ACADÉMIE DE MONTPELLIER
UNIVERSITÉ **M**ONTPELLIER **II**
Sciences et **T**echniques du **L**anguedoc

THÈSE

présentée au Laboratoire d'Informatique de Robotique
et de Microélectronique de Montpellier pour
obtenir le diplôme de doctorat

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Contribution à l'efficacité de la programmation par objets
Evaluation des implémentations de l'héritage multiple en typage statique

par

Floréal MORANDAT

Soutenue le 17, décembre 2010, devant le jury composé de :

Président du jury
Peter VAN ROY, professeur ..... Université de Louvain, Belgique

Directeur de thèse
Roland DUCOURNAU, professeur ..... LIRMM, Université Montpellier II

Rapporteurs
Manuel SERRANO, directeur de recherche ..... INRIA, Sophia-Antipolis
Jan VITEK, professeur ..... Purdue University, Indiana, USA

Examinateurs
Pascal GIORGI, maître de conférences ..... LIRMM, Université Montpellier II
Jean PRIVAT, professeur ..... UQAM, Montréal, Canada
Olivier ZENDRA, chargé de recherche ..... INRIA, LORIA, Nancy

A la mémoire d'un ami parti bien trop tôt …

Remerciements

Une thèse est une bien (trop) longue aventure qui n'aurait pas pu voir le jour sans de très nombreuses contributions scientifiques et humaines. Je vais m'efforcer d'en oublier le moins possible et je m'excuse par avance pour ceux qui pourraient être lésés.

Je tiens tout d'abord à remercier mon directeur de thèse Roland Ducournau avec qui j'ai pris un très grand plaisir à travailler. Je le remercie bien sûr pour toutes ses connaissances et son expérience sur les objets — mais bien au-delà [1] — qu'il a partagé durant toutes ces années avec moi. Mais mes plus grands remerciements vont à sa disponibilité, sa sagesse, sa rigueur, son enthousiasme en toutes circonstances et enfin à sa patiente sans fin envers ma prose. Merci !

Je remercie Jan Vitek et Manuel Serrano de m'avoir fait l'honneur d'être rapporteurs de ma thèse, ainsi que Peter Van Roy, Pascal Giorgi, Jean Privat et Olivier Zendra d'avoir accepté d'en constituer le jury. Je tiens tout particulièrement à remercier Jean pour le virus des langages qu'il a réussi à me transmettre et pour tout le plaisir que j'ai eu — et que j'aurais — à travailler avec lui, Olivier pour ses nombreuses conversations et Pascal quant à sa disponibilité pour partager ses connaissances sur l'architecture de nos ordinateurs.

Bien qu'une thèse soit un travail personnel, je remercie sincèrement toute l'équipe D'OC, mon équipe, avec qui j'ai passé des moments très agréables et enrichissants — bien que mouvementés. Avec une mention particulière pour André, Chouki, Christophe, Philipe et Pierre lors de ces échanges. Cependant je tiens tout spécialement à remercier Marianne et Clémentine pour à peu près tout, des petites attentions aux nombreuses discussions.

Durant ces années de recherche, il m'a aussi été possible d'enseigner avec de nombreuses personnes (talentueuses). Je commencerais pour remercier Ehoud pour les nombreuses charges qu'il a accepté à ma place avec le sourire — cela va de *tuteur* et du monitorat à *membre*de mon comité de suivi de thèse, en passant par la création d'une unité

1. si je puis me le permettre je dirais même bien plus qu'*au-delà*, je dirais *meta* !

d'enseignement. Je remercie Mathieu d'avoir accepté de compléter mon CST, ainsi que pour m'avoir fourni un si un bon sujet de discussion et de réflexion, jeuxdemots[2]. Un grand merci à Stéphane pour toutes les questions *systèmes* intrigantes qu'il a soulevées — et grâce auxquelles j'ai pu (ré)viser mes connaissances. Un second grand merci à mon Anne-Elisabeth pour les nombreuses heures passées à corriger des copies — et à rigoler aussi un peu, quand même. Enfin toutes les personnes avec qui j'ai appris à apprendre et apprécié d'enseigner, Chédi, Frédéric, Michel, Séverine et Yolande.

Je ne puis ignorer les personnes les plus méritantes du laboratoire, celles à qui je n'ai pas rendu la vie rose tous les jours, je veux parler de mes différents co-bureaux. Je commencerais donc par remercier Benoît pour son infinie patiente, Jean-Rémy pour les heures passées à s'entre-tuer — verbalement — pour être en fin de compte (quasi) systématiquement d'accord. Et plus récemment, Olivier qui n'aura pratiquement connu que ma période de rédaction — c'est à dire la période la plus gaie. Je remercie chaleureusement Raluca, Cécile, Fabio, J-P, Mathieu, qui ont partagé leur pain tous les midis avec moi. Enfin, un grand merci à tous les thésards — voire docteurs maintenant — que j'ai agréablement côtoyé, Benoît, Amine, Cécile, Céline, Gilles, Giullia, Gizmo, Grégory, Guillaume, Jean-Baka, Julien, Khalil, Lisa, Luc, Mehdi, Nicolas, Paola, Philippe, Rémy, Robin, Seb, Victor et Xavier.

Je remercie chaleureusement ma famille qui m'a soutenu tout au long de cette tumultueuse odyssée. Il me faut tout de même rajouter une mention spéciale à tata framboise pour la qualité et la promptitude de ses (trop) nombreuses relectures — non, papa, je ne t'oublie pas non plus — et merci à mon père pour les autres relectures. Si vous trouvez des fautes dans ce document, ne leur en voulez pas trop, c'est qu'à l'origine il y en avait vraiment tellement.

Je consacre les dernières lignes de ces remerciements à tous les amis (bien trop nombreux pour être cités ici) qui m'ont permis de tenir dans les moments de doute. A vous tous, merci : Aneso, Bazou, Ben, Boula, Damz, Deuf, Djow, Gond, Joan, Jouls, Laure, Loulou, Manu, Marion, Maudita, Olivier, Rafik, Seb, Steph, ...

Enfin, merci à vous, lecteurs anonymes, qui donnez un sens à ce document.

2. http://www.jeuxdemots.com

Table des matières

CHAPITRE

1 Introduction

Cette thèse s'inscrit dans le cadre de la programmation par objets. Ce paradigme de programmation est, de nos jours, universellement répandu, tant dans les milieux académiques qu'industriels. La liste des langages proposant les principes de l'objet est longue, on peut citer parmi les plus connus SMALLTALK, CLOS, C++, EIFFEL, OBJECTIVEC, JAVA, C# ou plus récemment PYTHON et RUBY. On retrouve les principes de l'objet même dans les langages dont la spécificité est ailleurs, comme ADA 2005 ou OCAML. La programmation par objets est principalement caractérisée par un mécanisme original, l'*invocation de méthodes*, aussi connue sous la métaphore anthropomorphique de l'*envoi de message — c'est l'objet qui détermine son comportement.*

Parmi les langages à objets, nous nous intéresserons seulement à une sous-catégorie, celle des langages à *classes*. Les classes factorisent les propriétés de leurs instances et elles sont reliées les unes aux autres par la relation de *spécialisation*. C'est cette relation de spécialisation qui fait la spécificité des langages objets à classes. La sémantique de l'approche par classe est simple et naturelle, on peut l'attribuer à la tradition aristotélicienne et plus particulièrement au syllogisme de *Socrate*. Son interprétation intuitive se formalise en termes ensemblistes. Dans les langages à classes, ce n'est donc plus l'objet qui détermine son comportement mais sa classe.

Nous nous plaçons aussi dans le contexte du *typage statique*. Le typage statique consiste à annoter par des types les divers éléments du langage de sorte que le type d'une expression puisse se déduire directement du contexte statique. Ces annotations restreignent les domaines de valeur que peuvent prendre les divers éléments du langage — variables, paramètres, etc. Elles augmentent le pouvoir expressif du langage — le programmeur exprime formellement ses intentions vis-à-vis des domaines de valeur — mais limitent la liberté du programmeur car il doit préciser les types statiques et s'y conformer. Bien que les annotations de type informent le compilateur sur les intentions du program-

meur, le *type dynamique* — la classe — des objets reste le seul à déterminer leur comportement à l'exécution. Dans un premier temps, on peut identifier, pour les langages à classes, les types aux classes et la relation de spécialisation à la relation de sous-typage. Même si cette identification semble naturelle, comme nous le verrons plus tard, elle ne l'est pas totalement et soulève quelques difficultés.

Le principe de la programmation par objets présente un problème d'implémentation [1] original, la *liaison tardive* qui est considérée comme le goulot d'étranglement de l'approche par objets. Cette liaison tardive correspond au fait qu'il est impossible statiquement de prévoir quelle sera la méthode appelée. C'est l'objet qui détermine son comportement. Au mécanisme principal d'appel de méthode, se rajoutent deux mécanismes accessoires mais aussi soumis à la liaison tardive, le *test de sous-typage* et l'*accès aux attributs*. Ces trois mécanismes sont les *mécanismes de base* spécifiques à la programmation par objets. L'implémentation des objets consiste à définir : la *représentation* des objets, une *structure de données statique* qui sert de support à la liaison tardive et que les objets doivent référencer, et enfin des algorithmes ou séquences de code qui mettent en œuvre les mécanismes de base.

Lorsqu'une classe en spécialise directement plusieurs autres, on parle d'*héritage multiple*. Depuis le début des objets, l'héritage multiple est un sujet de débats sans fin, pour des raisons de sémantique d'un côté et pour des raisons d'implémentation et d'efficacité de l'autre. Cependant le pouvoir expressif offert par l'héritage multiple est requis — tout du moins en typage statique — pour modéliser des problèmes réels. En pratique, l'héritage multiple est omniprésent dans les langages objet en typage statique. Outre les langages tels que C++ ou EIFFEL qui le proposent sans restrictions, tous les langages modernes en proposent au moins une forme dégradée, celle du *sous-typage multiple* où les classes sont en héritage simple et les *interfaces* en « héritage multiple ». Depuis JAVA, c'est le cas de C# et même de ADA 2005. De nouveaux langages, comme SCALA ou FORTRESS, reposent sur une notion intermédiaire, les *mixins* ou *traits*. Ce constat sur l'utilisation — même dégradée — de l'héritage multiple, nous amène à le mettre au cœur de cette thèse.

Nous considérons que la philosophie objet s'exprime pleinement avec l'*hypothèse du monde ouvert* , les classes sont des entités réutilisables qui doivent être conçues (vérifiées, compilées, etc.) indépendamment de leur usage futur. La modularité prônée par le génie logiciel n'est véritablement exprimable que dans cette hypothèse. Pour la programmation objet, cela signifie que la hiérarchie de classes n'est jamais connue dans son intégralité. Or l'hypothèse du monde ouvert a un effet sur l'héritage multiple, aussi bien sur le plan de la sémantique que sur celui de l'implémentation : il n'est jamais possible de savoir si deux classes, apparemment sans relations, auront une sous-classe commune ou pas. L'implémentation efficace de l'héritage multiple compatible avec l'hypothèse du monde ouvert

1. En français, on devrait utiliser le verbe *implanter* pour le verbe anglo-saxon *to implement*. Cependant le terme *implantation* est plutôt consacré aux problèmes de déploiements. Nous préférons donc l'utilisation du barbarisme *implémentation*.

demeure un problème entier. En effet, comme nous le montrerons, il n'existe actuellement aucune technique d'implémentation de l'héritage multiple, compatible avec cette hypothèse, aussi efficace que l'implémentation du sous-typage simple (héritage simple sans interfaces).

Cette dernière remarque, sur la compatibilité entre l'hypothèse du monde ouvert et une implémentation de l'héritage multiple, nous amène directement au sujet de cette thèse qui prend la suite de celle de Jean Privat [2006] : l'implémentation efficace des langages objet dans un cadre de typage statique et en héritage multiple.

L'efficacité des langages objet est conditionnée, d'un côté, par les techniques d'implémentation et, d'un autre côté, par le schéma de compilation. Le schéma de compilation représente la chaîne de production d'un programme, à partir d'un code source jusqu'à l'exécutable final. Cette chaîne de compilation met en jeu de nombreux éléments — compilation, analyses de types, éditions de liens, chargeur, ... Les schémas de compilation sont caractérisés par l'hypothèse avec laquelle ils considèrent le monde, de l'*hypothèse du monde ouvert* à l'*hypothèse du monde clos* , de la compilation séparée avec chargement dynamique utilisée par JAVA à la compilation purement globale faite par EIFFEL. Cependant toutes les techniques d'implémentation ne sont pas compatibles avec tous les schémas, les plus efficaces nécessitant l'hypothèse du monde clos. De plus, la plupart des techniques d'implémentation et certains schémas de compilation qui ont été proposés dans la littérature n'ont été testées que partiellement. Certaines combinaisons n'ont jamais été testées, quant aux autres elles ont généralement été évaluées en l'absence de référence objective.

Ceci nous permet de formuler clairement la question de fond essentielle de cette thèse : comment évaluer l'influence des schémas de compilation et des diverses techniques d'implémentation compatibles avec l'héritage multiple ?

Pour répondre à cette question, nous avons développé un compilateur, PRMC, qui propose la majorité des techniques d'implémentation que nous présentons dans cette thèse et ce dans le cadre de divers schémas de compilation. Jean Privat a cherché, lorsqu'il a commencé le travail qui nous sert de base, à modifier un compilateur existant. Cette tâche c'est avérée trop complexe et chronophage car les langages existants proposent trop de fonctionnalités non objet et leurs compilateurs ne sont tout simplement pas adaptés à la thèse qu'il défendait. Son travail l'a donc conduit à spécifier PRM, un langage purement objet, ainsi qu'à développer un prototype de son compilateur en RUBY. Nous avons repris le flambeau en développant, PRMC, un compilateur de PRM écrit dans ce même langage. Comme ce langage propose un système de *modules* et un mécanisme de *raffinement de classes*, le compilateur PRMC qui les utilise est modulaire.

Une fois le compilateur *bootstrappé*, nous l'avons utilisé pour proposer un protocole expérimental rigoureux basé sur la méta-compilation — c'est-à-dire la compilation du compilateur par lui même. En effet, le seul programme de taille significative disponible en PRM n'est autre que le compilateur lui-même. PRMC permet d'utiliser la plupart des techniques d'implémentation présentées dans cette thèse dans le contexte de différents schémas de compilation, de la compilation globale à la compilation séparée. Les expéri-

mentations que nous proposons se veulent systématiques et au maximum objectives, l'efficacité du programme testé n'est pas remise en question, seul l'impact sur le code généré est mesuré. Ce protocole rigoureux d'expérimentations nous permet de réaliser ces expériences « toutes choses égales par ailleurs », c'est-à-dire que la comparaison entre deux variantes est possible puisqu'elles ne différent que sur l'objet du test.

À notre connaissance, PRMC est le premier et unique compilateur qui propose une telle diversité d'implémentations et de schémas de compilation. PRMC est aussi le premier compilateur à implémenter, en vraie grandeur, des techniques comme la coloration, qui date pourtant d'une vingtaine d'année, ou le hachage parfait de classes, beaucoup plus récent. Les résultats qui seront présentés sont donc uniques.

Plan de la thèse

Cette thèse est découpée comme suit :

- Le chapitre 2 présente la problématique de la programmation par objets, la philosophie des objets et plus particulièrement celle des classes et l'héritage. La notion de typage statique est ensuite introduite et son rapport aux objets. Ce chapitre termine en présentant les spécificités de la programmation par objets qui nécessitent une implémentation.
- Le chapitre 3 présente la base de notre approche méthodologique des langages à objets, la méta-modélisation. La méta-modélisation donne à la fois une sémantique, aux notions de base comme la spécialisation et l'héritage multiple, Elle fournit aussi les objets qui sont manipulés par le compilateur. Le méta-modèle considéré donne aussi une sémantique claire à l'héritage multiple, et nous l'appliquons aux notions de modules et raffinement de classes de façon isomorphe. Le chapitre se termine par la présentation du langage PRM dont la sémantique est donnée par ces deux méta-modèles et qui sera utilisé dans la suite de cette thèse comme cible de nos expérimentations.
- La problématique de l'implémentation des objets est présentée au chapitre 4, à travers l'implémentation du sous-typage simple. Cette implémentation simple permet de mettre en évidence l'originalité de la programmation par objet, la liaison tardive. Le chapitre débute par la présentation du principe de l'évaluation abstraite de l'efficacité des implémentations. L'évaluation abstraite de cette implémentation du sous-typage simple permet de définir des critères souhaitables pour toutes les implémentations. Enfin, le chapitre analyses les propriétés de cette implémentation simple qui seront utilisées par la suite.
- Le chapitre 5 est consacré aux diverses techniques d'implémentation de l'héritage multiple. C'est le plus long chapitre de cette thèse, car les techniques sont nombreuses et souvent délicates. La majorité des techniques compatibles avec l'héritage

multiple existantes est examinée et évaluée. Cet état de l'art est complété par les techniques utilisées dans le contexte simplifié du sous-typage multiple à la JAVA.

- Le chapitre 6 est consacré aux divers schémas de compilation, des plus classiques à de plus innovants, et ce, de l'hypothèse du monde ouvert à l'hypothèse du monde clos. Les schémas de compilation sont présentés décomposés en deux phases, une locale et une globale, et une quantité de tâches à accomplir. L'organisation des tâches dans ces phases définit les spécificités de chacun des schémas qui sont considérés dans cette thèse.
- Le compilateur PRMC concrétise tous les éléments présentés. Son architecture est brièvement décrite à travers les modules qui le constituent au chapitre 7. Le problème intrigant de son amorçage (*bootstrap*) est ensuite examiné. Le chapitre se termine en décrivant comment PRMC met en œuvre les différentes alternatives qu'il propose — techniques d'implémentation et schémas de compilation.
- Les expérimentations réalisées avec et sur ce compilateur sont présentées au chapitre 8. En particulier, les questions de la reproductibilité des tests et de la comparabilité des expériences seront discutés Le protocole de méta-compilation rigoureux par lequel ces expérimentations sont réalisées est détaillé. Les résultats de ces expérimentations, qui ont donné lieu à des publications, sont présentés et soigneusement commentés.

Cette thèse se termine au chapitre 9 par un rappel des principaux éléments qu'elle contient et ouvre sur quelques perspectives connexes à ce travail.

CHAPITRE

2 Problématique de la programmation par objets

Dans ce chapitre, nous aborderons la philosophie des langages objet et ce qui les caractérise de façon générale et abstraite. Nous examinerons ensuite la question du typage et de son rapport aux objets. Au passage, nous présenterons les mécanismes de la programmation par objets qui font l'originalité de son implémentation.

Sommaire

Les langages à objets, maintenant universellement répandus, sont apparus il y a plus d'une quarantaine d'années. Le premier d'entre eux est probablement SIMULA en 1967 [Birtwistle *et al.*, 1973]. Il a été suivi par SMALLTALK en 1972, résultat des travaux d'Alan Kay, qui véhicula les principes de la programmation par objets dans sa version plus achevée de 1980 [Goldberg et Robson, 1983]. Dans les années 1980, de nombreux langages à objets ont vu le jour (OBJECTIVEC, C++, EIFFEL, ...), puis dans les années 1990 les principes de l'objet connaissent un âge d'or et seront utilisés dans tous les domaines, tant dans le milieu académique que dans le milieu industriel : programmation, modélisation, base de données, etc. Si les objets ont conquis autant de domaines, c'est dû à leurs bonnes propriétés en termes de génie logiciel [Meyer, 1988; Gamma *et al.*, 1995] et à leur sémantique intuitive.

2.1 La programmation par objets

La programmation par objets repose sur un principe très simple, celui de l'encapsulation et de l'envoi de message. Ce sont les notions de spécialisation et d'héritage d'une part, et de typage d'autre part, qui entraînent la plus grande partie de sa complexité.

2.1.1 Le principe des objets

Les langages objets sont caractérisés par un grand principe, *l'encapsulation*, et un mécanisme fondamental, *l'invocation de méthode*, qui est connue sous le métaphore d'*envoi de message* et mis en œuvre par la *liaison tardive*. Ce mécanisme original nécessite une implémentation qui, dans une certaine mesure, représente le sujet de cette thèse.

L'encapsulation Les objets encapsulent les données et les traitements qui leur sont propres.

L'envoi de message C'est l'objet qui détermine son comportement en interprétant le message qu'il reçoit.

Remarque. Ces deux éléments définissent une première catégorie de langages à objets appelés *langages à prototypes*, qui sont des langages à objets dits « sans classes ». Toutefois, dans cette thèse, nous ne les considérons pas et le terme *langage à objets* est synonyme de *langages à classes*.

On peut rajouter à ces deux éléments, le concept de *classe* et leur relation de *spécialisation*.

La classe Les objets sont issus d'une classe qui factorise leurs propriétés communes, c'est-à-dire que les instances d'une même classe partagent la même structure de données et le même comportement.

La spécialisation Une classe peut spécialiser une autre classe, de ce fait elle *hérite* de ses propriétés. Les notions d'héritage ou de spécialisation sont duales, la première est une conséquence de la seconde.

2.1.2 Envoi de message

C'est le mécanisme fondamental de la programmation par objets. Il a porté de nombreux noms suivant les langages et nous les utiliserons indistinctement : *envoi de message*, *appel de méthode*, *invocation de méthode*[1], *liaison tardive*[2], ...

1. Actuellement le terme *invocation de méthode* semble le plus utilisé.

2. Ce terme n'est pas censé représenter un appel de méthode mais plutôt le moyen par lequel cet appel est réalisé, bien qu'en pratique de nombreuses personnes l'utilisent comme s'il s'agissait d'un synonyme. Nous ne l'utiliserons que dans son sens premier.

Le principe de l'encapsulation consiste à rassembler les données et leurs traitements au sein d'un même objet. Dans les langages à classes, le comportement n'est pas déterminé par l'objet lui même mais par sa classe. Tout objet est *instance* d'une classe, classe qui factorise son comportement. Ce comportement est réalisé par le biais des fonctions attachées à la classe qui sont appelées des méthodes. Le principe d'un appel de méthode est bien illustré par la métaphore de *l'envoi de message* utilisée par SMALLTALK. Lorsque qu'un objet reçoit un message, c'est-à-dire qu'on réalise une invocation de méthode sur celui-ci, c'est l'objet lui même qui détermine — liaison tardive — sa réponse, c'est-à-dire la méthode effectivement appelée.

2.1.3 Sémantique intuitive des classes et de la spécialisation

Un des grandes qualités de l'approche objet est sa sémantique simple et naturelle, qui est proche de la manière dont nous, humains, catégorisons les choses. Cette sémantique s'interprète intuitivement en termes ensemblistes, ce qui simplifie sa formalisation.

La sémantique naturelle qu'on attribue à la spécialisation remonte au syllogisme de l'école aristotélicienne :

> « Socrate est un homme.
> Or les hommes sont mortels.
> Donc Socrate est mortel. »

Dans ce syllogisme, *Socrate* est un objet, tandis que `homme` et `mortel` sont des classes. L'objet *Socrate* appartient à la *classe* des `hommes` et possède, par conséquent, toutes les propriétés d'`homme`. On dit que *Socrate* est une *instance* (*propre*) d'`homme`. Comme les `hommes` sont `mortels`, la première classe est une spécialisation de la seconde,. La relation de spécialisation est un ordre partiel strict de classes et elle est notée $\prec$. Comme `Homme` $\prec$ `Mortel`, *Socrate* est une instance de `mortel`, il possède aussi toutes les propriétés de `mortel`. Les *propriétés* d'une classe forment son *intension* (ou *compréhension* chez les auteurs les plus anciens) de cette classe, tandis que les instances sont l'*extension* de cette classe.

La sémantique ensembliste continue de s'appliquer lorsqu'une classe en spécialise directement plusieurs et permet de définir des propriétés sur les relations entre spécialisation, intension et extension.

Lorsqu'on s'intéresse aux instances d'une classe, elles sont toutes instances de ses super-classes.

Propriété 2.1 *L'extension d'une classe est l'incluse dans l'intersection des extensions de ses super-classes.*

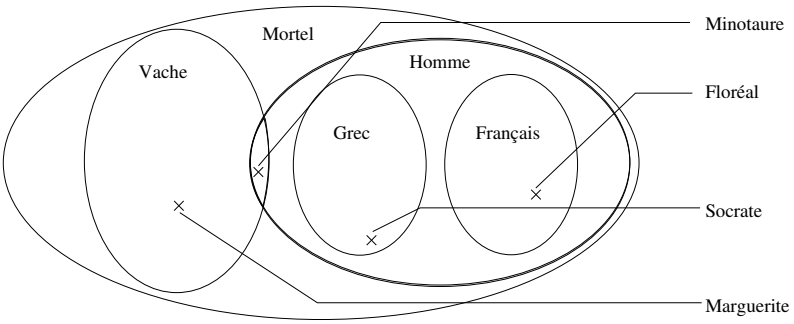
Tandis que si on s'intéresse à ces propriétés, la relation est inversée. Si une classe spécialise une autre classe, elle est au moins capable de faire tout ce que faisait sa super-classe. Elle *hérite* de toutes ses propriétés et peut en rajouter de nouvelles. L'héritage est donc une conséquence de la spécialisation.

