héritage répété impose un *casting* ascendant désambigüisé par des classes intermédiaires : le *casting* implicite est impossible. Quant au *casting* descendant, il est réalisable mais par des moyens plus compliqués, la classe cible pouvant être répétée (indépendamment du fait que la classe source soit ou non répétée).

[Ellis et Stroustrup, 1990, section 10.6c] refuse le *casting* descendant depuis une classe virtuelle. De fait, l'implémentation que nous avons proposée (table $\Delta^{\Uparrow}$) est valide si l'héritage est virtuel ou arborescent, mais elle ne marche pas si une classe cible est répétée, ce qui reste possible pour n'importe quelle classe source. Lorsque le chemin d'héritage entre la cible et la source est non virtuel, le *casting* pourrait se traiter statiquement, comme en héritage simple par la méthode des *tags*. Mais C++ faisant un *casting* à l'aveugle, sans vérification de type, aucune implémentation particulière n'est nécessaire : le décalage est statique.

Ce qui précède concerne l'ancien style de *casting*, à la C. Dans le nouveau style avec des opérateurs spécifiques — dont `static_cast` correspond à l'ancien style —, `dynamic_cast` serait l'opérateur à utiliser pour un *casting* descendant avec vérification de type. Si l'on en juge par les spécifications de [Koenig, 1998], l'implémentation de `dynamic_cast` repose à la fois sur les tables $\Delta^{\tau_s}_{\downarrow}$ (en cas d'héritage répété) et $\Delta^{\Uparrow}$ (pour le *casting* latéral). Si la classe cible est non ambigüe dans l'une des 2 tables, le *cast* réussit : aucune interdiction *a priori* (statique) n'est donc nécessaire.

Enfin, tout ce qui concerne la redéfinition de types fonctionne comme en héritage virtuel, tant qu'il n'y a pas d'ambigüité. Cependant, comme chaque type statique n'a pas sa propre table de méthodes, il peut être nécessaire de recourir à la technique de [Myers, 1995] présentée pour le sous-typage simple (page 11), si l'on veut éviter des vérifications de type systématiques.

4.2.3 Evaluation

L'héritage non virtuel apporte donc une simplification importante de l'implémentation, en particulier lorsqu'il se réduit à l'héritage simple. Il a l'avantage de ne coûter que si l'on s'en sert — si on l'implémente par des *thunks*, les expériences de [Driesen, 1999] montrant que les décalages systématiques ont un coût

[6] [Meyer, 1997] a un usage un peu différent du terme d'héritage répété, qui désigne la figure d'héritage en losange (figures 3.1 et 4.2) : EIFFEL permet alors de dupliquer certaines des propriétés de la racine du losange, après un renommage, de même qu'il permet, inversement, de fusionner des propriétés différentes héritées de diverses super-classes. L'implémentation en reste mystérieuse. EIFFEL permet aussi un héritage répété d'une même super-classe directe, modulo un renommage adéquat pour éviter les conflits. Au-delà de la curiosité sémantique, ce trait semble parfaitement redondant avec la possibilité d'expansion des attributs (mot-clé `expanded`, cf. section A.4). Un constat analogue est possible avec l'héritage `private` de C++ qui a tendance à tourner à l'agrégation. Le terme de *composant* (*component*) ou de membre (*member*) est courant, dans les langages et les méthodes, pour désigner les propriétés (attributs ou méthodes) des classes ou des objets. Encore faut-il distinguer les deux, le logique et le physique.

[7] [Sakkinen, 1992] qualifie cet héritage de *independant multiple inheritance* (IMI), par opposition à un *shared multiple inheritance* aussi appelé *fork-join inheritance* (FJI). L'auteur présente en particulier une discussion approfondie sur les rapports entre l'héritage et la protection, dont il tire quelques règles pour lier la « virtualité » et la protection. L'adoption de ces règles améliorerait sensiblement la sémantique du langage mais ne provoquerait aucun changement significatif dans l'implémentation, qui resterait entre les deux bornes des héritages virtuels et non virtuels.

type statique → ↓ dynamique	A	B	C	D
A	| A |	—	—	—
B	B	| A | B |	—	—
C	C	—	| A | C |	—
D	C/D	D	| A | C |	| A | B | C | D |

FIG. 4.3 – Tables des méthodes, avec partage de tables ou en héritage non virtuel : une classe fait référence à la table qui lui est associée en tant que type statique, sur la même ligne.

non négligeable —, contrairement à l'héritage multiple général dont le surcoût est indépendant du fait que l'on s'en serve ou pas. Mais l'héritage non virtuel a plusieurs inconvénients majeurs :

- l'héritage répété, c'est-à-dire la duplication de la classe A dans la classe D, n'a aucune justification sémantique : il faudrait donc restreindre cet héritage non virtuel à un héritage arborescent ;
- une compilation séparée ne permet pas de savoir que les classes B et C n'auront pas de spécialisation commune : seul un diagnostic *a posteriori* est possible, en interdisant la spécialisation de deux classes qui ont une super-classe commune non virtuelle ;
- lorsque le langage définit une classe `Object`, racine de la spécialisation et classe de tous les objets, l'héritage arborescent se réduit à un héritage simple.

Si l'on considère que l'héritage répété est une abomination — ce qui semble une opinion assez partagée [Eckel et Gil, 2000] — et donc que la différence entre `virtual` et non `virtual` n'a de raison que d'implémentation, il est possible de déterminer, par une analyse statique globales, les liens d'héritage qui doivent être virtuels et ceux pour lesquels c'est inutile [Eckel et Gil, 2000]. Sur l'exemple des figures 3.1 et 4.2, le gain est faible : il est juste possible de fusionner les sous-objets D et B (ou C). Dans la figure 3.2, seule une table de la dernière ligne est économisée.

4.2.4 Critique de C++

Le modèle d'héritage multiple non virtuel vient de SIMULA [Krogdahl, 1985]. L'héritage multiple de C++ a été abondamment commenté et critiqué [Cargill, 1991 ; Waldo, 1991 ; Sakkinen, 1992], mais souvent pour des raisons opposées. [Cargill, 1991] critique ainsi surtout l'héritage virtuel et il est particulièrement significatif que la réponse de [Waldo, 1991] repose en grande partie sur un exemple qui relève explicitement plus du sous-typage multiple que de l'héritage multiple.

La critique principale que l'on peut faire à C++ est que le langage est globalement incompréhensible sans une compréhension fine de son implémentation, en particulier pour ce qui concerne héritage virtuel et non virtuel, *casting* en particulier descendant, redéfinition du type de retour, etc., toutes choses autorisées, mais pas toujours. Et de fait, les programmeurs C++ pensent plus en termes (éventuellement faux) d'implémentation que de concepts. Dit autrement, les concepts de C++ sont plus ceux de C que ceux de la programmation par objets.

Cette implémentation de l'héritage non virtuel, sans modèle ni sémantique, a donné lieu à diverses tentatives de lui donner un sens *a posteriori*, en particulier avec des modèles de sous-objets [Rossie et Friedman, 1995 ; Rossie *et al.*, 1996 ; Snyder, 1991].

4.3 Réduire les indirections

A l'opposé des alternatives précédentes qui cherchent à réduire l'occupation mémoire statique, certaines optimisations cherchent à éviter des indirections en implémentant une partie des tables de classes dans les objets, au détriment de l'occupation mémoire dynamique : la table de décalage nécessaire au *casting* ascendant, $\Delta^{\uparrow}_{\tau_s}$, est la principale visée. Elle sert en effet en permanence pour les accès aux attributs — dès lors qu'ils n'ont pas été définis dans la classe courante (τ_s) même —, les passages de paramètres et les

affectations polymorphes[8]. On a vu que l'implémentation usuelle de $\Delta^{\uparrow}_{\tau_s}$ revenait à considérer les $\Delta_{\tau_s,t}$ comme des méthodes de la classe τ_s (cf. section 3.3.1, page 19). L'implémentation dans les objets revient à les considérer comme des attributs.

L'objectif de cette implémentation est de réduire l'accès à un attribut à la séquence suivante :

```
load [object + #castOffset], delta
add object, delta, object                                   2L + 1
load [object + #attributeOffset], attribute
```

voire à la séquence suivante si l'on remplace les décalages par des pointeurs :

```
load [object + #castOffset], object                           2L
load [object + #attributeOffset], attribute
```

En référence au vocabulaire C++ d'héritage virtuel et non virtuel, le contenu de ces tables est décrit comme des VBPTR, pointeurs sur des *virtual base classes*. Dans notre description de l'implémentation générale de l'héritage multiple, toutes les super-classes sont des *virtual base classes*. Ces VBPTR peuvent être de deux sortes : les `e-VBPTR` sont *essential* en ce qu'ils référencent des super-classes directes, alors que les `i-VBPTR` sont *inessential* car ils référencent des super-classes indirectes. Bien entendu, une indirection par un `i-VBPTR` se réduit, statiquement, à quelques indirections par des `e-VBPTR` : l'implémentation des seuls `e-VBPTR` dans l'objet imposerait aux `i-VBPTR` un coût identique, voire supérieur, à celui des tables $\Delta^{\uparrow}$. La cohérence suppose donc de mettre tous les VBPTR dans les objets.

Un avantage mineur de la technique est de permettre d'implémenter la classification d'instances — cas particulier de la migration (fonction générique `change-class` de CLOS) lorsqu'elle est monotone, c'est-à-dire quand la classe cible est une sous-classe de la classe source (cf. page 10). En effet, avec les VBPTR, l'implémentation d'un objet n'a plus besoin d'être connexe : spécialiser sa classe revient donc à rajouter quelques sous-objets et à changer les pointeurs sur les tables de méthodes.

Evaluation et optimisations

Or ces tables, en nombre quadratique et en espace cubique, comme toutes les tables de classes dans cette implémentation standard de l'héritage multiple, peuvent occuper plusieurs dizaines, voire centaines, d'entrées. Les mettre en mémoire dynamique, c'est-à-dire dans chaque objet, dupliquées autant qu'il y en a, peut avoir un coût très significatif.

[Gil et Sweeney, 1999 ; Eckel et Gil, 2000] présentent des statistiques assez développées sur plusieurs grandes hiérarchies de classes souvent utilisées dans les *benchmarks* et proposent diverses optimisations pour réduire la place dynamique occupée par ces VBPTR.

Certaines de ces optimisations sont globales et résistent mal à une comparaison avec les implémentations basées sur une compilation globale (cf. section 7.1), que ce soit avec l'heuristique de coloration (cf. section 7.2) ou avec les techniques de SMALL EIFFEL (cf. section 7.1.6) : la coloration réduit très fortement le nombre de décalages nécessaires et SMALL EIFFEL les éliminent tous.

D'autres optimisations sont utilisables en compilation séparée : elles utilisent la bidirectionnalité potentielle des tables que l'on retrouvera section 5.2.

4.4 Compilation des attributs par des accesseurs

Certains langages comme EIFFEL, SATHER ou CLOS traitent les attributs un peu comme des méthodes :
- en EIFFEL [Meyer, 1992 ; Meyer, 1997], il est, par exemple, possible de redéfinir une méthode sans argument comme un attribut ;
- en CLOS [Steele, 1990], on accède aux attributs par des *accesseurs* en lecture et écriture : ces accesseurs sont des méthodes (ou fonctions génériques dans la terminologie de CLOS) ;
- en SATHER [Szypersky *et al.*, 1994 ; Omohundro et Stoutamire, 1995], un attribut contribue au type d'une classe par les signatures de ses fonctions d'accès.

[8] De tous les mécanismes fondamentaux, seul l'appel de méthode s'en passe puisque la classe contenant la méthode n'est pas connue statiquement et qu'il faut donc expliciter le décalage dans la table des méthodes.

Une implémentation des objets s'en déduit, en abandonnant toute hypothèse d'invariance des indices des attributs, qui restent encapsulés dans les accesseurs : lorsque l'indice d'un attribut change dans une sous-classe, les accesseurs correspondants sont redéfinis. Bien entendu, tous ces accesseurs peuvent, voire doivent, être définis automatiquement par le compilateur, sans que le programmeur ait à s'en préoccuper[9]. L'invariant obtenu est alors :

Invariant 4.3 *L'indice des attributs est indifférent.*

Le principe étant posé, deux implémentations distinctes sont à considérer : les accesseurs peuvent être effectivement définis comme de véritables méthodes ou bien simulés, la table des méthodes ne contenant pas l'adresse d'une méthode mais la position de l'attribut dans l'objet. Cette simulation des accesseurs est plus satisfaisante, dans la mesure où elle évite un véritable appel de méthode. La séquence d'accès est alors la suivante :

```
load [object + #tableOffset], table
load [table + #attributeOffset], offset                    3L + 1
add object, offset, place
load place, attribute
```

Elle a exactement la même complexité que pour l'accès aux attributs, dans le cas général avec *casting* ascendant.

On constate alors que ces tables de conversion des indices peuvent être heureusement factorisées si les attributs sont regroupés par les classes qui les introduisent, ce qui ramène à une implémentation par sous-objets et à l'invariant 3.1. La table des méthodes doit alors contenir les Δ aux différents sous-objets, à partir du début de l'objet, et l'accès aux attributs se fait exactement comme en héritage multiple en cas de *casting* ascendant (cf. page 19).

```
load [object + #tableOffset], table
load [table + #castOffset], place
add object, place, place
load [place + #attributeOffset], attribute
```

La simulation des accesseurs est ainsi un peu un leurre, puisqu'elle ramène inéluctablement à l'implémentation par sous-objets, avec ce défaut que le décalage est nécessaire dans tous les cas. Pour économiser des décalages dans les accès aux attributs de `self`, [Myers, 1995] propose une double compilation des classes : dans l'une, les accès aux attributs de `self` se font avec décalage, alors que dans l'autre, ils se font sans, en supposant que les indices des attributs ne seront pas modifiés dans les sous-classes (par exemple, si la classe n'est spécialisée qu'en héritage simple). La bonne version est choisie à l'édition de liens.

Cependant, conserver des tables de conversion peut être une solution pour implémenter des mécanismes moins orthodoxes (héritage non monotone ou répété à la EIFFEL).

Mais, rajouter les accesseurs à l'implémentation par sous-objets ne procure qu'une simplification limitée : le problème de l'invariance des indices d'attributs est réglé ... à condition d'avoir réglé celui des méthodes. Si l'objet lui-même peut être constitué d'une unique table d'attributs, il reste toujours à trouver un accès direct aux méthodes. On verra plus loin plusieurs façons de résoudre ce problème, l'une qui conserve la non invariance de référence à l'objet (cf. section 5.2.3), les autres qui retrouvent au contraire l'invariance de l'héritage simple, au prix, soit d'un calcul global (cf. section 7.2), soit d'un flot de tables de méthodes (cf. section 5.3.2). De façon générale, dès que le traitement des méthodes est résolu, cette technique est applicable aux attributs.

En compilation séparée, les accesseurs sont apparemment la seule façon d'implémenter le mot-clé `:allocation` de CLOS avec une économie effective, ainsi que diverses fonctionnalités d'EIFFEL, comme la redéfinition d'une méthode par un attribut, ou le traitement de l'héritage répété (cf. section 4.2) où les attributs d'une même classe sont individuellement partagés ou répliqués. Mais la première de ces deux fonctionnalités nécessite de véritables accesseurs, leur simulation étant insuffisante, alors que la seconde semble nécessiter des tables de conversion, le décalage étant insuffisant.

[9] On ne confondra pas cette implémentation par accesseurs avec le style de programmation consistant à encapsuler tous les accès aux attributs dans des accesseurs redéfinissables dans les sous-classes. La présence d'accesseurs définis pas le programmeur ne devrait pas empêcher le compilateur d'en définir pour ses propres besoins. Le langage ILOGTALK/POWER-CLASSES proposait ainsi un « protocole des accesseurs » distinguant en particulier un niveau physique et un niveau logique [ILO, 1995 ; ILO, 1996]

4.5 Comparaisons

Les quelques alternatives que nous avons présentées consistent toutes à privilégier l'un (ou deux) des trois points de vue sur l'efficacité, au détriment des autres : efficacité temporelle contre mémoire dynamique ou, au contraire, mémoire contre temps. La recopie de méthodes a l'intérêt de privilégier le dynamique au dépens du statique, espace et temps confondus, mais elle est incomplète (l'invariance des méthodes n'est pas résolue) et incompatible avec une compilation séparée. Enfin, l'implémentation de C++ cherche à réduire le surcoût de l'héritage multiple quand on ne s'en sert pas, au dépens de la réutilisation ou de la sémantique.

Pour réduire l'impact de l'héritage multiple sur les classes qui sont en héritage simple, il faudrait permettre de définir des classes *qui ne seraient jamais spécialisées en héritage multiple*, ou qui ne permettraient pas d'exprimer le motif du losange à la source du problème de l'héritage répété : une version mieux spécifiée du mot-clé `virtual` de C++[10]. Ce serait une limitation importante à la réutilisation, mais c'est déjà le cas du mot-clé `virtual` : autant que la limitation soit bien spécifiée.

Une proposition naïve consisterait à faire une double compilation des classes : une pour l'héritage simple et une autre pour l'héritage multiple, un peu comme [Myers, 1995] le propose pour les accès aux attributs sur `self`. Malheureusement, cette idée séduisante se heurte à une explosion combinatoire de spécialisation et d'utilisation qui semble la rendre impraticable. Faire une double version des classes, avec des noms différents serait plus envisageable : les deux versions ne seraient pas en relation de sous-typage (pas de substituabilité des instances de l'une par celles de l'autre), mais la combinatoire des classes utilisées continue de se poser. Tout finit par se passer comme si la présence d'un seul héritage multiple imposait de tout compiler en héritage multiple. La double compilation reste possible, mais elle ne peut concerner que les accès à `self` et elle ne supporte pas des implémentations différentes mais uniquement un usage simplifié d'une même implémentation.

Reste un mystère[11], celui de l'implémentation d'EIFFEL en compilation séparée (ISE ou ...). La sémantique de l'héritage multiple d'EIFFEL répond à peu près aux spécifications du chapitre 3, avec quelques traits particuliers qui font penser qu'il peut difficilement se contenter de l'implémentation proposée, qui est, dans ses grandes lignes, celle de l'héritage virtuel de C++ :

- la redéfinition de méthode sans paramètre en attribut nécessite des accesseurs mais n'importe quelle implémentation peut générer des accesseurs en cas de besoin ;
- la possibilité d'héritage répété sélectif de certains attributs semble un défi à l'implémentation[12] : comment les méthodes de la classe répétée peuvent-elles différencier les deux versions d'un attribut répété ? il faut manifestement qu'elles soient redéfinies ou répétées ; mais comment sont implémentées des méthodes répétées, accédant à des attributs (ou méthodes) répétés ? y a-t-il recopie de code ? si oui, comment est-ce compatible avec une compilation séparée ?
- la redéfinition covariante, extensivement utilisée avec les types ancrés et malgré la menace inapplicable des « *catcalls* polymorphes », y compris pour la généricité (cf. section 6.3), justifierait de vérifier l'invariant 2.1.

La question se pose donc de l'implémentation de l'envoi de message dans ces conditions : nous verrons par la suite d'autres modèles de compilation qui répondraient bien à ce cahier des charges, au prix d'une édition de liens spécifique (cf. section 7.1). Mais la littérature sur ces implémentations d'EIFFEL semble confidentielle.

[10] Diverses tentatives ont été faites pour éliminer le mot clé `virtual`. [Sakkinen, 1992] propose de déduire la « virtualité » des protections d'héritage, alors que [Eckel et Gil, 2000] suggère une analyse globale. Mais cela ne change rien à l'affaire.

[11] Pour reprendre la terminologie de Roland Barthes, il s'agirait plutôt d'une mystification, puisque quelqu'un connaît la réponse.

[12] SMALL EIFFEL ne l'implémente d'ailleurs pas (D. Colnet, communication personnelle). [Meyer, 1997] cite quelques grands principes d'implémentation, comme le fait qu'un attribut partagé doit l'être physiquement ou que l'héritage répété n'entraîne aucun surcoût.

Chapitre 5

En héritage simple et sous-typage multiple

Entre les deux extrêmes du sous-typage simple et de l'héritage multiple, se situe le cas intermédiaire de langages qui différencient classes et types, tout en assimilant la spécialisation de classes au sous-typage, et qui se restreignent à un héritage simple. C'est typiquement le cas de JAVA, avec des classes en héritage simple et des interfaces en sous-typage multiple. Le langage THETA a une politique assez voisine [Myers, 1995 ; Day *et al.*, 1995], de même que EIFFEL#, la version d'EIFFEL pour la plate-forme .NET de MicroSoft [Simon *et al.*, 2000]. Dans un contexte assez différent, avec sélection multiple et types calculés, le langage CLAIRE repose aussi sur un compromis analogue [Caseau et Laburthe, 1999 ; Josset et Caseau, 2001].

5.1 Les spécifications

La spécification du problème est très exactement celle de JAVA [Arnold et Gosling, 1997 ; Grand, 1997] : les classes sont en héritage simple. Elles seules peuvent définir des attributs et le corps de méthodes, et avoir des instances. Les interfaces servent essentiellement à factoriser des signatures de méthodes.

L'implémentation des attributs est celle du sous-typage simple et `self` reste invariant puisque son type statique est toujours une classe. En revanche, la question se pose pour les entités typées par une interface. En effet, les indices des méthodes sont toujours invariants par héritage mais ne le sont plus par sous-typage : les indices des méthodes d'une même interface varient donc suivant les classes qui l'implémente. L'invariant 2.2 du sous-typage simple est donc toujours vérifié, mais pour les classes seulement. De même, la vérification de types et le *casting* descendant peuvent s'implémenter comme en sous-typage simple, par la double numérotation n_1/n_2, quand la cible est une classe.

Dans un tel contexte, l'implémentation standard de l'héritage multiple serait une complication inutile : il faut manifestement conserver les invariants du sous-typage simple, tant qu'il s'agit de classes.

On a néanmoins deux catégories de solutions, suivant que l'on se ramène au sous-typage simple, par complication, ou à l'héritage multiple, par simplification.

5.2 Variante de l'héritage multiple

Il s'agit manifestement d'un cas particulier d'héritage multiple et l'on peut donc traiter le problème comme pour l'héritage multiple (voir section 3) en cherchant juste à tirer parti de la structure pour simplifier.