Propriété 2.2 *L'intension d'une classe contient l'union des intensions de ses super-classes.*

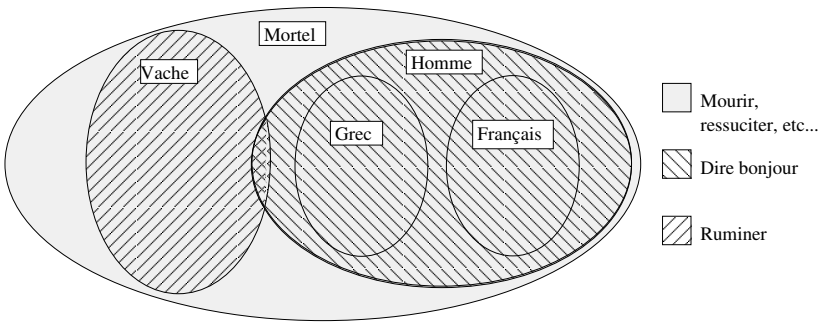
Cette vision ensembliste est naturelle et, à notre avis, c'est la seule façon correcte d'interpréter la sémantique du paradigme objet. Elle servira à poser les bases d'un méta-modèle pour l'héritage multiple (voir Section 3.2) et sera la justification de tous les choix qui pourraient être faits.

Exemple. On rajoute aux entités déjà présentes dans le syllogisme précédent, les classes `Grec`, `Français` et `Vache` sous-classes d'`homme`, pour les deux premières, et de `mortel`, pour la dernière. Si *Socrate* est un `Grec`, *Floréal* un `Français` et *Marguerite* une `Vache`, ils sont tous les trois mortels, donc appartiennent à l'extension de `Mortel`.

Si l'on rajoute la propriété « dire bonjour » aux hommes et « ruminer » aux vaches, *Floréal* et *Socrate* peuvent tout deux « dire bonjour », bien que le premier le fasse en français et le second en grec. Il s'agit bien de la même propriété mais elle a un comportement différent, on entrevoit ici le principe de la redéfinition de propriété. La propriété « dire bonjour » est redéfinie dans les deux sous-classes `Grec` et `Français`, et c'est la classe de l'instance qui détermine le comportement de l'objet. *Marguerite,* elle n'est qu'une `vache`, elle ne peut donc pas « dire bonjour » mais elle peut « mourir » et « ruminer ».



(a) Extension



(b) Intension

FIGURE 2.1 : Relation entre les intensions et les extensions d'une classe

Pour terminer l'exemple, considérons le *Minotaure*, seul représentant de son espèce, les `Hommes Vaches`, qui sont à la fois des `hommes` et des `vaches` et possèdent la capacité de « dire bonjour », de « ruminer » et de « mourir ». On constate aisément sur l'exemple que le *Minotaure* appartient bien à l'intersection des ensembles d'extensions des `Hommes` et des `Vaches` et que son intension — ensemble de ses propriétés — contient l'union des intensions de ses super-classes.

2.2 Typage

La section précédente suffit pour introduire les grands principes de la programmation par objets dans un contexte de typage dynamique, pour SMALLTALK par exemple. Cependant, c'est le typage statique qui introduit une grande part de la complexité des langages objets et tout le travail de cette thèse se place dans ce contexte.

Bien que certains des termes qui sont associés au typage soient un peu flous [3], nous les considérons uniquement dans leur signification la plus communément admise.

2.2.1 Typage statique

Le typage statique consiste à associer des annotations de type aux variables et aux méthodes dans le code source du programme, de sorte que le type statique de chaque expression se déduise directement du contexte de l'expression. Un type est, plus ou moins, la représentation informatique d'un *domaine* — en mathématique —, du moins c'est ce qui s'en rapproche le plus. L'ensemble des valeurs que peut prendre une entité est donc restreinte par son type, ceci augmente l'expressivité du langage puisqu'il est possible grâce aux types d'exprimer la notion de domaine. La généricité (voir Section 2.2.4), les types virtuels ou la covariance (voir Section 2.2.3) augmentent d'autant l'expressivité du langage. En contrepartie, le typage statique impose au programmeur de préciser les types des choses en rajoutant des annotations dans tout le programme qui peuvent freinent le prototypage rapide des applications.

Cependant, les avantages fournis par le typage statique sont nombreux, tant du point de vue du programmeur que de celui du compilateur. Pour le compilateur, les principaux bénéfices sont :

– Le compilateur d'un langage à typage statique détecte la majorité des erreurs de types avant que le programme ne soit exécuté. S'il détecte la totalité des erreurs de type, on parle de typage sûr.

3. « I spent a few weeks… trying to sort out the terminology of *strongly typed, statically typed, safe,* etc., and found it amazingly difficult… The usage of these terms is so various as to render them almost useless. » Benjamin C. Pierce

– Le compilateur peut tirer parti de l'information sur les types pour réaliser des optimisations sur le code produit. Notamment, tous les appels de méthodes font référence à une méthode qui existe, elle sera donc forcement présente dans la table des méthodes. De plus, en groupant les propriétés par leur classe d'introduction (voir Section 4.6.1), le compilateur peut utiliser la plupart des techniques qui serons présentées au chapitre 5. Enfin, il peut éliminer tout ou partie des tests de sous-typage.
– Puisque les types des objets manipulés sont connus, le compilateur peut éliminer cette information du code qu'il produit, réduisant ainsi sa taille.

Remarque. Il existe une intéressante dualité entre les langages à typage statique et ceux à typage dynamique. Les premiers utilisent des annotations de type dans le code source et les font disparaître lors de la compilation. Tandis que les langages à typage dynamique se passent des annotations dans les sources mais maintiennent l'information de type sur les valeurs durant l'exécution.

Pour le programmeur, on peut aussi citer quelques avantages :
– Les types lui permettent d'expliciter les abstractions qu'il manipule. Ce qui est encore renforcé par les langages à objets, où les types se confondent généralement avec les classes — ou interfaces.
– Le types permettent de désambiguïser le nom de propriétés homonymes en servant de contexte. Ceci assure l'existence des propriétés que le programmeur appelle et permet à un éditeur de développement [4] (EDI) de l'assister dans sa tâche de développement — par exemple, en lui proposant le nom des méthodes qu'il peut appeler.
– Enfin, les annotations de types peuvent servir à la documentation puisqu'elles illustrent une partie de ce que pense le programmeur. Elles sont en quelque sorte une condition nécessaire au bon fonctionnement du code.

Le débat entre défenseurs du *typage statique* et du *typage dynamique* n'est pas récent et risque de durer encore un moment. D'autant plus, qu'une troisième alternative est en train d'émerger, les systèmes de types facultatifs (ou *pluggable*) [Bracha, 2004] — cependant elle n'en est qu'à un stade balbutiant, par exemple langage STRONGTALK [Bracha et Griswold, 1993], et ses seules applications industrielles sont réalisées au travers des annotations de JAVA (ou C#). Toutefois, le langage PRM suivrait plutôt le principe de Meyer [1995] : *« Serious use of object technology requires static typing »* .

2.2.2 Typage sûr

La problématique du typage statique est de permettre au compilateur de vérifier qu'un programme, ou un fragment de programme, est sûr. Il existe au moins deux définitions de typage sûr [Cardelli, 2004] :

4. Éditeur de développement intégré. C'est un logiciel à la ECLIPSE, NETBEANS, VISUALSTUDIO, ... qui permet, entre autres, d'aider le programmeur dans sa compréhension du programme en lui proposant, par exemple, des *info bulles* sur l'objet, le type, ou le module qu'il utilise.

1. Un système de type est sûr, s'il ne peut pas y avoir d'erreur de types à l'exécution.
2. Un système de type est sûr si toutes les erreurs de types peuvent être rattrapées à l'exécution.

La politique de typage d'un langage peut être non-sûre d'après la première définition, c'est-à-dire qu'il peut avoir des erreurs de types à l'exécution. Cependant, si le langage est statiquement typé et son système de type bien formé (par opposition au terme anglais *ill-typed*), le compilateur sait où ces erreurs peuvent se produire. Il peut donc se prémunir en conséquence en générant les vérifications dynamiques nécessaires, sa politique de typage est donc sûre d'après la seconde définition.

2.2.3 Sous-typage

Chaque valeur est un objet et a un *type dynamique*, sa classe. Chaque variable, de son côté, a un *type statique*. À l'exécution, la variable doit être valuée par une valeur dont le *type dynamique* est conforme au *type statique*. Néanmoins, à l'exécution, c'est le `type dynamique` seul qui détermine le comportement d'un objet, c'est-à-dire la classe dont il est instance propre. Cette conformité, entre type statique et type dynamique, passe généralement par la notion de sous-typage, qui s'exprime, dans le cadre du typage sûr par le principe de *substituabilité* .

Définition 2.1 (Substituabilité) *Un type t_1 est un sous-type d'un type t_2 si toute valeur du type (dynamique) t_1 peut être substituée, à l'exécution, à toute expression du type (statique) t_2 , sans déclencher d'erreur de type.*

Dans les langages objets, il est tentant d'identifier les types à des classes et la relation de sous-typage à celle de spécialisation. Comme nous allons le voir, cette identification qui semble naturelle peut poser de nombreux problèmes. Tout d'abord elle n'est plus vraie si on tient compte des types d'ordre supérieur (types paramétrés par d'autres). Ensuite, elle impose des restrictions sur le type des que peuvent prendre les paramètres d'une méthode.

Supposons l'appel suivant `a.foo(u)`, avec `a:A` et `u:U`. Maintenant, considérons $B \prec A$ — B est une sous-classe de A, et $V \prec U \prec T$. La règle de substituabilité impose que ce code doive marcher pour les sous-types de A, donc pour B. Pour être valide, la méthode `B::foo` (la méthode `foo` définie dans la classe B) doit au moins accepter toutes les valeurs qu'aurait acceptées la méthode `A::foo`, son paramètre doit donc être plus général qu'un U, par exemple un T. Pour le type de retour, la substituabilité impose que `B::foo` ne retourne que des valeurs qu'aurait pu retourner `A::foo` — c'est-à-dire que le type doit être plus spécifique. Pour que la spécialisation soit du sous-typage la redéfinition doit donc respecter la règle dite de contravariance.

Définition 2.2 (Règle de contravariance) *La redéfinition d'une méthode dans une classe doit généraliser le type des paramètres et spécialiser le type de retour de la méthode redéfinie.*

Les attributs peuvent être considérés au travers de deux méthodes qui y accèdent, ces méthodes sont appelées accesseurs. Une des deux méthodes a un paramètre typé par le type de l'attribut — attribut en écriture —, l'autre sans paramètres mais qui retourne le type de l'attribut — attribut en lecture. En respectant la contrainte de la contravariance, ces deux méthodes imposent *l'invariance* aux attributs. Avec la covariance cette contrainte est relaxée et l'attribut peut être redéfinit de façon covariante..

Remarque. En JAVA (jusqu'à la 1.5), la spécification du langage imposait la règle d'*invariance*. Cependant, comme la spécialisation du type de retour est compatible avec un typage strict et une implémentation avec invariant de référence (voir Section 4.3), depuis les versions supérieures à 1.5, la spécialisation du type de retour est autorisée.

Dans les implémentations sans invariant de référence (par exemple les sous-objets de C++) même la covariance du type de retour peut poser un problème, car il faut ajuster le type statique de la référence retournée (voir Section 5.1).

2.2.4 Généricité

Dans les langages objets, la généricité est proposée au travers de classes paramétrées — par abus de langage on parle aussi de classes génériques. On peut voir les classes paramétrées de deux manières différentes. Soit d'un point de vue mathématique où une classe paramétrée est une fonction des types vers les types. Ou bien du point de vue de la modélisation, comme une méta-classe, c'est-à-dire une classe dont les instances sont des classes, et dont les méta-propriétés sont des *types formels*. Instancier la classe paramétrée revient à instancier les types formels par des types réels [5]. Dans la vision fonctionnelle, l'instanciation de la classe paramétrée correspond à l'application de la fonction avec les types réels comme arguments.

La *généricité* est un moyen de factoriser du code — on ne programme la classe paramétrée qu'une seule fois en faisant abstraction des types et on la réutilise pour n'importe quel type — tout en conservant le typage statique et la sécurité qu'il apporte. Les paramètres formels peuvent être bornés ou non et lorsqu'ils le sont, le paramètre formel peut servir à décrire la borne. On parle dans ce dernier cas de généricité F-bornée.

Les langages comme JAVA, C# ou SCALA rajoutent un nouveau [6] cas de généricité, il s'agit des méthodes paramétrées — ou génériques. Comme les classes paramétrées, les méthodes paramétrées peuvent faire référence à des types formels mais la portée de ces types est limitée à la méthode qui les déclare. Pour être utilisées ces méthodes ont besoin, elles aussi, d'être instanciées, ce qui en fait des méthodes relativement verbeuses à utiliser. Elles sont généralement utilisées pour factoriser des algorithmes dans des classes utilitaires comme `Collection` ou `Arrays` de JAVA. L'implémentation de ces méthodes présente

5. *Instancier* est le terme consacré dans les deux cas et il est ici manifestement polysémique.

6. Le terme « nouveau » est relatif aux langages objets car historiquement la généricité est d'abord apparue dans les langages fonctionnels qui proposaient des fonctions paramétrées.

les mêmes problèmes que l'implémentation de la généricité en toute généralité (voir Section 5.10.3) et le cas particulier de ces méthodes de ne sera discuté dans cette thèse.

2.2.5 Covariance et types virtuels

Le typage sûr et la règle de contravariance rajoutent des contraintes qui limitent la liberté du programmeur, au contraire de la covariance ou des types-virtuels qui augmentent l'expressivité.

Covariance

La règle de contravariance, qui est, du point de vue des types, très sensée, semble pourtant contre-intuitive et partiellement en désaccord avec la philosophie de la spécialisation. Le bon sens voudrait qu'une sous-classe, étant plus spécifique que ses parents, puisse spécialiser les méthodes avec des types plus spécifiques. Ce principe est connu sous le nom de *covariance*. La covariance est nécessaire pour modéliser le monde réel [Shang, 1996; Ducournau, 2002] et elle est utilisée par le langage EIFFEL [Meyer, 1988]. Cependant la covariance nécessite de relaxer la contrainte du typage sûr — dans sa première définition.

Pour illustrer la covariance, prenons l'exemple proposé par Shang des animaux, dont les vaches, et de la nourriture. Soient les classes `Vache` ≺ `Animal` et `Vegetal` ≺ `Nourriture`. La classe `Animal` introduit la méthode `mange(nourriture: Nourriture)`. Nous aimerions redéfinir cette méthode `mange` pour les vaches comme ceci : `mange(nourriture: Vegetal)`. Pour assurer la sûreté du typage, la règle de contravariance ne permettrait à la vache que de manger de la nourriture ou quelque chose de plus générique. D'où la fameuse question de Shang : « Are cows animals ? »

Si on accepte une politique de typage non sûre, la covariance permet au programmeur de mieux exprimer le domaine des méthodes qu'il utilise. Malgré les erreurs de types potentiels qui peuvent résulter de la règle de covariance [7], il est tout de même possible de rendre le typage sûr d'après sa seconde définition. Il suffit pour cela de rajouter des tests de sous-typage sur les arguments des méthodes ayant un redéfinition covariante. Dans le pire des cas, ces tests doivent être faits systématiquement au début des méthodes, alourdissant de ce fait tous les appels potentiellement covariants même lorsqu'ils sont appelés sur des formes sûres.

Types virtuels

Les types virtuels [Torgersen, 1998] permettent une alternative élégante à la covariance des paramètres tout en augmentant la sûreté du typage et l'efficacité. A première vue, les

7. Par exemple, si on nourrit une vache lorsqu'elle est typé par `Animal` avec de la viande. Finalement, la maladie de la vache folle ne serai qu'une erreur de types.

types virtuels ressemblent à de la généricité mais avec comme différence notable que ce sont les sous-classes qui redéfinissent les types et non le client qui les instancie.

Les types virtuels sont des propriétés de type *type* borné par une classe. A la définition du type virtuel, le programmeur peut imposer que sa borne soit finale. Si ce n'est pas le cas, il peut être redéfini de façon covariante par une sous-classe. Lorsque la borne est déclarée finale, les types virtuels sont sont sûr mais cela nuit à la réutilisabilité — comme toutes les propriétés finales, puisqu'elles ne peuvent plus être redéfinies.

Avec les types virtuels, une nouvelle politique de type peut être considérée, l'invariance des signatures.

Définition 2.3 (Invariance des signatures) *Toutes les méthodes ont des signatures invariantes par redéfinition. Seuls les types virtuels associés à une signature peuvent être spécialisés.*

Cette nouvelle règle ne rend toujours pas le système de type sûr — d'après la première définition — mais circonscrit les endroits où des erreurs de types peuvent survenir. Les erreurs sont localisées aux appels de méthodes utilisant des types virtuels, ce qui simplifie le travail du compilateur qui peut considérer tous les autres appels comme sûrs.

La covariance peut être simulée en utilisant des paramètres typés par des types virtuels ; chaque sous-classe pouvant spécialiser ces types à sa guise. Les méthodes ayant des types de paramètres qui peuvent varier doivent être introduites en utilisant des types virtuels. Les cas de covariance non prévus seraient à traiter par un test de sous-typage explicite (voir Section 2.3) [Shang, 1996].

2.2.6 Surcharge statique

La surcharge statique (*method overloading*) est une fonctionnalité souvent rencontrée dans les langages de programmation objet (C++, C#, JAVA, etc mais pas EIFFEL). Elle consiste à ce qu'une même classe puisse avoir plusieurs méthodes de même nom mais avec des types ou des nombres de paramètres différents.

Lorsque le compilateur rencontre un appel à une méthode surchargée, il doit désambiguïser l'appel en fonction du contexte statique, c'est-à-dire trouver la bonne signature (voir Section 3.2) en fonction du nombre de paramètres ou de leurs types. Lorsque le compilateur ne parvient pas à désambiguïser la méthode à appeler, le programmeur est mis à contribution pour l'aider en modifiant le type statique (voir Section 2.3.1). L'exemple 2.1 montre un cas de surcharge statique ambiguë.

La surcharge n'est clairement pas une redéfinition car il s'agit de méthodes réellement différentes mais homonymes. En pratique, la surcharge n'est pas nécessaire à un langage de programmation objet. Le programmeur peut introduire des méthodes nommées différemment, qu'il appellerait en fonction du type de paramètres qu'il souhaite.

Remarque. Il existe néanmoins des cas où la surcharge statique peut s'avérer utile. Pour simuler des méthodes ayant des arguments par défaut (voir Section 3.4.3), par exemple,

LISTING 2.1 : Exemple de surcharge statique ambiguë

```
class A
    def foo(bar: A, baz: B)
    ...
end
class B inherit A
    def foo(bar: B, baz: A)
    ...
end
...
let a, b: B
...
a.foo(a, b) # est ambigu
a.foo(A(a), b) # ok, appele la premiere methode
a.foo(a, A(b)) # ok, appele la seconde methode
```

ou dans les langages ne disposant que de la règle d'invariance — ou de contravariance —, pour simuler la covariance. Il est en effet possible de simuler la covariance par une utilisation conjointe de redéfinition, surcharge statique et *cast*.

Cependant comme la surcharge n'est pas indispensable dans un langage à objets et qu'elle complexifie le modèle, elle ne sera pas vraiment considérée dans le reste de cette thèse.

2.2.7 Sous-typage multiple

D'un côté, l'héritage simple strict est considéré comme trop restrictif pour les besoins de modélisation, en tout cas en typage statique car les défenseurs de SMALLTALK s'en contentent. De l'autre, l'héritage multiple est réputé comme trop compliqué à utiliser et à implémenter. Le sous-typage multiple est proposé comme un compromis entre les deux alternatives. Il est constitué d'un cœur en héritage simple et d'un système de types en héritage multiple. C'est une forme dégradée de l'héritage multiple, qui est utilisée dans les langages JAVA, C# ou ADA 2005.

Le système de type rajoute aux classes des *interfaces*. Une interface peut avoir un nombre quelconque de super-interfaces. Ce sont des classes abstraites restreintes, sans attribut, et où toutes les méthodes qui y sont définies sont obligatoirement abstraites. Elles servent principalement de types.

Les classes, elles, n'ont qu'une seule super-classe directe. Par contre, elles peuvent *implémenter*[8] plusieurs interfaces.

8. Pour une classe, *implémenter une interface* correspond à la spécialiser. Le mot *implémenter* est particulièrement ambigu dans une thèse sur l'implémentation des langages objets, cependant c'est le mot retenu en JAVA. Nous ne l'utiliserons avec ce sens que suivi du mot « interface ».

Il y a donc une dichotomie entre classes et interfaces. Ceci permet de profiter d'une part, d'une sémantique simple et d'une implémentation efficace pour les classes (voir Chapitre 4), et d'autre part, de la puissance de modélisation de l'héritage multiple, fourni par les interfaces. Dans ce contexte on ne devrait, en théorie, typer que par des interfaces et limiter les classes aux implémentations — c'est d'ailleurs le principe de SCALA (voir Section 5.9.3).

2.3 Test de sous-typage

La programmation par objets introduit le besoin d'un mécanisme original, le test de sous-typage. La question est de savoir si un objet x est instance d'une classe D. En typage statique, le test de sous-typage peut être formulé de la manière suivante : « Étant donné un objet x statiquement typé par C, x est-il une instance d'un type D. » Bien sûr D n'est pas supposé être un super-type de C, sinon la réponse serait trivialement « oui ». Cela se traduit de façon équivalente par la question : « La classe de x est-elle une sous-classe de D ? »

En général, ce test n'est pas utilisé directement par le programmeur mais au travers d'autres constructions — comme un *cast*, un appel de méthode covariant, un `typecase`, une tentative d'affectation, une clause `catch` de gestion d'exception, etc.

2.3.1 Changement de type statique

Le type statique d'une référence représente un point de vue sur cet objet. La *coercition* (ou *casting*) permet de changer le type statique d'une référence et donc de changer de point de vue sur l'objet. Même si les langages objets mettent en avant le mécanisme de *casting*, puisqu'il est syntaxiquement visible, dans les langages objet la plus grande partie du travail réside en fait dans le test de sous-typage lui même.

Remarque. De manière générale, le terme *cast* (ou *casting*) est emprunté à la coercition de type en C. Il est utilisé pour plusieurs opérations allant de la réinterprétation de zones mémoire aux conversions de types.

Dans cette thèse nous ne considérerons que les *casts* ayant un rapport avec les objets, c'est-à-dire l'ajustement du type statique d'une référence.

Il faut distinguer principalement deux cas de *casting* : le *cast ascendant* et le *cast descendant*.

Upcast

Si le type C est un sous-type d'un type cible D : on parle de *cast ascendant*, ou de *cast implicite*, puisqu'il ne nécessite aucune construction syntaxique particulière. Il peut être réalisé au travers d'un appel de méthode ou d'une affectation polymorphe du type x `:=` y

où x et y ont respectivement pour type X et Y tels que X soit un sous-type de Y (c'est-à-dire $Y \prec X$). Dans un cadre de typage statique, la validité de ce *cast* peut être vérifiée lors de la compilation et ne nécessite donc aucun traitement spécifique lors de l'exécution.

Remarque. L'explicitation de ce *cast* implicite est généralement utilisée pour lever l'ambiguïté en cas de conflits de méthode statiquement surchargée dans les langages disposant de *surcharge statique* (voir Section 2.2.6).

Downcast

Si C n'est pas un sous-type de D : on parle alors de *cast descendant* si D est un sous-type de C ($D \prec C$). Ce type de *cast* est utilisé en général en vue d'appeler une méthode non connue par le type source. Cette opération n'étant pas sûre, x pouvant ne pas être une instance de D, elle est réalisée par une construction syntaxique particulière. Par exemple la notation parenthésée (à la C) en JAVA et C# ; `dynamic_cast` en C++ ; `typecase` en THETA ; la tentative d'affectation en EIFFEL et PRM.

Remarque. Dans le cas où il n'y a aucune relation entre C et D, le terme *sidecast* ou *cross-cast* est utilisé. C'est une généralisation du *cast descendant* mais il se traite de manière identique — sauf dans l'implémentation de C++ (voir Section 5.1).

2.3.2 Prédicat *instanceof* et *typecase*

Il existe dans quelques langages à objets une construction qui rend explicite au programmeur le test de sous-typage. Il s'agit du prédicat qui retourne vrai si un objet est instance d'une classe — `instanceof` de JAVA ou `isa` de PRM. Conceptuellement ce prédicat ne présente aucune différence avec un test de sous-typage implicite ou un *cast*. En pratique, il s'utilise pour protéger l'entrée à une branche dans les énumérations de types. Dans ce cas, la branche commence souvent par changer le type statique de la variable qui a été testée, ce qui implique un second test de sous-typage, bien qu'il soit inutile.

Cette construction n'est, par conséquent, pas très adaptée et devrait être remplacée par une construction *ad hoc*. `typecase` semble être justement une construction plus adaptée. Un `typecase` prend en paramètre une variable — ou une expression qu'on peut associer à une variable — et en fonction de son type appelle la branche correspondante. Dans une branche, le type statique de la variable est remplacé par celui qui vient d'être testé. Cette construction proposée par le langage THETA [Liskov *et al.*, 1995] ne semble cependant plus être d'actualité et plus aucun langage récent ne la propose sous sa forme initiale. Les langages proposant des exceptions la proposent tout de même au travers de la construction `try/catch`. La succession de blocs formés par les *catchs*, ne sont en fait que les branches d'un `typecase` gardés par le type de l'exception. La sélection de la « bonne » branche est donc réalisée par une succession de tests de sous-typage.

Une alternative au `typecase` est possible en utilisant judicieusement l'appel de méthode. Elle est connue sous le nom de *double dispatch* et repose sur un double appel de méthode. La variable du `typecase` devient le receveur du premier appel qui, en fonction de son type dynamique, appelle une méthode différente sur son paramètre. Ce paramètre est le receveur du bloc qui aurait dû contenir le `typecase`. Ce principe est utilisé par le patron de conception *Visiteur* [Gamma *et al.*, 1994]

2.4 Mécanismes objet à implémenter

La programmation par objets repose sur deux mécanismes originaux qui nécessitent une implémentation, c'est-à-dire des structures de données et des algorithmes, non triviale. L'implémentation de l'invocation de méthode consiste à trouver la bonne méthode qui sera appelée par ce mécanisme. Le test de sous-typage sur un objet, dans le cadre des langages à classes, se ramène à implémenter le support à un prédicat entre deux classes qui, étant donné la relation de spécialisation, répond « vrai » si elles sont en relation. C'est donc une forme de codage de la relation d'ordre qu'est la spécialisation.

À ces deux mécanismes originaux se rajoute un troisième qui peut paraître trivial : l'*accès aux attributs*. Les attributs [9] des objets représentent les données. Une approche naïve des objets les identifierait à de simples structures, les attributs seraient dans ce cas les champs de la structure. Toutefois cette approche ne tient pas véritablement compte de la spécificité des objets, puisqu'un objet est le seul à connaître ses attributs. Comme pour les méthodes, dans les langages à classes, c'est la classe d'un objet qui détermine ses attributs et plus particulièrement leurs positions. L'implémentation de l'accès aux attributs est triviale en typage statique tant que l'héritage est simple (voir Section 4.4), ce n'est plus le cas lorsque l'héritage est multiple (voir Section 4.8.2) ou le typage dynamique .

2.5 Conclusion

La problématique des langages objet est relativement simple en soi, ce sont le typage statique et l'héritage multiple qui complexifient la sémantique et influencent l'implémentation. Les questions de types posent des problèmes intéressants de spécification de langages mais, dans le cadre de cette thèse, nous nous intéressons principalement aux questions d'implémentation. Cependant, le travail commencé par Jean Privat [2006] dans le cadre de sa thèse et poursuivi par moi-même, nous a conduit à spécifier un nouveau langage (voir Section 3.4). Ces contributions ont été motivées par le besoin d'un langage purement objet, simple et expressif afin de pouvoir nous concentrer uniquement sur les mécanismes objet. Les langages actuels sont souvent trop complexes et présentent de nom-

9. Les termes *variable d'instance, membre* ou *champ* sont aussi utilisés.

breuses caractéristiques non objet, caractéristiques souvent motivées par les besoins de l'industrie.

L'implémentation et la compilation des programmes objets présentent un point dur qui repose sur son principe même : c'est l'objet (ou sa classe) qui décide de son comportement. Cela se traduit par trois mécanismes — invocation de méthode, test de sous-typage et accès aux attributs — dont l'implémentation est non triviale et nécessite des structures adéquates. La totalité de cette thèse est dédiée à cette question.

CHAPITRE

3 Approche par méta-modélisation

Ce chapitre présente un élément essentiel de notre approche, le concept de méta-modélisation. Nous introduirons tout d'abord, ce concept et plus particulièrement son utilisation pour décrire la sémantique des langages objets. Nous présenterons ensuite un méta-modèle des classes et des propriétés (inspiré par [Ducournau et al.*, 1995; Ducournau et Privat, 2011]) qui analyse les relations entre ces entités dans le contexte de l'héritage multiple. Grâce à lui, nous décrirons et commenterons les conflits qui peuvent résulter de l'utilisation de l'héritage multiple. Enfin, nous introduirons les notions de modules et de raffinement de classes — un des mécanismes fondamentaux du langage* PRM*. Nous inspecterons ce mécanisme à la lumière d'un méta-modèle isomorphe à celui présenté pour l'héritage multiple. Outre leur rôle sémantique, ces méta-modèles sont pour nous essentiels car ils servent de sémantique au langage* PRM*, présenté dans la section 3.4, et de structure au compilateur (voir Chapitre 7).*

Sommaire

3.1 Introduction à la méta-modélisation

Dans le cadre des langages objets, la méta-modélisation est un outil puissant qui confirme l'intuition et qui peut servir de sémantique opérationnelle si on y ajoute un protocole — appelé Meta-Object Protocol (MOP).

La méta-modélisation a depuis longtemps servi la cause des objets. Son utilisation pour décrire les relations entre objets, remonte à ses débuts, avec la formalisation des concepts utilisés par SMALLTALK [Goldberg et Robson, 1983]. OBJVLISP est un très bel exemple de méta-modèle réflexif, il propose une analyse de la relation d'instanciation entre les objets et les classes [Cointe, 1987]. Ce modèle est pris comme exemple car il est simple, se concentre uniquement sur les objets et les classes, et fait partie du modèle objet réflexif de CLOS [Keene, 1989]. Enfin, il est à la base de nombreux travaux sur la réflection [Forman et Danforth, 1999]. Actuellement, la méta-modélisation est devenue une branche essentielle et à part entière du génie logiciel, avec ses modèles (UML), ses méta-modèles (ECORE, EMOF, etc) et ses procédés (ingénierie dirigée par les modèles, etc).

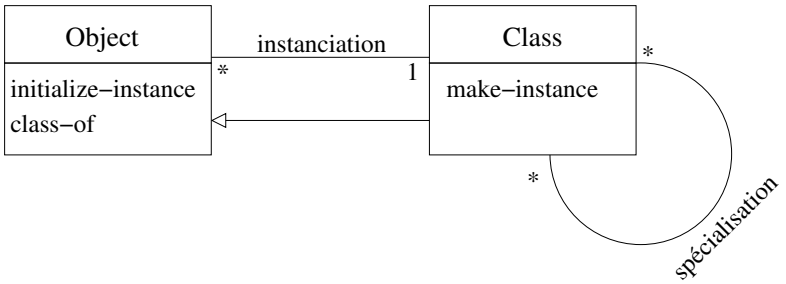


FIGURE 3.1 : Le modèle réflexif OBJVLISP, noyau de CLOS

La méta-modélisation d'un langage permet de définir ses méta-entités et son protocole méta-objet (MOP). La méta-programmation permet, à travers le MOP du langage, de modifier le comportement des objets à l'exécution. Les seuls langages à avoir exposé le MOP dans le langages sont des langages à typage dynamique (CLOS, PYTHON, RUBY) [Kiczales *et al.*, 1991]. Des tentatives d'ajout d'une couche méta ont été faites pour des langages à typage statique, il ne s'agissait au début que de transformation source-à-source [Chiba, 1995], jusqu'à devenir la programmation par aspect [Chiba, 1998; Kiczales *et al.*, 2001]. L'API réflexive (`java.reflect` pour JAVA et `System.Reflection` pour C#) représente, tout de même, une forme limitée de méta.

Nous utiliserons, dans le reste de ce chapitre, la méta-modélisation pour décrire la relation entre les classes et les propriétés en présence d'héritage multiple (voir Section 3.2), puis pour décrire un système de modules et son mécanisme de raffinement de classes (voir Section 3.3).

3.2 Méta-modèle pour l'héritage multiple

L'héritage multiple se rencontre dès lors qu'une classe spécialise directement plusieurs classes à la fois — donc qu'elle a plus d'une super-classe directe.

L'héritage multiple est souvent considéré par les programmeurs et les spécificateurs de langage comme trop compliqué, ou du moins n'apportant qu'une faible valeur ajoutée au prix d'une complexification abusive. En réalité le principal défaut de l'héritage multiple, en mettant de côté les problèmes d'implémentation, provient du défaut de compréhension globale de la problématique par les spécificateurs et concepteurs de langages. Des aberrations comme l'héritage répété et l'héritage d'implémentations n'ont fait qu'exacerber les problèmes, particulièrement dans les langages où la spécification est dictée par l'implémentation (pour n'en citer qu'un : C++). À cela, si l'on rajoute les problèmes d'implémentation, la plupart des nouveaux langages n'ont pas considéré le plein héritage multiple mais ont fait le choix d'utiliser une forme dégradée de celui-ci : le sous-typage multiple. En sous-typage multiple, une classe peut spécialiser plusieurs types — les interfaces dans la terminologie JAVA ou C#— mais ne peut spécialiser qu'une seule classe. Le sous-typage multiple est censé simplifier les problèmes. Il ne fait en fait que les marginaliser, les interfaces étant généralement nettement moins utilisées que les classes.

Cependant il est vrai que l'héritage multiple n'est pas simple et qu'il peut poser des problèmes délicats de spécifications — notamment pour ce qui est des conflits (voir Section 3.2.4). Et à notre avis, les arguments donnés par les détracteurs de l'héritage multiple ne sont pas rédhibitoires. Nous présentons dans cette section un méta-modèle qui représente la sémantique de l'héritage multiple et qui prend le soin d'en décrire les parties complexes. Nous présenterons dans le chapitre 5 des techniques d'implémentation compatibles avec l'héritage multiple.

3.2.1 Méta-modèle des classes et des propriétés

L'objectif de ce méta-modèle est de représenter la sémantique de l'héritage multiple, c'est-à-dire de la spécialisation dans le cas où une classe hérite directement de plusieurs classes sans aucun rapport. Il permet ainsi de mettre en évidence les cas de conflits liés à l'héritage multiple. Le méta-modèle est constitué de trois entités principales : les classes, les propriétés globales et les propriétés locales (Figure 3.2).

Les *classes* ne présentent pas de problème de compréhension pour le lecteur, et ce sont les deux dernières entités qui sont la clé du méta-modèle. Les *propriétés locales* représentent les propriétés telles qu'elles sont définies par une classe indépendamment de toutes les autres définitions possibles de cette « *même propriété* ». Les *propriétés globales* représentent cette notion de « *même propriété* » au travers de la hiérarchie de classes. En d'autres termes, elles représentent le *message*[1] auquel peut répondre une classe (dans le

1. La métaphore de l'*envoi de message* utilisée par SMALLTALK [Goldberg et Robson, 1983] est parfaite-

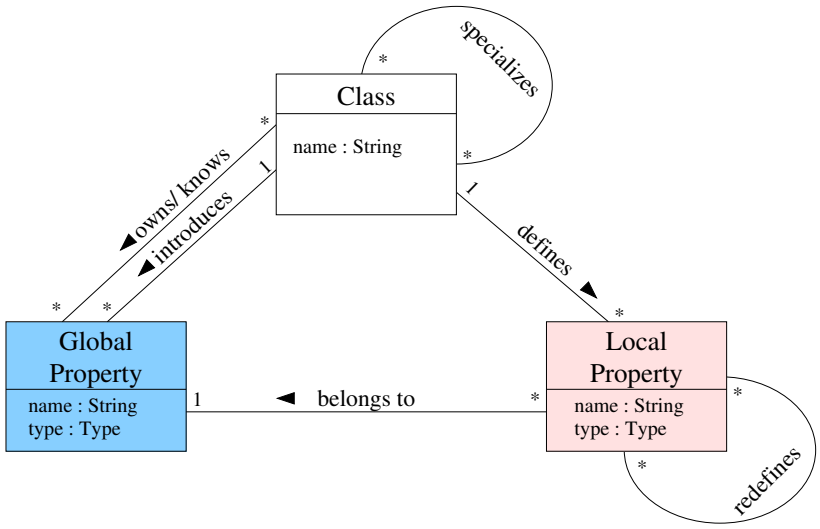


FIGURE 3.2 : Méta-modèle des classes et des propriétés, d'après [Ducournau et Privat, 2011]

cas des méthodes) ; la réponse étant l'invocation de la propriété locale associée au type dynamique du receveur. Une propriété locale est définie par une classe (relation *defines* de la figure 3.2) et elle appartient à une *propriété globale* (relation *belongs to*).

Remarque. Dans un méta-modèle plus concret, les propriétés locales seraient spécialisées en *méthodes*, *attributs*, *types virtuels* (Figure 7.4). À moins que la différence ne soit significative, dans la suite de nos propos nous utiliserons simplement le terme de *propriété* pour parler de toutes ces entités. Pour les exemples, nous prendrons le cas des méthodes.

Une définition de classe est un triplet ⟨`nom, super-classes, propriétés locales`⟩, où `super-classes` est un ensemble de classes déjà définies et `propriétés locales` l'ensemble des propriétés définies par cette classe. La modélisation du mécanisme d'héritage doit se faire en deux temps, d'abord l'héritage des propriétés globales puis celui des propriétés locales.

Héritage de propriétés globales : une classe hérite de toutes les propriétés globales de ses super classes — la sous-classe connaît toutes les propriétés connues par ses parents. Puis, pour chaque propriété locale définie par la classe, s'il existe une propriété globale du même nom, on l'associe à la propriété locale. Sinon, on crée une nouvelle propriété globale du même nom que celui de la propriété locale.

Héritage de propriétés locales : pour chaque propriété globale connue par la classe et non associée à une propriété locale, la classe hérite de la propriété locale la plus spécifique de

ment adaptée à ce méta-modèle. De nos jours, on utiliserait plutôt l'*invocation de méthode* de la terminologie JAVA.

ses super-classes (voir Section 3.2.4). Le typage statique garantit que cette propriété existe toujours, mais son unicité n'est pas garantie (voir Section 3.2.3). Cette phase n'est nécessaire que lors de l'exécution, toutefois la plupart des implémentations la pré-compilent. C'est cette pré-compilation qui est l'objet principal de cette thèse.

3.2.2 Exemple

Remarque. Dans cette thèse tous les exemples de code objet seront donnés dans la syntaxe du langage PRM (voir Section 3.4), qui est simple et intuitive.

Les exemples de code de bas-niveau, c'est-à-dire relatifs à l'implémentation, seront présentés en pseudo assembleur (principalement inspiré de celui de Driesen [2001] (voir Section 4.2.1).

Enfin, les algorithmes seront présentés en LISP [Steele, 1990] ou sous forme de pseudo-code.

Pour fixer les idées, nous présentons un exemple de l'instanciation de ce méta-modèle pour un programme objet très simple (Figure 3.3). L'exemple est constitué de deux classes : `A`, et `B` qui spécialise `A`. La classe `A` *introduit* la propriété `foo` et *définit* son contenu. Sa sous-classe `B` *redéfinit* la méthode `foo` et introduit une nouvelle propriété `bar`. Ce court exemple nécessite, tout de même, sept entités. La figure 3.3 montre le code associé à cet exemple (en bas) et les objets instanciés du méta-modèle (en haut).

Remarque. Le nom de ces classes n'est pas valide en PRM. En effet, un nom de classe valide doit au moins contenir une minuscule.

3.2.3 Formalisation

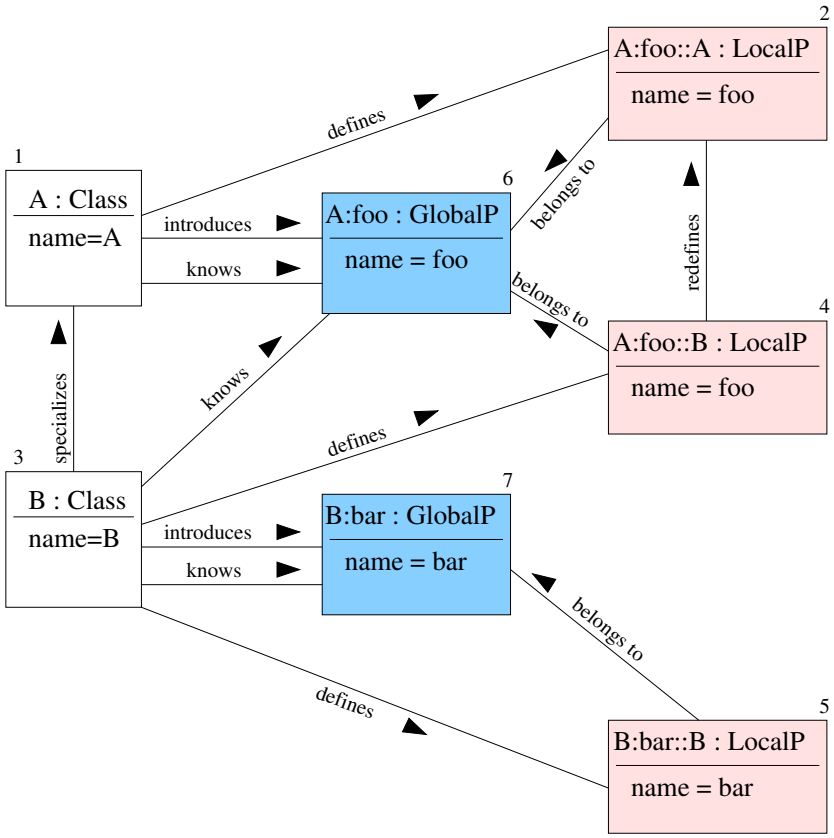
Nous rappelons la formalisation proposée dans [Ducournau et Privat, 2011] en la restreignant aux éléments directement nécessaires. Les instances de ce méta-modèle sont des *hiérarchies de classes*.

Définition 3.1 (Hiérarchie de classes) *Une hiérarchie de classes est un n-uplet :*

$$\mathcal{H} = \langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, \mathrm{nom}_{\mathcal{H}}, \mathrm{glob}_{\mathcal{H}}, \mathrm{intro}_{\mathcal{H}}, \mathrm{def}_{\mathcal{H}} \rangle,$$

où :

- $X^{\mathcal{H}}$ *est un ensemble de classes.*
- $\prec^{\mathcal{H}}$ *est une relation de spécialisation de classe — qui est transitive, antisymétrique, et antiréflexive. On note* $\preceq^{\mathcal{H}}$ *(resp.* $\prec_d^{\mathcal{H}}$*) la fermeture réflexive (resp. la réduction transitive) de* $\prec^{\mathcal{H}}$*. De plus,* $\langle X^{\mathcal{H}}, \preceq^{\mathcal{H}} \rangle$ *est un ordre partiel.*
- $G^{\mathcal{H}}$ *et* $L^{\mathcal{H}}$ *: deux ensembles disjoints des propriétés globales et locales.*
- $N^{\mathcal{H}}$ *: l'ensemble des identifiants (noms) des propriétés.*




```
class A
    def foo
    do ... end # Do something
end

class B
inherit A
    def foo
    do ... end # Do something else

    def bar
    do ... end # Do another thing
end
```

FIGURE 3.3 : Exemple d'instanciation du méta-modèle, d'après [Ducournau et Privat, 2011]

- $\text{nom}_{\mathcal{H}}$ *: une fonction de nommage des propriétés de* $xGH \uplus L^{\mathcal{H}} \to N^{\mathcal{H}}$ *; sa restriction à* $X^{\mathcal{H}}$ *est injective.*
- $\text{glob}_{\mathcal{H}}$ *: une fonction de* $L^{\mathcal{H}} \to G^{\mathcal{H}}$ *qui associe une propriété locale à sa propriété globale.*
- $\text{intro}_{\mathcal{H}}$ *: une fonction de* $G^{\mathcal{H}} \to X^{\mathcal{H}}$ *qui associe une propriété globale à sa classe d'introduction.*
- $\text{def}_{\mathcal{H}}$ *: une fonction de* $L^{\mathcal{H}} \to X^{\mathcal{H}}$ *qui associe une propriété locale à sa classe de définition.*

Pour alléger les notations, nous ne noterons l'exposant $\mathcal{H}$ que s'il y a ambiguïté, c'est à dire quand nous considérerons plusieurs hiérarchies de classes en même temps.

Ces définitions sont complétées par les équations et définitions suivantes :

Propriétés globales

Soit une classe c, $c \in X$. G_c représente l'ensemble des propriétés globales de c. Chaque propriété est, soit *héritée* d'une super-classe de c, soit *introduite* par c. Soit les $G_{\uparrow c}$ et G_{+c} les sous-ensembles correspondants. Tous les G_{+c} sont disjoints et

$$G_{+c} \stackrel{\text{def}}{=} \text{intro}^{-1}(c) \tag{3.1}$$

$$G_{\uparrow c} \stackrel{\text{def}}{=} \bigcup_{c \prec_d c'} G_{c'} \stackrel{\text{def}}{=} \biguplus_{c \prec c'} G_{+c'} \tag{3.2}$$

$$G_c \stackrel{\text{def}}{=} G_{\uparrow c} \uplus G_{+c} = \biguplus_{c \preceq c'} G_{+c'} \tag{3.3}$$

$$G = \bigcup_{c \in C} G_c = \biguplus_{c \in C} G_{+c} \tag{3.4}$$

Il y a conflit de propriétés globales quand il existe deux propriétés homonymes distinctes dans une classe. Les conflits, et donc ceux de propriétés globales, seront traités plus en détail dans la section 3.2.4.

Propriétés locales

Soit une classe $c \in X$, L_c représente l'ensemble des propriétés locales définies par c. Et la fonction réciproque def : $L \to C$ qui associe à chaque propriété locale sa classe de définition.

$$L = \biguplus_{c \in C} \quad \text{avec} \quad L_c \stackrel{\text{def}}{=} \text{def}^{-1}(c). \tag{3.5}$$

La définition des propriétés locales est contrainte pour éviter les ambiguïtés. Chaque classe ne peut définir qu'une seule propriété locale par propriété globale. Donc pour toutes les propriétés globales $g \in G$, la restriction de def à $\text{glob}^{-1}(g)$ est injective. De plus, chaque propriété locale est associée à une propriété globale de même nom.

Remarque. Si une propriété est considérée comme abstraite dans sa classe d'introduction — c'est-à-dire qu'elle n'a pas d'implémentation — une propriété locale doit quand même lui être associée.

A toute propriété locale correspond une propriété globale, soit héritée d'une de ces super-classes, soit introduite par la classe courante. Soit $L_{\uparrow c}$ et L_{+c} les deux sous-ensembles disjoints correspondants.

$$L_c = L_{\uparrow c} \uplus L_{+c} \quad \text{avec} \quad \left\{ \begin{array}{lcl} L_{\uparrow c} & \stackrel{\text{def}}{=} & L_c \cap \text{glob}^{-1}(G_{\uparrow c}) \\ L_{+c} & \stackrel{\text{def}}{=} & L_c \cap \text{glob}^{-1}(G_{+c}) \end{array} \right. \tag{3.6}$$

Et donc la *redéfinition de propriété* peut être exprimée par la définition ci-dessous.

Définition 3.2 (Redéfinition de propriété) *La redéfinition de propriété (*property overriding*) est définie comme la relation $\ll^{\mathcal{H}}$ (abrégée $\ll$) entre une propriété locale de $L_{\uparrow c}$ et la propriété locale correspondante dans une des super-classes de c :*

$$l \ll l' \stackrel{\text{def}}{\Longleftrightarrow} \text{glob}(l) = \text{glob}(l') \wedge \text{def}(l) \prec \text{def}(l').$$

Cette relation définit un ordre partiel (et nous noterons $\ll_d$ sa réduction transitive).

Instanciation du méta-modèle

L'instanciation du méta-modèle est tâche importante réalisée par le compilateur, c'est grâce à cela qu'il peut donner une sémantique aux différents éléments qu'il rencontre dans le code source.

La construction du modèle – donc l'instanciation du méta-modèle — est faite par définition de classes successives. Pour assurer la validité du modèle, chaque définition de classe est tout d'abord vérifiée, au besoin ses super-classes sont récursivement traitées au préalable. Puis la mise à jour du modèle est faite en respectant le protocole suivant :

1. *Mise à jour de la hiérarchie de classes.* La nouvelle classe c est ajoutée à l'ensemble des classes X. Le nom de la classe ne doit pas déjà exister dans l'ensemble des noms de X. Pour chaque super-classe d de c, la paire $\langle c, d \rangle$ est ajoutée à $\prec_d$. Si l'ensemble des super-classes rajoute un arc de transitivité, cet arc est tout simplement ignoré. Cette vision de la relation de spécialisation est cependant contraire à certaines spécifications de langage, plus particulièrement à celles qui autorisent l'héritage répété.
2. *Héritage de propriétés globales.* L'ensemble $G_{\uparrow c}$ est ensuite calculé (3.2) et la présence de conflit de propriétés globales est vérifiée (voir Section 3.2.4).
3. *Définitions locales.* Pour chaque définition locale l de la classe c, une nouvelle propriété locale est créée avec son nom. Pour chaque nom appartenant à $G_{\uparrow c}$, la propriété locale est associée à la propriété globale correspondante. Pour les autres, une nouvelle propriété globale est créée ce qui permet de constituer l'ensemble G_{+c} (3.1).

Les difficultés liées aux ambiguïtés de nom (conflit de propriétés globales) sont discutées en section 3.2.4.

4. *Héritage de propriétés locales.* Enfin, une propriété locale est *héritée* pour chaque propriété globale de c sans définition locale et les conflits éventuels qui y sont liés sont traités (voir Section 3.2.4). Le détail de cet héritage est traité juste après.

Héritage de propriétés locales

L'héritage de propriétés locales consiste à trouver la *bonne* propriété locale à associer à une propriété globale lorsqu'elle n'a pas de définition locale dans une classe.

Remarque. Cet héritage de propriétés locales est théoriquement possible avec tous les types de propriétés locales. Certains langages, comme C++ ou JAVA, ne permettent cependant de redéfinir que les méthodes. D'autres langages comme CLOS, YAFOOL [Ducournau, 1991] ou EIFFEL permettent également de redéfinir les attributs. Enfin, des langages comme PRM (voir Section 3.4) ou GBETA [Ernst, 1999] qui ajoutent aux méthodes et aux attributs des *types virtuels* (voir Section 2.2.5), permettent aussi de redéfinir ces derniers.

Pour des raisons de clarté, nous ne traiterons dans cette section que du cas des méthodes car c'est celui qui présente le plus de spécificité.

L'*invocation de méthode* implique deux mécanismes distincts : la liaison tardive (ou envoi de message) et l'appel à `super` [2].