5.2.1 Le principe

Dans un premier temps, on aura autant de tables de méthodes que la classe implémente d'interfaces mais avec une unique table pour la classe, ce qui permet de conserver l'invariance des indices des attributs :

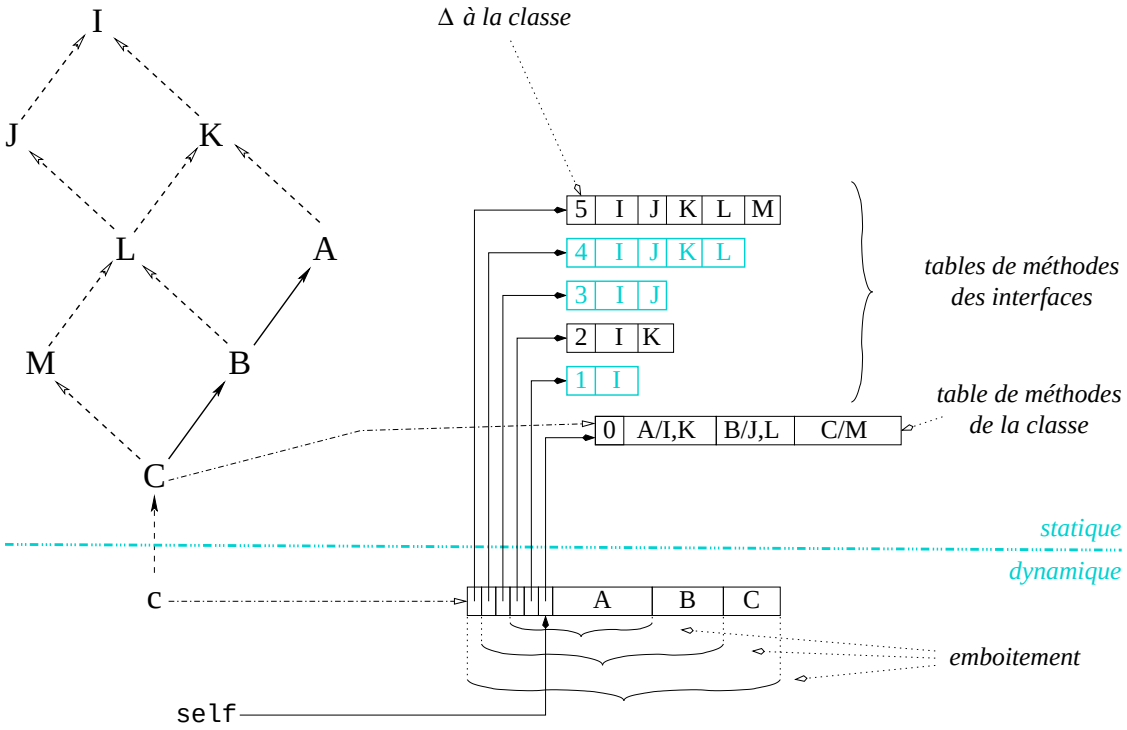

FIG. 5.1 – Sous-typage multiple et héritage simple : variante de l'héritage multiple, avec 3 classes A, B et C et 5 interfaces, I, J, K, L et M : variante à la C++. En grisé, les tables d'interfaces qu'une optimisation peut éliminer.

l'objet n'a plus une table de méthodes (comme en héritage simple), ou une table de méthodes par sous-objet (comme en héritage multiple), mais une table de tables de méthodes, qui constitue son en-tête. Chaque table de méthode commence par le décalage entre cette table et la table de classe (figure 5.1).

Une première optimisation consiste alors à ordonner les tables de méthodes, dans l'en-tête, pour que l'implémentation de la super-classe s'emboîte dans celle de la sous-classe : la sous-classe rajoute des tables en tête, pour les interfaces qu'elle implémente que sa super-classe n'implémente pas, et des attributs en queue.

Invariant 5.1 *L'implémentation de la super-classe s'emboîte dans celle de la sous-classe.*

Toute entité typée par une classe pointe sur l'unique table de méthodes de classe de l'objet, la dernière de l'en-tête : `self` est donc bien invariant. L'invariant 2.1 du sous-typage simple est vérifié, mais seulement pour les entités typées par une classe. Toute entité typée par une interface pointe dans l'en-tête, sur la table de méthodes correspondante : l'invariance de pointeur sur les objets n'est donc pas générale.

Une seconde optimisation permet ensuite de factoriser des tables. On constate en effet, dans l'exemple de la figure 5.1, que les tables de méthodes des interfaces peuvent être partagées : dans A (resp. B), celle de K (resp. L) vaut aussi pour I (resp. J). Il suffit pour cela que les indices des méthodes de K (resp. L) soient compatibles avec ceux de I (resp. J), mais c'est la manière la plus naturelle de calculer ces indices, par exemple suivant une extension linéaire qui ordonne les différentes interfaces comme pour l'implémentation de l'héritage multiple avec partage de tables (cf. section 4.1 et annexe A.2). L'ordre pour L sera soit $\{I, J, K, L\}$, soit $\{I, K, J, L\}$: l'une des deux tables de J ou de K sera donc incluse dans celle de L[1]. L'invariant 4.1 vaut donc ici aussi, mais pour les interfaces seulement.

Pour une entité typée par une classe, l'envoi de message se réalise comme en héritage simple. Si l'entité est typée par une interface, la séquence d'instructions de l'héritage multiple est nécessaire : la seule diffé-

[1] L'ordre sur les interfaces peut être compatible avec l'emboîtement (ici, $\{I, K, J, L\}$), mais il faudrait, si l'on veut en faire un invariant, qu'il soit associé aux classes et non aux interfaces. Considérer, par exemple, le symétrique de l'exemple, avec deux classes A' et B', implémentant respectivement les interfaces J et L.

cible → ↓ source	classe	interface
classe	ascendant * descendant *	ascendant * latéral
interface	envoi de message * descendant *	ascendant descendant

FIG. 5.2 – Les différents cas de *casting* en héritage simple et sous-typage multiple : le signe "*" marque les cas qui se traitent comme en héritage simple.

rence est que `deltaOffset` est une constante. Enfin, l'accès aux attributs se réalise exactement comme en héritage simple.

5.2.2 Le *casting*

Grâce à cet emboîtement et au fait que les classes ont une unique table de méthodes, les Δ pour passer des interfaces à la classe sont tous égaux aux $\Delta_{\Downarrow}$, ce qui permet de les factoriser en tête des tables au lieu de les associer à chaque méthode de la table. Le *casting* va aussi être grandement facilité, tant qu'il s'agit de classes, mais la combinatoire à examiner est plus grande (figure 5.2) :

1. de classe à classe, qu'il soit ascendant ou descendant, il s'effectue exactement comme en héritage simple ;
2. de classe à interface, en *casting* ascendant, le décalage est statique (constant et indépendant du type dynamique), grâce à l'emboîtement ;
3. d'interface à classe, le décalage est une constante en tête de table mais deux usages sont à considérer :
   – pour l'envoi de message à une entité typée par une interface, ce n'est pas du *casting* à proprement parler et aucun vérification de type n'est nécessaire ;
   – en *casting* descendant, la vérification de type se fait comme en héritage simple, en temps constant ;
4. d'interface à interface, que le *casting* soit ascendant ou descendant, il faut faire comme en héritage multiple ;
5. enfin, de classe à interface, il peut aussi s'agir d'un *casting* latéral, qui nécessite encore un recours aux techniques d'héritage multiple.

Aucun de ces cas ne peut être éliminé d'office. Pour les cas qui se ramènent à l'héritage multiple, la table des méthodes des classes doit être munie d'une table $\Delta^{\Uparrow}$ et celles des interfaces de tables $\Delta^{\uparrow}$ (voire aussi de $\Delta_{\downarrow}$). Pour que ces dernières tables puissent subir le même partage que les tables de méthodes, il est nécessaire que les méthodes soient regroupées par interface dans les tables d'interfaces, suivant un ordre déterminé par l'interface et indépendant des classes. En particulier, cet ordre doit être indépendant de l'emboîtement (cf. note 1, page 38).

Enfin, la redéfinition de types se réalise comme en héritage simple ou multiple, suivant que les types considérés sont des interfaces ou des classes.

5.2.3 Cas particulier de THETA

Le langage THETA, qui est comme JAVA en héritage simple mais sous-typage multiple, use d'une technique assez proche de celle décrite dans le paragraphe précédent, avec des optimisations pour réduire le nombre de tables de méthodes [Myers, 1995].

L'idée de base de l'implémentation est la *bidirectionnalité* des tables. On peut considérer que l'implémentation des objets qui précède est bidirectionnelle, `self` pointant sur la dernière table de méthodes de l'en-tête (indice 0). Les indices des attributs sont alors strictement positifs, et ceux des tables de méthodes des interfaces sont strictement négatifs.

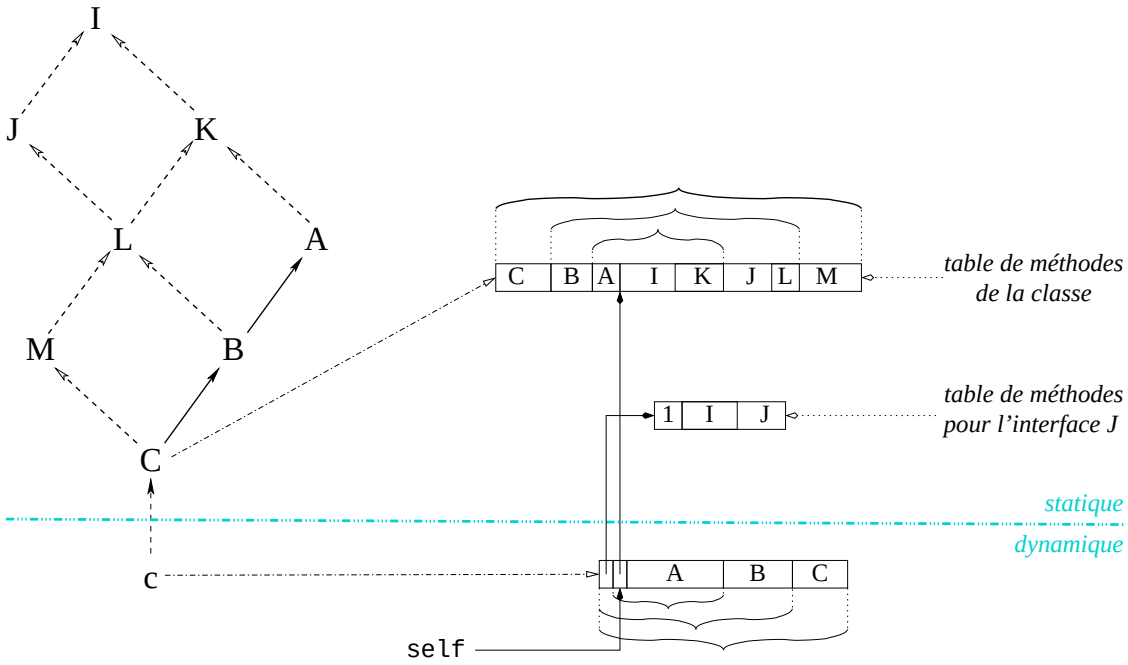

FIG. 5.3 – Table des méthodes en THETA pour l'exemple des figures 5.1 et 5.5.

THETA généralise cette idée de *bidirectionnalité* en l'appliquant aussi aux tables de méthodes[2]. Les tables de méthodes ont des indices positifs et négatifs : en première approximation, la partie positive contient les méthodes déclarées dans des types, alors que la partie négative contient les méthodes déclarées dans les classes mais pas dans les types qu'ils implémentent[3].

Sur l'exemple précédent, la table de classe vaut aussi pour les interfaces I, K, L et M, et une seule table d'interface est nécessaire, pour J (figure 5.3), au lieu de deux après l'optimisation décrite à la section précédente (figure 5.1). Dans un exemple symétrique, avec des classes A', B' et C' implémentant respectivement les interfaces J, L et M, ont aurait aussi une table de classe valable aussi pour les interfaces I et J, et une table supplémentaire d'interface pour les interfaces K, L et M. Dans les deux cas, l'ordre associé à M est supposé être $\{I, K, J, L, M\}$, qui est compatible avec l'emboîtement de A, B et C mais pas avec celui de A', B' et C'.

[Myers, 1995] propose aussi un algorithme de calcul des tables qui permet une relative optimisation.

5.2.4 Application à l'héritage multiple.

[Myers, 1995] propose deux extensions possibles de cette technique à l'héritage multiple. Les deux variantes reposent sur le principe suivant : lors d'un héritage multiple, l'une des super-classes directes est considérée comme principale, et les autres secondaires, et l'implémentation va vérifier l'emboîtement vis-à-vis de la super-classe principale.

Variante par copie de méthodes

La première variante est basée sur une recopie des méthodes et n'est valable que pour les langages comme SMALLTALK qui assurent une stricte encapsulation des attributs qui ne sont accessibles que sur

[2] On retrouve aussi cette idée de bidirectionnalité dans [Eckel et Gil, 2000], qui affecte aléatoirement un sens positif ou négatif aux classes sans super-classes : la spécialisation de deux classes de sens opposés économise des `VBPTR` (cf. section 4.3). Mais il semble que l'origine de cette idée soir dans [Pugh et Weddel, 1990].

[3] Il s'agira par exemple des méthodes non publiques, nécessaires à l'implémentation (donc propre à la classe) et ne relevant d'aucune interface. Même les méthodes `private` doivent figurer dans la table des méthodes, car une méthode peut être inaccessible tout en étant redéfinissable [Sakkinen, 1992].

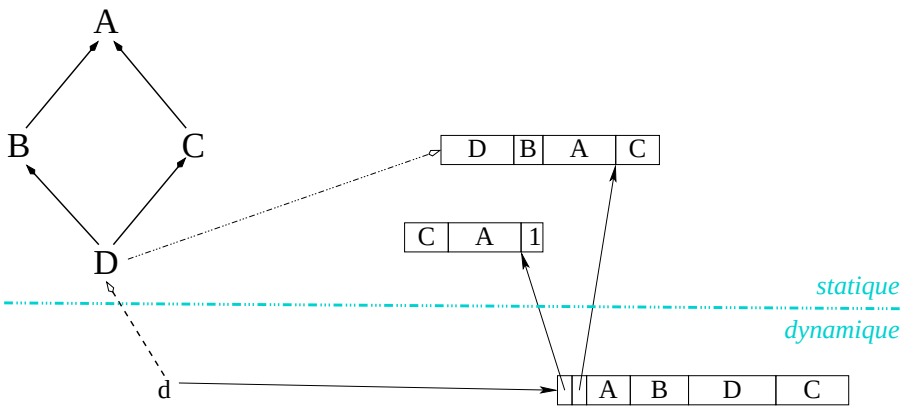

FIG. 5.4 – La technique de THETA en héritage multiple, où C est considérée comme secondaire.

`self`, ce qui peut s'obtenir par la génération automatique d'accesseurs (cf. section 7.1.2). Le partage de code ne s'effectue qu'avec la super-classe principale, les attributs et les méthodes des super-classes secondaires étant recopiées dans la classe. Cette recopie ne concerne que les attributs et méthodes non hérités de la super-classe principale, mais elle concerne aussi toutes les super-classes indirectes qui ne sont pas super-classe de la super-classe principale. La compilation peut ensuite traiter ces super-classes secondaires comme des types (figure 5.4).

Cette technique a un avantage indéniable : c'est une implémentation de l'héritage multiple qui ne coûte que si l'on s'en sert, tout en étant toujours au moins aussi efficace que l'implémentation de l'héritage multiple[4]. Mais son inconvénient n'est pas moins net : elle nécessite la connaissance du code des super-classes, au moins des secondaires, ce qui n'est pas très compatible avec une compilation séparée[5]. La technique s'applique cependant immédiatement à une approche par *mixins*, dans laquelle les classes sont en héritage simple et les *mixins* ne sont pas compilés (cf. section 5.5).

D'après les auteurs, l'évaluation expérimentale montre une sensible amélioration par rapport à C++ : sur l'exemple des figures 3.1 et 4.2, la figure 5.4 exhibe un résultat aussi bon que C++ en héritage non virtuel (figure 4.2).

Variante par décalage des attributs

Les auteurs proposent une technique alternative, plus flexible car compatible avec une compilation complètement séparée, qui consiste à simuler des accesseurs pour l'accès aux attributs : on ajoute donc à la table des méthodes, soit une table de conversion des indices des attributs, soit, mieux, une table de décalage des sous-objets associés aux super-types (cf. section 4.4).

Pour éviter des décalages pour tous les accès aux attributs, les auteurs proposent une double compilation des classes. Les deux compilations diffèrent suivant que l'accès aux attributs se fait par un décalage ou par un indice présumé constant : ce dernier cas est possible lorsque la classe considérée n'est spécialisée que comme super-classe principale, en héritage simple par exemple. Enfin, pour éviter les *castings* systématiques lors des redéfinitions covariantes, les auteurs proposent d'indicer les signatures de méthodes et non les sélecteurs, pour permettre aux *thunks* de ne faire les *casting* que lorsque les types statiques le nécessitent, comme nous l'avons décrit avec le sous-typage simple.

Cette implémentation possède un avantage indéniable sur l'implémentation standard : les décalages sont réduits — le *casting* ascendant vers les super-classes principales est inutile —, de même que le nombre de tables. Au total, elle améliore sans doute celle de l'héritage non virtuel, sans en avoir les inconvénients sémantiques.

[4] Avec une implémentation des décalages par les *thunks*, les classes en héritage simple s'implémentent exactement comme en héritage simple.

[5] La recopie du code des méthodes a pour objectif d'y faire trois modifications : les indices des méthodes et des attributs de `self`, ainsi que les décalages pour le *casting* de `self`. Elle serait donc compatible avec une compilation dans une machine virtuelle qui comporterait les traits syntaxiques nécessaires à la prise en compte de ces trois éléments.

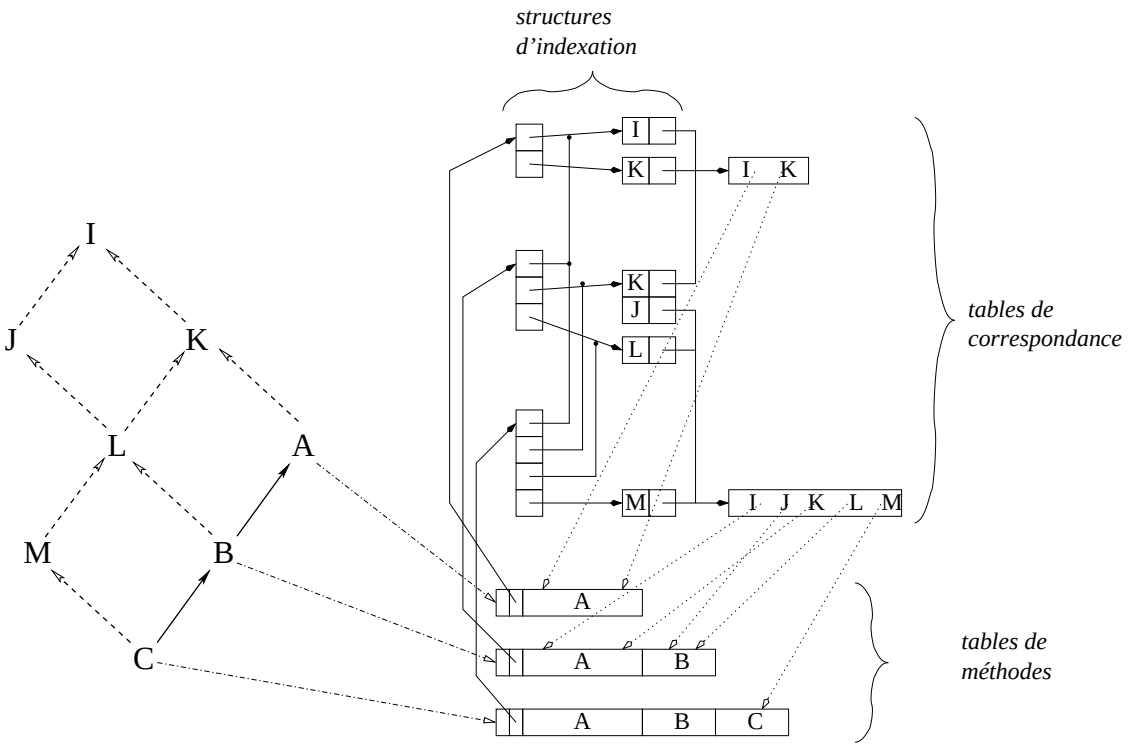

FIG. 5.5 – Sous-typage multiple et héritage simple : variante du sous-typage simple, avec tables de conversion (variante 1).

5.3 Variante du sous-typage simple

Au lieu de se ramener aux techniques d'implémentation de l'héritage multiple, en les simplifiant, on peut se ramener au sous-typage simple, en le complexifiant.

5.3.1 Le principe

On conserve alors de l'héritage simple les deux invariants 2.1 et 2.2, seul le second étant restreint aux entités typées par une classe. Les tables de méthodes sont donc calculées comme en sous-typage simple, mais il faut rajouter des structures d'indirections pour les cas où une entité est typée par une interface : l'invariance des indices de méthodes ne concerne que les classes.

Pour les tables de méthodes des interfaces, plusieurs possibilités sont envisageables :

1. des tables de conversion entre les indices des méthodes dans l'interface et les indices des méthodes dans la classe (figure 5.5) ;
2. des tables de méthodes de plein droit ;
3. si les méthodes sont regroupées par classe ou interface d'introduction, un décalage dans la table de méthode suffit ;
4. ou un pointeur sur le bloc des méthodes de l'interface (figure 5.6).

Chaque classe devra permettre d'accéder ainsi à toutes les interfaces qu'elle implémente.

Mais, contrairement au cas de la généricité bornée (cf. section 6.2), il n'est pas possible d'attacher cette table de correspondances à une entité particulière de façon à ce qu'elle soit accessible en temps constant : il faut rechercher la table de correspondance associée à l'interface considérée dans une structure d'indexation attachée à la table de méthodes de l'objet (cf. annexe A.1 et figure 5.5). La recherche de cette table de conversion est donc une sorte de *lookup* dynamique mais elle est faite une seule fois pour plusieurs appels

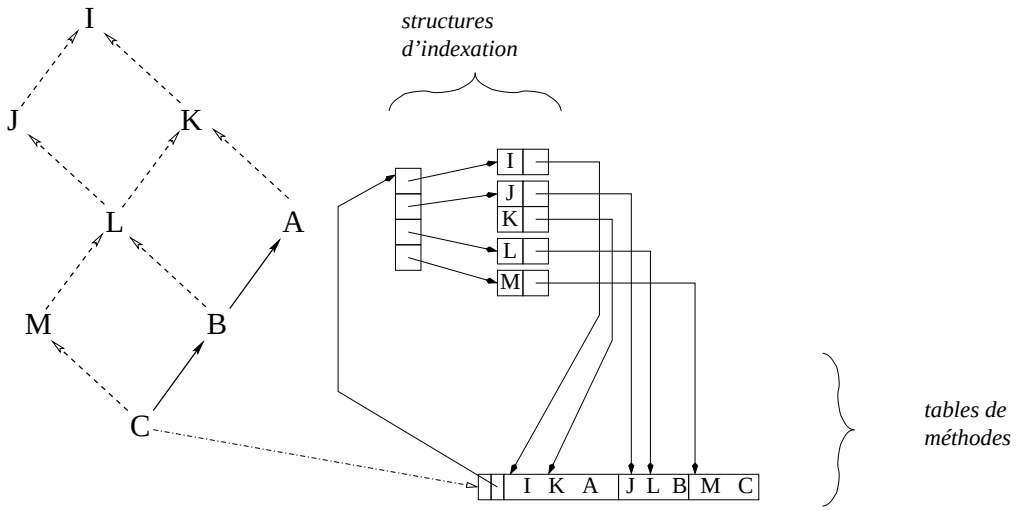

FIG. 5.6 – Sous-typage multiple et héritage simple : variante du sous-typage simple, avec pointeurs dans la table de méthodes (variante 4).

de méthodes car elle vaut pendant toute la durée de la référence à l'objet : la table servira pour toutes les méthodes de l'interface et elle pourra utilement être cachée en ligne (cf. section 5.3.2) [Ducournau, 1997].