Liaison tardive. Soit $c \in X$ une classe et une propriété globale $g \in G$, la liaison tardive de g sur c implique le choix d'une des propriétés locales associées à g définies par une des super-classes de c (c comprise). Soit $\mathrm{loc}(g,c)$ l'ensemble correspondant :

$$\mathrm{loc}(g,c) \stackrel{\text{def}}{=} \{l \in \mathrm{glob}^{-1}(g) \mid c \preceq \mathrm{def}(l)\}. \tag{3.7}$$

La propriété locale appelée par la *liaison tardive* est la propriété la plus spécifique de cet ensemble. Si g a une définition dans C, cette définition est la plus spécifique. Sinon la classe doit hériter d'une des propriétés locales d'une de ses super-classes. C'est la deuxième phase de l'héritage appelée *héritage de propriétés locales.* Cette propriété héritée est définie par :

$$\mathrm{spec}(g,c) \stackrel{\text{def}}{=} \min_{\ll} loc(g,c) \tag{3.8}$$

En héritage simple, $\mathrm{spec}(g,c)$ est un singleton, ce qui n'est pas toujours le cas en héritage multiple et conduit à un deuxième type de conflit, le *conflit de propriétés locales.* Ce conflit sera examiné à la section 3.2.4.

2. Nous utiliserons le terme `super` du fait de sa popularité — il est utilisé en SMALLTALK, JAVA, C#— bien que nous le considérions ici comme le mécanisme `Precursor` d'EIFFEL. En JAVA, l'utilisation du `super` n'impose pas que l'on parle de même propriété globale, ce qui est le cas de `Precursor`.

Appel à *super*. L'appel à `super` permet à une propriété locale l d'appeler une des propriétés locales (par exemple l') qu'elle redéfinit, c'est-à-dire telle que $l \ll l'$. On peut donc définir l'appel à super comme une sélection dans l'ensemble

$$\text{supl}(l) \stackrel{\text{def}}{=} \{l' \mid l \ll l'\}. \tag{3.9}$$

Comme pour la liaison tardive, la propriété sélectionnée est la plus spécifique de l'ensemble $\text{supl}(l)$, c'est-à-dire $\min_{\ll}(\text{supl}(l))$. Et comme précédemment, l'unicité n'est garantie qu'en héritage simple.

3.2.4 Conflits

Les conflits sont la principale difficulté rencontrée avec l'héritage multiple. Ils sont généralement exprimés en terme *d'ambiguïté* mais grâce au méta-modèle une analyse plus fine peut être faite. Les conflits sont décomposés en deux catégories, ceux relatifs à l'introduction et la désignation des propriétés, appelés conflits de propriétés globales et ceux relatifs à la redéfinition de propriétés, appelés conflits de propriétés locales.

Conflit de propriété globale

Le conflit de propriétés globales (anciennement *conflit de noms* dans [Ducournau *et al.*, 1995]) est essentiellement un problème de désignation de la « bonne » propriété. Il pourrait être évité par une convention de nommage (voir ci-dessous) ou corrigé par le renommage des propriétés conflictuelles à travers tout le programme — bien entendu il faudrait disposer de l'intégralité du code source, bibliothèques comprises.

Définition 3.3 (Conflit de propriétés globales) *Une classe C présente un* conflit de propriétés globales *quand C connaît deux propriétés globales distinctes homonymes, $g1, g2 \in G_{\uparrow c}$ introduites par des classes différentes, donc telles que*

$$\text{nom}(g1) = \text{nom}(g2) \wedge \text{intro}(g1) \neq \text{intro}(g2).$$

Lors de l'introduction d'une propriété globale g par une classe, le nom doit être non ambigu dans le contexte de cette classe — dans l'ensemble des propriétés *connues* par cette classe. Cette remarque implique qu'il ne peut y avoir de conflit de propriétés globales que dans le cas de l'héritage multiple. De plus, en cas de conflit, il existe toujours une classe qui présente un *conflit primitif*, c'est-à-dire telle que $g1$ et $g2$ soient héritées de deux superclasses directes distinctes.

Le conflit de propriétés globales peut être résolu de diverses manières :

Ne rien faire. Le langage ne propose aucune solution à ce problème mais signale une ambiguïté (erreur). Lever l'ambiguïté est laissé à la charge du programmeur qui doit renommer à travers tout le code source une des deux méthodes, ce qui peut être cause d'erreur. De plus, cette alternative n'est pas toujours possible, les bibliothèques pouvant avoir été achetées à divers tiers et leur code source n'est alors pas forcément disponible.

Qualification explicite. Cette solution consiste à proposer une syntaxe alternative permettant de nommer la propriété souhaitée, par exemple en la préfixant par la classe d'introduction de la propriété globale (`x.A::foo()`) — ou toute autre classe dans laquelle le nom est non ambigu. Le compilateur ne détecte une ambiguïté que quand le programmeur veut utiliser le nom ambigu dans un contexte où il est effectivement ambigu. La qualification explicite est donc une solution modulaire (hypothèse du monde ouvert) et paresseuse à ce problème, bien que légèrement verbeuse. Pour alléger la syntaxe, il est possible de définir une propriété à appeler par défaut — les autres devant obligatoirement être qualifiées — pour une unité ou un bloc de code. Cette solution est utilisée par PRM pour résoudre les conflits de propriétés globales.

Remarque. Cette qualification explicite peut être remplacée par un *cast* statique, donc sans ajout de nouvelle syntaxe, mais cela ne s'applique qu'aux invocations et non aux définitions — c'est la solution adoptée par C# (`((A)x).foo()`).

Renommage local. Le renommage local consiste à changer le nom d'une propriété, globale et locale, dans une classe et toutes ses sous-classes. Lors de la définition d'une classe possédant un conflit de propriétés globales, les propriétés conflictuelles doivent être héritées en changeant de nom — éventuellement pour le même nom. C'est la solution utilisée par EIFFEL [Meyer, 1994]. Le renommage local présente certains inconvénients, il est systématique et donc obligatoire même lorsque le programmeur n'utilise pas de nom ambigu. Une même propriété conflictuelle peut être héritée par différentes classes sous des noms différents. Enfin, la désignation d'une propriété dans le discours — ou dans la documentation — est bien plus complexe.

Unification. L'unification consiste à considérer deux propriétés homonymes comme identiques. C'est la solution généralement utilisée par les langages à typage dynamique (CLOS, etc.), qui n'ont pas d'autre moyen de distinguer les propriétés, et étrangement par C++— uniquement dans certains cas pour les méthodes (voir la remarque ci-dessous). JAVA a le même comportement en cas de conflit de propriétés globales introduites par deux interfaces distinctes. Dans un sens l'unification consiste à ne pas détecter le conflit de propriétés globales et à le transformer en un conflit de propriétés locales. Si le programmeur considère réellement que ces propriétés sont différentes il doit, comme dans le cas où le langage ne propose rien, renommer globalement les propriétés. En revanche, contraire-

ment au premier cas, le programmeur peut ne pas se rendre compte qu'il y a une erreur — puisqu'il n'est pas averti par le compilateur.

Remarque. Pour les méthodes, C++ a un comportement nettement plus alambiqué. Dans une classe où il a un conflit de propriétés globales, les propriétés conflictuelles sont unifiées si la classe redéfinit une propriété locale — elle sera valable pour les deux qui sont unifiées dans toutes les sous-classes. Sinon le compilateur ne produit une erreur qu'en cas d'appel d'une des propriétés conflictuelles dans un contexte ambigu. Le programmeur peut donc lever l'ambiguïté en changeant le type statique du receveur par un type où l'appel est non ambigu. Ceci autorise une certaine forme de renommage local pour les sous-classes, mais contrairement à EIFFEL, l'appel à une de ces propriétés renommées sur un type statique où il n'existait pas de conflit appelle la propriété globale associée à ce type statique et non celle qui résulte du renommage.

Convention de nom. Le conflit de propriétés globales peut être systématiquement évité en préfixant toutes les introductions de propriétés par le nom de la classe qui l'a introduit. En l'absence de type statique, cette alternative est souvent utilisée par les programmeurs pour éviter les conflits et l'unification de propriétés involontaires. Utiliser ce type de convention lorsque le langage et les outils qui l'accompagnent permettent de faire autrement est inutile voire nuisible. Car ceci entraîne une certaine lourdeur syntaxique pour tous les appels même non ambigus.

Pour conclure, le conflit de propriétés globales est un problème mineur qui peut facilement être résolu, dans tous les langages à typage statique, au prix de légères modifications dans le modèle et dans la syntaxe. La qualification explicite représente à notre avis la solution la plus simple et la plus élégante à ce problème. Elle ne rajoute que très peu de syntaxe — la qualification d'une propriété par le nom d'une classe, syntaxe déjà présente pour la qualification des *packages* en JAVA par exemple — et ne change quasiment pas le méta-modèle — seule la fonction nom doit être modifiée pour tenir compte du contexte. Du point de vue du programmeur, la documentation des méthodes reste claire et la lourdeur syntaxique rajoutée par les conflits est limitée aux cas ambigus dans un contexte ambigu.

Conflit de propriétés locales

Le conflit de propriétés locales (anciennement *conflit de valeurs* dans [Ducournau *et al.*, 1995]) est plus subtil et ne se rencontre que dans des cas d'héritage en losange lorsqu'une même propriété globale est héritée de deux classes différentes et qu'aucune des propriétés locales associées n'est plus spécifique — par rapport à la relation $\ll$.

Définition 3.4 (Conflit de propriétés locales) *Une classe C présente un conflit de propriétés locales pour une propriété globale g, si l'ensemble des propriétés locales les plus spécifiques de g dans C n'est pas réduit à un singleton, c'est-à-dire si $|spec(g,c)| > 1$.*

Définition 3.5 (Ensemble de conflit) *L'ensemble de conflit est défini comme l'ensemble de toutes les super-classes de c qui définissent une propriété locale et qui sont minimales par rapport à* $\preceq$.

$$cs(g,c) \stackrel{\text{def}}{=} \text{def}^{-1}(spec(g,c)) = \min_{\preceq} \text{def}^{-1}(loc(g,c)).$$

Remarque. Lorsque l'ensemble $spec(g,c)$ contient au moins une propriété locale concrète, toutes les propriétés locales abstraites peuvent être retirées de l'ensemble.

Le conflit de propriétés locales est un problème de sémantique qui n'a par conséquent pas de solution intrinsèque. Pour lever l'ambiguïté, le programmeur doit pouvoir rajouter du sens à son code et le langage doit lui en donner les moyens.

Les conflits de propriétés locales peuvent se résoudre globalement de trois manières différentes :

Ne rien faire. Le langage ne propose aucune solution mais signale l'erreur à la compilation. Le programmeur est forcé de redéfinir la méthode dans la classe où le conflit a eu lieu. Généralement, cette nouvelle propriété locale va réaliser un appel à `super` mais cet appel sera lui aussi ambigu (voir Section 3.2.4).

Sélection. Le programmeur a la possibilité de choisir de quelle méthode il hérite. Dans les langages à typage dynamique, ce choix est souvent fait par le langage lui même en utilisant une *linéarisation*. En EIFFEL, le choix est fait par le programmeur en utilisant le mot-clef `undefine`.

Combinaison. Intuitivement l'idée est de combiner les propriétés conflictuelles mais cette combinaison est dépendante du type de propriétés en conflit. Pour les méthodes, cette combinaison consiste à redéfinir la propriété conflictuelle par un enchaînement d'appels à `super` ou une linéarisation — pour éviter la double évaluation dans les losanges. Pour les contacts d'EIFFEL, la combinaison implique la disjonction des pré-conditions et la conjonction des post-conditions. En JAVA, lorsque le conflit porte sur les exceptions déclarées, la combinaison de méthodes implique une combinaison des exceptions — ce sont des post-conditions. L'invocation de la méthode sur un receveur typé par la dite classe ne peut signaler qu'une exception appartenant à l'intersection des exceptions — éventuellement vide. En présence de types virtuels, tous les conflits de signatures sont exprimés à travers un conflit de propriétés locales sur le type virtuel. Il devra être résolu spécifiquement (voir ci-dessous). Grâce aux types virtuels, les signatures de propriétés ne sont jamais conflictuelles et les attributs sont exempts de conflit de propriétés locales. En l'absence de types virtuels et en présence de covariance (voir Section 2.2.3) — éventuellement limitée au type de retour —, les problèmes de typages s'ajoutent aux problèmes sémantiques. Ils doivent être résolus comme le conflit de type virtuel. Le lecteur intéressé est renvoyé à [Ducournau et Privat, 2011] pour une explication détaillée.

Conflit de type virtuel. En présence de types virtuels, tous les problèmes liées aux types dans les définitions de classes, s'expriment au travers d'un conflit de type virtuel. S'il existe un conflit de cet ordre, sa résolution peut se faire automatiquement par *combinaison* et le type résultant est l'intersection des types en conflits — le langage doit donc posséder des types intersection. Sinon, comme pour les méthodes, une redéfinition explicite du type virtuel par un type appartenant à l'intersection règle le conflit.

Appel à `super`. Avec la redéfinition explicite des propriétés locales, l'appel à `super` présente un conflit similaire à celui du conflit de propriété locale La résolution se fait sur les ensembles $\sup(l)$ au lieu de $\log(g, c)$ en en prenant le minimal s'il est unique. Si ce n'est pas le cas, le conflit peut se résoudre, comme le conflit de propriétés locales, par linéarisation ou par qualification du super. La qualification du `super` ressemble à la qualification explicite (voir Section 3.2.4), en substituant `super` au nom de la propriété — `Classe::super`. Exactement comme pour la qualification des méthodes, le nom utilisé pour qualifier ne sert q'à lever l'ambiguïté — donc comme information de direction — et non à faire un appel statique comme c'est le cas en C++. PRM utilise la redéfinition explicite et le `super` qualifié comme seul moyen de régler les conflits de propriétés locales

Linéarisations. Les linéarisations sont très répandues dans les langages à typage dynamique, depuis CLOS à PYTHON, et même cachées dans certains traits de langage à typage statique. SCALA ou GBETA utilisent des linéarisations pour la combinaison des *mixins* (voir Section 5.9.3) et C++ les utilise pour l'implémentation de ses constructeurs et destructeurs.

Le principe des linéarisations est de construire un ordre total à partir de l'ordre partiel de la spécialisation et, pour rester compatible avec les règles du méta-modèle présenté ci-dessus, cet ordre doit être une extension linéaire de l'ordre partiel (tri topologique). Malheureusement la construction d'un tel ordre n'est pas une tâche aisée et maintenir la monotonie de l'ordre en considérant l'hypothèse du monde ouvert peut s'avérer impossible.

Les linéarisations permettent de définir un opérateur du genre de `super` qualifié mais qui évite les évaluations multiples (dans les cas de losanges) — cet opérateur est nommé `call-next-method` en CLOS. L'implémentation de `call-next-method` dans un langage statiquement typé avec l'hypothèse du monde ouvert nécessite d'introduire une nouvelle propriété globale pour chaque propriété locale l'utilisant.

En conclusion, ce conflit est réellement l'une des difficultés majeure de l'héritage multiple et il n'existe pas de solution miracle pour le résoudre. En l'absence de linéarisation, le programmeur est généralement mis à contribution.

3.3 Modules et raffinement de classes

La modularité et la réutilisabilité font partie des concepts clé du génie logiciel. Dans les langages objets, les classes et le mécanisme d'héritage ont parfois été considérés comme la solution définitive aux problèmes de modularité. Depuis SIMULA [Birtwistle *et al.*, 1973], les classes jouent le rôle de modules, rôle amplifié par les *classes imbriquées* (*nested classes*) de JAVA.

Or cette vision des langages objets est souvent critiquée, et de nombreux auteurs affirment que les classes n'ont pas le rôle de modules. Les premières sont là pour décrire et créer des objets, tandis que les modules sont là pour structurer les programmes. Pour reprendre les mots de Szyperski [1992] : « Import is not inheritance. Why we need both : Modules and classes ». Ceci est connu sous le nom de problème de l'« expression » ou séparation des préoccupations transversales (*cross-cutting concerns*) et de nombreuses propositions ont été faites pour tenter d'y répondre [Bergel *et al.*, 2003; Clifton *et al.*, 2000; Ducasse *et al.*, 2005; Ernst, 2003; Ichisugi et Tanaka, 2002; Nystrom *et al.*, 2004; Tip et Sweeney, 2000].

Bien qu'il existe de nombreuses autres propositions, dans cette section nous en présentons une appelée *modules et raffinement de classes* [Privat, 2006; Ducournau *et al.*, 2007]. Ce choix est principalement motivé par le fait que les *modules et le raffinement de classes* sont une des fonctionnalités clé de PRM (voir Section 3.4). La seconde raison, et non la moindre, tient à ce qu'ils sont sont spécifiés par une forte analogie au méta-modèle de l'héritage multiple présenté en section 3.2.

3.3.1 Principe

Un *module* est une hiérarchie de classes, c'est-à-dire un ensemble de classes ordonnées par une relation de spécialisation. C'est une unité de réutilisation [Szyperski, 1992] qui peut être compilée séparément, puis liée à d'autres modules en vue d'en faire un programme exécutable (voir Chapitre 6). Donc les classes n'ont plus ce rôle d'unités de réutilisation mais seulement celui d'unités d'extension. Enfin, il n'y a plus de besoin pour les classes imbriquées [3], et elles ne seront de ce fait aucunement traitées dans cette thèse.

Un module *dépend* d'autres modules (appelés *super-modules*) et il *importe* les classes définies par ces super-modules qu'il peut aussi *raffiner*. Cette relation de dépendance est acyclique, comme la relation de spécialisation, et il n'y a pas d'imbrication de modules [4] (comme pour les classes).

On peut voir le raffinement de classe comme une définition incrémentale de classe (par exemple la classe C_A qui est raffinée par la classe C_B), dans lequel les propriétés de la

3. Les classes imbriquées (*nested classes*) sont principalement motivées par le besoin de modularité en l'absence de mécanisme plus adéquat [Nystrom *et al.*, 2004, 2006].

4. Nous utiliserons les termes *super-modules* et *sous-modules* (par analogie aux classes, super-classes et sous-classes), bien qu'il n'y ait pas d'imbrication, pour des raisons de commodité.

classe raffinée (C_B) s'ajoutent ou remplacent les propriétés de la classe précédente (C_A). Ceci ressemble clairement à la spécialisation, à cela près qu'avec le raffinement, la classe C_B se substitue complètement à la classe C_A, c'est-à-dire que toutes les instances de C_A du programme sont remplacées par des instances de C_B. En pratique, les classes C_A et C_B portent le même nom et une fois que l'opération de raffinement a eu lieu ce nom désigne les instances de C_B. Pour le moment, c'est la seule différence intuitive entre le raffinement et la spécialisation. Ce mécanisme, relativement nouveau pour les langages statiquement typés, se rencontre plus facilement dans le cadre des langages à typage dynamique, car ils ont la possibilité de recharger les définitions de classes [Ducournau, 1991] — du moins en ce qui concerne les méthodes [DeMichiel et Gabriel, 1987; Steele, 1990]. La hiérarchie (l'ordre) de raffinement est déduite de la hiérarchie de dépendance des modules — alors que pour les langages à typage dynamique cet ordre est basé sur le flot de contrôle.

Du point de vue des mécanismes, nous avons identifié quatre mécanismes atomiques :

1. *L'ajout de propriété*, c'est-à-dire la définition d'une nouvelle propriété (méthode, attribut, type virtuel).
2. *La redéfinition de propriété* (*overriding*).
3. *L'ajout de super-classe*
4. *La généralisation de propriété*, c'est-à-dire la définition d'une propriété dans une super-classe de la classe courante.

Les trois premiers mécanismes existent également avec la spécialisation, alors que le dernier est une réelle nouveauté du raffinement.

A ces quatre mécanismes, on peut en ajouter un cinquième *l'unification de classe*, c'est-à-dire l'affirmation qu'une classe introduite par deux modules différents est la même (voir Section 3.3.6). Ce cinquième mécanisme ne sera cependant pas davantage développé dans cette thèse, pas plus qu'il n'est géré par le compilateur PRMC.

Comme pour la spécialisation et l'héritage, ce mécanisme est parfaitement intuitif quand les ordres considérés sont totaux — par analogie à l'héritage simple. Cependant quand le raffinement est multiple ou qu'il est utilisé conjointement à la spécialisation, le comportement n'est plus trivial et peut engendrer un certain nombre de conflits (voir Section 3.3.6). Les langages à typage dynamique règlent ces problèmes grâce à l'ordre chronologique des définitions, ce qui est impossible quand les hiérarchies sont calculées statiquement durant la compilation.

3.3.2 Exemple

Afin d'illustrer ce mécanisme nous présentons dans cette section un exemple d'utilisation de modules et de raffinement de classe. Bien que cet exemple soit totalement fictif et que son utilité puisse être débattue, il met en évidence les différents mécanismes décrits ci-dessus.

Cet exemple est constitué de quatre modules. Le premier module introduit la classes `Computer` et une sous-classe `PC`. Nous ne considérons que la méthode `switch_on_off` qui allume ou éteint l'ordinateur.

Le second module dépend du premier. Il généralise la méthode qui allumait ou éteignait l'ordinateur dans la classe `Appliance`. Par la même occasion, il rajoute cette classe comme super-classe d'ordinateur. Enfin, deux autres classes qu'on peut allumer ou éteindre sont rajoutées (`Lamp` et `ExpressoMaker`).

Vient ensuite un module qui introduit les notions d'hyper-graphe, d'arête et de sommets (`Hypergraph`, `Edge` et `Vertex`).

Enfin, un dernier module importe les modules `computer` et `hypergraph` pour réaliser des réseaux d'ordinateurs. Un ordinateur étant un sommet dans un réseau, la super-classe `Vertex` est ajoutée aux `Computer`.

LISTING 3.1 : Computer Module

```
class Computer
    def switch_on_off ...
end

class PC
    inherit Computer
...
end
```

LISTING 3.2 : Appliance Module

```
import computer

class Appliance
    def switch_on_off ...
end

class Computer
inherit Appliance
end

class Lamp
inherit Appliance
    def switch_on_off ...
end

class ExpressoMaker
inherit Appliance
    def switch_on_off ...
end
```

LISTING 3.3 : Hypergraph Module

```
class Hypergraph
    def @nodes: Array[Vertex]
    def @edges: Array[Edge]
    def is_connected ...
    def diameter ...
end

class Vertex
    def @edges: Array[Edge]
    ...
end

class Edge
    def @nodes: Array[Vertex]
    ...
end
```

LISTING 3.4 : Network Module

```
import hypergraph
import computer

class Computer
inherit Vertex
    def @hostname: String
    def switch_on_off ...
end
```

3.3.3 Méta-modèle des modules et des classes

Le méta-modèle des *modules* et du *raffinement* de classes part de l'observation suivante : les classes sont aux modules ce que les propriétés sont aux classes ; la relation de dépendance entre les modules s'apparente à la relation de spécialisation entre les classes et l'importation des classes s'apparente à l'héritage de propriété. Cette observation nous permet de définir un méta-modèle isomorphe à celui des classes et des propriétés (voir Section 3.2.1) avec deux entités associées aux classes — par analogie : *classes locales* et *classes globales* — et une entité associée au concept de module (figure 3.4).

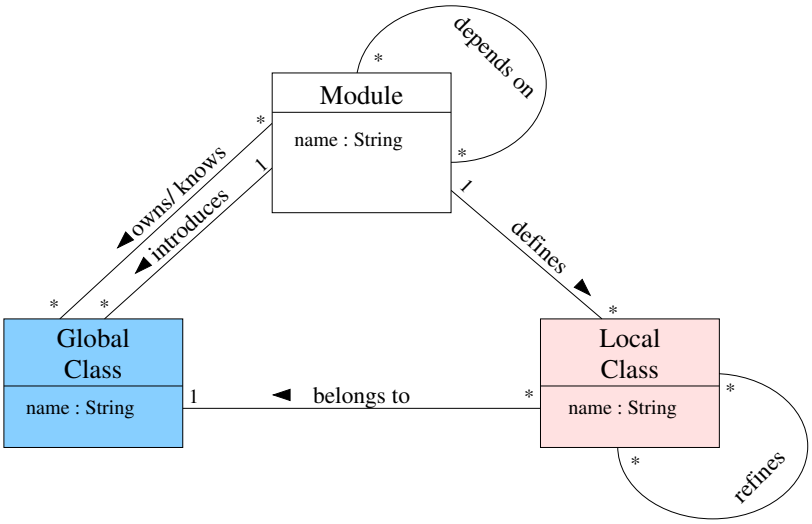


FIGURE 3.4 : Méta-modèle des modules et des classes

Modules

Les *modules* sont des hiérarchies de classes. Un module dépend soit d'aucun, soit d'un ou de plusieurs autres modules appelés *super-modules* — ce module est donc un *sous-module* de ses super-modules. Comme la relation de spécialisation, la relation de dépendance est un ordre partiel strict. Les *classes locales* sont définies dans un module. Une classe locale est définie comme une classe ordinaire (voir Section 3.2.1). Les *classes globales* rassemblent les classes locales orthogonalement aux modules. Chaque module connaît un ensemble de classe globales, c'est-à-dire l'ensemble des classes utilisables par un module.

Un module est un triplet ⟨nom, {super-modules}, {classes locales}⟩, où super-modules est un ensemble de modules déjà définis. La relation de dépendance possède un mécanisme d'importation (comparable à la relation entre spécialisation et héritage), c'est-à-dire qu'un module importe les classes de ses super-modules. Comme pour l'héritage des propriétés, l'importation de classes est constituée de deux niveaux d'importation. D'abord,

l'importation des classes globales : un module importe toutes le classes globales de ses super-modules. Ensuite, chaque définition de classe locale est traitée. Si le nom d'une classe locale est identique à celui d'une classe globale importée, la classe locale est attachée à cette classe globale. Sinon, une nouvelle classe globale de même nom est introduite dans le module.

Remarque. Il n'y a aucune différence, au niveau du modèle, entre les classes importées seulement pour être utilisées et celles importées pour être raffinées. Comme la visibilité pour la spécialisation, ceci présenterait cependant un intérêt pour le programmeur.

Pour chaque classe globale connue par un module, l'existence d'une classe locale correspondante est supposée. La définition de cette classe est soit une définition explicite, soit un raffinement implicite — par extension la classe locale associée est dite classe implicite. La relation de raffinement de classe est analogue à la relation de redéfinition de propriétés, elle est déduite de la relation de dépendance entre les modules. De plus, comme les modules sont des hiérarchies de classes, leur classes locales sont reliées par une *relation de spécialisation*. Cette relation de spécialisation est déduite des déclarations explicites de super-classes dans les définitions de classes locales ainsi que de la relation correspondante dans les super-modules, c'est-à-dire qu'un module importe la relation de spécialisation depuis ses super-modules. Enfin, l'héritage de propriétés est donc maintenant géré par les deux relations, celle de spécialisation et celle de raffinement.

Sémantique d'un programme par aplatissement

Un programme est un ensemble de modules fermé par la relation de spécialisation. Il correspond à un module, appelé *bottom module*, et à toutes ses dépendances. Ce module peut éventuellement être implicite. La sémantique d'un programme donnée par l'*aplatissement* (*flattering*) des classes locales sur ce *bottom module*. Cette idée d'aplatissement correspond à une transformation par laquelle le programme n'utiliserait plus que la relation de spécialisation.

Il y'a plusieurs manières de réaliser cet aplatissement, qui sont conceptuellement légèrement différentes.

1. Le programme final correspond à l'ensemble des classes globales définies par tous les modules. La relation de spécialisation entre les classes résulterait de la projection de la relation de spécialisation définie dans chaque module et les propriétés locales associées seraient les plus spécifiques par rapport à la relation de raffinement. Le cas des appels à *super* peut être traité par fusion du code ou par un appel à une nouvelle propriété globale générée pour l'occasion.
2. Le programme final résulte de l'ensemble des classes locales du *bottom module* avec la relation de spécialisation importée de ses super-modules. L'ensemble des propriétés locales serait le même que précédemment.

3. Enfin, le programme peut résulter de l'union de toutes les classes locales définies dans tous les super-modules avec la relation de spécialisation définie comme l'union de toutes les relations de spécialisation plus celle de raffinement. Les propriétés locales sont les mêmes que celles définies dans chaque module et chaque classe. Dans cette vision, toutes les classes locales sont abstraites, sauf celles du *bottom module*.

La dernière proposition est conceptuellement très simple mais elle produirait une très grande hiérarchie de classes qui comporterait de nombreux cas d'héritage multiple. De plus, il n'est pas raisonnable de considérer la relation de spécialisation au même niveau que la relation de raffinement. Intuitivement la relation de raffinement est plus *forte* que la relation de spécialisation, de la même manière que la redéfinition d'une propriété est plus *forte* que l'héritage de propriétés. Les deux autres propositions sont approximativement similaires, bien que dans la première les classes globales soient concrètes — ce sont les classes utilisées réellement par le programme.

Dans la suite de ce chapitre nous utiliserons indistinctement les trois définitions, tandis qu'en pratique, le compilateur PRMC se servirait plutôt de la seconde (voir Section 7.5.2).

3.3.4 Formalisation

Dans cette section, nous présentons la formalisation des modules et du raffinement de classes proposée dans [Ducournau *et al.*, 2007].

Un *module* est une hiérarchie de classes et un *programme* est une hiérarchie de modules.

Définition 3.6 *Le modèle d'un programme est un le tuple :*

$$\mathcal{P} = \langle X^{\mathcal{P}}, \prec^{\mathcal{P}}, G^{\mathcal{P}}, L^{\mathcal{P}}, N^{\mathcal{P}}, \mathrm{nom}_{\mathcal{P}}, \mathrm{glob}_{\mathcal{P}}, \mathrm{intro}_{\mathcal{P}}, \mathrm{def}_{\mathcal{P}} \rangle$$

où :

- $X^{\mathcal{P}}$ *est un ensemble de* modules.
- $\prec^{\mathcal{P}}$ *est une relation de dépendance entre modules — qui est transitive, antisymétrique, et antiréflexive. On note* $\preceq^{\mathcal{P}}$ *(resp.* $\prec^{\mathcal{P}}_{d}$*) la fermeture réflexive (resp. la réduction transitive) de* $\prec^{\mathcal{P}}$*. De plus,* $\langle X^{\mathcal{P}}, \preceq^{\mathcal{P}} \rangle$ *est un ordre partiel.*
- $G^{\mathcal{P}}$ *et* $L^{\mathcal{P}}$ *: deux ensembles disjoints des classes globales et locales.*
- $N^{\mathcal{P}}$ *: l'ensemble des identifiants (noms) des classes.*
- $\mathrm{nom}_{\mathcal{P}}$ *:* $G^{\mathcal{P}} \uplus L^{\mathcal{P}} \to N^{\mathcal{P}}$*, une fonction de nommage des modules et des classes.*
- $\mathrm{glob}_{\mathcal{P}}$ *: une fonction de* $L^{\mathcal{P}} \to G^{\mathcal{P}}$ *qui associe une classe locale à sa classe globale.*
- $\mathrm{intro}_{\mathcal{P}}$ *: une fonction de* $G^{\mathcal{P}} \to X^{\mathcal{P}}$ *qui associe une classe globale à son module d'introduction.*
- $\mathrm{def}_{\mathcal{P}}$ *: une fonction de* $L^{\mathcal{P}} \to X^{\mathcal{P}}$ *qui associe une classe locale à son module de définition.*

Définition 3.7 *Chaque module* $\mathcal{M} \in X^{\mathcal{P}}$ *est une hiérarchie de classes définie par*

$$\mathcal{M} = \langle X^{\mathcal{M}}, \prec^{\mathcal{M}}, G^{\mathcal{M}}, L^{\mathcal{M}}, N^{\mathcal{M}}, \text{nom}_{\mathcal{M}}, \text{glob}_{\mathcal{M}}, \text{intro}_{\mathcal{M}}, \text{def}_{\mathcal{M}} \rangle$$

où :

- $X^{\mathcal{M}} = L^{\mathcal{P}}_{\mathcal{M}}$ *est l'ensemble des classes locales de* $\mathcal{M}$ *dans* $\mathcal{P}$.
- *Tous les* $N^{\mathcal{M}}$ *sont les mêmes ensembles de noms.*
- *Tous les* $G^{\mathcal{M}}$ *sont des parties d'un ensemble* G *de propriétés globales.*
- *Tous les* $L^{\mathcal{M}}$ *sont disjoints, et* def *(resp.* glob*) est un raccourci non ambigu pour une fonction* $\text{def}_{\mathcal{M}}$ *(resp.* $\text{glob}_{\mathcal{M}}$*).*

À l'exception des définitions 3.1 et 3.3, toutes les notations, définitions et contraintes existantes pour $\mathcal{H}$ (voir Section 3.2) le sont aussi pour $\mathcal{P}$ en substituant $\mathcal{H}$ (resp. classe, propriétés) par $\mathcal{P}$ (resp. module, classes). Naturellement, les exposants $\mathcal{P}$ et $\mathcal{M}$ deviennent maintenant obligatoires.

La dépendance de module entraîne le *raffinement de classe* qui est l'analogue de la redéfinition de propriété (Définition 3.2).

Définition 3.8 (Raffinement de classes) *Le raffinement de classe est défini comme la relation* $\ll^{\mathcal{P}}$ *entre une classe locale de* $L^{\mathcal{P}}_{\uparrow m}$ *définie dans un module* m *et la classe locale correspondante dans un super module de* m

$$c \ll^{\mathcal{P}} c' \stackrel{\text{def}}{=} \text{glob}_{\mathcal{P}}(c) = \text{glob}_{\mathcal{P}}(c') \wedge \text{def}_{\mathcal{P}} \prec^{\mathcal{P}} \text{def}_{\mathcal{P}}(c')$$

La relation $\ll^{\mathcal{P}}$ définit un ordre partiel strict ($\ll^{\mathcal{P}}_{d}$ représente sa réduction transitive).

Remarque. Pour simplifier le modèle, quand on considère l'ensemble des modules $X^{\mathcal{M}}$, on suppose qu'il contient une définition de classe locale — potentiellement vide — pour chaque classe globale connue par $\mathcal{M}$.

Étant donné deux modules $\mathcal{M}' \prec^{\mathcal{P}} M$, la fonction $id^{\mathcal{M}}_{\mathcal{M}'} : X^{\mathcal{M}} \to X^{\mathcal{M}'}$ qui associe une classe $c \in X^{\mathcal{M}}$ au singleton $c' \in X^{\mathcal{M}'}$ tel que $c' \ll^{\mathcal{P}} c$ est injective.

Qui plus est, $\prec^{\mathcal{M}}_{e}$ représente la relation de spécialisation explicitement définie pour un module $\mathcal{M}$ par des définitions successives de classes (voir Section 3.2.3), alors $\prec^{\mathcal{M}}$ est défini comme la fermeture transitive de l'union de $\prec^{\mathcal{M}}_{e}$ et de $id^{\mathcal{M}}_{\mathcal{M}'}$ (pour tout $\mathcal{M}'$ tel que $\mathcal{M} \prec^{\mathcal{P}}_{d} \mathcal{M}'$).

Toutefois, les modules sont des hiérarchies de classes légèrement étranges, car pour des hiérarchies normales $\prec_e$ et $\prec_d$ coïncident. Alors que, pour que le modèle soit valide, $\prec^{\mathcal{M}}$ doit être antisymétrique. Cette observation d'étrangeté est renforcée, dès lors qu'on considère les propriétés globales, le raffinement de classes entraîne que les modules sont des hiérarchies de classes qui ne vérifient pas la contrainte suivante : une propriété locale doit être connue dans le contexte d'une classe.

Contrainte 3.9 (Héritage de raffinement) *Soit* $\mathcal{M}, \mathcal{M}' \in X^{\mathcal{P}}$*, tel que* $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$*. Alors l'ensemble* $G^{\mathcal{M}'} \subseteq G^{\mathcal{M}}$.

Soit $c \in X^{\mathcal{M}}$ *et* $c' \in X^{\mathcal{M}'}$ *tel que* $c \ll^{\mathcal{P}} c'$. *Alors l'ensemble* $G^{\mathcal{M}'}_{c'} \subseteq G^{\mathcal{M}}_{c}$. *Enfin, pour toute propriété* $g \in G^{\mathcal{M}'}$, $\text{intro}_{\mathcal{M}}(g) = id^{\mathcal{M}'}_{\mathcal{M}}(\text{intro}_{\mathcal{M}'}(g))$

Cette dernière contrainte ne tient pas compte de la *généralisation de propriétés* qui doit être traitée de façon spécifique.

L'ensemble des propriétés globales pour une classe ($G^{\mathcal{M}}_{c}$) est maintenant séparé en trois sous-ensembles : celui des propriétés introduites par c dans $\mathcal{M}$ ($G^{\mathcal{M}}_{+c}$), celui des propriétés héritées par spécialisation ($G^{\mathcal{M}}_{\uparrow c}$) et celui des propriétés héritées par raffinement ($G^{\mathcal{M}}_{\Uparrow c}$). Les équations suivantes formalisent ces ensembles — et remplacent les deux exceptions faites par les définitions (3.1) et (3.3) :

$$G^{\mathcal{M}}_{c} \stackrel{\text{def}}{=} G^{\mathcal{M}}_{+c} \uplus (G^{\mathcal{M}}_{\uparrow c} \cup G^{\mathcal{M}}_{\Uparrow c}) = G^{\mathcal{M}}_{+c} \uplus G^{\mathcal{M}}_{\uparrow c} \uplus G^{\mathcal{M}}_{\Uparrow +c}) \tag{3.10}$$

$$G^{\mathcal{M}}_{\uparrow c} \stackrel{\text{def}}{=} \bigcup_{c \ll^{\mathcal{P}} c'} G^{\text{def}_{\mathcal{P}}(c')}_{c'} \tag{3.11}$$

$$G^{\mathcal{M}}_{\uparrow +c} \stackrel{\text{def}}{=} \bigcup_{c \ll^{\mathcal{P}} c'} G^{\text{def}_{\mathcal{P}}(c')}_{+c'} \tag{3.12}$$

$$G^{\mathcal{M}}_{+c} \cup G^{\mathcal{M}}_{\Uparrow c} = \text{intro}^{-1}_{\mathcal{M}}(c) \tag{3.13}$$

$$\tag{3.14}$$

Ces quelques modifications rajoutent l'ensemble des propriétés introduites par les classes raffinées par c. Comme les propriétés ne sont plus seulement introduites par des classes mais par des classes dans le contexte d'un module, l'ensemble des équations est modifié pour en tenir compte. Le domaine de la fonction $\text{intro}_{\mathcal{P}} : G^{\mathcal{P}} \uplus G \to X^{\mathcal{P}}$ doit maintenant être étendu à toutes les propriétés $g \in G$:

$$g \in G^{\mathcal{M}}_{+c} \stackrel{\text{def}}{\iff} \text{intro}_{\mathcal{P}}(g) = \mathcal{M}. \tag{3.15}$$

Enfin, le raffinement de classes entraîne une modification sur la relation de redéfinition des propriétés locales :

Définition 3.10 (Raffinement de propriété) *Soit deux modules* $\mathcal{M} \prec^{\mathcal{P}} \mathcal{M}'$, *le* raffinement de propriétés *est la relation notée* $\lhd$, *définie sur les ensembles de propriétés locales* $L^{\mathcal{M}}$ *et* $L^{\mathcal{M}'}$ *telle que, étant donné une propriété* $lp \in L^{\mathcal{M}}$ *et* $lp' \in L^{\mathcal{M}'}$,

$$lp \lhd lp' \stackrel{\text{def}}{\iff} \text{glob}_{\mathcal{M}}(lp) = \text{glob}_{\mathcal{M}'}(lp') \wedge \text{def}_{\mathcal{M}}(lp) \ll^{\mathcal{P}} \text{def}_{\mathcal{M}'}(lp')$$

.

3.3.5 Instanciation du méta-modèle

De la même manière que pour l'héritage multiple, le modèle d'un programme est construit par définitions successives de modules. Le protocole, qui va être décrit ci-

dessous, est utilisé par le compilateur pour construire le modèle de l'application qu'il doit compiler.

Une définition de module est analogue a une définition de classe, excepté que classe (resp. propriétés) est remplacé par module (resp. classes). Cependant le protocole d'instanciation est légèrement différent de celui des classes.

1. *Mise à jour de la hiérarchie de modules.* Le nouveau module m est ajouté à l'ensemble des modules $X^{\mathcal{P}}$. Pour chaque super-module de m une paire est ajoutée à $\prec_d^{\mathcal{P}}$. Comme dans le cas de la mise à jour de la hiérarchie de classes (voir Section 3.2.3) les ambiguïtés (voir Section 3.3.6) et les arcs de transitivité sont vérifiés.
2. *Importation des classes globales.* L'ensemble $G^{\mathcal{P}}_{\uparrow m}$ est construit en accord avec (3.2) et les conflits de classes globales (voir Section 3.3.6) sont vérifiés.
3. *Hiérarchie de classes locales.* Puis, chaque définition de classe locale est traitée comme une définition de classe (voir Section 3.2.3) sans tenir compte des définitions locales de propriétés. Les nouvelles classes locales sont créées. On suppose pour simplifier qu'une définition de classe locale explicite est fournie pour chaque classe globale. L'ensemble des classes locales est déterminé par l'équivalent pour les modules de l'équation (3.6). La relation de spécialisation est importée de ses super-modules direct, puis l'absence de cycle est vérifié (voir Section 3.3.6). Puis l'ensemble $G^{\mathcal{P}}_{+m}$ des nouvelles classes globales est constitué avec l'ensemble des classes locales non associés à une classe globale.
4. *Héritage de classes locales.* Dans la définition d'un module m, chaque définition de classe locale c est un triplet de forme ⟨`nom, super-classes, propriétés locales`⟩ (voir Section 3.2.1). Une fois que la hiérarchie complète des classes $\langle X^{\mathcal{M}}, \prec^{\mathcal{M}}\rangle$ a été traitée, les définitions et les héritages de propriétés peuvent avoir lieu. Tout d'abord on considère l'héritage de propriété globale (voir section 3.2.3, seconde étape) dans lequel on tient maintenant compte du raffinement de classe, c'est-à-dire on construit l'ensemble des propriétés hérités par la spécialisation ($G^{\mathcal{M}}_{\uparrow c}$) et l'ensemble hérité par le raffinement ($G^{\mathcal{M}}_{\Uparrow c}$). Les conflits de propriétés globales sont calculés sur l'union des deux ensembles hérités. L'ensemble des nouvelles propriétés globales est construit. Enfin, toutes les propriétés locales définies par `localdef` sont traités comme précédemment (voir section 3.2.3, troisième et quatrième étapes). Seule la *généralisation de propriétés* doit être traitée de manière *ad hoc*.

3.3.6 Conflits

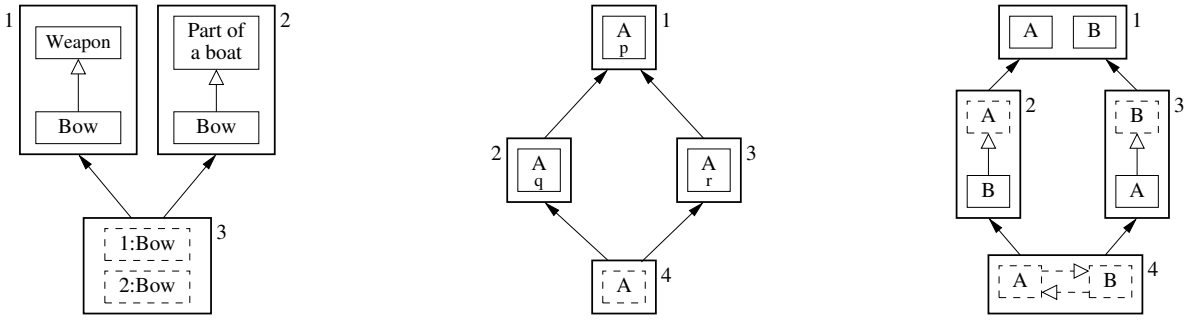
La raffinement d'une sous-classe — c'est-à-dire d'une classe qui hérite d'une autre classe — introduit automatiquement un genre d'héritage multiple, même s'il est obtenu par cumul des deux relations $\prec^{\mathcal{M}}$ et $\ll^{\mathcal{P}}$. Les conflits de l'héritage multiple ne peuvent donc plus être ignorées dans le contexte des modules et du raffinement de classe. A l'instar de l'héritage multiple, le raffinement de classes présente un ensemble de conflits, certains

d'entre eux reposent sur les mêmes bases que l'héritage multiple, tandis que d'autres leurs sont spécifiques.

Importations multiples

L'importation des classes globales et locales introduit le même genre de conflits que l'héritage des propriétés globales et locales.

Conflit de classes globales. Par analogie, il existe un *conflit de classes globales* quand deux classes homonymes mais distinctes sont importées dans un même module (Figure 3.5(a)). Comme la définition de ce conflit est identique au conflit de propriétés globales — aux termes *classes* et *propriétés* près —, les résolutions possibles sont analogues (voir Section 3.2.4). Pour une propriété, la qualification explicite consiste à l'annoter par sa classe d'introduction — ou du moins par n'importe quelle classe dans laquelle la propriété est non ambiguë —, tandis que pour qualifier une classe, c'est le module d'introduction qui sert d'annotation.



(a) Conflit de classes globales

(b) Conflit de classes locales

(c) Conflit de spécialisation

FIGURE 3.5 : Importations multiples

Conflit de classes locales. De la même manière qu'il existe un conflit de classes globales, il existe un *conflit de classes locales* quand un même module importe deux classes locales d'une même classe globale et qu'aucune des deux n'est plus spécifique (Figure 3.5(b)). Néanmoins, cette fois il y a fondamentalement une différence avec le conflit de propriétés locales. En effet, une propriété est un élément concret du langage qui doit être unique pour pouvoir être appelé, tandis qu'une classe est un ensemble de propriétés globales et locales. La résolution de ce conflit est donc directe, par opposition au conflit de propriétés locales qui n'avait pas solution intrinsèque, elle consiste à raffiner la classe globale associée au conflit.

Remarque. Si un module raffine systématiquement les classes globales qu'il importe, ce conflit n'a pas lieu d'être — il est systématiquement résolu.

Cependant, même une fois résolu le conflit de classes locales, il peut demeurer des conflits de propriétés dans la classe ; nous les examinerons juste après.

Conflit de spécialisation. Les modules et le raffinement entraînent un nouveau cas de conflit, le *conflit de spécialisation*. Ce conflit arrive lorsque deux modules introduisent une relation de spécialisation dans des modules distincts qui forme un cycle dans la relation de spécialisation d'un sous-module commun (Figure 3.5(c)). La nouvelle relation de spécialisation ne formerait plus un ordre partiel mais seulement un pré-ordre.

Une solution intuitive consiste à interdire au programmeur la définition d'une spécialisation entre deux classes qui entraîne un tel cycle. Malheureusement, cette solution n'est pas suffisante, ce conflit est généralement engendré par deux modules distincts — comme dans l'exemple — importés par un troisième module — éventuellement un *bottom module* implicite.

Une alternative plus robuste consiste à dire que ce cycle dans la relation de spécialisation constitue une classe d'équivalence — puisque la spécialisation implique l'inclusion de l'ensemble des instances et de l'ensemble des propriétés. La résolution naturelle de ce conflit est donc l'unification des classes conflictuelles.

Attention. Le compilateur PRMC n'implémente pas l'unification de classe. Il annonce simplement un cycle dans le graphe de spécialisation avant de s'arrêter

Héritage de propriétés globales

En plus des conflits spécifiques liés aux modules et au raffinement de classe, ces derniers exacerbent les conflits de l'héritage multiple. Un conflit de propriétés globales, qui était circonscrit à des cas de réel héritage multiple, est maintenant possible même en héritage simple. En effet, deux propriétés globales homonymes mais distinctes peuvent être introduites par une même classe mais dans des modules incomparables.

Les solutions proposées pour le conflit de propriétés globales (voir Section 3.2.4) sont toujours valables mais doivent être complétées. Dans le cas de la qualification explicite et du renommage local, les classes globales ne sont plus suffisantes pour lever l'ambiguïté. Le recours à la qualification par un module devient nécessaire. Si on considère le cas d'un appel à une méthode, qui participe un conflit de propriétés globales, dans une classe, qui participe, elle même, à un conflit de classes globales. La désignation explicite d'une telle méthode est quand même nettement plus verbeuse :

```
module_intro_classe:classe_intro:module_intro_methode::nom_methode
```

La généralisation de propriété permet de simuler l'unification des propriétés globales en conflit. Ceci ne constitue pas, à proprement parler, une nouvelle solution mais permet au programmeur d'utiliser une autre politique de résolution de conflit que la politique par

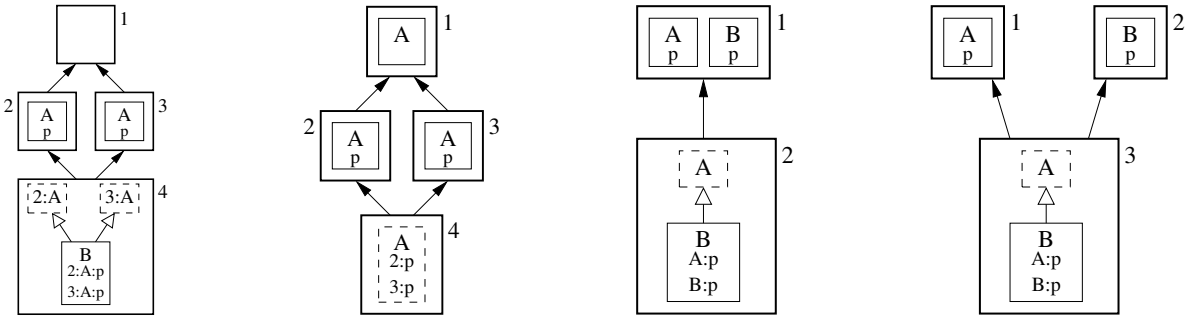


FIGURE 3.6 : Nouveaux cas de conflits de propriétés globales

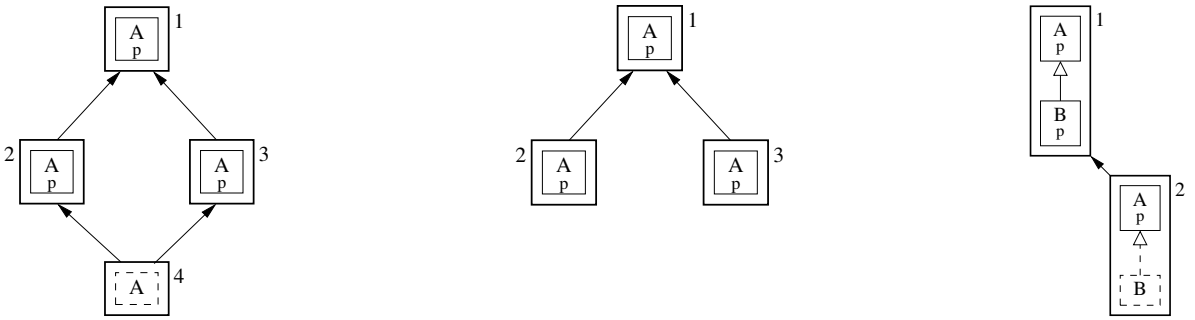


FIGURE 3.7 : Nouveaux cas de conflits de propriétés locales

défaut du langage. Puisqu'il est possible de simuler l'unification, la généralisation constitue un argument contre ce choix de politique par défaut.