Des optimisations basées sur le partage de ces structures sont possibles dans le cas des tables de conversion (variante 1) : elle sont partagées par les interfaces dont la numérotation des méthodes est compatible, comme dans la section 5.2 ; les tables de correspondances s'héritent, ainsi que les structures d'indexation lorsque la sous-classe n'implémente pas de nouvelles interfaces. Dans l'exemple de la figure 5.5, les tables de conversion de K et de M contiennent toutes les autres et les structures d'indexation des interfaces I et K sont héritées de A, celles J et L sont héritées de B. Dans la variante 2, les optimisations sont possibles mais elles sont réduites au contexte du même type dynamique.

Aucune optimisation n'est possible dans les autres variantes. Les deux premières variantes ont l'avantage d'être globales pour toutes les méthodes connues par une interface, alors que les deux dernières ne valent que pour les méthodes introduites par l'interface. On peut envisager un cache en ligne de l'information associée à l'interface, dont la durée de vie sera plus importante dans le premier cas que dans le second.

Quant au *casting*, il s'implémente à peu près comme en sous-typage simple : comme les pointeurs sur les objets sont invariants, seule la vérification de type du *casting* descendant est à étudier. Pour un *casting* descendant vers une classe, c'est la même chose qu'en héritage simple. Pour un *casting* descendant ou latéral vers une interface, qui n'est pas à exclure, on utilisera la structure de conversion que l'on vient de décrire pour faire la vérification de type, comme une table $\Delta^{\Uparrow}$ dont tous les décalages seraient nuls.

Enfin, la redéfinition de types se réalise comme en héritage simple, avec *casting* ascendant transparent ou *casting* descendant qui se réduit à une vérification de type.

En JAVA — tout au moins dans certaines implémentations de la machine virtuelle —, c'est une solution de ce genre qui semble avoir été adoptée, la recherche séquentielle s'effectuant par le biais d'une table de hachage (cf. annexe A.1 et figure 5.5), dans la primitive `invokeinterface` [Meyer et Downing, 1997, chapitre 9][6].

[6] D'après [Driesen, 1999], les (des) implémentations JAVA utiliseraient un cache global, ce qui est assez surprenant : le cache global est une optimisation statistique d'une technique moins efficace, comme un *lookup* [Ducournau, 1997]. Quelle serait alors la technique de base utilisée en JAVA, pour rattraper les défauts de cache ?

La littérature sur l'implémentation des JVM ne donne pas beaucoup de précisions sur ces détails d'implémentation : [Alpern *et al.*, 1999] se contente de préciser qu'il n'a pas été possible d'expanser en ligne l'appel de méthodes par `invokeinterface`.

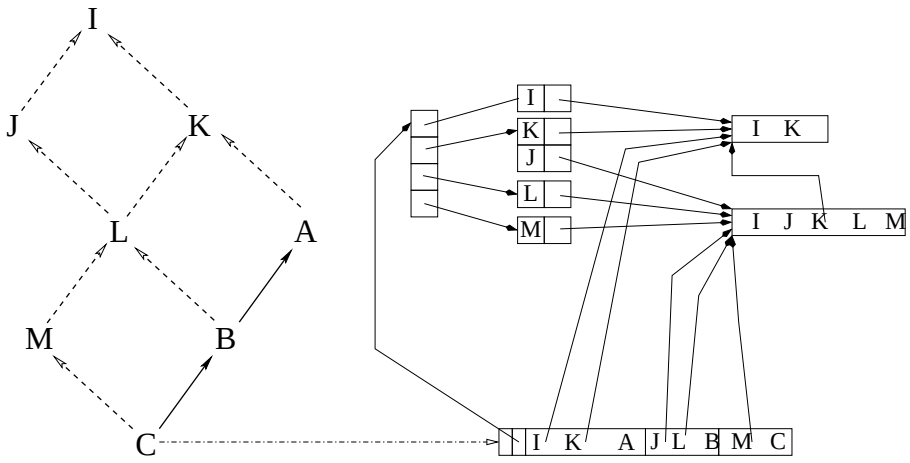

FIG. 5.7 – Sous-typage multiple et héritage simple : variante du sous-typage simple, avec cache en ligne.

5.3.2 Variante en temps constant, par cache en ligne

Il est possible d'éviter l'accès dynamique aux tables des interfaces en les recherchant de façon statique à partir de la valeur que l'on affecte, ou que l'on passe comme paramètre, à une entité typée par une interface. En effet, dans un appel de méthode, lorsque l'on passe comme argument une valeur liée à une entité typée par une classe, pour la lier à un paramètre typé par une interface, le problème est que l'appelée ne connaît pas statiquement le type source alors que l'appelante le connaît. L'idée est donc de faire rechercher la table de l'interface par l'appelante et non par l'appelée. Pour l'affectation, c'est encore plus simple puisque appelante et appelée sont confondues.

Il faut pour cela redoubler toute entité (variable) `X` typée par une interface `I`, par une variable `tableX` qui contient la table de conversion des indices de `I` (variante 1). De plus, chaque table de méthodes, qu'elle soit de classe ou d'interface, doit contenir des pointeurs sur les tables des super-interfaces, un peu comme les tables $\Delta^{\uparrow}$ (figure 5.7). Lorsque l'on affecte une valeur `Y` à `X`, deux cas se présentent, tous deux similaires à un *casting* ascendant en héritage multiple :

– `Y` est typé par une classe : on retrouve la table de conversion dans la table des méthodes de `Y` :
```
load [Y + #tableOffset], table
load [table + #interfaceOffset], tableX
move Y, X
```
– `Y` est typé par une interface sous-type de `I` : on retrouve la table de conversion dans la table `tableY` :
```
load [tableY + #interfaceOffset], tableX
move Y, X
```
Le changement de table est inutile si les 2 interfaces vérifient l'invariant 4.1.

L'envoi de message à `X` s'effectue alors comme suit :

```
load [tableX + #selectorOffset], offset
load [object + #tableOffset], table
add table, offset, table
load table, method
call method
```

Bien entendu, pour un surcoût relativement faible en mémoire statique, il est possible de mettre dans `tableX` les adresses des méthodes elles-mêmes et non des indices (variante 2). Le partage de tables entre deux classes est alors impossible. L'envoi de message se simplifie d'autant :

```
load [tableX + #selectorOffset], method
call method
```

Dans les deux variantes, un certain partage est possible, qui peut éviter certains changements de tables. Les deux autres variantes sont inadaptées car les tables de méthodes des interfaces sont réduites aux méthodes introduites par l'interface.

Outre une certaine augmentation de la taille du code, cette technique présente un inconvénient pour le retour des méthodes et pour les attributs, lorsqu'ils sont typés par une interface, puisqu'il faut retourner ou stocker 2 valeurs. Pour ces deux cas, qui doivent être relativement rares[7], il vaut sans doute mieux conserver la recherche dynamique.

Application à l'héritage multiple

Bien entendu, cette technique s'appliquerait aussi à l'héritage multiple : il faudrait alors l'utiliser pour n'importe quel type d'entité. On peut voir cette implémentation comme ce que deviendrait l'implémentation standard s'il n'y avait pas d'attributs : au lieu de contenir des décalages, la table $\Delta^{\uparrow}$ contiendrait des pointeurs sur les tables de méthodes des super-classes. Les décalages permanents de l'implémentation standard sont alors remplacés par un flot de tables de méthodes en parallèle du flot de données.

Le problème des méthodes étant réglé, il suffit alors de traiter les attributs par décalage dans la table de méthodes, comme dans la simulation des accesseurs (cf. section 4.4).

Les exceptions que nous venons d'écarter seraient trop nombreuses : il faudrait donc doubler la mémoire dynamique.

5.4 Application à JAVA.

Le langage JAVA pourrait adopter une implémentation de l'une ou l'autre catégorie si sa machine virtuelle et son *garbage collector* ne lui imposait pas, semble-t-il, de conserver l'invariant de référence de l'héritage simple.

5.4.1 Receveur typé par une interface : `invokeinterface`

[Alpern *et al.*, 2001a] propose une implémentation efficace et fait un état de l'art des implémentations existantes. Ces implémentations reposent en général sur les techniques utilisées en typage dynamique[8] [Ducournau, 1997].

Comme on l'a vu, le problème de recherche d'une méthode, dans le contexte d'une classe donnée, peut se ramener au problème de la recherche d'une interface. Diverses techniques reprennent donc l'approche d'une grande table classe-sélecteur, éventuellement en la transposant en une table classe-interface, de taille sensiblement inférieure : c'est le cas des machines virtuelles Cacao [Krall et Grafl, 1997] ou Sable [Gagnon et Hendren, 2001]. D'autres techniques comme le cache en ligne, éventuellement polymorphe, sont citées.

De leur côté, les auteurs proposent, dans Jalapeño, d'associer à chaque classe une table de hachage de taille fixe, contenant soit l'adresse de la méthode, soit un arbre de décision indexé sur l'identifiant de l'interface.

5.4.2 Test de sous-typage

Le chargement dynamique des classes interdit d'utiliser la double numérotation pour le test de sous-type lorsque le type cible est une classe. En revanche, la technique de coloration de classes — proposée par [Cohen, 1991] dans le cadre de l'héritage simple et reprise par [Queinnec, 1997] — a l'avantage d'être incrémentale.

Cette technique est utilisée dans Jalapeño [Alpern *et al.*, 2001b]. Lorsque le type cible est une interface, cette technique ne s'applique plus : les auteurs ont alors recours à un tableau de bits qui fait correspondre à chaque classe les interfaces qu'elle implémente.

[7] Est-ce sûr ? et la généricité ?

[8] Cela ne doit pas surprendre : le système de types de JAVA est en fait le plus petit système de types permettant de typer statiquement SMALLTALK en lui conservant un héritage simple des classes : la notion d'interface de JAVA est la seule solution pour traduire la surcharge de définition propre au typage dynamique.

5.4.3 Autres spécificités de JAVA

L'une des spécificités de JAVA est de proposer des fonctionnalités générales, rattachées à des interfaces particulières, voire à `Object`, mais qui nécessitent une implémentation particulière dans les objets. Le cas le plus typique est la synchonisation qui nécessite des *locks* dans les objets. L'implémentation de ces fonctionnalités pourrait se faire simplement par des attributs, mais l'héritage simple de JAVA imposerait de les définir dans `Object`, ce qui provoquerait un surcoût non négligeable dans tous les objets, même ceux qui ne s'en servent pas[9]. Notons qu'une implémentation dans l'en-tête des objets est strictement équivalente à une implémentation par un attribut défini dans `Object`.

[Bacon *et al.*, 2002] propose un schéma d'implémentation qui évite de pénaliser les classes qui ne s'en servent pas, ainsi que des statistiques permettant de comparer les diverses techniques d'implémentation.

5.5 Application à l'héritage multiple : les *mixins*

Une abondante littérature a été consacrée à la notion de *mixin*, sans qu'il soit possible, pour autant, d'en déduire une définition claire [Stefik et Bobrow, 1986 ; Bracha et Cook, 1990 ; Ancona *et al.*, 2000]. Ce serait un moyen stylistique, au choix, de pratiquer ou de se passer de l'héritage multiple.

Un *mixin* serait une classe abstraite — c'est-à-dire sans instances propres, mais pas forcément sans implémentation, attributs ou code de méthodes, au contraire même, toujours avec du code — qui pourrait être rajoutée aux super-classes de n'importe quelle classe pour lui rajouter une fonctionnalité prétendument orthogonale. Les classes « normales », c'est-à-dire concrètes et non *mixins*, seraient en héritage simple.

Sans rentrer dans le débat sémantique, qui n'est pas notre objet ici, force est de constater que cette approche ressemble beaucoup à l'alliance de l'héritage simple et du sous-typage multiple que nous venons de décrire, tout en s'en éloignant sur des points essentiels. Les deux approches ont en commun un héritage simple des classes concrètes et un héritage multiple des classes abstraites. Mais il ne s'agit pas des mêmes classes abstraites car les *mixins* sont des types qui ont, en plus, des attributs et du code, contrairement aux interfaces de JAVA : il ne leur manque que les instances. Une différence supplémentaire est qu'un *mixin* peut spécialiser une classe (cf. note 10, page 47). Pour reprendre une distinction de la linguistique, une classe serait un *catégorème*, alors qu'un *mixin* ne serait qu'un *syncatégorème* [Lalande, 1926].

Variante de l'héritage multiple

La ressemblance est pourtant frappante entre les *mixins* et l'application de la technique du sous-typage multiple à l'héritage multiple par recopie, telle que l'expose [Myers, 1995] (cf. section 5.2.4). Cette approche s'applique immédiatement aux *mixins* en compilation séparée, sans nécessiter d'adaptation particulière : il suffit de faire, pour les *mixins*, ce que C++ fait pour les *templates*, ne pas les compiler (cf. section 6). Le rapprochement entre *mixins* et *templates* n'est pas fortuit : les *mixins* sont souvent présentés comme des classes génériques, paramétrées par la super-classe de la classe résultant de leur « instanciation ». Si A est une classe et M un *mixin*, définir une sous-classe B de A et M revient à définir B comme une sous-classe de $M\langle A\rangle$ [Bracha et Cook, 1990 ; Ancona *et al.*, 2000]. Mais il n'est pas possible de tirer de cette analogie une implémentation : celle qui est proposée en section 6 ne permettrait pas de faire de $M\langle A\rangle$ un sous-type de A.

La deuxième variante de [Myers, 1995] s'applique ici aussi sans problème : les accès aux attributs de `self` dans les méthodes des classes sont alors optimisés par la double compilation.

Cela étant, il ne semble pas qu'en compilation purement séparée, les *mixins* permettent une implémentation beaucoup plus efficace qu'en héritage multiple général. Plus généralement, le fait qu'une classe C soit abstraite ne permet sans doute pas d'optimiser l'implémentation de façon significative : la seule économie semble être l'allocation des tables correspondant au cas où $C = \tau_d$. Dans le cas des *mixins*, un élément fondamental est apporté par le fait que les classes concrètes sont censées être en héritage simple, ce qui entraîne une invariance des indices des attributs *dans les objets*. Mais cette invariance ne s'étend pas aux méthodes des *mixins* qui font référence à ces attributs.

[9] D'après [Bacon *et al.*, 2002], le nombre moyen d'attributs par classe est assez faible, de l'ordre de 4 ou 5 : le surcoût serait donc de l'ordre de 20 %.

Variante de l'héritage simple

Si l'on veut aussi compiler les *mixins* séparément, le schéma de compilation précédent (section 5.3.1 et figure 5.5) peut être étendu aux *mixins* comme suit :

1. dans chaque *mixin*, les attributs et méthodes propres au *mixin*, c'est-à-dire non hérités de leur éventuelle super-classe, ont un indice, calculé à partir de 0 (un *mixin* peut avoir une super-classe[10], pas de super-*mixin*) ;
2. les indices des attributs et des méthodes des classes, y compris ceux qui sont hérités d'un *mixin*, sont calculés comme en héritage simple, sans tenir compte des *mixins* ;
3. dans une classe, pour chaque *mixin* « spécialisé » par la classe, on calcule la table de conversion, ou le décalage, entre les indices du *mixin* et les indices de la classe[11] ;
4. ces tables de conversion ou décalages sont associés aux classes dans une structure d'indexation analogue à celle de la figure 5.1, mais qu'il serait souhaitable de ne pas baser sur la profondeur : les *mixins* ne sont pas censés se spécialiser ;
5. une méthode de *mixin* commencera par chercher la table de conversion lui correspondant, qu'elle utilisera pour accéder à ses attributs et méthodes propres sur `self` : `self` étant immutable, cette table est valable pendant toute la durée d'activation de la méthode ; les accès aux attributs et méthodes de la *mixin* qui seraient héritées d'une super-classe se font comme pour une classe normale ;
6. les méthodes de classes ont une implémentation normale ;
7. l'envoi de message sur un receveur typé par une classe est normal : l'indice de la méthode est invariant ;
8. l'envoi de message sur un receveur différent de `self` et typé par un *mixin* consiste à rechercher la table de conversion, de la même manière que pour les receveurs typés par une interface (section 5.3.1) : le receveur n'étant pas immutable, il faut soit recalculer la table à chaque appel, soit ne la recalculer qu'à chaque affectation du receveur.

A cause de la liaison tardive et d'une possible combinaison des *mixins*[12], il n'y a pas de raison que la table utilisée dans la méthode de *mixin* (étape 5) soit la même que celle de l'envoi de message (étape 8).

La définition des *mixins* reste assez floue. Les exemples donnés par [Bracha et Cook, 1990] les rendent plutôt orthogonaux aux interfaces, le *mixin* n'apportant pas un nouveau type par lui-même : l'objectif serait plutôt de discipliner l'usage de `call-next-method`, dont l'implémentation ne doit pas changer de façon significative (cf. section 3.5). En revanche, rien n'empêche d'avoir à la fois des interfaces et des *mixins* : une interface n'étant qu'un *mixin* sans code ni attribut. La seule différence avec le schéma proposé ci-dessus est que la numérotation de l'étape 1 comportera tous les attributs et méthodes de l'interface ou du *mixin*, qu'ils soient propres ou hérités, à l'exclusion de ceux qui sont hérités d'une super-classe.

5.6 Evaluation

Deux variantes restent en lice : à la JAVA ou à la THETA.

Dans les deux cas, le surcoût en espace provoqué par les interfaces est relativement faible car le partage est important. D'un point de vue temporel, l'efficacité est celle de l'héritage simple pour les entités typées par des classes ainsi que pour l'accès aux attributs. Dans le cas des interfaces, l'envoi de message reste, en THETA, en temps constant, avec des décalages et des indirections qui sont au pire du même ordre qu'en héritage multiple. En revanche, la variante à la JAVA évite les décalages mais elle repose sur une recherche dynamique dont l'impact peut être important si le typage par une interface est courant.

Cela étant, l'une et l'autre implémentations ont des coûts raisonnables et assez voisins si le programmeur se sert de façon limitée des interfaces. Lorsque les interfaces sont nombreuses, par exemple si elles sont calculées de façon automatique [Huchard et Leblanc, 2000], la variante à la THETA risque d'avoir

[10] Ce qui revient à faire de la généricité bornée : $M \prec C$ équivaut à $M\langle T < C\rangle$: dans ce cas, le *mixin* M ne peut être spécialisé (instancié) que pour (par) des sous-classes de C.

[11] Pour les *mixins* spécialisés directement : pour les autres, il suffit d'hériter la table de correspondance de la super-classe.

[12] Le fait qu'une classe soit définie comme spécialisation de plusieurs *mixins*, directement ou par l'intermédiaire de sa super-classe.

un coût spatial dynamique (en-tête des objets) important et il faut choisir la variante à la JAVA. Mais si les interfaces sont en plus beaucoup utilisées pour typer les entités, par exemple si ce typage est généré automatiquement, la dégradation des performances risque d'être sensible.

Les départager plus précisément demanderait une analyse et des expérimentations plus poussées.

Enfin, de la même manière que l'implémentation de l'héritage multiple décrite en section 3, les tables bidirectionnelles obligent à pointer à l'intérieur des objets, ce qui peut augmenter le coût d'un *garbage collector* (cf. annexe A.4). Ce dernier argument peut faire pencher la balance en faveur de la variante à la JAVA.

Parmi les applications de ces techniques à l'héritage multiple, seule l'approche de THETA par décalage des attributs est convaincante : elle réduit sensiblement le surcoût de l'héritage multiple, sans en dénaturer la sémantique. L'approche de THETA par recopie relève d'une compilation globale, pour laquelle on peut faire beaucoup mieux (cf. section 7.1). Quant aux *mixins*, c'est une restriction de l'héritage multiple qui présente peu d'intérêt, d'un point de vue fonctionnel comme d'un point de vue implémentatoire, ou, plus exactement, dont la pauvreté fonctionnelle n'est pas compensée par la facilité d'implémentation.

Chapitre 6

La généricité

La compilation des classes paramétrées pose un problème spécifique qu'il est possible d'éluder, comme en C++, en ne les compilant pas. Dans ce cas, chaque instanciation d'une classe paramétrée se traduit par la génération d'une nouvelle classe non paramétrée, obtenue par copie du code source et substitution du ou des types formels. Tous les arguments en faveur de la compilation séparée rendent souhaitable une compilation séparée des classes paramétrées, dont le code serait partagé par toutes les instanciations futures : c'est ce que [Odersky et Wadler, 1997] appellent une implémentation *homogène*, par opposition à l'implémentation *hétérogène* de C++.

Rappelons que nous avons exclu de notre problématique l'implémentation des types primitifs. C'est sans doute pour la généricité que cette exclusion sera le moins justifié : pour implémenter $\texttt{OrderedSet}\langle\texttt{Integer}\rangle$ et $\texttt{OrderedSet}\langle\texttt{Float}\rangle$, il nous faudrait recourir à une implémentation *hétérogène* ou aux *wrappers* de JAVA.

6.1 En sous-typage simple et généricité bornée

Les classes paramétrées peuvent s'implémenter simplement, dans le cas où la généricité est « bornée », c'est-à-dire pour définir des types paramétrés $A\langle T < B\rangle$ dont le type formel T est contraint à être un sous-type d'un type B, et si cette relation de sous-typage est simple : les indices des méthodes et attributs de T sont alors connus statiquement puisque ce sont ceux de B (invariant 2.2).

Si la généricité n'est pas bornée, il faut avoir recours à la technique décrite en cas de sous-typage multiple (cf. section 6.2). Notons que la généricité peut faire implicitement sortir du cadre du sous-typage simple, en ce qu'elle induit un produit d'ordre (cf. figure 6.2).

6.2 En héritage simple et sous-typage multiple

L'implémentation de la généricité est à peine plus compliquée dans le cas du sous-typage multiple et héritage simple. Lorsque la borne B du type paramétré $A\langle T < B\rangle$ est une classe, les indices des méthodes appliquées au paramètre formel restent constants : l'implémentation du sous-typage simple fonctionne donc toujours. Le traitement des attributs est identique. En revanche, si la borne B est une interface — ou lorsque la généricité n'est pas bornée —, ce n'est plus le cas. Mais il suffit d'étendre la table des méthodes de la classe paramétrée, par une table de conversion des indices des méthodes du type formel aux indices des méthodes du type concret qui l'instancie : l'envoi de message au type formel se résout avec une seule indirection supplémentaire (figure 6.1). Notons qu'il est indispensable de passer par une conversion d'indices — ou un décalage, comme on l'a vu à diverses occasions (sections 4.4, 5.3.1) — car la table de $A\langle C\rangle$ ne peut pas contenir directement les adresses des méthodes de C sans violer le polymorphisme, c'est-à-dire la possibilité qu'une instance de $A\langle C\rangle$ contienne une référence à une instance d'une sous-classe de C.