Héritage de propriétés locales

Le conflit de propriétés locales est, quant à lui, exacerbé par les modules et le raffinement de classes. En effet, même si ces derniers ne changent pas la nature du conflit ni ses solutions, ce conflit peut maintenant survenir dans des cas où l'héritage est simple mais où l'importation est multiple — c'est-à-dire que le module courant possède au moins deux super-modules.

Appel à super

Comme les solutions au conflit de propriétés locales sont inchangées, c'est uniquement le mécanisme utilisé pour combiner les méthodes qui doit être adapté aux modules et au raffinement de classes.

On peut ainsi utiliser deux mots-clefs distincts pour réaliser l'appel à `super`, un pour le raffinement et l'autre pour la redéfinition, ou bien s'en contenter d'un seul qui doit être

qualifié respectivement, par un nom de module ou un nom de classe. L'utilisation simultanée des deux annotations n'aurait, par contre, aucun sens [5].

La combinaison automatique des propriétés conflictuelles par linéarisation (`call-next-method` de CLOS), nécessiterait de tenir compte des redéfinitions par raffinement en plus de celles par spécialisation. Intuitivement la relation de raffinement est plus « forte » que la relation de spécialisation, elle doit donc être considérée comme prioritaire dans la linéarisation.

3.4 Le langage PRM

Le langage PRM [Privat, 2006] est un langage de programmation par objets avec héritage multiple et statiquement typé. Il se classe dans la même famille de langages que C++ ou EIFFEL. C'est un langage avant tout universitaire — mais néanmoins utilisable — cible et fruit de nos recherches sur les langages objet statiquement typés. Il intègre un certain nombre de fonctionnalités rarement rencontrées dans des langages utilisés en production — du moins au moment de sa spécification — tout en restant simple et clair.

3.4.1 Un langage de laboratoire : PRM

Le langage objet PRM a été conçu par Privat [2006] dans le cadre de sa thèse car il ne parvenait pas à adapter un compilateur existant à ses besoins, car cela aurait représenté un investissement trop coûteux en temps. Il décida donc de spécifier et d'implémenter un langage purement objet, statiquement typé et en héritage multiple, dont la sémantique, notamment celle de l'héritage, serait donnée par les méta-modèles présentés dans ce chapitre (voir Section 3.2). Il le dota de plus, de modules et d'un mécanisme de raffinement de classes, dont il proposait des spécifications originales (voir Section 3.3).

Les spécifications de PRM cherchent à respecter un grand principe qui a trouvé, historiquement, de nombreuses formulations :

> Pluralitas non est ponenda sine necessitate. *(Une pluralité ne doit pas être posée sans nécessité.)*
>
> Rasoir (ou Minimalité) [Ockham, 1319]

> Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.
>
> Perfection [de Saint-Exupéry, 1939]

> Un bon langage de programmation doit permettre au programmeur de s'exprimer facilement, il doit donc être le plus proche [possible] du mode de pensée humain.
>
> Expressivité [Privat, 2006]

5. A l'exception d'une utilisation pour désambiguïser un conflit d'ordre global, mais il ne s'agit plus du même problème.

En pratique, ce principe est appliqué en PRM de la façon suivante :

minimalité des concepts le noyau du langage contient le minimum des concepts significatifs. Il contient uniquement des concepts objets, donc aucun mécanisme statique.

syntaxe simple et concise le langage se veut clair et précis sans pour autant être verbeux. Les mots-clefs ont une seule signification. La langage propose du sucre syntaxique pour rendre le code plus agréable à lire, comme le montre l'exemple du plus court, néanmoins fonctionnel, programme PRM :

```
println("Hello␣world")
```

La syntaxe de PRM est à base de mots-clefs, dans la lignée des langages PASCAL, EIFFEL ou plus récemment RUBY. Par exemple, les blocs sont définis à l'aide de `do ... end` et, plus généralement, `end` termine toutes les constructions (`class`, `then`). Toutes les propriétés sont définies par un mot-clef unique `def` et leur nom doit être entièrement en minuscule. Les attributs se distinguent des méthodes par le caractère `@` qui les préfixe[6]. Les noms contenant des majuscules sont réservés aux noms de types et un nom de classe doit au moins contenir une minuscule — ce qui fait au moins deux caractères.

Remarque. Cette particularité n'est pas dérangeante car dans les « vrais » programmes les classes ont toujours des noms plus explicites. Il n'y a que pour les exemples et les programmes *jouets* qu'elle s'avère contraignante et nous ferons comme si elle n'existait pas.

Après ces quelques considérations, le lecteur devrait être en mesure de comprendre les exemples de cette thèse. Nous n'entrerons donc pas plus dans les détails syntaxiques.

3.4.2 Héritage multiple, modules et raffinement de classes

PRM permet l'héritage multiple sans restrictions et une classe peut hériter directement de plusieurs super-classes directes. Les modules et le raffinement de classe constituent sans aucun doute la plus grande spécificité du langage PRM. La sémantique de tous ces mécanismes est directement dictée par les méta-modèles présentés dans les sections précédentes (3.2 et 3.3).

Héritage multiple

Comme le langage propose un héritage multiple le langage se doit de gérer les conflits — conflits qui sont exacerbés par les modules et le raffinement de classes. Pour résoudre les conflits de propriétés globales, PRM utilise la qualification de nom (voir Section 3.2.4). La syntaxe adoptée est proche de celle des appels statiques en C++, c'est-à-dire les noms de propriétés ambigus sont préfixés par un qualificateur séparé par `::`, mais n'a pas le même sens. Pour les conflits de propriétés locales, PRM impose la redéfinition explicite, combinée à l'utilisation de `super` qualifié.

6. Le caractère _ est prévu en remplacement dans les prochaines versions et il est aussi géré comme synonyme.

Exemples de conflits en PRM. La figure 3.8(a) montre deux classes `Etudiant` et `Salarie` qui introduisent deux propriétés globales distinctes mais homonymes, ainsi qu'une sous-classe commune `EtudiantSalarie`. Dans cette dernière, il existe deux propriétés globales `departement`. Le listing 3.5 montre comment en PRM on qualifie de façon non ambiguë ces deux propriétés.

Dans l'exemple suivant (Listing 3.6), une classe racine `Vehicule` introduit la méthode abstraite `klaxonne`. Cette méthode est redéfinie par deux sous-classes différentes (`Voiture` et `VehiculeUrgence`). Leur sous-classe commune, `Ambulance`, doit redéfinir cette propriété car aucune des définitions n'est plus spécifique. Le programmeur a, dans cet exemple, considéré que pour la propriété `klaxonne`, le comportement d'un `VehiculeUrgence` était souhaitable.

Remarque. Ces spécifications limitées de l'héritage multiple ne sont pas définitives. Elles représentent le strict necessaire pour utiliser l'héritage multiple, l'essentiel de nos efforts étant tournés vers le développement d'un compilateur fonctionnel. La prochaine version devrait proposer un mécanisme de linéarisation en complément du `super` qualifié.

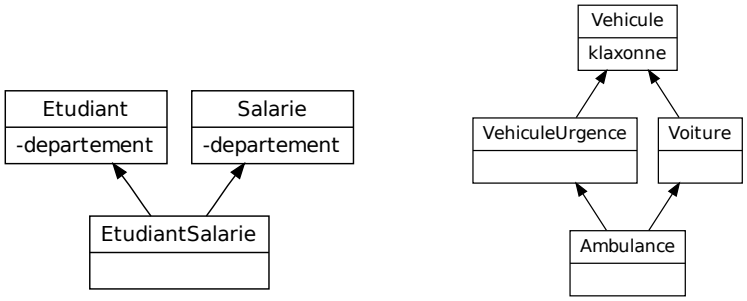
Modules et raffinements de classes

Les modules permettent une séparation claire des préoccupations en unités fonctionnelles, tandis que le raffinement de classe permet une extension des concepts utilisés, réduisant de ce fait le nombre d'entités nécessaires dans un programme. Les modules sont l'unité de code du langage, ils regroupent un ensemble de classes et ils dépendent les un des autres.

D'un côté, le mécanisme de raffinement permet de simplifier de nombreux patrons de conceptions (visiteur acyclique [Gamma *et al.*, 1994], ...) — car les méthodes nécessaires à l'ajout d'une fonctionnalité par un module peuvent être ajoutées dans le module en question même si elles n'étaient pas prévues à l'origine. D'un autre côté, le prix à payer pour cette (trop) grande flexibilité est une complexification des schémas de compilation.

Un module se présente sous la forme d'un fichier qui peut contenir plusieurs définitions de classes et un ensemble de méthodes hors classe. Une méthode hors classe, est transformée en une méthode qui appartient à la classe racine `Any` avec une visibilité `selfish` (voir Section 3.4.4). Donc tous les objets peuvent invoquer ce message, mais seulement sur eux-mêmes (`self.message`, ou `message`). Cela permet donc l'équivalent des méthodes statiques, mais en en conservant le caractère objet. Il est aussi possible de définir directement du code dans un module, auquel cas il est déclaré comme un raffinement de la méthode principale du programme (`Sys::main`).

Comme le raffinement de classe et de propriété est l'un des concepts clés du langage, il se fait naturellement, c'est-à-dire exactement comme une introduction de classe ou de propriété. Il n'y a pas de différence syntaxique entre l'introduction et le raffinement d'une classe, pas plus qu'entre l'introduction et la redéfinition d'une propriété. Nous pensions,



(a) Conflit de propriétés globales (b) Conflit de propriétés locales

FIGURE 3.8 : Diagramme des classes utilisé par les exemples de conflits

LISTING 3.5 : Résolution d'un conflit de propriétés globales en PRM

```
let etudiant_salarie := new EtudiantSalarie

let etudiant: Etudiant := etudiant_salarie
let salarie: Salarie := etudiant_salarie

println(etudiant.departement)
println(etudiant_salarie.Etudiant::departement)

println(salarie.departement)
println(etudiant_salarie.Salarie::departement)

# println(etudiant_salarie.departement) est illegal, car ambigu
```

LISTING 3.6 : Résolution d'un conflit de propriétés locales en PRM

```
class Vehicule
    def klaxonne as abstract
end


class Voiture
inherit Vehicule
    def klaxonne
    do
        println("*tuut-tuut*")
    end
end
```

```
class VehiculeUrgence
inherit Vehicule
    def klaxonne
    do println("*pin-pon*") end
end

class Ambulance
inherit VehiculeUrgence
inherit Voiture
    def klaxonne
    # Doit etre redefinie
    do VehiculeUrgence::super end
end
```

au début, que c'était une bonne chose. Mais après utilisation du langage, ce point de syntaxe s'est avéré plutôt problématique — particulièrement sans aide d'un EDI complet.

Remarque. Dans la dernière version du langage, il existe un mot-clef qui informe le compilateur qu'il s'agit d'un raffinement. Son utilisation entraîne une erreur si c'est pas le cas. Par contre, la non utilisation du mot-clef n'implique pas qu'il s'agit d'une nouvelle introduction, comme la surcharge statique est interdite en PRM, cette définition n'est pas une nouvelle introduction mais bien un raffinement. Ce comportement devrait disparaître dans la prochaine version du langage.

L'utilisation des modules et du raffinement peut entraîner des nouveaux cas de conflits (voir Section 3.3.6). Les solutions à ces conflits sont identiques à celles utiliser pour l'héritage multiple, sauf que les modules remplacent les classes pour désambiguïser les situations. Les appels aux méthodes raffinées sont fait par le même mot-clefs, `super`, que pour les méthodes redéfinies. Ce choix est très certainement une erreur et en l'absence d'un mécanisme de linéarisation unique, il devrait y avoir deux mots-clefs distincts. L'utilisation de `super` non qualifié — par un module ou une classe — provoque un avertissement, et il est considéré prioritairement comme un super de raffinement.

3.4.3 Typage statique

PRM est un langage *statiquement typé, sûr* au sens de Cardelli [2004].

Typage fort

Le typage fort consiste à interdire les conversions implicites, le programmeur est donc contraint à les expliciter. Par exemple, en JAVASCRIPT `"2"+4` → `"24"`, tandis qu'en PRM cette expression s'écrirait `"2" + 4.to_s`. C, C++ et PL 1 sont des langages qui ont un typage faible.

Types primitifs

PRM est un langage purement objet où toutes les valeurs de première classe du langage sont des objets. Ceci est aussi valable pour les types primitifs. En effet, les classes `Int` et `Real` sont sous-classes de `Number` ; la classe des entiers est en plus sous-classe de `Discrete`. Cette réification rajoute de l'expressivité au détriment des performances dans leurs usages polymorphes.

Cependant, par rapport aux autres classes, les types primitifs ont certaines spécificités. Tout d'abord ces classes ne peuvent pas être spécialisées (à l'exception de `Pointer`). De plus, les méthodes invoquées sur des types statiques primitifs sont court-circuitées par des appels statiques et ne sont donc pas des appels polymorphes.

LISTING 3.7 : La classe paramétrée `Pile`

```
class Stack[E]
inherit Collection[E]
    def @capacity: Int
    def @length: Int
    def @items: Array[E]

    def push(item: E)
    do
        assert stack_full: @length < @capacity
        @items[length] := e
        @length := @length + 1
    end
    def pop: E
    do
        assert stack_empty: @length > 0
        let item := @items[@length]
        @length := @length - 1
        return item
    end
constructor
    def init(max: Int)
    do
        @capacity := max
        @items := new Array[E].with_capacity(max)
    end
end
```

Généricité

Les classes paramétrées de PRM, contrairement à C++ et à l'instar de JAVA, sont bornées[7] et peuvent donc être compilées séparément. Lors de la compilation de la méta-classe, le type de la borne est utilisé pour la vérification de type.

Comme la généricité est implantée de manière homogène (voir Section 5.10.3) par le compilateur PRM et que les constructeurs ne sont pas hérités, il n'est pas possible d'instancier une variable typée par un type formel. On peut, tout de même, s'en sortir en utilisant un patron de conception du type *fabrique* (voir par exemple les classes `Poset` et `LinearisablePoset` de la librairie standard de PRM).

Syntaxe. La définition d'une classe générique se fait en rajoutant entre crochets le nom des types formels, éventuellement complété par une borne. Les noms de types formels sont entièrement en majuscules.

7. Le type universel (`Any`) est utilisé si aucune borne n'est fournie.

LISTING 3.8 : Exemple de typage implicite

```
let number := 3 # number is an Int
let map := new ArrayMap[String,Int] # map is an ArrayMap[String, Int]
let range := [ 'a' .. 'z' ] # range of Char
let array := [ number, fibo(2), fact(3) ] # array of Int
for x in range do ... end # x is a Char
```

Types virtuels et covariance

Initialement, les spécifications de PRM incluaient une covariance sans restriction, comme en EIFFEL. Les types virtuels se sont finalement imposés mais les traces de la covariance n'ont pas complètement été éliminées du code, d'où une certaine schizophrénie dans notre traitement de la covariance : en théorie, il n'y en a plus qu'au travers des types virtuels. En pratique, le compilateur doit bien les traiter.

Les types virtuels se définissent comme des attributs mais leur nom doit être entièrement en majuscules [8] comme les types formels.

Typage implicite

Pour alléger l'écriture des procédures, PRM propose d'inférer le type des variables en fonction de l'expression qui leur est affectée (*R-value*). Par exemple, dans le code précédent (Listing 3.6), l'instruction **let** `etudiant_salarie := new EtudiantSalarie` déclare une variable `etudiant_salarie` de type `EtudiantSalarie`.

Remarque. Le typage implicite est une forme légère d'inférence de type qui n'en a pas la puissance. L'inférence de type est utilisée par OCAML, un langage à typage statique mais sans annotation de type.

Le typage implicite a été rajouté sous le nom de *synthèse de type* dans la version 3.0 de C# .

Le typage implicite n'est pas réservé aux déclarations de variables. Il est aussi utilisé pour d'autres constructions syntaxiques telles que les définitions de tableaux, d'intervalles ou pour les itérations automatiques. Dans le cas des itérations automatiques (`for`) le type de l'expression qui sert à faire l'inférence doit obligatoirement être itérable (`Iterable`). Tandis que pour la définition d'un type d'intervalle le type doit être discret [9] (`Discrete`).

Attention. Le type absurde `None`, sous-type de tous les types, a une seule instance : `nil`.

8. En PRM, les noms entièrement en majuscules représentent tous les types formels, qu'il s'agisse de type virtuel ou de paramètre de classe générique

9. Le terme discret est particulièrement mal choisi, il s'agirait plutôt d'une classe dont les valeurs ont un prédécesseur et un successeur. Les rationnels sont énumérables et ont une topologie discrète mais ne vérifient pas cette propriété.

LISTING 3.9 : Une méthode avec un paramètre par défaut

```
class Helloer
    def say_hello
    do
        say_hello("world")
    end

    def say_hello(name: String)
    do
      println("Hello␣#{name}␣!")
    end
end
```

Comme ce type ne peut pas être utilisé pour typer les éléments du programme, `nil` est illégal aux endroits où un type implicite est attendu.

Comme le typage implicite s'est avéré pratique et prête peu à confusion — notre plus grande peur était qu'il rende le code illisible — la prochaine version du langage généralisera son utilisation pour typer les paramètres des méthodes non exportées (`private` et `selfish`).

Surcharge statique

PRM propose une forme limitée de surcharge statique. Deux méthodes homonymes peuvent exister dans une même classe si elles différent par leur nombre de paramètres. Cela a initialement été rendu obligatoire à cause de la méthode `-`, qui sert aussi bien à la soustraction, lorsqu'elle a deux paramètres, qu'à la négation lorsque le paramètre est unique. On peut utiliser cette surcharge pour définir des méthodes avec des paramètres par défaut (Listing 3.9). Cependant une certaine discipline du programmeur est nécessaire pour ne pas tordre le cou à la sémantique des objets. Il faut redéfinir la méthode avec tous les paramètres pour redéfinir effectivement la méthode, les redéfinitions des méthodes avec moins de paramètres ne doivent être faites que pour changer la valeur des paramètres par défaut.

3.4.4 Autres traits du langage

Nous terminons cette présentation du langage par quelques derniers points de spécifications.

Visibilité

Le contrôle de la visibilité est utile dans les vrais programmes, aussi bien pour le programmeur que pour le compilateur. Comme PRM ne considère que des mécanismes réellement objet, aucune optimisation particulière n'est possible pour le compilateur. La visibilité se résume donc à son usage premier, permettre au programmeur de choisir l'interface qu'il expose. Cependant elle n'est pas un point essentiel du langage et peu d'attention lui a été prêtée en PRM.

Le contrôle de la visibilité se fait au moyen de deux spécificateurs, `public` et `selfish` suivi de `:`. Ces spécificateurs sont valables pour toutes les définitions qui suivent jusqu'au spécificateur suivant ou la fin de la classe. Si aucun spécificateur n'est précisé, `public` est utilisé. Le spécificateur `selfish` [10] a un comportement identique aux attributs de SMALLTALK, seule l'instance courante (`self`) peut accéder à la propriété.

Constructeurs

PRM autorise un nombre variable de constructeurs — le mot constructeur n'est pas particulièrement bien choisi mais est communément admis, il s'agirait plutôt d'un initialiseur. Ce sont des méthodes nommées ordinaires dont le statut de constructeur est conféré par un spécificateur de visibilité `constructor` [11]. Le statut de constructeur est perdu par héritage, ce qui impose au programmeur de redéfinir un constructeur pour les sous-classes. À l'instanciation d'un objet, `init` est utilisé par défaut si aucun nom de constructeur n'est fourni.

Constantes

Actuellement, PRM ne permet plus au programmeur de définir de constantes. Ce choix est volontaire, tout simplement car les solutions actuellement utilisées dans les langages à objets ne sont pas objet. Pour pallier a ce défaut, nous utilisons une expression spéciale, `once`. Cette expression ne sera exécutée qu'une seule fois dans tout le programme, la valeur de sa première exécution est cachée et le contenu du cache est utilisé par les évaluations suivantes.

Cette alternative n'est pas aussi pratique qu'un système de constante mais permet tout de même de définir assez facilement des classes singletons ou des instances partagées.

10. Anciennement `private`. Mais nous entrevoyons une autre utilisation pour ce mot clef.

11. Le choix d'utiliser un spécificateur de visibilité est très certainement discutable et sera changé dans la prochaine version.

LISTING 3.10 : Un module définissant un dictionnaire commun

```
def the_dictionary: Map[String, String]
do
  return once new ArrayMap[String, String]
end
```

Ramasse-miettes

La gestion mémoire est une tâche complexe et contraignante pour le programmeur. PRM comme de nombreux langages modernes, utilise un ramasse-miettes (*garbage collector*) pour décharger le programmeur de cette tâche [Jones et Lins, 1996].

L'avantage d'un ramasse-miettes pour le programmeur est immédiat, il peut se concentrer sur sa tâche de développement sans se soucier des désallocations. Les fuites mémoire sont limitées, le code est plus court et plus simple à lire — donc plus maintenable.

Cependant, cette simplicité a un coût en termes de performance. Quand la mémoire disponible devient insuffisante, le ramasse-miettes doit parcourir la mémoire à la recherche des éléments inaccessibles pour les libérer.

Détecter les éléments inaccessibles est une tâche complexe et souvent indécidable. Pour cela, de nombreuses heuristiques, plus ou moins efficaces, sont disponibles — *conservatif*, *mark&sweep*, *mark&compact*, etc. En général, le travail d'un ramasse-miettes consiste à scanner les racines dans la pile, puis naviguer à travers toutes les références pour savoir lesquelles sont utilisées.

Une autre difficulté inhérente aux ramasse-miettes est de ne pas libérer la mémoire encore utilisée. Ce point, particulièrement important, n'est pourtant pas garanti par tous les ramasse-miettes — par exemple celui de Chailloux [1992]. En cas de doute, les ramasse-miettes *conservatifs*, comme celui de Boehm [1993] ne libèrent pas la mémoire, ce qui peut entraîner des fuites de mémoire — c'est-à-dire la mémoire utilisée par le programme grandit au fur et à mesure du temps.

Enfin, la performance d'un ramasse-miettes est un critère déterminant. Pour un ramasse-miette *générationel*, il y'a une distinction entre les *jeunes* objets et les *vieux*. En pratique, un jeune objet a une tendance à « mourir » rapidement et donc ces ramasse-miettes le considèrent comme candidat prioritaire au ramassage. Comme ces ramasse-miettes ont moins de mémoire à parcourir, ils sont souvent plus efficaces et sont utilisés dans la machine virtuelle de C# et certaines implémentations de JAVA [Lieberman et Hewitt, 1983]. Pour éviter, des parcours de mémoire trop grands d'un seul coup, les ramasse-miettes incrémentaux ne la *scannent* que partiellement, réduisant de ce fait le temps où l'application est *bloquée*. Mais, au final ces ramasse-miettes mettent plus de temps à parcourir la mémoire et réduisent les performances générales de l'application. Leur utilisation doit donc principalement être limitée aux applications interactives — ou nécessitant une forte réactivité.

3.5 Conclusion

L'héritage multiple, nécessaire dans les langages à typage statique, complique nettement l'approche objet. Néanmoins la méta-modélisation parvient à fournir une sémantique claire aux différents éléments mis en jeu et s'applique aussi bien à l'héritage multiple qu'aux modules et à la relation de raffinement de classes. Elle permet, d'une part, de donner une sémantique au langage PRM et, d'autre part, de définir les éléments qui seront manipulés par le compilateur du langage. Ce langage et surtout son compilateur seront utilisés dans ce qui suit comme source et comme cible de nos travaux.

CHAPITRE

4 Implémentation du sous-typage simple

Dans ce chapitre la problématique de l'implémentation des mécanismes objet est présentée à travers le contexte simplifié du sous-typage simple *. L'implémentation des mécanismes de base de la programmation par objets nécessite une structure de données statique adéquate et un ensemble d'algorithmes permettant d'y accéder. Cette implémentation est caractérisée par des invariants forts et un ensemble de propriétés qui seront intensivement utilisés dans le chapitre suivant. De plus, ce chapitre introduit les principes de l'évaluation abstraite et les applique à cette implémentation simple. Les résultats de cette évaluation fournissent des critères d'efficacités souhaités pour toutes les techniques d'implémentation.*

Sommaire

Nous commencerons par présenter la problématique de l'implémentation des mécanismes objet à travers le contexte simplifié du *sous-typage simple* , où les types sont directement identifiables à des classes et où une classe a au maximum une super-classe directe. Chaque type a ainsi un unique super-type direct.

Cette implémentation serait celle d'un langage dans le genre de JAVA ou de C# sans interface. Il n'existe plus vraiment de langage à typage statique qui soit strictement en sous-typage simple. C'était par exemple le cas d'ADA 95, cependant le sous-typage multiple a été rajouté dans sa version 2005 [Taft *et al.*, 2006].

4.1 Introduction

L'originalité de la programmation par objets réside dans le fait que l'héritage de propriétés locales nécessite une pré-compilation. À chaque classe doit ainsi être associée une structure de données *statique*. L'implémentation des objets consiste à définir la *représentation* des objets — c'est-à-dire les attributs d'un objet —, et la structure de donnée qui est partagée et référencée par toutes les instances d'une même classe et qui sert de support à la liaison tardive. A ces structures, on doit rajouter l'algorithme simple — ou la séquence de code — qui met en œuvre chacun des mécanismes, ainsi que les algorithmes éventuellement complexes, qui servent à calculer ces structures pour chaque classe. Nous ne connaissons pas d'autres paradigme de programmation qui ait des besoins similaires.

4.2 Évaluation a priori

L'implémentation des objets concerne les trois mécanismes objet de base qui s'appuient sur le méta-modèle décrit à la section 3.2 : l'invocation de méthode, le test de sous-typage et l'accès aux attributs. L'implémentation des mécanismes objet est concernée par la recherche efficace des positions de propriétés globales, donc des attributs et des méthodes.

Sauf exception, nous nous intéresserons dans cette thèse seulement aux techniques en temps constant et ne nécessitant aucune recompilation. Pour chaque technique, nous considérerons son évaluation abstraite en espace ainsi que son évaluation théorique en temps.

4.2.1 Pseudo-code de Driesen

Dans ce chapitre et le suivant, nous décrirons les séquences de code dans la syntaxe du pseudo-assembleur de Driesen [2001]. Cet assembleur virtuel est accompagné d'un modèle général d'exécution de processeur.

Dans ce modèle, on considère qu'un branchement direct ne coûte rien et qu'un branchement conditionnel bien prédit ne coûte qu'un cycle. Pour les branchements indirects ou mal prédits, leur coût est nettement plus élevé et nous noterons B la latence du branchement par la suite. Une valeur réaliste pour B est 10.

Les chargements depuis la mémoire ont une latence de L cycles. Grâce à l'architecture en pipeline, deux chargements consécutifs sans relation entre eux peuvent se faire en pa-

rallèle, décalés de un cycle et le total des deux ne prend que $L+1$ cycles — au lieu d'une valeur de $2L$.

Nous ajoutons à ce modèle pour le hachage parfait de classes avec la fonction `mod` (voir Section 5.3), la latence de la division entière — notée D — qui, d'après nos expérimentations sur les processeurs INTEL, aurait un coût de 10 à 25 cycles. Enfin, toutes les autres instructions sont estimées à un coût constant de 1 cycle.

L'évaluation en nombre de cycles peut s'accompagner d'une estimation des risques de défauts de cache, liés au nombre de zones mémoire non reliées qui sont accédés, ainsi que du risque de mauvaise prédiction des sauts conditionnels.

4.2.2 Évaluation spatiale

La consommation mémoire d'une technique est évaluée du point de vue statique (mémoire calculée à la compilation et allouée au chargement du programme) et dynamique (mémoire allouée à l'exécution). L'espace statique est considéré du point de vue du code (longueurs des séquences de code dans le pseudo-assembleur de Driesen) et de la table de méthodes (nombre d'entrées).

4.3 Principe de l'implémentation du sous-typage simple

Dans le cas du sous-typage simple, le graphe de spécialisation d'un programme est une arborescence. Il n'existe donc qu'un chemin reliant une classe (sommet dans le graphe) à la classe racine (Figure 4.1). Cette propriété permet une implémentation intuitive et efficace de l'héritage simple.

Une instance est représentée par un tableau d'attributs précédé d'un pointeur vers la table des méthodes de cette classe — qui est commune à toutes les instances d'une même classe. La table des méthodes — appelée *Virtual Function Table* (VFT) dans le jargon C++— est constituée d'un tableau contenant les adresses des méthodes.

FIGURE 4.1 : Hiérarchie de classes en héritage simple

Une table des méthodes (resp. des attributs) pour une classe donnée est obtenue par

concaténation de la table de sa super-classe directe — unique dans ce contexte — avec l'ensemble des méthodes (resp. attributs) introduits par cette classe (voir Figures 4.1, 4.2).

Cette implémentation est caractérisée par deux invariants majeurs :

Invariant 4.1 (Référence) *Les références à un objet sont indépendantes du type statique de la référence.*

Invariant 4.2 (Position) *Chaque méthode (resp. attribut)* p *a une position dans la table des méthodes (resp. représentation de l'objet) invariante par spécialisation et notée* δ_{p}.

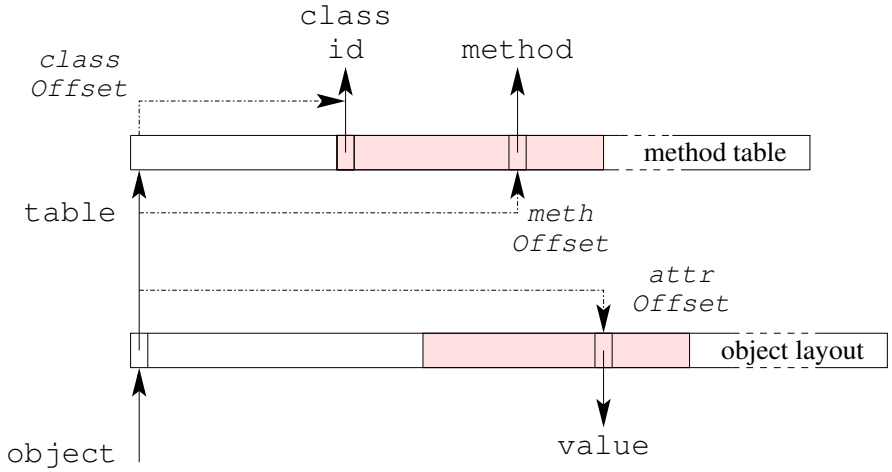


Diagramme d'une instance et sa table de méthodes

FIGURE 4.2 : Implémentation du sous-typage simple

Cette implémentation est donc caractérisée par une invariance totale par rapport aux types statiques durant l'exécution d'un programme. Ils n'ont donc pas d'utilité durant l'exécution du programme et peuvent être effacés à la compilation après avoir été vérifiés.

L'effacement de type renforce la sémantique de la programmation par objets, où l'essence d'un objet est déterminée uniquement par son type dynamique, les types statiques n'étant là que pour aider le programmeur à faire des programmes plus sûrs et pour aider le compilateur à les vérifier et à générer des exécutables plus efficaces.

4.4 Appel de méthode et accès aux attributs

Étant donné l'invariant 4.2, l'accès à un attribut en lecture et en écriture se fait simplement par un accès à un tableau indexé par δ_p.

LISTING 4.1 : Accès en lecture à un attribut

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

LISTING 4.2 : Accès en écriture à un attribut

```
store value, [object + #attributeOffset]                   1
```

L'appel de méthode est réalisé selon le même principe en rajoutant seulement une indirection sur la table des méthodes.

LISTING 4.3 : Appel de méthode

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

Avec `#attributeOffset` et `#methodOffset` étant la position δ_p de la propriété considérée et `#tableOffset` valant généralement 0.

Le passage des paramètres se fait de la même manière que dans les langages non objet, c'est-à-dire qu'en général les paramètres sont empilés avant de réaliser l'appel lui-même. Il existe d'autres conventions pour le passage de paramètres mais non standardisées, par exemple *fastcall* où les premiers paramètres sont passés par des registres — sur x86 avec GCC on utilise `ecx`, `edx`. Même si le passage de paramètres a un réel impact sur l'efficacité du code généré, le receveur est toujours passé comme premier paramètre d'une méthode, il est trop spécifique au couple ⟨architecture, système d'exploitation⟩ et nous n'en parlerons plus durant cette thèse — pas plus qu'il ne sera explicité dans les séquences de code.

4.5 Test de sous-typage

Le test de sous-typage dans le cadre du sous-typage simple est réalisé généralement par une des deux implémentations classiques : le test de Cohen ou la numérotation relative de Schubert.

4.5.1 Test de Cohen

Une première implémentation du test de sous-typage, appelée *test de Cohen [1991]*, a été proposée d'après le *Display de Dijkstra [1960]*. Le principe est d'associer à chaque classe un identifiant et un vecteur contenant les identifiants de toutes les super-classes placés à l'indice de leur profondeur — de la racine (premier indice) à la classe courante. Le test consiste à regarder si le vecteur contient au *bon* indice l'identificateur cherché.

Plus formellement, un objet de type dynamique τ_d est une instance de la classe C si et seulement si le vecteur des identifiants (noté Γ_C) contient l'identifiant de C (noté id_C) à la

LISTING 4.4 : Test de Cohen avec vecteur fusionné sans test de bornes

```
load [object + #tableOffset], table                                    2L + 2
load [table + #classIdPosition], class
comp #classId, class
bne #checkFailed
```

position profondeur de C (noté δ_C) :

$$\tau_d \preceq C \Longleftrightarrow \Gamma_C[\delta_C] = id_C$$

Le vecteur Γ_C peut être fusionné à la table des méthodes en rajoutant l'identifiant id_C à une position déterminée dans la table — par exemple au début des méthodes introduites par C. Dans ce cas, les identifiants id_C ne doivent pas pouvoir être confondus avec des adresses de méthodes, ce qui peut être facilement vérifié en utilisant des nombres impairs — ou du moins non multiples de 4 ou 8 suivant l'alignement puisque les adresses des méthodes sont alignées. De plus un test de borne doit aussi être effectué au préalable. Pour éviter ce test supplémentaire, il faut allouer les tables de méthodes de façon contiguë dans une zone de mémoire dédiée. La mémoire après la dernière table (au moins jusqu'a $\max(\{\delta_C\})$) ne doit pas contenir d'identifiant de classe valide — elle peut donc être remplie de nombres pair — ou des identifiant trivialement valide (par exemple l'identifiant de la racine). Le lecteur intéressé peut se référer à [Ducournau, 2008] pour une discussion détaillée des alternatives d'implémentation du test de Cohen.

Ce test, compatible avec la compilation séparée et le chargement dynamique, est efficace en temps, même s'il n'est pas optimal en espace. La taille de la table Γ_C est la profondeur de la classe C dans l'arborescence de la spécialisation.

Ce test a souvent été utilisé pour l'implémentation de différents langages [Pfister et Templ, 1991; Queinnec, 1998; Alpern *et al.*, 2001b], particulièrement dans des langages avec chargement dynamique.

4.5.2 Numérotation de Schubert

Une autre implémentation simple et commune du test de sous-typage, parfois appelée *numérotation relative*, est due à Schubert *et al.* [1983]. Ce test est basé sur une double numérotation des classes. La première, notée n_1, est une numérotation strictement croissante des classes suivant un parcours en profondeur de l'arbre d'héritage — n_1 forme un pré-ordre. La seconde numérotation est définie par $n_2(C) = max_{D \preceq C}(n_1(D))$. Enfin, le test en lui-même est réalisé par la double comparaison suivante :

$$\tau_d \preceq C \Longleftrightarrow n_1(C) \leq n_1(\tau_d) \leq n_2(C)$$

LISTING 4.5 : Numérotation de Schubert

```
load [object + #tableOffset], table                    2L + 4
load [table + #n1Offset], classId
comp classId, #n1
blt #fail
comp classId, #n2
bgt #fail
; succeed
```

Pour ce test, les deux entiers $n_1(C)$ et $n_2(C)$ doivent être stockés dans la table de C, ce qui, en pratique, est linéaire dans le nombre de classe [1]. Comme les n_1 sont uniques, ils peuvent servir d'identifiant de classe.

Ce test nécessite trois accès mémoire dans le cas général mais si le test est statique — c'est-à-dire que C est connu lors de la compilation — $n_1(C)$ et $n_2(C)$ peuvent être traités comme des valeurs immédiates

Bien que ce test soit en pratique optimal en espace — deux entiers par classe — et relativement efficace en temps, il présente deux inconvénients majeurs. Il ne se généralise pas à l'héritage multiple, tout du moins en maintenant le temps constant. Et comme la numérotation n'est pas incrémentale, elle est peu compatible avec le chargement dynamique.

L'utilisation de ce test dans un contexte de chargement dynamique nécessiterait un recalcul des numérotations au chargement de chaque classe — ce qui n'est pas si grave car l'algorithme est linéraire en temps — mais comme pour le *cast* dynamique, ce test nécessiterait deux accès mémoire supplémentaires, même en cas de *cast* statique. Le recalcul complet des numérotations peut être évité en construisant paresseusement les numérotations [Palacz et Vitek, 2003] ou en réservant certains identifiants. Dans le pire des cas, le recalcul complet ne peut pas être évité. Enfin, comme les valeurs $n_1(C)$ et $n_2(C)$ peuvent être modifiées, il faut que l'intégralité de la table des méthodes soit allouée dans un espace mémoire accessible en écriture.

Pour toutes ces raisons, nous ne considérerons par la suite que le test de Cohen dont nous présenterons plusieurs généralisations en temps contant..

4.6 Propriétés de l'implémentation du sous-typage simple

Le sous-typage simple présente un certain nombre de *bonnes* propriétés. Ces qualités seront intensivement utilisées dans la suite de ce chapitre pour expliquer et justifier les implémentations de l'héritage multiple.

1. En réalité, la complexité nécessite de tenir compte du codage des nombres, le coût de ce test est donc $n\log(n)$. Cependant nous considérons, dans cette thèse, tous les nombres comme ayant une taille fixe, nous omettrons donc le codage des nombre dans tous les calculs de complexité.

4.6.1 Condition du préfixe et groupes de propriétés

Si on *groupe* les méthodes et les attributs par classe d'introduction, cette implémentation du sous-typage simple satisfait à ce que nous nommerons la *condition du préfixe*. Pour $B \prec A$, l'implémentation de A forme un *préfixe* de l'implémentation de B.

Avec l'héritage multiple, aucune implémentation ne peut satisfaire cette condition pour toutes les classes reliées par la relation de spécialisation. Cependant, lorsque cette condition est vérifiée pour un couple de classes, elle donne une bonne opportunité d'optimisation.

Les groupes de propriétés (méthodes ou attributs) seront intensivement utilisés par la suite.

4.6.2 Compatibilité avec l'hypothèse du monde ouvert

Cette implémentation est totalement incrémentale, donc pleinement compatible avec le chargement dynamique et l'hypothèse du monde ouvert. Le calcul des positions peut être retardé au chargement d'une classe. Une classe n'a besoin d'être recompilée que si on modifie son code ou, suivant le mécanisme de résolution des symboles, si on change le modèle d'une de ses super-classes.

4.6.3 Équivalence des mécanismes

L'implémentation du sous-typage simple est remarquable non seulement par sa simplicité et son efficacité, mais également par le fait que tous les mécanismes objet fondamentaux s'intègrent et s'implémentent directement. Bien que les mécanismes fondamentaux ne soient pas équivalents, on peut réduire tous les mécanismes les uns aux autres. Ces réductions permettent de dériver une technique d'implémentation d'un mécanisme à un autre.

Accès aux attributs ⇒ Invocation de méthodes. Les méthodes peuvent être considérées comme des attributs immutables partagés au niveau de la classe et la table des méthodes représente l'instance de cette classe. L'appel en lui même est implémenté par un branchement indirect sur un des champs de cet attribut.

On pourrait aussi, au prix d'une plus forte consommation mémoire, recopier toutes les méthodes directement dans l'instance ; auquel cas l'appel se ferait directement comme un appel sur la valeur de l'attribut.

Invocation de méthodes ⇒ Accès aux attributs. Dans l'autre sens, l'accès à un attribut peut se faire au travers de deux méthodes spéciales appelées *accesseurs* — l'un en lecture et l'autre en écriture. Plus généralement, tous les accès aux attributs peuvent se réduire à un appel de méthode retournant, ou passant en paramètre, la valeur de l'attribut concerné.

Et l'appel complet — c'est-à-dire empilage des paramètres, branchement et retour — peut être évité en simulant les accesseurs (Section 5.5, page 94).

Invocation de méthodes ⇒ Test de sous-typage. Dans un cadre de typage dynamique, le test de sous-typage, « c est-il instance de C » peut être considéré comme l'accès à une méthode `amIaC` définie dans C qui répondrait toujours « vrai ». Un échec à ce test serait matérialisé par l'absence de cette méthode. En typage statique, `amIaC` ne peut être qu'une *pseudo-méthode* car le compilateur refuserait tout appel non trivial à `amIaC`, mais cette pseudo méthode peut être, en général, implémenté comme une méthode.

Test de sous-typage ⇒ Invocation de méthodes. En typage statique, on peut grouper les méthodes par classe d'introduction. La position p d'une méthode dans ce groupe est invariante par spécialisation et $\delta_p = \delta_C + \delta_{p|C}$, avec δ_C la position du groupe de méthode dans la table des méthodes et $\delta_{p|C}$ la position de la propriété dans son groupe de méthode. Pour passer du test de sous-typage à l'appel de méthode, il suffit d'associer aux réussites du test de sous-typage l'ensemble des δ_C. Cependant ceci n'est possible que si le test de sous-typage utilise des entrées sous forme de tables — comme par exemple le test de Cohen. Si ce n'est pas le cas — comme la numérotation relative de Schubert — il faut se ramener à la situation précédente. Par exemple, en ajoutant une indirection par une nouvelle table contenant aux index des indentifiants de classes la position δ_C.

Pour éviter de compliquer la présentation des diverses techniques d'implémentation, nous présenterons principalement le cas de l'appel de méthode. Dans le chapitre suivant, nous nous baserons sur ces équivalences pour généraliser aux autres mécanismes.

4.7 Évaluation de l'efficacité du sous-typage simple

Cette implémentation intuitive du sous-typage simple nous fournit une bonne référence — on ne peut pas en espérer une meilleure implémentation sans considérer des optimisations plus spécifiques. Les diverses techniques d'implémentation présentées par la suite serons comparées à cette implémentation du point de vue spatial et temporel.

4.7.1 Évaluation du temps

Les trois mécanismes de base ainsi que la création des instances sont en temps constant. Comme les mécanismes sont réalisés avec une seule indirection dans une table, leur efficacité est considérée comme optimale — du moins dans le contexte où le polymorphisme est effectivement utilisé.

4.7.2 Évaluation de l'espace

Du point de vue de l'espace dynamique, l'efficacité de cette implémentation est aussi optimale. La représentation de l'objet est limitée aux attributs qu'il possède et seul un pointeur vers la table de méthodes est rajouté.

Les tables de méthodes ne dépendent que du type dynamique d'un objet et leurs tailles sont exactement le nombre de méthodes connues par classe. La taille totale des tables est égale au nombre de paires ⟨classe, méthode⟩ valides, ce qui est la compaction optimale de la grande matrice ⟨classe, méthode⟩ de sélection souvent considérée en compilation globale. Si m_C représente le nombre de méthodes introduites par une classe C, la taille d'une table de méthodes est $M_C = \sum m_C$ et la taille totale des tables est $\sum_C M_C$.

Enfin, le test de Cohen ajoute exactement à la taille des tables la taille de la relation de spécialisation — $|\preceq|$. Si on suppose que m_c ne dépend pas de C (le nombre d'introduction de méthodes est uniforme dans toutes les classes) la taille totale des tables est linéaire par rapport à $|\preceq|$.

Ce critère de linéarité sera essentiel dans nos évaluations de l'héritage multiple

4.7.3 Coût de l'abstraction

La programmation par objet utilise les classes comme abstraction des entités manipulées. Ces classes factorisent les propriétés communes des objets. La conception d'une hiérarchie de classes procède souvent par spécialisation et généralisation. Dans ce dernier cas, la séparation d'un classe en deux augmente la réutilisabilité, la première pouvant maintenant être réutilisée par de nouvelles sous-classes. Dans l'implémentation du sous-typage simple, cette nouvelle abstraction ne coûte rien. Le temps d'invocation des mécanismes est constant, donc indépendant du nombre de super-classes. La taille d'un objet est définie par le nombre d'attributs qu'il possède ce qui ne dépend pas du nombre de classes qui le définissent. Enfin, la taille de la table de méthodes dépend du nombre de méthodes connues par la classe, et pas du nombre de classes qui les introduisent, modulo une entrée supplémentaire pour le test de sous-typage.

L'absence de coût à l'abstraction est donc une propriété souhaitable pour toutes les implémentations — bien que ça ne soit pas toujours le cas — puisqu'elle permet au programmeur d'expliciter les concepts qu'il manipule, de mieux les factoriser, ce qui augmente d'autant la réutilisabilité des classes qu'il crée.

4.8 Conclusions

L'implémentation proposée pour le sous-typage simple n'est pas unique, on peut citer celle de Muthukrishnan et Muller [1996] qui est nettement plus compacte mais ne présente plus de temps constant pour l'invocation des mécanismes. Un certain nombre d'implémentations alternatives du test de sous-typage ont été proposées [Raynaud et Thierry,

2001; Gil et Zibin, 2005; Alavi *et al.*, 2008], ces techniques découlent généralement des deux tests présentés dans ce chapitre (voir Section 4.5.1, 4.5.2).

Toutefois, aucune autre implémentation ne présente autant de qualités, c'est pourquoi nous nous servirons de cette implémentation, simple et efficace, du sous-typage simple comme référence pour la suite. Toutes les techniques présentées doivent ce comparer à cette implémentation de référence, aussi bien pour l'efficacité spatiale que temporelle, et lorsqu'elles différent sur certains points il faut se demander pourquoi. Outre sa simplicité et son efficacité, elle définit un ensemble de propriétés souhaitables pour toutes les autres implémentations ainsi qu'un certain nombre de critères d'évaluation.

4.8.1 Critères d'évaluation

En plus des évaluations abstraites, on peut considérer quatre critères de qualité dérivés de l'implémentation du sous-typage simple pour une technique d'implémentation.

Temps constant. Tous les mécanismes de base dans l'implémentation du sous-typage simple sont en temps constant. Le *temps constant* permet de borner l'évaluation de l'exécution d'un mécanisme dans le pire des cas. Actuellement, la taille des programmes, par conséquent des hiérarchies de classes, augmente sérieusement. Ce critère permet donc d'envisager le passage à l'échelle d'une technique d'implémentation malgré l'augmentation de la taille des programmes.

Espace linéaire (dans la taille de $\preceq$). La taille des structures de l'implémentation du sous-typage simple, nous permet de définir ce critère d'espace raisonnable. La spécialisation nécessitant une forme d'encodage, la taille des structures de l'implémentation du sous-typage simple est linéaire dans la taille de la relation de spécialisation. Cette taille semble incompressible tout au moins en conservant le temps constant.

Comme la taille des hiérarchies de classes augmente, la taille des structures augmente en conséquence. La linéarité des structures de données mises en jeu garantit que la technique d'implémentation passera bien à l'échelle.

Incrémental. Avec l'hypothèse du monde ouvert une nouvelle classe peut être chargée n'importe quand, ce qui implique une modification dynamique de la hiérarchie de classes. Le calcul de l'implémentation du sous-typage simple est incrémental, il n'est donc pas nécessaire de toucher aux structures déjà construites au chargement d'une nouvelle classe.

Cette propriété est souhaitable, car sans elle une technique d'implémentation ne peut être considérée dans le cadre du chargement dynamique qu'avec circonspection — le coût des recalculs que la technique impose au chargement d'une nouvelle classe peut être élevé. Nous verrons, toutefois, que certaines techniques intrinsèquement non incrémentales ont été adaptées au contexte du chargement dynamique.

Mise en ligne. Les séquences de code, pour l'invocation des mécanismes de base, sont courtes dans l'implémentation du sous-typage simple. Comme ces mécanismes vont être appelés des millions de fois par seconde, leur mise en ligne est souhaitable — ils n'en seront que plus efficaces. La mise en ligne est toujours possible, toutefois si les séquences de code ne sont pas courtes elles augmenteront d'autant la taille du code généré. Par conséquent, il est préférable pour une technique d'implémentation que la taille — en nombre d'instructions — des séquences de code soit assez courte pour pouvoir être mise en ligne sans trop pénaliser le programme résultant.

4.8.2 Du sous-typage simple à l'héritage multiple

L'héritage multiple complique considérablement l'implémentation des mécanismes objet, en particulier lorsqu'il est couplé au chargement dynamique. Dans cette configuration, les deux invariants qui caractérisent le sous-typage simple (4.1 et 4.2) ne sont pas maintenables simultanément. En effet, considérons deux classes *B* et *C* sous-classe d'une même classe *A* (Figure 4.3). En utilisant l'implémentation proposée pour le sous-typage simple, les méthodes (resp. attributs) de *B* et de *C* occuperaient les mêmes positions dans la représentation d'une sous-classe commune (*D* dans l'exemple). Cette sous-classe serait impossible à créer car les méthodes introduites par *C* et *B* entreraient en collision.

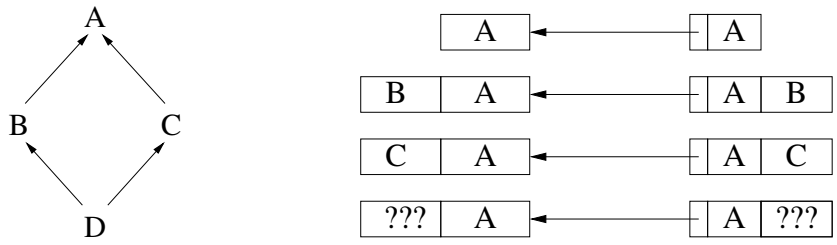


FIGURE 4.3 : Conflit de positions

Le typage dynamique, même sans héritage multiple, rajoute des difficultés du même ordre — une même propriété pouvant être introduite par des classes incomparables. Néanmoins, la problématique de ce nouveau contexte est très différente de celle de l'héritage multiple en typage statique. Une étude de ce contexte représenterait une nouvelle thèse en soi, nous ne le considérerons donc pas plus dans celle-ci.

CHAPITRE

5 Techniques d'implémentation de l'héritage multiple

Ce chapitre présente un état de l'art des techniques d'implémentation en typage statique des mécanismes objet compatibles avec l'héritage multiple, ainsi que les détails de leur mise en œuvre. Les techniques considérées, sauf exceptions, sont en temps constant et ne nécessitent pas de recompilations.