Le pseudo-code pour un appel de méthode en héritage simple est alors le suivant :

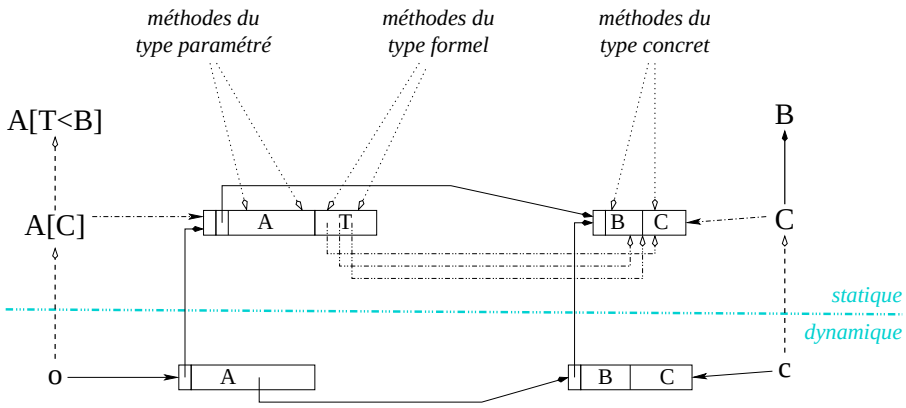

FIG. 6.1 – Table d'indirection des méthodes en sous-typage multiple pour les types paramétrés

```
load [generic + #tableOffset], table1
load [object + #tableOffset], table2
load [table1 + #selectorOffset], offset
add table2, offset, method
load method, method
call method
```

En PIZZA [Odersky et Wadler, 1997], la généricité peut ne pas être bornée : à l'instanciation du type paramétré, il est alors obligatoire de passer en argument les méthodes à utiliser sur le paramètre. L'implémentation se ramène aisément au cas précédent : ce sont les indices des méthodes qui sont passés en argument et qui servent à valuer la table de méthodes du type paramétré.

Plus généralement, comme cette technique consiste à considérer les méthodes du type formel comme des méthodes du type paramétré, elle est *a priori* compatible avec toutes les implémentations décrites pour le sous-typage multiple, tant que l'on instancie les types formels par des classes. Si le type formel est instancié par une interface, quelques restrictions sont nécessaires. Dans la variante de l'héritage multiple (cf. section 5.2), l'accès à une interface nécessite un décalage : tous les accès aux types formels devront donc implémenter ce décalage si la possibilité d'une telle instanciation est prévue. Quant à la variante de l'héritage simple (cf. section 5.3.1), elle ne permet pas cette instanciation, car les accès aux interfaces et aux classes se font de façon complètement différente, ce qui est explicité dans la JVM par les instructions `invokevirtual` et `invokeinterface` [Meyer et Downing, 1997]. Pourtant, une telle instanciation n'est pas toujours sans objet : ainsi par exemple du type `Set`$\langle$`Point`$\rangle$, où `Point` est une interface, implémentée par deux classes `PointPolaire` et `PointCartésien`.

6.2.1 Généricité et spécialisation

Cette implémentation autorise la spécialisation de classes paramétrées, sans faire sortir du cadre de l'héritage simple et du sous-typage multiple (figure 6.2) :

- si $A'\langle T < B'\rangle \prec A\langle T < B\rangle$[1], alors, par définition, $A'\langle C\rangle \prec A\langle C\rangle$. L'implémentation proposée est compatible avec cette spécialisation : il suffit d'étendre la table de méthodes de $A\langle T < B\rangle$ par les méthodes introduites dans A'. La table de méthodes d'une classe paramétrée est alors une alternance d'adresses de méthodes de la classe et d'indices de méthodes du type formel.
- si $D \prec C$ et en admettant que $A\langle D\rangle \prec A\langle C\rangle$ — ce qui n'est possible que si T n'est jamais utilisé en position contravariante [Cook, 1989 ; Weber, 1992], ou bien dans un langage comme EIFFEL qui ne respecte pas la règle de contravariance — alors $A\langle C\rangle$ et $A\langle D\rangle$ se partagent la même table de méthodes.

Enfin, si le *casting* de A à A', ou inversement, ne pose de problèmes, il n'en est pas de même pour un *casting* entre $A\langle C\rangle$ et $A\langle D\rangle$ ou $A'\langle D\rangle$: il faudrait pour cela que le type paramétré dispose d'informations

[1] Ce qui signifie que $A'\langle T\rangle \prec A\langle T\rangle$, quelque soit $T \preceq B'$, avec $B' \preceq B$.

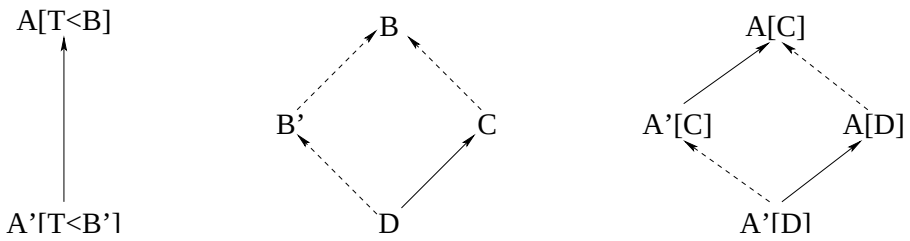

FIG. 6.2 – Spécialisation de types paramétrés

sur le type qui instancie le type formel, C ou D. Une solution est que la table de A pointe sur la, ou les, tables des types qui instancient ses types formels : le *casting* descendant de $A\langle C\rangle$ à $A'\langle D\rangle$ impose alors deux vérifications de types, l'une sur le type paramétré, l'autre sur le paramètre.

Si la généricité est bornée, il est aussi envisageable de faire du *casting* ascendant, du type formel T à la borne B ou à un super-type de la borne : il est juste nécessaire de faire une conversion d'indice dans la table des méthodes, exactement comme pour les méthodes (cf. section 3.3.1). Tout autre *casting* à partir du type formel T paraît exclu.

6.2.2 La généricité en JAVA

De nombreuses extensions de JAVA ont été proposées pour la généricité : PIZZA [Odersky et Wadler, 1997], GENERIC JAVA [Bracha *et al.*, 1998], NEXTGEN [Cartwright et Steele, 1998] ou d'autres encore [Agesen *et al.*, 1997 ; Solorzano et Alagić, 1998]. Aucune ne reprend le schéma que nous décrivons ici parce qu'il n'est pas compatible avec la JVM. Pour que ce soit compatible, il faudrait remplacer l'indirection par un indice par une indirection par une méthode, ce qui pénalise trop.

En pratique, les propositions *homogènes* typent le paramètre formel par la borne et font des *casting* descendants prouvés sûrs par le compilateur : la vérification de type est donc inutile.

6.3 En héritage multiple

En héritage multiple, les appels de méthodes sur les types formels se traitent comme en sous-typage multiple. En revanche, les accès aux attributs ne peuvent s'implémenter, ni comme les attributs en héritage simple, ni comme les méthodes en héritage multiple. En effet, les attributs ne sont invariants que relativement au type statique qui les introduit. Pour chaque attribut utilisé du type formel, le type paramétré doit donc stocker la position du décalage vers le sous-objet — c'est-à-dire $i_{\tau_s}(t_p)$ et non pas Δ_{τ_s,t_p} — ainsi que la position de l'attribut dans le sous-objet, δ_p (cf. page 19). La séquence est alors la suivante :

```
load [generic + #tableOffset], table1
load [object + #tableOffset], table2
load [table1 + #castOffset], delta
load [table1 + #attrOffset], offset
add table2, delta, table2                    4L + 3
load [table2], place
add object, place, place
add place, offset, place
load [place], attribute
```

En héritage multiple, les types paramétrés eux-mêmes peuvent être en héritage multiple : il est donc nécessaire de les implémenter comme en héritage multiple (section 3), avec une table de méthodes par « sous-objet ». Dans chaque table de méthodes, l'accès au(x) type(s) formel(s) s'implémente de la même façon qu'en sous-typage multiple, par une table de correspondance entre les indices des méthodes du type formel dans le type paramétré et les indices de ces méthodes dans le type concret.

En revanche, avec cette implémentation par « sous-objets », la spécialisation $A\langle D\rangle \prec A\langle C\rangle$ n'est pas possible sans une certaine complexité : en effet les entités de type C et D pointent sur des parties différentes

du même objet et ne peuvent se substituer l'une à l'autre. Le problème est exactement le même que pour la redéfinition covariante du type des attributs (invariant 3.6).

Pour un langage au typage orthodoxe, cette limitation est mineure puisque ce cas de spécialisation est très rarement licite [Day *et al.*, 1995]. En revanche, pour un langage comme EIFFEL, il faut assurer un *casting* descendant systématique lors de l'écriture des attributs typés par T, et un *casting* ascendant dynamique en lecture des attributs ou pour le type de retour des méthodes (cf. section 3.4). Les tables de méthodes de $A\langle D\rangle$ et $A\langle C\rangle$ ne sont alors plus partagées.

De plus, le *casting* ascendant du type formel T à la borne B, ou à un super-type de la borne, ne peut pas se faire directement, puisque l'instanciation de T n'est pas connue statiquement, donc l'indice du type cible dans $\Delta^{\uparrow}_{T}$ non plus. Il est donc nécessaire d'inclure dans la table de méthodes de A les indices de la table de méthodes de T qui contiennent les Δ à B : il s'agit encore une fois de conversion d'indices, qui peut être traitée globalement par décalage.

Chapitre 7

Comparaison avec une compilation globale

L'objectif d'une compilation séparée est d'effectuer la compilation d'une unité de programme — ici, une classe — valable quelles qu'en soient les utilisations et spécialisations futures. La compilation séparée est une traduction, au niveau de la compilation, des exigences de modularité du génie logiciel. Elle répond à trois catégories de justifications :

- la rapidité de la recompilation : sans être aussi rapide que dans un environnement interprété, le cycle de développement incrémental permet de tester au fur et à mesure les modifications d'un gros programme, en ne recompilant que les parties modifiées et celles qui s'en servent, voire moins si l'on dispose d'une notion de *compatibilité binaire* [Drossopoulou *et al.*, 1998] ;
- la localité des erreurs : idéalement, la correction d'un système devrait être équivalente à la correction de chacun de ses composants : en particulier, il est souhaitable que la définition d'une nouvelle classe soit une *extension conservative* des classes préexistantes[1] ;
- la sécurité des sources : la non distribution du source des logiciels assure une sécurité importante, commerciale pour les vendeurs et technique pour les acheteurs, qui ne risquent pas d'y faire des modifications hasardeuses.

Enfin, la compilation séparée est suivie par une étape d'édition de liens qui va produire un exécutable n'incluant que les classes explicitement utilisées, ou en dépendant, par une relation de spécialisation ou de clientèle.

En toute généralité, la compilation va donc reposer sur une phase séparée, module par module, classe par classe, et sur une phase globale précédant ou généralisant l'édition de liens. En contrepoint de l'analyse qui précède, nous décrivons donc ci-dessous, d'une part des techniques propres à une compilation globale, d'autre part diverses façons de répartir les rôles entre une compilation séparée et une édition de liens globale.

7.1 Compilation globale

7.1.1 Compilation globale et fermeture

Les diverses techniques de compilation globale décrites dans [Ducournau, 1997] sont toutes applicables dans un cadre de typage statique. Ces techniques sont *a priori* destinées à des langages triplement dynamiques :

- leur typage est dynamique (il n'y a pas d'annotation de types dans les programmes et aucune analyse des types n'est faite statiquement, comme en OBJECTIVE-CAML par exemple [Rémy et Vouillon, 1997]),

[1] Mais la programmation par objets fait apparaître un certain nombre de contre-exemples, en particulier pour les langages ayant une politique de typage covariante [Meyer, 1997 ; Weber, 1992 ; Ducournau, 2001b] ou pour les sémantiques à la base de la classification d'instances [Ducournau et Pavillet, 2001].

– ils sont interprétés (même s'ils peuvent aussi être compilés) et la totalité des bibliothèques de classes d'un environnement donné doit être compilée car il n'y a pas de programme principal (`main`) permettant de déterminer le code effectivement utilisé ;
– ils sont destinés au développement rapide dans un cadre *ouvert* à la définition de nouvelles classes.

L'archétype de ces langages est SMALLTALK.

Le typage statique apporte à ces techniques globales des optimisations significatives par rapport au typage dynamique. En effet, outre les vérifications dynamiques que le typage statique permet d'éviter — on en verra un exemple avec la coloration —, dans le contexte des langages interprétés, la compilation globale joue le rôle d'une édition de liens et s'adresse uniquement à des programmes *fermés* auxquels il n'est pas question d'ajouter la moindre ligne de code[2]. Comme dans le cas de la compilation séparée, on ne va donc considérer qu'un sous-ensemble des classes, mais les techniques utilisées sont linéaires au lieu d'être quadratiques, dans le cas général de l'héritage multiple.

La compilation globale va pouvoir se servir, successivement, d'une analyse globale des types (cf. section 7.1.3), qui détermine le code mort (cf. section 7.1.4), les appels monomorphes[3] ou plus généralement une prédiction de types (cf. section 7.1.5), de la recopie de codes (cf. section 7.1.2), et de l'expansion en ligne (*inlining*) des appels monomorphes.

7.1.2 Recopie physique des méthodes

L'héritage a été souvent qualifié de *copie virtuelle* : certains auteurs proposent d'aller au bout de la métaphore en implémentant l'héritage des méthodes par une *copie physique*. Si les attributs sont bien encapsulés de façon orthodoxe, à la SMALLTALK, cette recopie rend la variance des indices d'attributs parfaitement transparente. Pour les attributs qui ne seraient pas encapsulés, il est nécessaire d'effectuer cette encapsulation au moyen d'accesseurs, dont la génération peut être à la charge du compilateur. Le traitement des appels à `super` et la désignation explicite (cf. sections 2.3 et 3.5) se traitent alors par une compilation (ou expansion) en ligne (*inlining*).

Invariant 7.1 *`self` est monomorphe (cf. note 20, page 66).*

Invariant 7.2 *L'indice des méthodes et des attributs est indifférent pour `self`.*

Le principal avantage de cette technique, proposée sous le nom de *customization* par [Chambers et Ungar, 1989], est que la pseudo-variable `self` n'est plus polymorphe : tout appel de méthode sur `self` peut donc se résoudre statiquement. De plus, toutes les méthodes qui ne peuvent s'appliquer qu'à `self` (par la contrainte d'un mécanisme de protection spécifique)[4], ou qui ne s'appliquent de fait qu'à `self` (par un constat lors d'une compilation globale), n'ont plus besoin de figurer dans les tables de méthodes, ou dans n'importe quelle structure d'appel de méthode polymorphe. Ce sont ces propriétés qui rendaient possible son utilisation dans l'application à l'héritage multiple de la technique de THETA (cf. section 5.2.3). Enfin, la recopie de méthodes donne des informations de type plus précises, dans tous les cas de redéfinitions : sur le type de retour des méthodes[5] et sur le type des paramètres et des attributs[6].

Bien entendu, cette technique a plusieurs inconvénients notables :
– la compilation d'une classe nécessite le source de ses super-classes, ce qui est totalement contradictoire avec la compilation séparée ;
– l'objectif de simplification de l'implémentation n'est pas atteint pour tous les appels de méthodes : seuls les appels à `self` sont concernés et la réduction des tables obtenue n'est que partielle ;

[2] On exclut donc ici une édition de liens dynamique.

[3] Il n'y a pas de consensus pour désigner l'« entité générique » qui se cache derrière les méthodes de même nom (et de même signature pour JAVA ou C++) qui s'héritent, se masquent, etc. En CLOS, on parle de *fonction générique*. En SMALLTALK, seul le sélecteur, c'est-à-dire le nom, est identifié. Quant à C++ ou JAVA, il n'y a rien et le nom lui-même est insuffisant à cause de la surcharge.

[4] Parmi les langages cités jusqu'ici, seul EIFFEL permet de pratiquer une encapsulation à la SMALLTALK, où les droits d'accès de `self` sont distingués. En C++ ou JAVA, les mots-clés `private` ou `protected` traitent indifféremment toutes les instances de la classe.

[5] Pour tous les langages, sauf JAVA, mystérieusement.

[6] Pour les langages comme EIFFEL qui pratiquent une politique de redéfinition covariante [Ducournau, 2001b]. La recopie permet de ne vérifier le type des paramètres que dans les cas de redéfinition effective : les types ancrés permettent en effet de redéfinir la signature d'une méthode sans redéfinir son code.

– une amélioration non négligeable des performances temporelles est obtenue, au prix d'un dégradation très importante des ressources spatiales : la duplication du code des méthodes coûte considérablement plus cher que la petite amélioration obtenue sur les appels de méthodes.

Cette technique n'est donc envisageable que dans un cadre de compilation globale, associée à d'autres améliorations permettant de réduire significativement la taille du code généré : de toute évidence, l'élimination du code non utilisé (« mort ») est une nécessité. C'est ainsi que la technique est utilisée dans le compilateur SMALL EIFFEL [Zendra *et al.*, 1997] (cf. section 7.1.6).

7.1.3 Analyse globale des types

L'un des objectifs d'une analyse globale est de déterminer, pour chaque expression du programme, l'ensemble des types dynamiques que pourront prendre les valeurs de cette expression, pour toute exécution. Dans sa généralité, le problème est exponentiel — voire indécidable puisqu'il pose trivialement le problème de l'arrêt d'un programme — mais des hypothèses simplificatrices le rendent polynomial [Gil et Itai, 1998].

Cette analyse peut être plus ou moins sophistiquée, en calculant une approximation (borne supérieure) plus ou moins précise. Elle peut être insensible au flot de données, ou sensible au flot intra-procédural, inter-procédural, ou aux deux. Dans le cas d'une analyse de flot inter-procédural, elle peut se placer dans l'espace du produit cartésien des types des différents paramètres, ou, au contraire, le projeter sur chacun d'eux, ou, de façon intermédiaire, déterminer les types des paramètres secondaires pour chaque type de receveur. Dans tous les cas, un graphe d'appel doit être construit [Chambers *et al.*, 1997].

L'un des objectifs de l'analyse globale des types est de vérifier la sûreté du *casting* descendant et des redéfinitions covariantes [Wang et Smith, 2001].

7.1.4 La détection du code mort

L'analyse globale des types a pour résultat intéressant de permettre de distinguer le code vivant, qui peut être atteint dans certaines exécutions, du code mort, qui ne sera jamais exécuté.

La détection du code mort permet d'automatiser la sélection des classes nécessaires pour une application par une analyse du code d'un programme depuis le programme principal (`main`). A l'intérieur du code des classes reconnues comme vivante, elle permet enfin d'éliminer les méthodes non utilisées.

Outre la sélection automatique des classes à utiliser, deux cas de figure apparaissent naturels :

– un ensemble de classes peut être utilisé dans différentes applications : chaque programme principal va déterminer implicitement le code vivant de l'application correspondante. ;
– un programme peut être compilé avec différentes options — par exemple, en EIFFEL avec ou sans les assertions — et le code vivant pourra être différent suivant le niveau de compilation.

En revanche, certaines catégories d'applications ne semblent pas pouvoir se prêter de façon satisfaisante à cette optimisation : les applications pour lesquelles l'instanciation des classes résulte, pour partie, d'un processus interactif, avec un utilisateur, un SGBD, un autre programme ou un réseau, voire l'ensemble. Dans un tel contexte, toutes les classes « applicatives » sont *a priori* vivantes, ainsi que tout leur code qui peut être activé par des procédés du même genre.

7.1.5 Prédiction de type et arbres de décisions

Plusieurs travaux récents ont montré la relative inefficacité des techniques de « tables de fonctions virtuelles » sur les processeurs modernes avec *pipe-line* : l'indirection par une table rend toute anticipation impossible [Driesen *et al.*, 1995 ; Zendra *et al.*, 1997 ; Driesen, 1999]. Les auteurs proposent alors de remplacer ces tables par des techniques de prédiction de types [Hölzle *et al.*, 1991], voire d'arbres de décision, appelés *lookdown* dans [Ducournau, 1997 ; Queinnec, 1997].

Dans ce cadre, deux approches sont possibles :

– La plus extrême consiste à énumérer tous les types possibles, dans une succession de tests de type qui implémentent un arbre de décision. Le typage statique rend cette technique plus envisageable qu'en typage dynamique, puisque l'ensemble des types à considérer est borné par le type statique.

Mais une compilation globale est nécessaire puisque c'est la seule façon de connaître la totalité des types : la hiérarchie doit être « fermée ».
- Une solution plus modérée — le cache en ligne, éventuellement polymorphe — consiste à prévoir (prédire) le cas d'un (ou plusieurs) types plus probables, en faisant appel au mécanisme général d'envoi de message dans les autres cas.
- pour l'une ou l'autre approche, le test de type lui-même peut être un test d'égalité ou de sous-typage (codage d'ordre, cf. section A.3) : dans le premier cas, il faut énumérer les types, alors que dans le second il faut énumérer les méthodes ;
- enfin, l'ensemble des tests peut être structuré de diverses manières — séquentiellement ou de façon arborescente — en garantissant des propriétés de la structure (par exemple, arbre équilibré) ou des propriétés d'efficacité dynamique (probabilités décroissantes).

Qui plus est, l'alliance de la compilation globale et du typage statique peut rendre la prédiction quasi-déterministe : le code « mort » étant détecté statiquement, le nombre de classes restantes peut être suffisamment réduit pour qu'une grande partie des appels de méthodes puisse être résolue statiquement, et parmi les appels polymorphes restant, beaucoup mettent en jeu des types peu nombreux.

Discussion

Cela étant, l'influence des architectures de *pipe-line* n'a d'effets que sur le premier (ou les deux premiers) type(s) prédits : si le nombre de types — ou de méthodes — en concurrence dépasse 2 (ou 4), ou si le type prédit n'a pas une grande probabilité, les techniques de tables sont plus efficaces[7]. *A contrario*, un compilateur comme SMALL EIFFEL n'utilise aucun codage d'ordre : pour vérifier que $D \prec C$, où C est connu statiquement, mais pas D, le type D va être comparé par un test d'égalité à C et à chacun des sous-types de C dont une analyse globale a prouvé qu'il n'était pas exclu. Les classes C étant remplacées par un numéro $n(C)$, généré automatiquement et relativement arbitrairement, ces tests vont être organisés entre un arbre binaire équilibré d'inégalités numériques (cf. annexe A.2) : avec k sous-types à examiner, le test de sous-typage va requérir $log_2(k)$ tests numériques, alors qu'il serait possible de le faire en temps constant. Dans le cas plus général de la sélection de méthodes, si le nombre m de méthodes à sélectionner est proche de k, un test de sous-typage efficace n'y changera rien : c'est typiquement le moment d'utiliser des tables de méthodes[8]. En revanche, si ce nombre de méthodes est petit devant k, typiquement si la méthode a été peu redéfinie mais qu'il y a beaucoup de sous-classes, un bon test de sous-type peut apporter un gain significatif[9].

[7] Pas si sûr ! D'après [Zendra, 2000], les capacités de prédictions de branchements, même indirects, des processeurs paraissent diaboliques. Tellement diaboliques que l'on se demande s'ils ne pourraient pas aussi prédire les branchements dans les tables... [Driesen, 1999] travaille en ce sens.

[8] Comme [Queinnec, 1997], dont le schéma de compilation cherche un compromis entre l'usage des tables et l'usage de *look-downs* : c'est une solution que pourrait étudier SMALL EIFFEL.

[9] S'il y a m méthodes à discriminer, pour k types possibles, avec $k \geq m > 1$, la technique de SMALL EIFFEL a un coût compris entre $log_2(m)$ et $log_2(k)$. Vu la numérotation utilisée (cf. annexe A.2), il n'y a aucune raison que des classes héritant de la même méthode aient des numéros contigus. En mode `debug`, quand la possibilité d'une erreur `doesntunderstand` est envisagée, la situation est même pire puisque le facteur important n'est plus k mais $\min(3k, \max_C(n(C)) - \min_C(n(C)))$, où C parcourt les k classes, puisqu'il faut considérer tous les intervalles délimités par ces k numéros. Mais le log_2 de ce nouveau facteur ne doit pas être supérieur à k, ce qui ne justifie pas le `switch`, de coût k, utilisé par SMALL EIFFEL.

Une numérotation par un parcours en profondeur ou une extension linéaire en profondeur [Ducournau *et al.*, 1995] — en héritage simple, les deux sont identiques à n_1 — permettrait de conserver une certaine contiguïté des numéros : ce pourrait être la numérotation initiale de SMALL EIFFEL.

Cela étant, comme SMALL EIFFEL pratique une recopie des méthodes systématique, on devrait toujours avoir $k = m$. La seule exception serait pour les méthodes qui ne sont pas recopiées car elles ne font pas d'accès à `self`. Comme la recopie peut rendre artificiellement polymorphes des appels qui ne le sont pas ou peu, la recopie pourrait être évitée dans des cas plus nombreux (méthode non redéfinie, sous-hiérarchie en héritage simple, etc.).

Enfin, pour terminer la pondération de cette remarque, les chiffres donnés par [Zendra, 2000] semblent donner l'avantage à SMALL EIFFEL, même si $m = 50$! Pour [Driesen, 1999], dans l'implémentation des PIC (*polymorphic inline caches*) à la base de SMALL EIFFEL, un test supplémentaire ne coûte qu'un cycle du processeur (en cas de succès de la prédiction, bien entendu). Le chiffre de 50 est par ailleurs une borne très relative : dans un système d'interfaces graphiques, par exemple, le nombre de classes de composants graphiques peut être de plusieurs centaines, et les méthodes les plus génériques s'appliquent à des receveurs dont le type statique est très général, avec plusieurs centaines de types possibles. A l'exécution, plusieurs dizaines de composants graphiques de type différent peuvent coexister, parmi lesquels l'utilisateur se promène de façon plus ou moins erratique, ce qui réduit forcément la qualité de la prédiction.

Schéma mixte

[Queinnec, 1997] et [Chambers et Chen, 1999] proposent des schémas mixtes basés sur un mélange d'arbres de décision et de tables.

Prédiction de type et compilation séparée

En ce qui concerne la compilation séparée, ces constats expérimentaux pourraient être exploités dans des heuristiques simples, comme la prédiction d'un type plus probable — par exemple le type statique du receveur, qui a le mérite d'être le seul connu par le compilateur, mais il faut que ce soit une classe instanciable, pas une interface ou une classe abstraite. On n'accèderait alors à la table que si le type du receveur diffère du type prédit. La présence des décalages ne gêne pas puisqu'ils sont connus statiquement.

Dans une implémentation qui vérifie l'invariant 2.1 de référence aux objets, on rajoutera l'identifiant de la classe dans l'objet pour éviter une indirection supplémentaire lors de la prédiction de type.

```
      load [object + #idOffset], classid
      load [object + #tableOffset], table
      comp classid, #id
      bne 1
      call #method                                          L + 3
      bne #suite
1     load [table + #selectorOffset], method
      call method                                           2L + B + 2
```

Le chargement de la table (italiques) est anticipé dans la séquence qui n'en a pas besoin : cela permet de réduire le coût d'une mauvaise prédiction, mais au détriment des éventuelles autres parallélisations, par exemple sur les arguments.

7.1.6 Le cas SMALL EIFFEL

Le compilateur SMALL EIFFEL d'EIFFEL repose sur deux *a priori*, qui sont la compilation globale et l'absence de tables de méthodes. Le compilateur utilise les techniques suivantes :

1. la recopie de méthodes est systématique : `self` n'est donc plus polymorphe ;
2. une inférence de types permet de déterminer, pour chaque appel de méthode, le ou les types dynamiques possibles du receveur : cet ensemble de types est d'autant plus réduit que `self` est monomorphe après l'étape (1) ;
3. le code mort peut alors être identifié et éliminé, qu'il s'agisse de classes entières, non instanciées, ou de méthodes non utilisées d'une classe « vivante » ;
4. chaque appel de méthode encore polymorphe après la phase (2) est effectué par une petite procédure qui implémente un arbre de décision à base de comparaisons numériques (basées sur une simple numérotation des classes) ; un traitement du même ordre est effectué pour les accès polymorphes aux attributs — lorsque l'indice de l'attribut varie suivant les différents types déterminés par l'étape (2) — de même que pour la vérification de type associée au *casting* descendant ou à la redéfinition covariante du type des paramètres[10] ;
5. enfin, une compilation en ligne (*inlining*) est effectuée dans divers cas.

La rapidité de la recompilation est assurée par la production de code C : seule la compilation d'EIFFEL à C est globale, les fichiers C non modifiés n'étant pas recompilés. [Zendra *et al.*, 1997 ; Collin *et al.*, 1997 ; Zendra, 2000] détaillent ces diverses techniques.

[10] La recopie de méthodes a un effet intéressant en ce qui concerne cette redéfinition covariante : le résultat de la copie peut violer les règles de typage de façon flagrante, dans la mesure où le type statique de l'argument peut ne pas être un sous-type du type statique du paramètre de la méthode du type statique du receveur (cf. figure 3.7). On peut considérer que la règle de typage la plus simple (mais insuffisante pour assurer qu'un programme est correct du point de vue des types) se définit par le fait que, dans une affectation ou un passage de paramètre, le type statique de la source doit être un sous-type du type statique de la cible. La recopie d'une méthode correcte au sens de cette règle n'est pas forcément correcte.

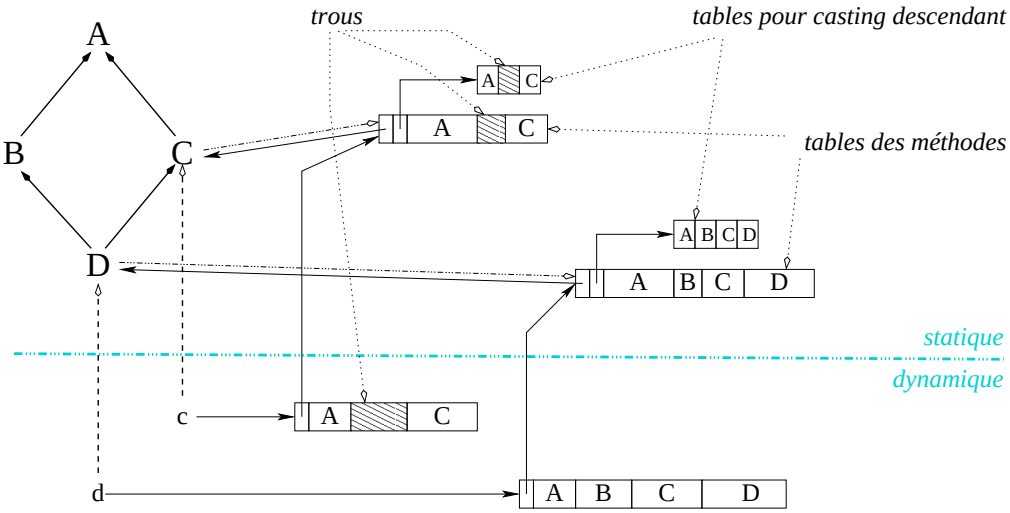

FIG. 7.1 – Exemple de la figure 3.1, avec l'heuristique de coloration unidirectionnelle, appliquée aux classes, aux méthodes et aux attributs

7.2 Heuristique de coloration

Cela étant, tout usage de table ou d'une quelconque structure d'indexation ne peut pas être exclu par principe. Parmi les différentes techniques de compilation globale, celle de l'heuristique de coloration se rapproche le plus des techniques de tables utilisées en compilation séparée.

Le contenu de ce paragraphe est plus longuement développé dans [Ducournau, 2001a ; Ducournau, 2002].

7.2.1 Coloration de méthodes et d'attributs

Le principe de la coloration a été décrit, à peu près indépendemment, dans [Dixon *et al.*, 1989] et [Pugh et Weddel, 1990] et implémenté en YAFOOL [Ducournau, 1991]. [Dixon *et al.*, 1989] propose d'abord une coloration de sélecteurs qui est expérimentée par [André et Royer, 1992], à la suite de quoi elle a été écartée à cause de son apparente inefficacité. Le schéma d'algorithme proposé par [Ducournau, 1997] démontre des performances qui justifient pleinement sa prise en considération.

La coloration permet de retrouver l'invariance caractéristique de l'héritage simple, formulée plus précisément ainsi :

Invariant 7.3 *Chaque attribut (resp. méthode) a un indice (couleur) invariant par héritage. Deux attributs (resp. méthodes) de même couleur n'appartiennent pas à la même classe.*

En corollaire deux classes ayant deux attributs (resp. méthodes) différents mais de même indice, ne peuvent pas avoir de sous-classe commune.

Comme nous l'avons déjà remarqué, la technique que nous avons décrite pour le sous-typage simple est un cas particulier de l'heuristique de coloration.

Dans le cas général, c'est-à-dire en héritage multiple, l'heuristique de coloration permet d'associer à un sélecteur de méthode un indice unique et non ambigu dans toutes les classes qui possèdent la méthode. Plusieurs sélecteurs peuvent se partager le même indice, mais ils n'appartiennent jamais à la même classe. Si, en typage dynamique, il faut vérifier dynamiquement que chaque envoi de message s'adresse au bon sélecteur, ce n'est plus nécessaire en typage statique. Par ailleurs, le typage statique éliminant le risque de « surcharge en typage dynamique »[11], l'heuristique est notablement simplifiée : l'algorithme n'a plus besoin de procéder méthode par méthode, mais seulement classe par classe.

[11] C'est-à-dire le fait que deux méthodes de même nom puissent être définis (introduits) dans deux classes incomparables, sans qu'il soit possible de les distinguer à cause de l'absence d'annotation de type (cf. note 3, page 54).

De plus, l'heuristique de coloration a un autre usage : elle peut aussi s'utiliser pour calculer les indices des attributs. C'était d'ailleurs le but originel de cette heuristique dans le langage YAFOOL [Ducournau, 1991], ainsi que pour [Pugh et Weddel, 1990].

7.2.2 *Casting* descendant et coloration de classes

Comme en héritage simple, le *casting* ascendant n'a plus de raisons d'être. Quant au *casting* descendant, il se ramène à un test booléen puisqu'il n'y a pas de décalage à retourner, `self` étant invariant. Une technique de codage d'ordre est donc envisageable [Caseau, 1993 ; Queinnec, 1997 ; Habib *et al.*, 1997 ; Krall *et al.*, 1997 ; Ducournau, 1997 ; Vitek *et al.*, 1997 ; Raynaud et Thierry, 2001] (cf. annexe A.3). Le *casting* descendant peut aussi être implémenté par l'heuristique de coloration, en généralisant l'implémentation qui a été proposée pour l'héritage simple : on colore les classes de façon à ce que deux classes de même indice n'aient pas de sous-classe commune[12] et on associe à chaque classe un vecteur contenant toutes ses super-classes, à l'indice de leur couleur. Comme pour les tables $\Delta^{\uparrow}$, et contrairement aux tables $\Delta^{\Uparrow}$, cette table de coloration des classes peut être intégrée dans la table de méthodes, par exemple dans les indices négatifs, ce qui évite une indirection supplémentaire : la seule condition est qu'un identifiant de classe $n(C)$ ne puisse pas être confondu avec une adresse de méthode, si l'on veut éviter un test sur la longueur de la table. Pour faire un *casting* vers la classe C, il suffit de vérifier dans la table T_{τ_d} de la classe de l'objet (τ_d) que $T_{\tau_d}[k(C)] = n(C)$, où k est la couleur des classes et n leur identifiant[13]. L'invariant suivant renforce, globalement, l'invariant 3.5 qui n'est que local :

Invariant 7.4 *Chaque classe a un indice (couleur). Deux classes de même couleur n'ont pas de sous-classe commune.*

Le pseudo-code pour une vérification de type, donc pour un *casting* descendant, est alors le suivant :

```
load [object + #tableOffset], table
load [table + #colorOffset], classid
load [table + #lenOffset], len
comp classid, #id                                   2L + 2/3
bne  #fail
comp len, #id
blt  #fail
// casting réussi
```

Le test sur la longueur de la table, en italiques, est inutile si ces tables sont allouées dans un espace contigu : la valeur $n(C)$ ne peut figurer qu'à l'indice $k(C)$ d'une table[14]. Si l'on plonge la coloration de classes dans la table de méthodes — ce qui évite une indirection ou un pointeur supplémentaire dans les objets — on peut encore éviter ce test s'il n'est pas possible de confondre l'identifiant avec une adresse : c'est certainement toujours possible, en choisissant une convention adéquate pour les indentifiants (par exemple en les prenant négatifs ou impairs)[15].

La première technique proposée pour le *casting* descendant en sous-typage simple [Cohen, 1991] était une variante de cette technique, pas optimale dans ce cadre-là. [Queinnec, 1997] retrouve par la suite cette technique, dont [Vitek *et al.*, 1997] propose une généralisation à l'héritage multiple, sans voir qu'il s'agit d'un cas particulier de coloration de méthodes et, comme [Dixon *et al.*, 1989 ; André et Royer, 1992], avec le nombre de couleurs comme critère de minimisation.

Comme technique de codage d'ordre, la coloration est loin de l'optimal souhaité par les algorithmiciens [Raynaud et Thierry, 2001], mais son coût est relativement marginal, puisqu'elle correspond à la définition

[12] On obtient, en même temps, une extension linéaire des super-classes de chaque classe. Cette coloration des classes pourra servir de base à la coloration des méthodes et des attributs, les super-classes étant parcourues dans l'ordre de leur coloration pour la coloration des attributs et des méthodes.

[13] En comparaison, la table $\Delta^{\Uparrow}$ de l'héritage multiple usuel est telle que $\Delta^{\Uparrow}(n(C)) = \Delta_{\tau_d,C}$ (si le *casting* réussit).

[14] A condition que l'identifiant $n(C)$ soit un identifiant global, injectif sur l'ensemble des classes : pour compacter ces tables T, on pourrait prendre un identifiant $n_{k(C)}(C)$ qui ne serait injectif que sur les classes de même couleur. Le couple $(k(C), n_{k(C)}(C))$ est bien un identifiant global, mais il devient nécessaire de tester la longueur des tables.

[15] Outre ces couleurs de classes, la table de méthodes ne contient pas que des adresses. En particulier, la redéfinition covariante des attributs nécessite d'inclure leur type dans la table : il sera donc nécessaire, dans la coloration d'un attribut p, d'éviter que la couleur de p, $k(p)$ soit égale à $k(t_p)$, où t_p est l'un des types de p.

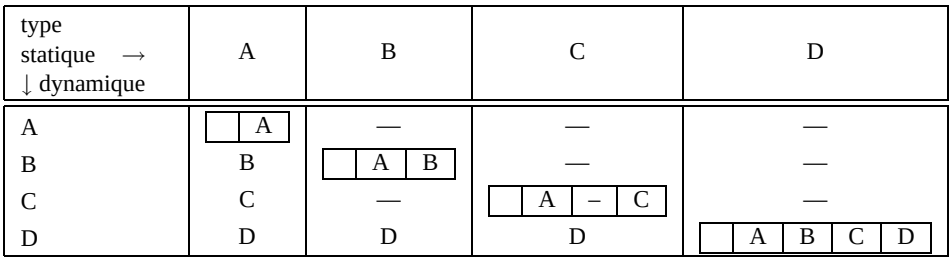

FIG. 7.2 – Tables des méthodes nécessaires pour la coloration unidirectionnelle.

d'une méthode supplémentaire par classe. Elle ressort de l'étude de [Vitek *et al.*, 1997] comme une des plus efficaces : sur le plan temporel, il est peu probable que l'on fasse jamais beaucoup mieux. D'un point de vue spatial, la coloration de classes a autant d'entrées occupées que la taille de la fermeture transitive, soit hN, si h est le nombre moyen de super-classes indirectes d'une classe et N le nombre de classes, à comparer aux N^2 d'une table de fermeture transitive. Les statistiques de [Vitek *et al.*, 1997] donnent à h (resp. N/h), des valeurs de l'ordre de 10 à 30 (resp. 50 à 300).

La coloration de classes peut aussi être utilisée pour retrouver les Δ dans un cadre de compilation séparée : il suffit de doubler les entrées pour y inclure le décalage. Le résultat est beaucoup plus compact que la table naïve de fermeture transitive proposée pour l'héritage multiple[16] : la taille de chaque table est de l'ordre du nombre de super-classes au lieu d'être de l'ordre du nombre total de classes. La technique peut en fait être utilisée pour accéder à toute information associée aux super-classes du type dynamique, par exemple les tables de conversion, décalages ou pointeurs utilisées en section 5.3.1. Mais il ne semble pas très intéressant de se servir de la coloration à cette seule fin.

7.2.3 Schéma d'algorithme

On décrira d'abord le schéma d'algorithme pour la coloration de classes :

```
chaque classe est munie d'une table illimitée
pour chaque classe C, prise dans un certain ordre :
     affecter à C le premier indice libre dans la table de C ;
     propager l'occupation de cet indice dans les super-classes
          des sous-classes de C ;
finpour
```

Bien entendu, tout repose sur le *certain ordre* : la correction est assurée dans tous les cas, mais une extension linéaire descendante, de la super-classe à la sous-classe, permettra une minimisation relative. Le choix de l'extension linéaire ne sera sans doute pas sans effet : mais tout dépendra alors de savoir s'il s'agit d'une coloration de classes, de méthodes ou d'attributs.

Pour la coloration d'attributs (resp. méthodes), il est possible de partir de la coloration des classes, en plaçant les attributs (resp. méthodes) introduits par une classe par blocs contigus, dans l'ordre des couleurs des classes : la difficulté est que ces blocs ne seraient pas de même taille, d'où des risques de trous et des difficultés pour les boucher sans provoquer d'erreurs. Il est donc sans doute plus sûr, et pas beaucoup plus lent, de refaire une coloration spécifique pour les méthodes (resp. attributs) :

```
 chaque classe est munie d'une table illimitée
 pour chaque classe C, prise dans un certain ordre :
      pour chaque méthode m (resp. attribut) introduite par C,
1          affecter à m le premier indice libre dans la table de C ;
      finpour
2     propager l'occupation de ces indices dans les super-classes
3                    des sous-classes de C ;
 finpour
```

[16] En moyenne mais pas dans le pire des cas.

Si le critère d'optimisation est le même, la coloration des attributs et des méthodes peut se faire en même temps.

En héritage simple et typage statique, l'algorithme de l'implémentation standard est identique, avec une simplification : la propagation est inutile, car le premier indice libre est le successeur de la couleur de la super-classe, d'où l'absence de trous.

Coloration bidirectionnelle

La coloration décrite jusqu'ici sous-entendait des couleurs ou indices positifs, les tables s'étendant de l'indice 0 à l'indice maximal occupé. L'intuition et diverses expérimentations [Myers, 1995 ; Eckel et Gil, 2000] montrent une plus grande compacité en utilisant des tables aux indices aussi bien négatifs que positifs. Curieusement, cette idée de la bidirectionnalité semble en fait venir de [Pugh et Weddel, 1990], qui l'appliquait à ... la coloration d'attributs.

Dans une coloration bidirectionnelle, la taille de chaque table est la différence entre les indices maximaux et minimaux occupés. Les algorithmes sont modifiés de façon minimale, dans la définition *du premier indice libre* : il faut considérer les premiers indices libres, positif et négatif, et choisir celui qui cause moins de trous, le positif en cas d'égalité. L'annexe **??** précise un peu le schéma d'algorithme pour obtenir une certaine forme d'optimalité. Dans la figure 7.1, la bidirectionnalité enlèverait tous les trous, en affectant à C l'indice -1. La conjecture suivante est vraisemblable :

Conjecture 7.1 *A extension linéaire identique, la bidirectionnalité réduit au moins de moitié le nombre de trous produits par la coloration.*

On ne peut espérer mieux : un peigne de degré d — une classe avec d super-classes directes incomparables — aura de l'ordre de $d^2/2$ trous en coloration unidirectionnelle, et $d^2/4$ en coloration bidirectionnelle. Par ailleurs, il est nécessaire de comparer à extension linéaire identique[17].

Notons que la coloration bidirectionnelle n'a aucun rapport avec la bi-coloration proposée par [Huang et Chen, 1992], qui consiste à colorer le couple classe-sélecteur.

Coloration en typage dynamique

Le typage dynamique introduit deux complications par rapport au typage statique. Deux méthodes de même nom peuvent être introduites dans plusieurs classes incomparables, sans qu'il soit possible de les distinguer à cause de l'absence d'annotation de type, (« surcharge en typage dynamique »), ce qui oblige, dans le schéma d'algorithme, à vérifier la liberté d'un indice (ligne 1) et à propager une affectation dans toutes les classes de définition, pas seulement dans C (ligne 3) : la propagation ne peut pas se faire globalement, pour toutes les méthodes d'une classe, mais doit se faire méthode par méthode (lignes 2-3 dans le `pour-finpour` précédent). L'absence de vérification statique des types impose aussi une vérification dynamique, dans chaque appel de méthode, en passant le sélecteur en paramètre pour que la méthode appelée vérifie que c'est bien le bon : il y a sinon une erreur `message inconnu`.

7.2.4 Décalage d'attributs et coloration de classes

L'heuristique de coloration permet donc d'implémenter l'héritage multiple de façon aussi efficace que l'héritage simple, au prix d'un inconvénient non négligeable : l'existence de trous dans les tables.

Les trous ne sont pas trop gênants pour les tables de méthodes ou de *casting*, où ils seront plus ou moins compensés par la réduction du nombre de tables. En revanche, les trous dans les instances peuvent provoquer un surcoût non compensé par la perte des pointeurs sur les tables de méthodes des super-classes : et ce surcoût peut être important dans chaque instance et multiplié par un nombre important d'instances. Le surcoût de mémoire dynamique ne gêne pas tous les implémenteurs et on ne sait pas comment se comparent, en moyenne, les trous de la coloration avec le surcoût des `VBPTR` (cf. section 4.3). Il n'est même pas forcément très important en moyenne : [Pugh et Weddel, 1990] rapporte un taux de trous de

[17] [Pugh et Weddel, 1990] montre un rapport inférieur à 2, mais sa définition du taux de trous est relative au total de la taille des tables et non à la place occupée : en corrigeant, on obtient bien un rapport supérieur à 2. Sur la hiérarchie la plus significative, le rapport est supérieur à 10.

6 % pour les 563 classes et 2245 attributs des FLAVORS. Cela étant, ces trous constituent un handicap sérieux : leur minimisation serait un objectif ambitieux, mais le problème est au moins NP-difficile, car il se réduit facilement à la coloration de graphes. Par ailleurs, l'optimal doit pouvoir conserver un surcoût très significatif.

La solution la plus raisonnable est donc certainement de ne pas se servir de la coloration d'attributs, mais de coupler coloration de méthodes et de classes avec la simulation des accesseurs, dont on a vu qu'elle se ramenait à une implémentation par sous-objets mais nécessitait une technique complémentaire pour obtenir l'invariance des méthodes et des décalages pour atteindre ces sous-objets (cf. section 4.4). La coloration de méthodes assurant cette invariance, la solution va donc être la coloration de classes, en associant à chaque couleur, non seulement l'identifiant de la classe, mais aussi son Δ relativement au début de la classe. Chaque entrée de la table de méthodes est un mot qui contient, soit une adresse, soit deux petits entiers, l'identifiant et le décalage. Suivant que l'identifiant occupe les octets de poids fort ou faible, on adoptera une convention sur les bits de poids fort (négatif) ou faible (impair) pour distinguer une couleur de classe d'une couleur de méthode et éviter les tests sur la longueur des tables. Une distinction analogue devra être faite sur les couleurs des attributs, pour la redéfinition covariante.

Pour éviter un décalage systématique et retrouver une efficacité optimale en cas d'héritage simple, on adopte aussi la double compilation proposée par [Myers, 1995]. Dans tous les cas d'héritage multiple, on détermine une super-classe principale, sur la base d'un critère heuristique, comme la maximisation du nombre de super-classes ou du nombre total d'attributs. Dans la version la plus efficace, sans décalage, les attributs de la nouvelle sous-classe sont constitués par la concaténation des attributs de la super-classe principale et de ceux des super-classes indirectes qui ne sont pas super-classes de cette super-classe principale. Cette compilation plus efficace sera utilisée pour toutes les classes qui ne sont jamais spécialisées en tant que super-classe secondaire. Les statistiques de [Pugh et Weddel, 1990] montrent un exemple en FLAVORS, de 563 classes, dont 66 sont en héritage multiple, les 497 autres en étant dérivées en héritage simple : sur ces 66 classes, seulement 28 nécessiteraient une compilation par décalage, soit 5 % du nombre total de classes.

On annule ainsi le surcoût de mémoire dynamique, pour une petite perte d'efficacité dans l'accès aux attributs, dans les cas où le décalage est nécessaire, avec, au total, une amélioration probable par rapport à l'héritage multiple standard.

7.2.5 Schéma complet utilisant la coloration

Au total, un schéma d'implémentation complet basé sur la coloration utiliserait :

- la coloration bidirectionnelle des méthodes, avec une coloration des signatures, au lieu des sélecteurs, pour la redéfinition covariante de méthodes ;
- la coloration bidirectionnelle des classes, associant à chaque couleur de classe, l'identifiant de la classe et le décalage pour les attributs ;
- le décalage des attributs associé à la double compilation pour éviter autant que possible les décalages sur les accès à `self` ;
- la coloration bidirectionnelle des attributs, dans la table des méthodes, pour stocker l'identifiant du type, en cas de redéfinition covariante, donc pour les attributs dont le type est redéfinissable ;
- la généricité bornée s'implémente aussi simplement qu'en héritage simple, en ce sens que l'accès à un type formel est identique à l'accès à un type normal.

7.2.6 Evaluation

La coloration procure donc une implémentation efficace, au prix d'un dernier inconvénient : c'est une technique globale qui ne peut être calculée lors d'une compilation séparée. Mais elle peut s'effectuer à l'édition de liens, en calculant les tables et tous les indices comme des variables dans le code compilé. En revanche, l'incompatibilité avec une édition de liens dynamique est totale.

Quant à la mémoire statique, les tables de méthodes de la coloration sont celles de l'héritage simple, avec un facteur légèrement supérieur à 1 et, dans toutes les expérimentations, largement inférieur à 2, alors que l'implémentation standard a un nombre quadratique de tables, entraînant un facteur supérieur à 3 dans la cadre d'un usage très limité [Driesen, 1999], auxquels s'ajoutent les décalages ou les *thunks*.

Dans tous les cas, une expérimentation serait intéressante pour juger sur pièces le surcoût de mémoire statique[18]. Enfin, les décalages qui parsèment le programme augmentent significativement la taille du code dans l'implémentation standard. Dans tous les cas, une expérimentation serait intéressante pour comparer les coûts statiques des deux implémentations.

Une chose est sûre : avec la double compilation, *l'application de la coloration à des classes en héritage simple avec une encapsulation des attributs à la* SMALLTALK *donne exactement la même implémentation que l'implémentation standard de l'héritage simple.*

Dans le cas de l'héritage simple et du sous-typage multiple, la coloration pourra ne concerner que les méthodes, ainsi que les interfaces pour d'éventuelles vérifications de types, la coloration de classes étant avantageusement remplacée par la double numérotation n_1/n_2.

7.3 Intermédiaires entre compilations globale et séparée

7.3.1 Diverses catégories d'informations

Les diverses techniques reposent sur plusieurs catégories d'informations exploitées plus spécifiquement par le compilateur pour la compilation d'une classe :

- la connaissance du code des méthodes de ses super-classes ;
- la connaissance du code des méthodes des classes utilisées ou de leurs sous-classes ;
- la connaissance des sous-classes de ces classes, de leur existence et de leur interface (fermeture au moins partielle de la hiérarchie).

Ces diverses catégories d'information sont nécessitées par diverses fonctionnalités des langages, par des techniques de compilation qui ne sont pas réellement séparées sans être tout à fait globales, ou par des optimisations spécifiques :

- la *customization* consiste à recopier les méthodes des super-classes dans la classe (cf. section 7.1.2) ;
- pour que l'implémentation puisse tirer parti des *mixins*, en ouvrant la voie à un intermédiaire entre les sous-typage et héritage multiples, une telle recopie est nécessaire ;
- l'héritage non monotone nécessiterait de savoir de quels attributs se servent toutes les méthodes des super-classes (cf. section 2.5.2) : le code n'est pas nécessaire, une information sur les attributs utilisés suffisant ;
- la prédiction de type et les *catcalls* polymorphes d'EIFFEL[19] [Ducournau, 2001b] nécessitent de connaître tout ou partie des sous-classes des classes utilisées ;
- l'héritage arborescent nécessite de connaître toutes les sous-classes de la classe à compiler, voire les sous-classes de ses super-classes si l'on veut déterminer automatiquement quels liens d'héritage doivent être `virtual` (cf. section 4.2 et [Eckel et Gil, 2000]).

7.3.2 Génération de code et de structures de données

Dans la compilation, il faut distinguer la génération du code des méthodes et la génération des structures de données implémentant l'envoi de message. La compilation séparée a besoin de savoir comment générer le code de chaque accès à un objet : accéder à une table, à un certain indice, avec un certain décalage, etc. La technique à utiliser doit être déterminée de façon séparée et il n'est pas possible de varier les techniques utilisées pour les accès aux objets suivant des considérations globales. Mais la taille des tables, leur contenu, la valeur des indices et des décalages peuvent parfaitement être déterminés ultérieurement, à l'édition de liens ou entre la compilation et l'édition de liens. Par ailleurs, le langage cible du compilateur

[18] La comparaison des 2 techniques n'est pas si simple, leurs pires des cas étant orthogonaux. Pour l'héritage multiple virtuel, c'est la chaîne : pour une profondeur p, la taille des tables est en $p^3/6$, alors que la coloration est en $p^2/2$. En revanche, pour la coloration, le pire des cas est le peigne, qui est en $d^2/2$ ($d^2/2$ pour la coloration bidirectionnelle) pour un degré d, alors qu'il est en $3d$ pour l'implémentation par sous-objets.

[19] Notons que ce traitement des *catcalls* polymorphes décrit par [Meyer, 1997, chapitre 17] n'est pas implémenté : les compilateurs EIFFEL faisant tous de la compilation séparée, à part SMALL EIFFEL, qui ne l'implémente pas (D. Colnet, communication personnelle). De fait, c'est une règle inapplicable, qui consiste, en gros, à autoriser la covariance à condition de ne pas se servir des méthodes ainsi redéfinies.

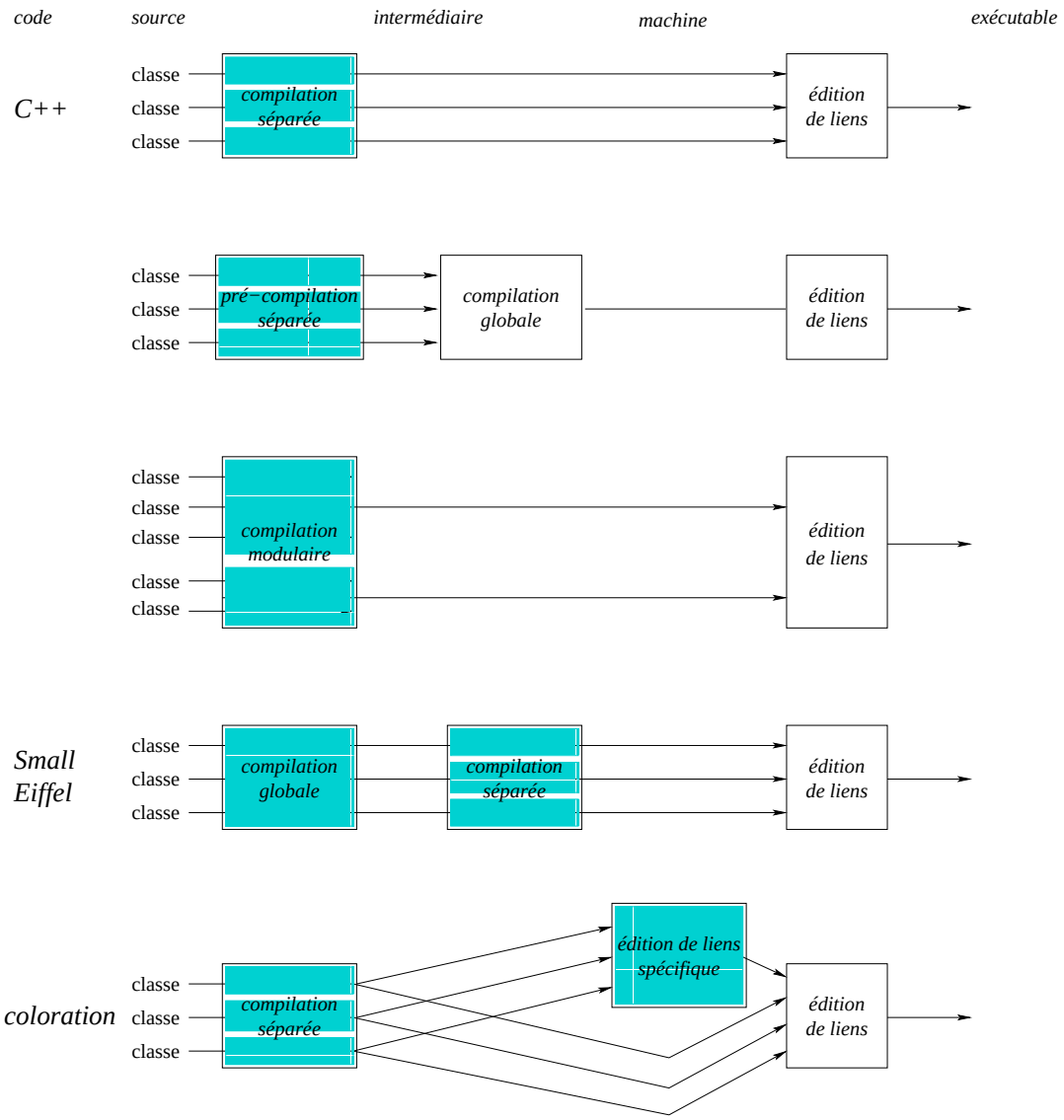

FIG. 7.3 – Schémas de compilation

peut être un langage machine spécifique ou, comme en JAVA et SMALLTALK, un langage intermédiaire qui peut permettre d'abstraire les différentes catégories s'accès aux objets.

Typiquement, le calcul des identifiants de classes $n(C)$, ou la double numérotation n_1/n_2 doit s'effectuer globalement, à l'édition de liens ou entre la compilation et l'édition de liens. La plupart des techniques d'implémentation peuvent être mises en œuvre globalement, la compilation séparée se contentant de générer le code adéquat se servant de constantes qui seront calculées ultérieurement. C'est une complication inutile pour toutes les techniques conçues pour être entièrement calculées de façon séparée, mais c'est la seule solution pour la coloration ou pour la double compilation de [Myers, 1995].

Une telle approche a aussi l'avantage de réduire les recompilations nécessitées par des modifications : le changement de valeur de ces constantes n'est pas suffisant pour imposer une recompilation. A la limite, une classe ne doit être recompilée que si son code a changé, alors que, dans un schéma traditionnel, les sous-classes et les classes clientes doivent aussi être recompilées. *A contrario*, les concepteurs de SMALL EIFFEL sont contraints à des heuristiques compliquées pour limiter la simple modification de l'identifiant d'une classe [Zendra, 2000].

7.3.3 Schémas de compilation et de génération d'un exécutable

La combinatoire des méthodes de compilation se peuple ainsi de nombreuses possibilités, suivant que les générations de code et de structures de données soient, séparément, séparées ou globales, et suivant le langage cible de la génération de code :

1. compilation séparée et édition de liens standard (cas de C++) : les structures de données sont calculées séparément ;
2. compilation séparée et édition de liens spécifique (ou pré-édition de liens) : la génération de code est séparée, donc les parti-pris d'implémentation sont uniformes, mais les structures de données sont calculées globalement, par exemple par l'heuristique de coloration ; la rapidité des recompilations est assurée en ne recalculant ces structures qu'en cas de besoin : la coloration des classes n'est recalculée qu'en cas de modification de la hiérarchie de classes, la coloration des méthodes et des attributs, qu'en cas de modification du schéma d'une classe ; le langage THETA [Liskov *et al.*, 1995] semble ainsi disposer d'un éditeur de liens spécifique, qui choisit les implémentations des différents types ;
3. pré-compilation séparée, c'est-à-dire génération de code intermédiaire (machine virtuelle) et édition de liens standard (cas de JAVA ?) : comme dans le premier cas ;
4. pré-compilation séparée (machine virtuelle) et compilation globale : si la machine virtuelle distingue bien tous les cas — envoi de message à `self`, à une entité typée par une classe ou par une interface, différents cas de *casting*, etc. — la technique utilisée pour la compilation de chaque accès peut être déterminée suivant des considérations globales ; plus généralement, toutes les techniques utilisées dans la compilation globale (recopie de méthodes, détection du code mort, etc.) peuvent être appliquées au code intermédiaire ; il semble que ce soit un schéma de cet ordre qui soit esquissé dans [Meyer, 1997] ;
5. compilation globale (cas de SMALL EIFFEL) : comme dans le cas précédent.

7.3.4 Analyse globale à l'édition de liens

Les techniques d'analyse statique globale qui reposent sur une abstraction du code source pourrait être effectuées tout aussi bien, localement à la compilation séparée et globalement à l'édition de liens.

Coloration

Le dernier inconvénient de la coloration était qu'il s'agissait d'une technique globale. Mais c'est une technique globale qui n'a besoin de connaître que ce qui est déjà exporté par le compilateur pour les classes clientes : la hiérarchie de types et les atrributs et méthodes introduits et définis dans chaque classe. En retour, la coloration construit des tables et calcule diverses constantes — couleurs, indices des méthodes et attributs, etc. — dont les valeurs peuvent être insérées dans le code par l'éditeur de liens.

Analyse de types

C'est typiquement le cas des analyses de types comme la *Rapid Type Analysis* (RTA) de [Bacon et Sweeney, 1996], qui reposent sur le graphe d'appel de méthodes, mais c'est vrai de toute analyse globale de types.

En compilation séparée, le compilateur pourrait faire l'analyse intra-procédurale et intra-classe et générer un fichier d'en-tête donnant pour chaque méthode de la classe

- le type de retour de la méthode ;
- la liste des instanciations de classes effectuées par le corps de la méthode ;
- la liste des appels de méthodes, avec les types de leurs différents paramètres.

Une analyse globale inter-procédurale est alors possible, qui permet de déterminer le code mort. Couplée à une précompilation, cette analyse de types doit aussi permettre de choisir la meilleur implémentation pour chacun des appels.

7.3.5 Précompilation

La précompilation séparée répond en grande partie aux exigences de la compilation séparée :
- relative sécurité des sources,
- localité des erreurs,
- efficacité des recompilations, le plus gros du travail étant effectué séparément : analyses lexicales et syntaxiaques, vérification de type, génération de code et optimisations pour les boucles, récursions etc.

Si le langage intermédiaire distingue tous les accès aux objets suivant les possibilités d'implémentation — suivant le receveur (`self` ou une entité typée par une classe, une interface, etc.), la méthode ou l'attribut (introduit dans le type statique ou un super-type), etc. — la technique d'implémentation peut être décidée, à l'édition de liens, suivant les cas : appels directs pour les receveurs monomorphes, arbres de décision pour les appels mettant en jeu peu de types, tables pour les appels mégamorphes, etc.

Avec une précompilation, l'expansion de code (*inlining*) reste possible et la double compilation, proposée par [Myers, 1995] et citée ici à plusieurs reprises (cf. section 5.2.3 et 7.2), devient inutile.

Bien entendu, la précompilation se prête parfaitement aux techniques globales décrites pour la compilation séparée.

7.3.6 Compilation modulaire

Dans les langages traditionnels, l'unité de compilation séparée est le fichier. Très vite est apparue une notion de *module* permettant de structurer les unités de compilation, de spécifier leurs dépendances, par des imports et des exports, et d'encapsuler la totalité de ce qui n'est pas exporté. Les langages à objets, qui se présentaient un peu vite comme le comble du modulaire, ont identifié cette notion de module avec la classe, un peu comme si les langages fonctionnels avaient identifié le module et la fonction.

Ainsi, les *packages* de JAVA ne sont pas des modules mais des espaces de nom, à l'instar de COMMON LISP.

Or une compilation modulaire est certainement le chaînon manquant actuel, du point de vue de l'encapsulation qu'elle procure autant que comme intermédiaire entre la compilation séparée et la compilation globale. En effet, si les *packages* de JAVA ne sont pas des modules, ce n'est pas seulement parce qu'ils ne sont pas des unités de compilation, mais surtout parce qu'ils sont « ouverts » : n'importe qui peut y définir de nouvelles classes permettant de se servir de ce qui est censé être encapsulé.

Cette idée de la nécessité de modules dans la programmation par objets n'est pas nouvelle [Szypersky, 1992 ; Bracha et Lindstrom, 1992]. Il semble que JAVA s'oriente vers une notion de *scellement* (*sealing*) des *packages*, qui consiste à les « fermer », au sens où aucune autre définition ne peut être ajoutée au *package*, assurant ainsi l'encapsulation [Biberstein *et al.*, 2001]. L'étape suivante consistera à réaliser des compilateurs effectivement modulaires, qui pourront effectuer diverses optimisations : le fait que la hiérarchie soit partiellement fermée — la classe non exportée du *package* n'est plus spécialisable — permet de détecter des appels de méthodes *oligomorphes*[20], voire monomorphes. Diverses propositions de modules ont été faites pour JAVA [Ancona et Zucca, 2001].

Analyse globale restreinte aux sources disponibles

Un autre intermédiaire entre compilations globales et séparées consiste à générer une application par une compilation globale des sources disponibles, avec une édition de liens des autres composants dont seuls le compilé est disponible. Cela revient, d'une certaine manière à compiler un unique module qui n'exporte rien et qui importe toutes les classes déjà compilées.

7.3.7 Perspectives

Trois démarches complémentaires sont donc possibles pour améliorer les techniques de compilation, dans une optique de compilation séparée :

[20] Le polymorphisme étant la capacité qu'a une entité d'être valuée par des valeurs de types différents, le *monomorphisme* est la propriété de n'être valué que par des valeurs d'un seul type et l'*oligomorphisme* celle d'être valué par un ensemble de types limité. Quant au *mégamorphisme*, il désigne le fait que le nombre de types est élevé.

– renforcer l'étape d'édition de liens, ou définir une étape globale précédant l'édition de liens, pour y calculer l'ensemble des structures nécessaires à l'implémentation (tables, indices, décalages, etc.) ; suivant le cas (usage de machine virtuelle ou pas), la technique sera soit différenciée suivant des considérations globales, soit uniforme ;
– définir une machine virtuelle possédant toutes les abstractions nécessaires pour distinguer les différents cas d'envoi de message (suivant que le receveur est `self`, ou typé par une classe ou une interface), de *casting*, etc. de façon à ce que l'édition de liens puisse choisir la technique la plus adaptée pour chaque sélecteur de méthode et pour chaque attribut : le code de cette machine virtuelle peut, par exemple, se prêter à une recopie de méthode, avec une simple substitution, pour donner lieu, dans la sous-classe à une compilation optimisée.
– changer la granularité des unités de compilation, en faisant des *packages* à la JAVA, non seulement des espaces de noms mais aussi de vrais modules : il serait alors possible de fermer la hiérarchie de classes, d'un module, par exemple en interdisant certaines spécialisations, en explicitant une relation d'incompatibilité ou d'appliquer à l'intérieur du module des techniques globales efficaces comme la recopie de méthodes et la compilation en ligne.

A la limite, toutes les techniques utilisées pour SMALL EIFFEL peuvent s'appliquer, soit, dans cette phase globale, non pas aux sources EIFFEL, mais au code de cette machine virtuelle, soit sur les sources d'un module. De façon alternative, la compilation globale peut aussi bien s'effectuer, pour l'essentiel, avec des techniques de tables à la C++.

Enfin, la coloration de méthodes et de classes, alliée à l'implémentation des attributs par une simulation d'accesseurs, offre une technique pour l'héritage multiple, aussi efficace que l'héritage simple pour l'envoi de message, le *casting* et la mémoire dynamique, mais avec le même coût que l'héritage multiple pour l'accès aux attributs et la mémoire statique. La seule contrepartie à ce tableau idyllique est la nécessité d'une phase spécifique d'édition de liens, ou de pré-édition de liens, pour calculer les colorations.

héritage	sous-typage	technique	section	efficacité temporelle: méthodes	efficacité temporelle: attributs	efficacité temporelle: covariance	efficacité temporelle: GC	efficacité spatiale: dynamique	efficacité spatiale statique: tables	efficacité spatiale statique: code	compilation: séparée	compilation: reliure spécifique	compilation: globale	invariants: globaux	invariants: interfaces	invariants: classes
simple	simple	STS	2	1	1	1	1	1	1	1	*			2.1, 2.2		
		HMNV *thunks*	4.2	1	1	1	1	1	1	1	*			2.1, 2.2		
		HM	3	1	2	2	2	2	4	3	*			3.1, 3.2, 3.3, 3.4, 3.5		
		THETA	5.2.3	1	1	1	2	1	1	1			*	3.2, 3.3		
		coloration	7.2	1	1	1	1	1	1	1		*		2.1, 2.2		
	multiple	vHM	5.2	1+	1+	1+	2	2-	1+		*				3.2, 3.3	2.1, 2.2
		THETA	5.2.3	1+	1+	1+	2	2-	1+		*				3.2, 3.3	2.1, 2.2
		vSTS	5.3.1	1+	1+	1+	1	1	1+		*			2.1	3.2	2.2
		vSTS tc	5.3.2	1	1	1	1	1	2		*			2.1	3.2	2.2
		coloration	7.2	1	1	1	1	1	1+	1		*		2.1, 2.2		
multiple	multiple	HM	3	2	2	2	2	2	4	3	*			3.1, 3.2, 3.3, 3.4, 3.5		
		HM covar	3.4	2	2+	2+	2	2	4	3+	*			3.1, 3.2, 3.3, 3.4, 3.5, 3.6		
		HMNV	4.2	2-	2-	2-	2	2-	3	2	*			3.1, 3.2, 3.3, 4.2		
		VBPTR	4.3	2	2-	2-	2	3	4-	3	*			3.1, 3.2, 3.3, 3.4, 3.5		
		partage	4.1	3	3	3	2	2	3	4	*			3.1, 3.2, 3.3, 3.4, 3.5, 4.1		
		THETA cp	5.2.3	2-	1+	1+	2	2-	3	3-			*	3.2, 3.3		
		THETA déca	5.2.3	2-	2-	1+	2	2-	3	3-	*			3.2, 3.3		
		coloration	7.2	1	1+	1	1	1	2	1+		*		2.1, 2.2, 7.3, 7.4		
		SMALL EIFFEL	7.1.6	1?	1?	1?	1?	1	0	3?			*	2.1, 7.1		

FIG. 7.4 – Comparaisons des diverses techniques : les scores indiquent une inefficacité croissante. La signification numérique est faible, l'étalon étant les 2 implémentations standard du sous-typage simple et de l'héritage multiple. En particulier, les scores de deux catégories différentes ne sont pas comparables.

mécanisme	STS		HMV				coloration	
			VTBL pur		*thunks*			
méthode	2L+B	(3)	2L+B+1	(5)	2L+B [+2]	(3 [+2])	2L+B	(3)
redéf. retour	0	(0)	3L+2	(6)	[2L+1]	([3])	0	(0)
param. invar.	1	(1)	1 [+2L]	(1 [+2])	1 [+2L]	(1 [+2])	1	(1)
param. covar.	L+4	(5)	4L+4	(10)	4L+4	(10)	2L+2	(4)
affectation	1	(1)	1 [+2L]	(1 [+2])	1 [+2L]	(1 [+2])	1	(1)
casting ↑	0	(0)	2L+1	(3)	2L+1	(3)	0	(0)
casting ↓	L+4	(5)	4L+4	(10)	4L+4	(10)	2L+2/3	(4/7)
casting dynamique	2L+4	(7)	4L+4	(10)	4L+4	(10)	3L+4	(6/9)
attribut lecture	L [+L]	(1 [+1])	L [+2L+1]	(1 [+3])	L [+2L+1]	(1 [+3])	L [+2L+1]	(1 [+4])
attribut covar	L [+L]	(1 [+1])	3L+3 [+2L]	(6 [+3])	L [+2L+1]	(1 [+3])	L [+2L+1]	(1 [+4])
attribut écriture	L [+L]	(1 [+1])	L [+2L+1]	(1 [+3])	L [+2L+1]	(1 [+3])	L [+2L+1]	(1 [+4])

FIG. 7.5 – Comparaisons des diverses techniques : nombre de cycles et d'instructions.

Chapitre 8

Conclusion

Au terme de cette analyse, il apparaît ainsi clairement que
- la compilation séparée de l'héritage multiple est difficile et coûteuse, avec une aggravation non négligeable en cas de redéfinition covariante du type de retour ;
- la compilation séparée du sous-typage simple est très simple et efficace, même en cas de redéfinition covariante, mais l'expressivité du sous-typage simple est très en-deça du standard actuel ;
- le sous-typage multiple en héritage simple, tel qu'il est offert par JAVA, THETA ou EIFFEL#, offre un bon compromis entre l'expressivité et le coût.

Les implémentations alternatives de l'héritage multiple ne sont en général pas convaincantes car elles nécessitent toutes une certaine dose d'analyse globale ou de connaissance du code des autres classes, qui sont actuellement incompatibles avec une compilation séparée. Dans cette optique, seule la coloration paraît sérieusement envisageable, car elle ne nécessite qu'une pré-éditions de liens, qui peut se contenter des informations contenues dans les fichiers d'en-tête des classes. Comme la coloration a, sur tous les points, une efficacité supérieure à celle de l'héritage multiple standard, et, sur certains, la même efficacité qu'en sous-typage simple, il paraît raisonnable d'envisager sérieusement son utilisation dans un langage à la C++. Des simulations restent à faire pour étudier plus précisément le comportement des 2 implémentations en ce qui concerne la mémoire statique.

Outre l'emploi de la coloration, les perspectives se situeraient donc parmi les divers modèles de compilation suivants, qui ne sont pas incompatibles mais dont certaines peuvent être utopiques :
- un modèle d'implémentation de l'héritage multiple supportant une compilation séparée et n'ayant pas les défauts de l'implémentation de C++, aussi bien d'un point de vue sémantique que du point de vue de l'efficacité ;
- un modèle d'implémentation de l'héritage multiple, supportant une compilation séparée et ne faisant payer cette implémentation qu'aux classes qui ont besoin d'héritage multiple ;
- un modèle de précompilation et de compilation globale qui assure à la fois une analyse statique locale minimale et une certaine confidentialité des sources ;
- un modèle de compilation mixte séparée ou globale, dans lequel la compilation globale accepterait de prendre en compte la présence de bibliothèques compilées de façon séparées ;
- un modèle de compilation globale répartie, dans laquelle les sources seraient compilés à distance en fonction des besoins des clients. Il s'agirait typiquement d'une version répartie de SMALL EIFFEL, dans laquelle les sources seraient répartis sur différents sites en restant inaccessibles aux autres : chaque site devrait donc compiler ses sources en fonction des besoins des sources des autres sites.
- un modèle de compilation séparée exportant des informations sur le flot de types et le graphe d'appel, permettant diverses analyses globales à l'édition de liens ;
- des modèles de compilation reposant sur une granularité différente de la classe : soit avec une notion de module regroupant plusieurs classes, soit, au contraire, en faisant produire à la compilation séparée plusieurs fichiers, à la limite un par méthode, ce qui permettrait à l'édition de liens d'éliminer le code mort.

Aussi séduisant qu'il soit par ses résultats, le schéma de compilation de SMALL EIFFEL n'est ni la panacée, ni la fin de l'histoire. Il ne s'applique pas, ou mal, à des applications pour lesquelles le code mort

ne peut pas être connu statiquement : typiquement, les applications interactives dans lesquelles l'utilisateur instancie lui-même les différentes classes du système, par exemple par des requêtes dans une base de données ou au travers d'interfaces graphiques. Dans ce cas, toutes les classes sont *a priori* vivantes, ainsi que tout leur code accessible interactivement.

Il est enfin possible, à plus court terme, d'améliorer les compilateurs existants en en combinant les techniques. Ainsi, SMALL EIFFEL a prouvé qu'il était possible de faire des compilateurs efficaces, voire plus efficaces que les compilateurs existants, en mettant en pratique des idées simples. Mais il gagnerait à utiliser des techniques plus compliquées dans les cas qui le méritent : des tables de méthodes basées sur l'heuristique de coloration et restreintes aux méthodes qui le nécessitent — car redéfinies dans suffisamment de sous-classes — remplaceraient avantageusement les arbres de décisions pour les appels de méthodes dits *mégamorphes*[1].

On peut constater que le standard actuel est réparti entre 3 langages :

- le premier fait une compilation séparée en héritage multiple, au prix de nombreux défauts de spécification et de réutilisation ;
- le second fait une compilation globale ou séparée, suivant les compilateurs, en héritage multiple et au prix de certaines libertés avec ses spécifications ;
- le troisième fait une précompilation en héritage simple, dans une machine virtuelle qui compense heureusement, pour ses concurrents, les gains de performance qu'il aurait pu en attendre.

Ces divers langages ont un point commun, leur imperfection et leur incomplétude. Nous avons ainsi relevé diverses fonctionnalités, comme de vraies variables de classes, avec liaison tardive, qui gagneraient à être spécifiées correctement car elles sont utiles et que leur implémentation serait peu coûteuse. Pour les implémentations de l'héritage multiple dont le surcoût est supposé trop élevé, la possibilité d'exprimer proprement des restrictions de spécialisation — par exemple, héritage simple ou arborescent seulement — serait un élément de simplification et d'optimisation des langages.

Trois axes complémentaires donc : la mise en œuvre séparée des techniques globales, des machines virtuelles plus orientées vers les traits spécifiques à optimiser et une vraie compilation modulaire.

[1] Modulo la conclusion de la note 9, page 56.

Annexe A

Éléments d'implémentation

A.1 Structure d'indexation dynamique des super-classes

Le *casting* descendant en héritage multiple, aussi bien que l'accès aux indices des méthodes dans le cas des interfaces de JAVA, nécessitent un *lookup* qui peut s'implémenter par une structure d'indexation des super-classes ou des interfaces d'une classe. Une table de hachage générale est envisageable, mais il est possible de profiter de la structure d'ordre pour obtenir une fonction de hachage statique. Dans tous les cas, il est préférable que la fonction de hachage soit calculable à la compilation, pour qu'un *casting* statique se traduise par une constante dans l'exécutable.

On peut ainsi utiliser comme fonction de hachage un poids $w(C)$ associé à chaque classe ou interface C, tel que w soit une fonction décroissante, c'est-à-dire que $C \prec D \Rightarrow w(C) > w(D)$. On pourra prendre par exemple le *rang* de la classe, avec $w(C) = 1 + \max_{C \prec D}(w(D))$.

La figure 5.5 donne un exemple de cette structure d'indexation pour la correspondance des indices des méthodes entre les interfaces et les classes.

L'heuristique de coloration offre une structure d'indexation des super-classes très efficace en temps, pour un surcoût en espace qui n'est peut-être pas supérieur à celui de toute autre structure d'indexation : son utilisation devrait être examinée attentivement.

A.2 Ordonner et numéroter les classes

En héritage ou sous-typage multiples, l'implémentation repose sur plusieurs tables qui ont chacune besoin d'un ordre sur les super-classes d'une classe : ordre des « sous-objets » dans l'objet, des blocs de méthodes dans les tables de méthodes, ordre des super-classes dans la table des Δ, etc.

Dans le cas général (section 3), tous ces ordres sont indépendants, mais il n'est pas nécessaire de compliquer l'implémentation en les faisant varier à plaisir : on pourra donc en définir un seul. Dans les implémentations qui partagent des tables (cf. sections 4.1, 5.2 et 5.2.3), le partage repose sur des ordres communs pour une même classe, mais aussi sur des ordres communs entre une classe et certaines de ses super-classes. On utilisera de préférence une extension linéaire, par exemple celle qui est utilisée pour ordonner les constructeurs et destructeurs [Huchard, 2000].

En plus de cet ordre local aux super-classes d'une classe, l'implémentation nécessitera une numérotation des classes — $n(C)$ — qui fournit à la fois un identifiant des classes, utilisé comme *tag* pour accéder au type d'un objet et un ordre sur les classes. Là encore, une extension linéaire est possible. Par ailleurs, sous réserve de contraintes spécifiques, par exemple un ordre compatible avec l'emboîtement (cf. section 4.2) — les ordres locaux pourraient être définis comme des restrictions de cet ordre global. A titre d'exemple, SMALL EIFFEL numérote les classes avec le seul objectif de conserver la numérotation de la compilation précédente, pour éviter d'avoir à recompiler des arbres de décision qui serait la seule modification d'un fichier [Zendra, 2000] (cf. section 7.1.6).

A.3 Codage d'ordre

La vérification dynamique de type joue un rôle marginal mais fréquent dans l'implémentation des langages à objets à typage statique : le *casting* descendant et la vérification de type des arguments covariants en sont l'exemple le plus frappant. Le problème consiste, de façon générale et abstraite, à répondre à la question : D est-il une sous-classe (un sous-type) de C ($D \prec C$) ? En général, C est connu statiquement, mais D ne l'est que dynamiquement. Le problème est aussi borné : on sait que D est une sous-classe de B, qui est connue statiquement. En pratique $D = \tau_d$ et $B = \tau_s$. Mais on a vu que, dans les différentes implémentations envisagées, cette connaissance statique n'avait aucun effet.

Un codage d'ordre est la solution générale à ce problème : il consiste[1] en une fonction i telle que $D \preceq C \Leftrightarrow i(D) \leq_i i(C)$. La relation de spécialisation $\prec$ étant donnée par son graphe de couverture (sans arcs de transitivité), on appelle souvent ce problème un codage de la fermeture transitive. Le problème est d'obtenir le codage le plus compact possible, avec un test $\leq_i$ aussi efficace que possible. La section 2.2.2 en a donné un exemple simple et optimal pour le sous-typage simple. La généralisation à l'héritage multiple est malheureusement moins simple et plus complexe [Caseau, 1993 ; Queinnec, 1997 ; Habib *et al.*, 1997 ; Krall *et al.*, 1997 ; Vitek *et al.*, 1997 ; Ducournau, 1997 ; Raynaud et Thierry, 2001].

Un codage d'ordre est surtout crucial en typage dynamique ou en compilation globale.

On a vu dans cet article 3 techniques différentes pour faire ce codage d'ordre :

- la simple numérotation des classes, en héritage simple uniquement, qui ne coûte que deux entiers par classe ;
- la table de fermeture transitive, qui coûte en moyenne $(N-1)/2$ entiers par classe et donne les décalages relatifs au type dynamique ;
- l'heuristique de coloration, dont le coût empirique n'est pas beaucoup supérieur à la taille de la fermeture transitive.

Avec des hiérarchies de classes réelles, l'heuristique de coloration se comporte bien mieux que la table de fermeture transitive[2] : on peut d'ailleurs voir la coloration de classes comme l'introduction d'une méthode supplémentaire pour chaque classe, ce qui revient à augmenter la taille des tables de 20 %, si l'on prend une valeur moyenne de 5 méthodes introduites dans chaque classe (cf. annexe **??**).

A.4 *Garbage collector*

A.4.1 Prise en compte du *Garbage collector* dans les objets

Les langages qui possèdent un *garbage collector* doivent intégrer les informations nécessaires à ce mécanisme dans l'implémentation des objets. Les techniques de *GC* sont nombreuses : elles nécessitent des informations variées : bits de marquage, pointeurs en avant pour les déplacements d'objets lors du compactage ou du changement de génération, etc.

Parmi les nombreuses solutions proposées, [Bacon *et al.*, 2002] propose d'intégrer les bits de marquage dans les bits de poids fort ou faible du pointeur sur la table de méthodes. D'autres auteurs proposent des tables extérieures aux objets, comme [ILO, 1995 ; ILO, 1996] qui réserve un bit par double mot (8 octets) de mémoire, soit un surcoût uniforme de 12,5 %. [Colnet *et al.*, 1998] regroupe les objets par type sur une même page mémoire : il est donc envisageable d'implémenter les bits de marquage dans un tableau de bits propre à chaque page, ce qui rendrait insignifiant le surcoût.

[1] Nous décrivons ici une grossière simplification : en toute généralisation, i représente une application monotone dont le co-domaine est ordonné par $\leq_i$ (qui se teste en temps constant). Le cas de la section 2.2.2 s'exprime comme une inclusion d'intervalles.

[2] Mais dans le pire des cas, les deux techniques sont en $N^2/2$ (où N est le nombre de classes) : c'est le cas aussi bien avec une hiérarchie complètement plate, avec 1 classe ayant $N-1$ super-classes directes, où avec une hiérarchie linéaire, ordonnant totalement les N classes. Comme la coloration impose de doubler les entrées de la table avec $n(C)$, la coloration pourrait être beaucoup moins bonne : mais une grande hiérarchie de classes peut difficilement reproduire le pire des cas et deux demi-hiérarchies indépendantes, reproduisant chacune ce pire des cas, ramènerait à une complexité en $N^2/2$.

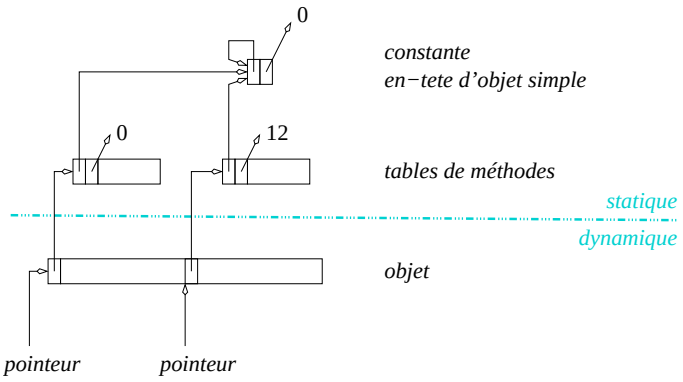

FIG. A.1 – Schéma d'implémentation uniforme des objets, avec *garbage collector*.

A.4.2 *Garbage collector* compatible avec des sous-objets

Dans les implémentations de l'héritage multiple pour lesquelles le pointeur sur un objet dépend du type statique du pointeur, le *garbage collector* doit être capable de retrouver l'origine de la zone mémoire occupée par l'objet.

Si l'information permettant de retrouver le début de l'objet ($\Delta^{\mathcal{T}^s}_{\Downarrow}$, si on décide de placer en premier le sous-objet correspondant au type dynamique) est présente, elle est d'autant moins accessible à un *GC* que tous les « objets » de la mémoire ne sont pas forcément du même type (les tables de méthodes, par exemple, n'ont elles-mêmes pas de tables de méthodes).

Dans un langage à la LISP ou SMALLTALK où les objets sont des vecteurs uniformes de pointeurs, les pointeurs n'ont pas l'habitude de pointer ainsi au milieu des objets et le *GC* n'est pas prévu pour cela. Mais l'obstacle n'est pas insurmontable : sans chercher des solutions plus sophistiquées, il suffit que tout pointeur sur un objet pointe sur un mot mémoire pointant lui-même sur une table (elle-même pointant sur ...) dont le deuxième élément désigne le Δ au début de l'objet (0 pour une table normale). La régression infinie est arrêtée par une boucle réflexive sur un doublet constant pointant sur lui-même et contenant un décalage nul (cf. figure A.1). Mais ce contournement a un surcoût non négligeable. [Ellis et Detlefs, 1993] propose un schéma de ce genre mais juge finalement plus simple de traiter de façon générale tous les pointeurs à l'intérieur des objets.

Dans un langage comme C++ — qui est dépourvu de GC pour des motifs autant techniques que culturels — ou EIFFEL — qui, lui, en est pourvu pour des raisons essentiellement identiques mais contraires —, l'implémentation des objets n'est pas uniforme, puisque la valeur d'un attribut peut être aussi bien une référence sur un objet que l'objet lui-même (mot-clé `expanded` en EIFFEL)[3]. Dans ces langages, la phase de marquage de l'algorithme *mark and sweep* doit donc s'effectuer par l'intermédiaire d'une méthode `mark`, générée automatiquement par le compilateur suivant la description de la classe pour tenir compte des variations d'implémentation des attributs [Colnet *et al.*, 1998]. Cette méthode effectuera naturellement le décalage décrit précédemment. On notera que cette méthode `mark` se heurte au même problème de réflexivité, qu'elle résout de la même façon, par une boucle réflexive d'un objet sur lui-même : une fois marqué, il n'est plus nécessaire de régresser à l'infini. Cela étant, la présence de cette méthode `mark` et la

[3] Voilà encore un trait de langage assez criticable ! Lorsqu'il s'applique à autre chose qu'un type primitif non spécialisable, il viole deux des fondements des objets : d'une part ce que l'on appelle polymorphisme d'inclusion, sous-typage ou spécialisation, d'autre part l'identité d'objet. En effet, l'affectation d'une valeur d'un sous-type devient physiquement impossible. En fait, l'affectation d'une valeur source à un tel attribut cible revient à affecter, à l'objet valeur de la cible, le contenu, c'est-à-dire les valeurs des attributs, de la source. Une telle affectation peut être aussi bien spécifiée d'un sous-type à un super-type, qu'en sens inverse : l'opération considérée relève en effet plus de la copie que de l'affectation proprement dite, et le sous-typage ne fournit pas de contraintes particulières. La spécialisation ne fournit pas non plus un guide : en toute généralité, on peut recopier tous les attributs communs à la source et à la cible, donc définis dans des super-classes communes. Curieusement, cette spécification très générale de l'affectation rend l'attribut immutable : il est donc possible de le spécialiser de façon covariante [Ducournau, 2001b] et son caractère constant peut autoriser diverses optimisations.

Pour les types primitifs non spécialisables, le compilateur devrait décider de l'implémentation sans qu'un mot-clé particulier soit nécessaire : c'est ce que propose [Dolby et Chien, 1998]. Des optimisations significatives d'occupation mémoire sont possibles pour les types qui utilisent moins d'un mot mémoire, booléens par exemple [Pavillet et Ducournau, 1999].

non uniformité de l'implémentation permettent d'exclure les tables de méthodes de la gestion mémoire, en les implémentant dans le code et non dans le tas[4].

Pour tous les langages, la deuxième phase de l'algorithme est effectuée par une méthode `sweep`, qui fera par exemple appel à un analogue des *destructeurs* de C++.

Mais un *GC* ne se résume pas aux deux étapes *mark* et *sweep* (sans parler des algorithmes de type *stop* et *copy*). Une troisième étape de compactage est nécessaire pour éviter la fragmentation. Ce compactage repose souvent sur une indirection systématique dans les pointeurs sur les objets, ce qui permet de déplacer l'objet sans modifier tout ce qui pointe sur lui. La fonction `change-class` de CLOS, ou le `become` de SMALLTALK, reposent souvent sur cette implémentation[5]. Pointer à l'intérieur d'un objet est alors impossible, du moins sans y ajouter une certaine complication. On réduira l'effet de ce `wrapper` en lui faisant inclure le pointeur sur la table des méthodes : seul l'accès aux attributs subira une indirection supplémentaire, qui devrait bien se paralléliser avec l'éventuelle indirection qui passerait par la table[6]. Le pseudo-code en coloration, dans le cas d'un accès avec ou sans décalage (italiques) donnerait alors la séquence suivante :

```
load [object + #tableOffset], table
load [object + #slotsOffset], slots
load [table + #castOffset], delta                    2L/3L + 1
add slots, delta, slots
load [slots + #attributeOffset], attribute
```

L'indirection par le *wrapper* coûte en tout cas moins cher que le décalage.

De fait, le compactage du tas et les fonctionnalités de migration d'instances reposent sur la même implémentation. [Ingalls *et al.*, 1997] montre comment il est possible d'éviter l'indirection par un *wrapper*, au prix d'une part d'une sophistication du GC, d'autre part d'une sous-utilisation de la fonctionnalité `become`.

[4] Toutes les illustrations d'implémentation de l'article distinguent une partie statique — allouée une fois pour toutes, à la limite dans le code, et qui ne devrait pas être concernée par le *GC* — et une partie dynamique, l'allocation des objets, qui seule relève du *GC*.

[5] [Drossopoulou *et al.*, 2001] propose une spécification moins générale destinée à décrire les états des objets par des changements de `state` classes. L'implémentation proposée serait basée sur le calcul statique de la taille mémoire nécessaire à tous les états d'un même type. Au prix d'une plus grande consommation mémoire, une telle implémentation pourrait être compatible avec des sous-objets : il faudrait juste calculer statiquement la place des sous-objets de façon à ce qu'elle ne varie pas en changeant d'état.

L'implémentation des `VBPTR` comme des pointeurs dans les objets, de son côté (cf. section 4.3), permet d'implémenter la classification d'instances, c'est-à-dire une version monotone de `change-class`, par alloocation de sous-objets supplémentaires et mise à jour des pointeurs sur les tables.

[6] Mais, attention ! Comme on l'a remarqué plus haut en analysant les conclusions de [Driesen, 1999] sur les *thunks*, cet argument de la bonne parallélisation ne vaut sans doute pas grand chose.

Annexe B

Analyse de types séparée

Chacun des types considérés est décrit par une expression de type formée sur la grammaire suivante :

```
Type  = { Atom* }
Atom  = Prim | =Class | <Class | =self | <self | Param | p.Type | stat p.Type
Prim  = type primitif non spécialisable
Class = n'importe quel type ou classe
self  = le type du receveur lui-même
Param = type formel représentant le type d'un paramètre, indicé
p     = le type de retour de p (type ancré like p)
stat p = le type statique de p
```

Une expression de type est donc un ensemble, ce qui généralise l'approche de [Palsberg et Schwartzbach, 1991] pour qui un type est un ensemble de classes[1]. Les symboles `=` et `<` précisent respectivement un type exact, ou le type et n'importe quel sous-type. Ainsi, `<self` fait référence à un type ancré `like current`[2], alors que `=self` fait référence exactement au type du receveur. Les types de retour se distinguent suivant qu'ils sont statiques (`Stat p.Type`) ou inférés (`p.Type`) : les premiers ne servent que pour les instanciations.

Encore une fois, les attributs sont décrits comme des méthodes dont le type de retour est égal au type du paramètre.

Chaque définition de méthodes sera donc décrite avec la syntaxe suivante :

```
MethDescr = method: Methodname
               parameters:   { ParameterType* }
               returns:      ReturnType
               instanciates: { InstanciationType* }
               calls:        { MethodCall* }
MethodCall = Methodname ReceiverType { ArgumentType* }
```

les différents types étant décrits par des sous-ensembles de la grammaire générale :

```
ParameterType  = Prim | Param | <Class
ReturnType = Type
InstanciationType = =Class | =self | Stat p.self
ReceiverType = Type
ArgumentType = Type
```

Le type du paramètre est en effet, soit un type formel, soit le type statique du paramètre dans la méthode, si le type formel n'est pas utilisé dans les appels extérieurs. Quant aux instanciations, elles ne peuvent porter que sur une classe précise (`=Class`), ou sur un type ancré, à la EIFFEL, que l'encapsulation le fait alors restreindre à `self` : les autres types comme `<Class` sont sans objet. Une généralisation des types

[1] Cette approche est donc parfaitement duale de l'approche classique dans laquelle un type est un ensemble d'opérations : les *treillis de Galois* unifient les deux visions [].

[2] Il semble que `<self` ne soit utilisé qu'en absence d'inférence de types, par une analyse intra-procédurale minimale. En effet, `<self` n'a de sens que dans la méthode appelée, l'appelante ne l'utilisant jamais : mais ce sera alors soit un type `Param`, soit un type de retour d'attribut, donc un `p.Type`.

d'instanciation consisterait à pouvoir créer un objet de même type (instance de même classe) qu'un autre : n'importe quelle expression de `Type` serait donc admissible. Les méthodes de copie ou de clonage ont cette fonctionnalité, sans qu'il soit nécessaire de l'inclure dans la grammaire d'instanciation, puisque cela serait pris en compte simplement au niveau de la signature de la méthode :

```
method: klone
     returns:      =self
     instanciates: { =self }
```

Les autres types sont *a priori* quelconques.

Le degré zéro de l'analyse intra-classe est accessible à n'importe quel compilateur et consiste à utiliser le sous-langage :

```
Type  = <Stat
Stat  = le type statique de l'entité
```

La version la plus sophistiquée distinguera les appels du même sélecteur qui diffèrent par leurs types de paramètres, avec une expression : $m.r_i(a_i^1, .., a_i^{k_m})$ par appel de méthode, alors que la plus grossière fusionnera tous ces appels en une seule expression : $m.\cup_i r_i(\cup_i a_i^1, .., \cup_i a_i^{k_m})$, et qu'une version intermédiaire ne fusionnera que les appels sur les mêmes receveurs : $m.r_i(\cup_{r_j=r_i} a_j^1, .., \cup_{r_j=r_i} a_j^{k_m})$.

A l'édition de liens, le graphe d'appel peut être construit et une analyse globale, comme la RTA [Bacon et Sweeney, 1996] ou plus sophistiquée, peut être appliquée. Le schéma d'appel de chaque méthode peut, en particulier, être recopié dans les sous-classes qui en héritent, ce qui permet de rendre `=self` monomorphe dans les expressions de type : `=self` est remplacé partout par `=Class`, où `Class` est la classe où figure l'occurrence de `=self`.

Quelle que soit la technique choisie, la détermination des classes vivantes et leur seule inclusion dans l'exécutable est immédiate. On peut faire la même chose pour distinguer les méthodes vivantes et les méthodes mortes d'une même classe, sous réserve que le format de sortie du compilateur le permette. Enfin, pour accélérer les recompilations, divers niveaux d'analyse sont possibles.

Si cette approche permet de concilier la compilation séparée et des analyses globales, elle présente néanmoins un ultime inconvénient : l'*inlining* est impossible, de même que le changement de techniques d'implémentation.

La spécification détaillée de l'analyse de types en compilation séparée est développée dans le mémoire de DEA de Jean Privat [Privat, 2002].

Bibliographie

[Agesen *et al.*, 1997] O. Agesen, S. Freund, et J.C. Mitchell. Adding type parameterization to Java. In OOPSLA [1997], pages 49–65.

[Alpern *et al.*, 1999] B. Alpern, C.R. Attanasio, J.J. Barton, A. Cocchi, S.F. Hummel, D. Lieber, T. Ngo, M. Mergen, J.C. Shepherd, et S. Smith. Implementing Jalapeño in Java. In OOPSLA [1999], pages 314–324.

[Alpern *et al.*, 2001a] B. Alpern, A. Cocchi, S. Fink, et D. Grove. Efficient implementation of Java interfaces : Invokeinterface considered harmless. In *Proc. OOPSLA'01*, SIGPLAN Notices, 36(10). ACM Press, 2001.

[Alpern *et al.*, 2001b] B. Alpern, A. Cocchi, et D. Grove. Dynamic type checking in Jalapeño. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[America, 1991] éditeur P. America. *Proceedings of the Fifth European Conference on Object-Oriented Programming, ECOOP'91*, LNCS 512. Springer-Verlag, 1991.

[Ancona *et al.*, 2000] D. Ancona, G. Lagorio, et E. Zucca. JAM — a smooth extension of Java with mixins. In Bertino [2000], pages 154–178.

[Ancona et Zucca, 2001] D. Ancona et E. Zucca. True modules for Java-like languages. In Knudsen [2001], pages 354–380.

[André et Royer, 1992] P. André et J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In *Proc. OOPSLA'92*, pages 110–126. ACM Press, 1992.

[Arnold et Gosling, 1997] K. Arnold et J. Gosling. *The JAVA programming language, Second edition*. Addison-Wesley, 1997.

[Bacon *et al.*, 2002] D. Bacon, S. Fink, et D. Grove. Space- and time-efficient implementation of the java object model. In *Proc. ECOOP'2002*, éditeur J. M. Troya, LNCS. Springer-Verlag, 2002.

[Bacon et Sweeney, 1996] D.F. Bacon et P. Sweeney. Fast static analysis of C++ virtual function calls. In OOPSLA [1996], pages 324–341.

[Bertino, 2000] éditeur E. Bertino. *Proceedings of the 14th European Conference on Object-Oriented Programming, ECOOP'2000*, LNCS 1850. Springer-Verlag, 2000.

[Biberstein *et al.*, 2001] M. Biberstein, J. Gil, et S. Porat. Sealing, encapsulation and mutability. In Knudsen [2001], pages 28–52.

[Borne et Godin, 2001] éditeurs I. Borne et R. Godin. *Actes des journées Langages et Modèles à Objets, LMO'2001*, in *L'Objet vol. 7*. Hermès, 2001.

[Bracha *et al.*, 1998] G. Bracha, M. Odersky, D. Stoutamire, et P. Wadler. Making the future safe for the past : Adding genericity to the Java programming language. In OOPSLA [1998], pages 183–200.

[Bracha et Cook, 1990] G. Bracha et W. Cook. Mixin-based inheritance. In *Proc. OOPSLA/ECOOP'90*, SIGPLAN Notices, 25(10), pages 303–311. ACM Press, 1990.

[Bracha et Lindstrom, 1992] G. Bracha et G. Lindstrom. Modularity meets inheritance. In *Proc. Int. Conf. on Computer Languages*, pages 282–290. IEEE, 1992.

[Cargill, 1991] T. A. Cargill. Controversy : The case against multiple inheritance in C++. *Computing Systems*, 4(1) :69–82, 1991.

[Cartwright et Steele, 1998] R. Cartwright et G.L. Steele. Compatible genericity with run-time types for the Java programming language. In OOPSLA [1998], pages 201–215.

[Caseau et Laburthe, 1999] Y. Caseau et F. Laburthe. Introduction to the CLAIRE programming language. Rapport technique, LIENS, 1999.

[Caseau, 1993] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. OOPSLA'93*, pages 271–287. ACM Press, 1993.

[Castagna, 1995] G. Castagna. Covariance and contravariance : Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3) :431–437, 1995.

[Castagna, 1997] G. Castagna. *Object-Oriented Programming : A Unified Foundation*. Progress in Theorical Computer Science. Birkhäuser, Berlin, 1997.

[Chambers *et al.*, 1997] C. Chambers, D. Grove, G. DeFouw, et J. Dean. Call graph construction in object-oriented languages. In OOPSLA [1997], pages 108–124.

[Chambers et Chen, 1999] C. Chambers et W. Chen. Efficient multiple and predicate dispatching. In OOPSLA [1999], pages 238–255.

[Chambers et Ungar, 1989] C. Chambers et D. Ungar. Customization : Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *Proc. OOPSLA'89*, pages 146–160, New Orleans, 1989. ACM Press.

[Chambers, 1993] C. Chambers. Predicate classes. In *Proc. of ECOOP'93*, éditeur O. Nierstrasz, LNCS 707, pages 268–296. Springer-Verlag, Berlin, 1993.

[Cohen, 1991] N.H. Cohen. Type-extension type tests can be performed in constant time. *Programming languages and systems*, 13(4) :626–629, 1991.

[Cointe, 1987] P. Cointe. Metaclasses are first class : The ObjVlisp model. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 156–167, 1987.

[Collin *et al.*, 1997] S. Collin, D. Colnet, et O. Zendra. Type inference for late binding. the SmallEiffel compiler. In *Joint Modular Languages Conference*, LNCS 1204, pages 67–81. Springer Verlag, 1997.

[Colnet *et al.*, 1998] D. Colnet, P. Coucaud, et O. Zendra. Compiler support to customize the mark and sweep algorithm. In *ACM Sigplan International Symposium on Memory Management (ISMM'98)*, pages 154–165, 1998.

[Cook, 1989] W. R. Cook. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89*, éditeur S. Cook, pages 57–70. Cambridge University Press, 1989.

[Day *et al.*, 1995] M. Day, R. Gruber, B. Liskov, et A. Myers. Subtypes vs. where clauses. constraining parametric polymorphism. In *Proc. OOPSLA'95*, pages 156–168. ACM Press, 1995.

[Dixon *et al.*, 1989] R. Dixon, T. McKee, P. Schweitzer, et M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*. ACM Press, 1989.

[Dolby et Chien, 1998] J. Dolby et A.A. Chien. An evaluation of automatic object inline allocation techniques. In OOPSLA [1998], pages 1–20.

[Driesen *et al.*, 1995] K. Driesen, U. Hölzle, et J. Vitek. Message dispatch on pipelined processors. In *Proc. ECOOP'95*, éditeur W. Olthoff, LNCS 952, pages 253–282. Springer-Verlag, 1995.

[Driesen et Hölzle, 1995] K. Driesen et U. Hölzle. Minimizing row displacement dispatch tables. In OOPSLA [1995], pages 141–155.

[Driesen, 1999] K. Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. Phd thesis in computer science, University of California, Santa Barbara, 1999.

[Drossopoulou *et al.*, 1998] S. Drossopoulou, D. Wragg, et S. Eisenbach. What *is* Java binary compatibility. In OOPSLA [1998], pages 341–358.

[Drossopoulou *et al.*, 2001] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, et P. Giannini. *Fickle* : Dynamic object re-classification. In Knudsen [2001], pages 130–149.

[Ducournau *et al.*, 1995] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, et A. Napoli. Le point sur l'héritage multiple. *Technique et Science Informatiques*, 14(3) :309–345, 1995.

[Ducournau et Pavillet, 2001] R. Ducournau et G. Pavillet. Langage à objets et logique de descriptions : un schéma d'intégration. In Borne et Godin [2001], pages 233–249.

[Ducournau, 1991] R. Ducournau. *Y3 : YAFOOL, le langage à objets, et YAFEN, l'interface graphique*. Sema Group, Montrouge, 1991.

[Ducournau, 1997] R. Ducournau. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet*, 3(3) :241–276, 1997.

[Ducournau, 2001a] R. Ducournau. La coloration : une technique pour l'implémentation des langages à objets à typage statique. i. la coloration de classes. Rapport de Recherche 01-225, L.I.R.M.M., 2001.

[Ducournau, 2001b] R. Ducournau. Spécialisation et sous-typage : thème et variations. Rapport de Recherche 01-013, L.I.R.M.M., 2001. *L'Objet*.

[Ducournau, 2002] R. Ducournau. La coloration pour l'implémentation des langages à objets à typage statique. In *Actes LMO'2002* in *L'Objet vol. 8*, éditeurs M. Dao et M. Huchard, pages 79–98. Hermès, 2002.

[Eckel et Gil, 2000] N. Eckel et J. Gil. Empirical study of object-layout and optimization techniques. In Bertino [2000], pages 394–421.

[Ellis et Detlefs, 1993] J. R. Ellis et D. L. Detlefs. Safe, efficient garbage collection for C++. Rapport technique, DEC and Xerox Corp, 1993.

[Ellis et Stroustrup, 1990] M.A. Ellis et B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading (MA), USA, 1990.

[Ernst *et al.*, 1998] M. Ernst, C. Kaplan, et C. Chambers. Predicate dispatching : A unified theory of dispatch. In Jul [1998], pages 186–211.

[Gagnon et Hendren, 2001] E. M. Gagnon et L.J. Hendren. SableVM : A research framework for the efficient execution of Java bytecode. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[Gil et Itai, 1998] J. Gil et A. Itai. The complexity of type analysis of object oriented programs. In *Proc. ECOOP'98*, LNCS 1445, pages 601–634. Springer-Verlag, 1998.

[Gil et Sweeney, 1999] J. Gil et P. Sweeney. Space and time-efficient memory layout for multiple inheritance. In OOPSLA [1999], pages 256–275.

[Goldberg et Robson, 1983] A. Goldberg et D. Robson. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading (MA), USA, 1983.

[Grand, 1997] M. Grand. *JAVA Reference manual*. O'Reilly, 1997.

[Habib *et al.*, 1997] M. Habib, L. Nourine, et O. Raynaud. A new lattice-based heuristic for taxonomy encoding. In *Proc. KRUSE'97*, pages 60–71, 1997.

[Hölzle *et al.*, 1991] U. Hölzle, C. Chambers, et D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In America [1991], pages 21–38.

[Huang et Chen, 1992] S.-K. Huang et D.-J. Chen. Coloring approaches for method dispatching in object-oriented programming systems. In *Proc. ICSA'92*, pages 39–44, 1992.

[Huchard et Leblanc, 2000] M. Huchard et H. Leblanc. Computing interfaces in Java. In *Proc. of IEEE Int. Conf. on Automated Software Engineering (ASE'2000)*, pages 317–320, 2000.

[Huchard, 2000] M. Huchard. Another problematic multiple inheritance mechanism : Construction and destruction in C++ in the virtual multiple inheritance case. *Journal of Object Oriented Programming*, july 2000.

[ILO, 1995] ILOG, Gentilly. ILOG TALK *reference manual, Version 3.2*, 1995.

[ILO, 1996] ILOG, Gentilly. POWER CLASSES *reference manual, Version 1.4*, 1996.

[Ingalls *et al.*, 1997] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, et A. Kay. Back to the future : The story of Squeak - a usable Smalltalk written in itself. In OOPSLA [1997], pages 318–326.

[Josset et Caseau, 2001] F.-X. Josset et Y. Caseau. Optimisation de code dans le langage CLAIRE. In *Actes LMO'2001* in *L'Objet vol. 7*. Hermès, 2001.

[Jul, 1998] éditeur E. Jul. *Proceedings of the Twelth European Conference on Object-Oriented Programming, ECOOP'98*, LNCS 1445. Springer-Verlag, 1998.

[Knudsen, 2001] éditeur J. L. Knudsen. *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'2001*, LNCS 2072. Springer-Verlag, 2001.

[Koenig, 1998] A. Koenig. Standard – the C++ language. Report ISO/IEC 14882 :1998, Information Technology Council (NCTIS), 1998. `http ://www.nctis.org/cplusplus.htm`.

[Krall *et al.*, 1997] A. Krall, J. Vitek, et R.N. Horspool. Near optimal hierarchical encoding of types. In *Proc. ECOOP'97*, éditeurs M. Aksit et S. Matsuoka, LNCS 1241. Springer-Verlag, 1997.

[Krall et Grafl, 1997] A. Krall et R. Grafl. CACAO - a 64 bits JavaVM just-in-time compiler. *Concurrency : Practice ans Experience*, 9(11) :1017–1030, 1997.

[Krogdahl, 1985] S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 35 :318–326, 1985.

[Lalande, 1926] A. Lalande. *Vocabulaire technique et critique de la philosophie*. Presses Universitaires de France, Paris, 1926.

[Liskov *et al.*, 1995] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, et A. C. Myers. THETA reference manual. Technical report, MIT, 1995.

[Meyer et Downing, 1997] J. Meyer et T. Downing. *JAVA Virtual Machine*. O'Reilly, 1997.

[Meyer, 1992] B. Meyer. *Eiffel : The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1992.

[Meyer, 1995] B. Meyer. *Object Success : A Manager's Guide to Object Technology, its Impact on the Corporation, and its Use for Reengineering the Software Process*. The Object-Oriented series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[Meyer, 1997] B. Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, second édition, 1997.

[Myers, 1995] A. Myers. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*, pages 124–139. ACM Press, 1995.

[Odersky et Wadler, 1997] M. Odersky et P. Wadler. Pizza into Java : Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[Omohundro et Stoutamire, 1995] S.M. Omohundro et D. Stoutamire. The Sather 1.0 specification. Technical Report TR-95-057, Int. Computer Science Institute, Berkeley (CA), USA, 1995.

[OOPSLA, 1995] *Proceedings of the Tenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'95*, SIGPLAN Notices, 30(10). ACM Press, 1995.

[OOPSLA, 1996] *Proceedings of the Eleventh ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'96*, SIGPLAN Notices, 31(10). ACM Press, 1996.

[OOPSLA, 1997] *Proceedings of the Twelfth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*, SIGPLAN Notices, 32(10). ACM Press, 1997.

[OOPSLA, 1998] *Proceedings of the Thirteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'98*, SIGPLAN Notices, 33(10). ACM Press, 1998.

[OOPSLA, 1999] *Proceedings of the Fourteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'99*, SIGPLAN Notices, 34(10). ACM Press, 1999.

[Palsberg et Schwartzbach, 1991] J. Palsberg et M. J. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91*, SIGPLAN Notices, 26(10), pages 146–161. ACM Press, 1991.

[Pavillet et Ducournau, 1999] G. Pavillet et R. Ducournau. Implémentation des attributs booléens par un Meta Object Protocol. In *Actes LMO'99*, éditeurs J. Malenfant et R. Rousseau, pages 55–68. Hermès, 1999.

[Privat, 2002] J. Privat. *Analyse de types et graphe d'appels en compilation séparée*. Mémoire de dea, Université Montpellier II, 2002.

[Pugh et Weddel, 1990] W. Pugh et G. Weddel. Two-directional record layout for multiple inheritance. In *Proc. SIGPLAN'90 Conf. on Programming Language Design and Implementation*, Special issue of ACM SIGPLAN Notices, 25(6), pages 85–91, 1990.

[Queinnec, 1997] Ch. Queinnec. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*, 1997.

[Raynaud et Thierry, 2001] O. Raynaud et E. Thierry. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proc. ECOOP'2001*, LNCS 2072, pages 165–180. Springer-Verlag, 2001.

[Rémy et Vouillon, 1997] D. Rémy et J. Vouillon. Objective ML : A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages (POPL'97), Paris, France*, pages 40–53, 1997.

[Rossie *et al.*, 1996] J. G. Rossie, D. P. Friedman, et M. Wand. Modeling subobject-based inheritance. In *Proc. ECOOP'96*, éditeur P. Cointe, LNCS 1098. Springer-Verlag, 1996.

[Rossie et Friedman, 1995] J. G. Rossie et D. P. Friedman. An algebraic semantics of subobjects. In OOPSLA [1995], pages 187–199.

[Sakkinen, 1992] M. Sakkinen. A critique of the inheritance principles of C++. *Computing Systems*, 5(1) :69–110, 1992.

[Simon *et al.*, 2000] R. Simon, E. Stapf, C. Mingins, et B. Meyer. Eiffel for e-commerce under .NET. *Journal of Object-Oriented Programming*, pages 42–47, october 2000.

[Snyder, 1991] A. Snyder. Modeling the C++ object model : An application of an abstract object model. In America [1991], pages 1–20.

[Solorzano et Alagić, 1998] J. H. Solorzano et S. Alagić. Parametric polymorphism for Java : A reflective solution. In OOPSLA [1998], pages 216–225.

[Steele, 1990] G.L. Steele. *Common Lisp : The Language, Second Edition*. Digital Press, Bedford (MA), USA, 1990.

[Stefik et Bobrow, 1986] M. Stefik et D.G. Bobrow. Object-oriented programming : Themes and variations. *The AI Magazine*, 6(4) :40–62, 1986.

[Szypersky *et al.*, 1994] C. Szypersky, S. Omohundro, et S. Murer. Engineering a programming language : The type and class system of Sather. In *Proc. of First Int. Conference on Programming Languages and System Architectures*, LNCS 782. Springer Verlag, 1994.

[Szypersky, 1992] C. Szypersky. Import is not inheritance. Why we need both : Modules and classes. In *Proc. ECOOP'92*, éditeur O. L. Madsen, LNCS 615, pages 19–32. Springer-Verlag, 1992.

[Vitek *et al.*, 1997] J. Vitek, R.N. Horspool, et A. Krall. Efficient type inclusion tests. In OOPSLA [1997], pages 142–157.

[Waldo, 1991] J. Waldo. Controversy : The case for multiple inheritance in C++. *Computing Systems*, 4(2) :157–171, 1991.

[Wang et Smith, 2001] T. Wang et S.F. Smith. Precise constraint-based type inference for java. In *Proc. ECOOP'2001*, LNCS 2072, pages 99–117. Springer-Verlag, 2001.

[Weber, 1992] F. Weber. Getting class correctness and system correctness equivalent — how to get covariant right. In *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, éditeurs R. Ege, M. Singh, et B. Meyer, pages 192–213, 1992.

[Zendra *et al.*, 1997] O. Zendra, D. Colnet, et S. Collin. Efficient dynamic dispatch without virtual function tables : The SmallEiffel compiler. In *Proceedings of OOPSLA'97, Atlanta (GA), USA*, special issue of ACM SIGPLAN Notices, 32(10), pages 125–141, 1997.

[Zendra, 2000] O. Zendra. *Traduction et optimisation globale dans les langages de classes*. Thèse d'informatique, Université Nancy 1, 2000.