Sommaire

Dans ce chapitre nous considérons les diverses implémentations de l'héritage multiple en typage statique. Nous avons vu dans le chapitre précédent que l'héritage multiple ne permet pas de conserver, à la fois, les deux invariants du sous-typage simple et la compatibilité avec le chargement dynamique. Nous commencerons par examiner les implémentations des langages qui conservent le chargement dynamique mais relaxent certaines contraintes (implémentation par sous-objets de C++), puis nous considérerons celles qui

ajoutent de nouvelles restrictions comme l'hypothèse du monde clos. Enfin nous présenterons des implémentations alternatives qui peuvent s'appliquer au plein héritage multiple comme au cas dégénéré du sous-typage multiple.

5.1 Implémentation par sous-objets

Comme expliqué dans la section 4.8.2, le cas de l'héritage multiple complique considérablement les implémentations. Les deux invariants du sous-typage simple sont trop forts et ne sont pas maintenables simultanément tout en conservant le chargement dynamique. Afin de conserver ce dernier, l'implémentation proposée pour C++ [Ellis et Stroustrup, 1990; Lippman, 1996; Stroustrup, 2000] relaxe les deux invariants du sous-typage simple (invariants 4.1 et 4.2) en se reposant sur le type statique des références. De ce fait, toutes les opérations polymorphes sur un objet nécessitent un ajustement de pointeur. Comme cet ajustement est dépendant du type dynamique de la référence, la table des méthodes doit contenir — en plus des informations habituelles — les décalages à utiliser, faisant ainsi augmenter sa taille.

Des tentatives de formalisation de l'implémentation de C++ ont été faites [Calder et Grunwald, 1994; Rossie et Friedman, 1995].

5.1.1 Tables des méthodes et représentation des objets

Le principe de cette implémentation est de relaxer les deux invariants du sous-typage simple (4.1 et 4.2) en tenant compte du type statique des références. La représentation d'un objet est faite par concaténation des *sous-objets* d'une classe et de toutes ses super-classes — c'est-à-dire pour chaque type statique τ_s qui est un supertype du type dynamique τ_d. La référence à un objet pointe sur le sous-objet correspondant au type statique courant de cette référence et le changement de type statique nécessitent un ajustement de sous-objet (voir section 5.1.2). Un sous-objet pour une classe donnée est un pointeur vers une table de méthodes suivi par les attributs qu'elle *introduit*. Cette table de méthodes contient l'ensemble les méthodes *connues* par le type statique du sous-objet — c'est-à-dire les méthodes introduites comme les méthodes héritées — mais les valeurs (adresses) sont celles des méthodes redéfinies par le type dynamique. Dans le pire des cas, la taille totale des tables de méthode est *cubique* dans le nombre de classes (voir Section 5.11.1).

L'invariant de référence est relaxé de la manière suivante :

Invariant 5.1 (Référence dépendante du type statique) *Chaque référence dont le type statique est T est liée à un sous-objet unique correspondant au type T. Deux sous-objets de type statique différent sont différents.*

Afin de proposer une implémentation efficace en temps, l'invariant de position est relaxé de manière différente pour les attributs et les méthodes [1].

Invariant 5.2 (Position des attributs) *Chaque attribut a un offset (noté δ_a) invariant dans le sous-objet du type statique qui l'a introduit, quel que soit le type dynamique.*

Invariant 5.3 (Position des méthodes) *Étant donné un type statique τ_s et une méthode m connue par celui-ci, la position de m est déterminée par un offset (noté $\delta_m^{\tau_s}$) indépendant du type dynamique du receveur.*

Les positions des méthodes (resp. des attributs) ne sont donc plus absolues mais dépendantes du type statique de la référence (resp. du type d'introduction de l'attribut). Deux instances de type dynamique différent ne partagent aucune table même si elles ont des super-types statiques communs — les tables sont isomorphes mais leur contenu est différent.

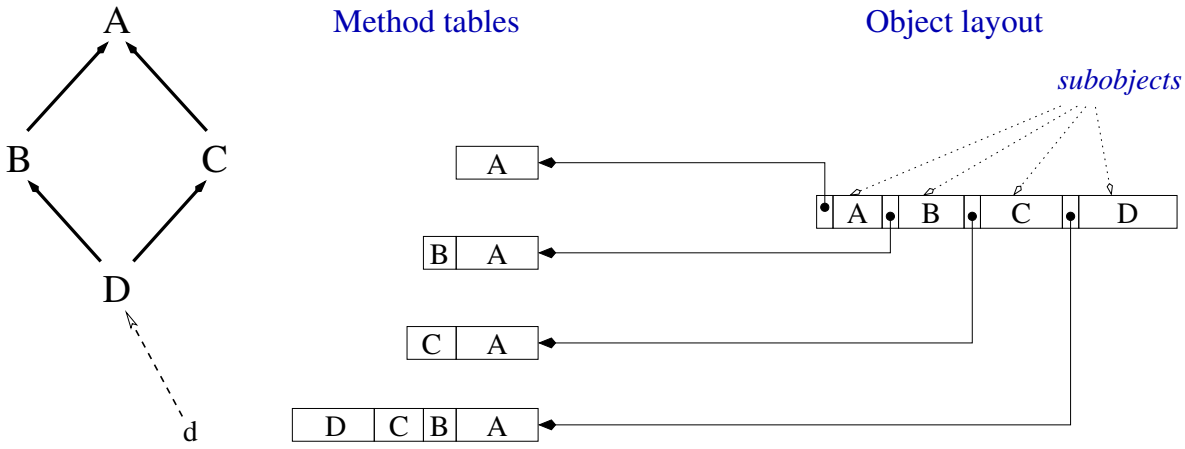


FIGURE 5.1 : Représentation par sous-objets

Pour un type statique donné, l'ordre des méthodes dans sa table n'a aucune importance ni influence sur l'efficacité du code généré. Bien que leur regroupement par classe d'introduction ait un sens, pour simplifier aussi bien le travail du compilateur que le code généré mais aussi pour permettre la condition du préfixe (voir section 4.6.1), condition indispensable à l'optimisation des sous-objets vides (voir section 5.1.4).

Alors que nous défendons la thèse de l'unification des méthodes et des attributs sous le même concept de propriété, cette implémentation fait apparaître une flagrante dissymétrie entre les attributs et les méthodes. Cette différence est due à la volonté de proposer

1. Il est possible de dériver une implémentation plus homogène entre les méthodes et les attributs mais cette implémentation nécessite une indirection supplémentaire pour l'appel de méthode (voir Section 5.1.5).

une implémentation efficace en temps et s'explique par le fait que les méthodes sont des attributs immutables. Elles peuvent donc être copiées sans changer la sémantique du programme, ce qui n'est pas le cas pour les attributs.

5.1.2 Ajustements de pointeurs

Cette implémentation repose sur le type statique des receveurs, un changement de type statique implique un ajustement de pointeurs. Ces ajustements ont lieu, lors d'un appel de méthode, de l'accès à un attribut, d'un *cast* ou d'un test d'égalité.

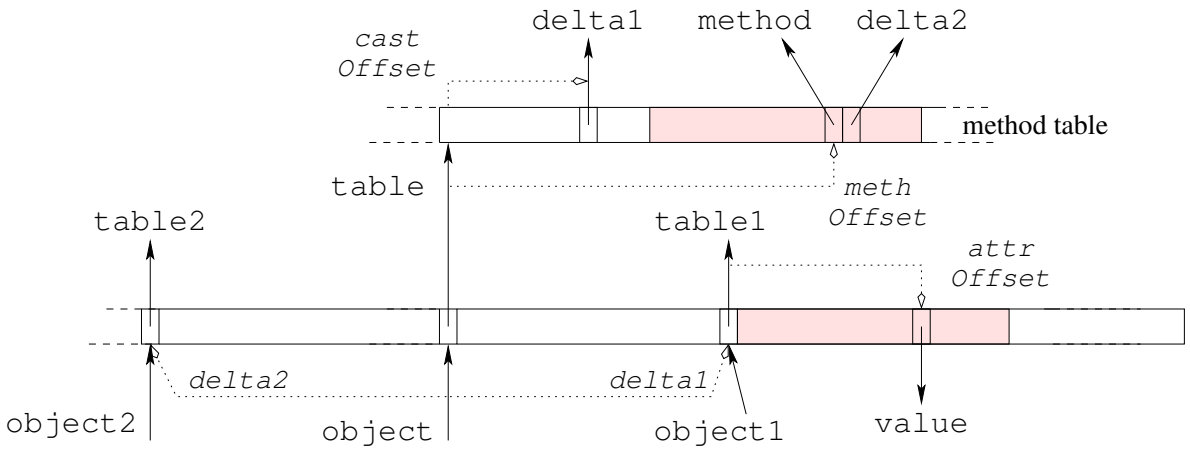
Appel de méthode

Lors d'un appel de méthode, à cause de l'invariant 5.1, le type statique de la référence du receveur (`self`) de l'appelant doit être ajusté au type statique de l'appelé car le receveur dans une méthode est statiquement typé par sa classe de définition. Le type statique du receveur dans l'appelé est déterminé par la classe W ayant défini la méthode appelée. L'appel de méthode doit donc connaître la position relative entre le sous-objet W et le sous-objet actuellement pointé par τ_s, nous appellerons cette position un *décalage*. Ce décalage étant propre à chaque classe, la notation (ainsi que la valeur du décalage) est dépendante du type dynamique. Nous noterons $\Delta^{\tau_d}_{U,V}$ le décalage de U à V dans le contexte du type τ_d avec $\tau_d \preceq U, V$. Pour prendre en compte ce décalage au moment de l'appel il existe deux techniques.

La première consiste à utiliser des entrées doubles dans les tables de méthodes, la première partie de l'entrée contenant l'adresse de la méthode et la seconde le décalage. La séquence de code nécessaire pour réaliser l'appel est légèrement plus longue que celle utilisée pour l'appel du sous-typage simple mais bien que la séquence contienne un `load` supplémentaire, un de ces `load` peut être parallélisé et donc le nombre de cycles n'est augmenté que d'un.

La seconde technique pour tenir compte des décalages lors de l'envoi de message est d'appeler des petits bouts de code intermédiaires, nommés *thunks*[2], pour réaliser le décalage suivi de l'appel réel. Cette alternative présente plusieurs avantages par rapport à la première implémentation proposée. La séquence de code de l'appel lui même est identique à celle du sous-typage simple. Le décalage utilisé par le *thunk* (`#delta` ou $\Delta^{\tau_d}_{U,V}$) est une valeur immédiate, et lorsqu'elle est nulle — c'est-à-dire que la méthode a été définie par le type statique courant — le *thunk* peut être court-circuité en appelant directement la méthode. Par contre un *thunk* différent doit être généré pour chaque $\tau_d \prec \tau_s$

2. Ces bouts de code intermédiaire portent plusieurs noms dans la littératures : *thunks* [Ellis et Stroustrup, 1990], *trampolines* [Myers, 1995] ou plus généralement *stubs*. Nous ne conserverons que le premier terme dans le reste de ce document.




```
; Method invocation using shifts in table                    2L + B +2
load [object + #tableOffset],  table
load [table + #methodOffset + 1], delta
load [table + #methodOffset], method
add object, delta, object ;
call method
```

FIGURE 5.2 : Représentation par sous-objets avec décalage dans la table

LISTING 5.1 : Thunk pour l'appel de méthode par sous-objets

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

Propriétés des décalages

Les décalages représentent les positions relatives d'un sous-objet par rapport à un autre. Ce sont des bi-points dans un espace unidimensionnel, ils obéissent aux propriétés basiques des vecteurs.

$$\forall \tau_d, U: \quad \tau_d \preceq U \Rightarrow \Delta^{\tau_d}_{U,U} = 0 \tag{5.1}$$

$$\forall \tau_d, T, U, V: \quad \tau_d \preceq T, U, V \Rightarrow \Delta^{\tau_d}_{T,V} = \Delta^{\tau_d}_{T,U} + \Delta^{\tau_d}_{U,V} \tag{5.2}$$

$$\forall \tau_d, U, V: \quad \tau_d \preceq U, V \Rightarrow \Delta^{\tau_d}_{V,U} = -\Delta^{\tau_d}_{U,V} \tag{5.3}$$

Changement de type statique

Le même genre d'ajustement de pointeur doit être réalisé dans tous les cas de changement de type statique, même dans le cas normalement trivial du `cast` ascendant vers un

type statiquement connu et donc pour chaque affectation ou passage de paramètre polymorphe.

Cast ascendant. A cause de cela, la valeur des $\Delta_{U,V}^{\tau_d}$ doit être encodée dans la structure de chaque sous-objet, par exemple dans une table $\Delta_{\tau_s}^{\uparrow\tau_d}$. Les index des éléments de cette table sont régis par le même genre d'invariant que celui des méthodes (Invariant 5.3).

Invariant 5.4 (Positions des valeurs d'upcast) *Chaque classe T a une position dans le contexte statique de chacune de ses sous-classes τ_s et invariante pour tout type dynamique τ_d tel que $\tau_d \preceq \tau_s \prec T$.*

Cette position notée $\sigma_{\tau_s}(T)$, avec $\tau_s \preceq T$ est indépendante du type dynamique du receveur et vérifie : $\Delta_{U,V}^{\tau_d} = \Delta_{\tau_s}^{\uparrow\tau_d}[\sigma_{\tau_s}(T)]$. Cette table peut être fusionnée à la table des méthodes de chaque sous-objet permettant ainsi de régir les décalages en substituant l'invariant 5.4 par l'invariant 5.3. De plus cette fusion permet d'éviter un `load` supplémentaire qui serait autrement requis pour accéder à la table. En fait, ces *upcast* peuvent être considérés comme si chaque classe introduisait une méthode retournant l'instance courante statiquement typée par elle-même.

Test d'égalité

Dans toutes les implémentations possédant l'invariant 4.1 de référence, l'égalité physique de deux références se limite à comparer la valeur des deux pointeurs — donc un simple `cmp` suffit. Dans l'implémentation par sous-objets, à cause de l'invariant 5.1, seules deux références de même type statique sont directement comparables. Dans le cas contraire, c'est-à-dire si les deux références sont de types statiques différents, il faut ramener les deux références à un type commun, donc procéder à un ajustement de pointeur. Soit a et b deux instances respectivement typées par A et B, il faut distinguer plusieurs cas :

Si $A \preceq B$ (resp. $B \preceq A$), il faut ramener a au type statique de b (resp. b vers a) avant de pouvoir les comparer. Sinon, c'est que a et b sont incomparables, mais elles peuvent disposer d'un type en commun. Lorsque ce type commun existe, chacune des instances doit y être ramenée, deux *upcast* seront nécessaires avant de pouvoir les comparer. Dans les cas des hiérarchies enracinées, la classe racine est un type commun. Dans les hiérarchies sans racine, comme en C++, on doit ajouter dans la table des méthodes de chaque sous-objet un décalage $\Delta_{\tau_s,\tau_d}^{\tau_d}$ — noté $\Delta_{\Downarrow}^{\tau_s}$ avec $\Delta_{\Downarrow}^{\tau_d} = 0$ — permettant de se ramener au type dynamique de l'instance. Une fois les instances ramenées à leur type dynamique, elles sont comparables. Toutefois, ce décalage $\Delta_{\Downarrow}^{\tau_s}$ doit avoir une position invariante dans toutes les tables de méthodes.

LISTING 5.2 : Accès à un attribut avec un type statique différent de l'introduction

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

Accès aux attributs

D'après l'invariant 5.2, l'accès à un attribut n'est possible que dans le contexte du type statique l'ayant introduit. Dans tous les autres cas, accéder à un attribut requiert de se ramener au type statique de l'introduction et un *upcast* implicite est donc nécessaire. Soit un attribut a introduit par une classe U et τ_s un type statique tel que $\tau_s \preceq U$, la position de cet attribut par rapport a un type statique τ_s notée $\delta(a,\tau_s)$. De plus, on peut poser l'égalité suivante :

$$\delta(a,\tau_s) = \Delta^{\tau_d}_{\tau_s,U} + \delta(a,U) = \Delta^{\uparrow\tau_d}_{\tau_s}(U) + \delta_a$$

Si τ_s et U sont confondus, l'accès à un attribut se fait comme en sous-typage simple, dans le cas contraire la séquence de code est considérablement plus longue.

Le test $a.x = b.y$ exacerbe encore la situation, puisque si les types statiques de a et b sont différents du type d'introduction de x et y, avec x différent de y, quatre *upcast* implicites sont nécessaires — 2 *upcasts* si le type de x et y sont incomparables, 1 *upcast* si le type de a (resp. b) n'est pas la classe d'introduction de x (resp. y). La séquence de code devient donc cinq fois plus longue que celle utilisée pour du sous-typage simple— $5L+3$ contre $L+1$ et 11 instructions contre 2.

5.1.3 Test de sous-typage

Malgré les réductions de mécanismes proposés en section 4.6.3, l'implémentation par sous-objets présente une étonnante singularité : il est impossible de dériver un test de sous-typage de l'appel de méthode. Ceci est dû au fait que, pour maintenir l'invariant de position, l'implémentation par sous-objets utilise pour chaque type dynamique, une table de méthode différente par type statique. La pseudo-méthode `amIaC` devrait avoir une position unique dans toutes les tables des sous-classes de C, ce qui est incompatible avec l'héritage multiple sans l'hypothèse du monde clos.

Cast descendant. Le *downcast* dans les implémentations sans l'invariant de référence (Invariant 4.1) doit être réalisé en deux temps. Premièrement le test de sous-typage lui même « x est il une instance de C ? » En cas de réussite, pour respecter l'invariant de référence dépendante du type statique (invariant 5.1), le test doit être suivi d'un ajustement de pointeur $\Delta^{\tau_d}_{\tau_s,C}$.

Sans perte de généralité, chaque type dynamique τ_d doit avoir une structure d'association qui apparie à chaque super-classe T de τ_d le décalage $\Delta^{\tau_d}_{\tau_d,T}$, cette structure sera notée $\Delta^{\Uparrow}$. Des implémentations pour cette structure seront proposées dans la section 5.1.3. Cette structure peut être référencée depuis la table de méthodes de chaque type statique ou propre à chacun des types statiques, évitant ainsi une indirection au prix d'une plus grande consommation mémoire. Comme cette alternative paraît peu raisonnable, nous ne considérerons dans la suite que la version où cette table est partagée. Étant donné l'invariant 5.1, les décalages ne peuvent pas être les mêmes quel que soit le type statique de la référence, il faut donc se ramener à un type commun. Dans les langages où il existe une classe racine, elle peut faire office de type commun et on a :

$$\Delta^{\tau_d}_{\tau_s,T} = \Delta^{\tau_d}_{\tau_s,Any} + \Delta^{\tau_d}_{Any,T}$$

Dans les langages sans racine, comme C++, le seul type commun à chaque type dynamique est le type dynamique lui même. La table $\Delta_{\Downarrow}$ doit donc contenir les décalages du type dynamique à chacun des types statiques. On a donc :

$$\Delta^{\tau_d}_{\tau_s,T} = \Delta^{\tau_d}_{\tau_s,\tau_d} + \Delta^{\tau_d}_{td,T} = \Delta^{\tau_s}_{\Downarrow} + \Delta^{\Uparrow}(T)$$

Les tables $\Delta^{\uparrow\tau_d}$ et $\Delta^{\Uparrow}$ ont le même contenu mais des structures (et des utilisations) différentes. La première est utilisée quand $\tau_s = \tau_d$ et que τ_s est connu statiquement. Tandis que la seconde est utilisée quand τ_d est statiquement inconnu.

L'utilisation d'une matrice directe ⟨classe, classe⟩ pour la table $\Delta^{\Uparrow}$ est possible, cependant sa taille serait déraisonnable ($N \times N$ avec N le nombre de classes). Avec une numérotation des classes dans l'ordre de la spécialisation — les super-classes avec les sous-classes —, cette matrice carrée peut être transformée en une matrice triangulaire — qui peut s'implémenter par un vecteur par classe — mais dont la taille demeure déraisonnable. Nous proposerons dans la section 5.3.4 une adaptation du hachage parfait de classes pour représenter cette structure $\Delta_{\Downarrow}$.

5.1.4 Optimisation des sous-objets vides (ESO)

Le surcoût de l'implémentation par sous-objets sur l'implémentation sous-typage simple est important, en temps comme en espace. L'optimisation des sous-objets vides (ESO) permet de limiter ce surcoût quand une classe n'introduit pas d'attributs, donc que son sous-objet est vide [Ducournau, 2011].

Le principe de cette optimisation est de rajouter une exception à l'*invariant de référence dépendante du type statique* (invariant 5.1) lorsqu'une classe n'introduit pas d'attribut et que le sous-objet peut être fusionné avec celui d'une super-classe ou d'une sous-classe. Soit F une sous-classe de E qui n'introduit pas d'attribut, le sous-objet de F peut être fusionné au sous-objet de E — cette fusion est appelé *fusion de bas en haut*. La statistiques

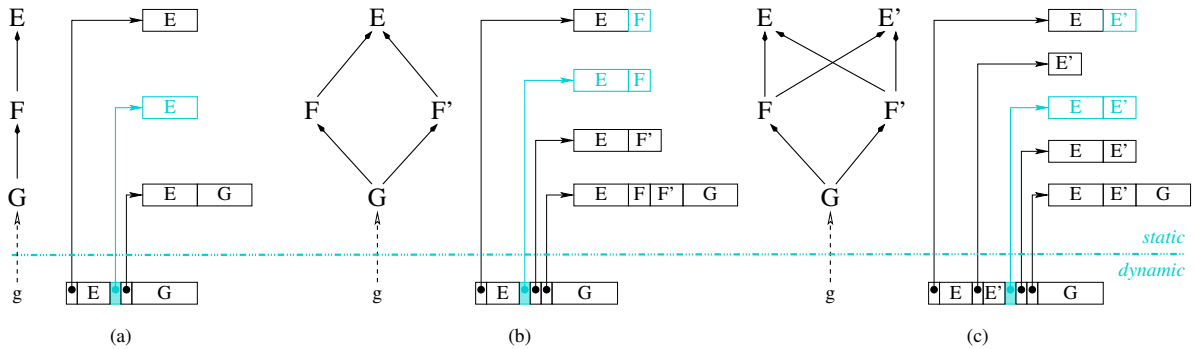


FIGURE 5.3 : Trois différents cas de sous-objets vides

données par [Ducournau, 2011] montrent que ce cas n'est pas rare dans la plupart des *benchmarks*. Nous examinerons à la fin de cette section le cas de la *fusion de haut en bas*.

Cette optimisation nécessite de distinguer plusieurs cas (Figure 5.3). Si E est la seule super-classe de F (Figure 5.3.a) et F n'introduit pas non plus de méthodes, le sous-objet de F peut être véritablement fusionné au sous-objet de E — le contenu des tables est identique (mêmes adresses de méthodes). Dans ce cas, la fusion est statique et invariante par spécialisation, c'est-à-dire E et F sont fusionnés dans toutes les sous-classes de F. Les problèmes liés à l'héritage multiple sont évités, car F n'a pas plus de méthodes que E et E est sa seule super-classe — si une autre sous-classe de E, par exemple F', est dans la même situation, les trois sous-objets E, F et F' peuvent être fusionnés et ce dans toutes les sous-classes de F et F'. Comme les sous-objets sont totalement fusionnés, il n'y a plus besoin de décalage entre E et F (car $\Delta^{\tau_d}_{E,F} = 0$ et $\sigma_{\tau_s}(E) = \sigma_{\tau_s}(F)$, pour tout $\tau_d \preceq \tau_s \preceq F \prec_{\mathrm{d}} E$) et ce dans toutes les sous-classes. De plus, le code généré pour tout les $\tau_s \preceq F$ peut tenir compte de cette fusion. Il n'y a plus besoin d'ajustements de pointeurs et l'accès à un attribut introduit par E peut se faire depuis un receveur typé par F sans *upcast*.

Un autre cas est à considérer. Soit la classe F a plus de méthodes que E (Figure 5.3.b), soit elle a plus d'une super-classe directe (Figure 5.3.c). Le dernier cas est un cas particulier du précédent, car si F a plus d'une super-classe, elle possède au moins les pseudo-méthodes liées aux *upcasts*. Si l'ordre des méthodes de E est un préfixe de l'ordre des méthodes de F (voir Section 4.6.1) — c'est-à-dire si les positions des méthodes de E sont les mêmes dans E et F —, les sous-objets de E et F peuvent être fusionnés dans l'implémentation d'une classe $G \preceq F$. Cependant, le sous-objet de E ne doit pas déjà faire l'objet d'une fusion dans une autre classe F' avec $G \prec F'$ — en d'autre termes, le sous-objet E ne peut être fusionné qu'une seule fois par classe. Cette optimisation est possible grâce à la condition du préfixe qui est invariante vis-à-vis du type dynamique. Cependant, comme E n'est pas fusionné dans toutes les sous-classes de F, la fusion permet d'économiser de l'espace — un pointeur dans l'instance et une table de méthode — mais elle ne permet plus d'économiser du temps. En effet, la génération de code ne peut pas considérer que pour $\tau_s \prec F$

les sous-objets sont fusionnés, donc les ajustements de pointeurs et les *upcasts* pour accéder à un attribut sont toujours requis — bien que leurs valeurs soit 0. Si l'implémentation utilise des *thunks*, il est quand même possible d'en économiser quelques-uns (le décalage entre E et F étant nul).

La *fusion de haut en bas* peut aussi être considérée. C'est la seule façon de fusionner la racine avec une sous-classe non vide mais c'est aussi une alternative quand la fusion vers le haut est impossible. Dans la figure 5.3.b, par exemple, le sous-objet E peut être fusionné dans F et le sous-objet F' peut être fusionné dans G. Cette fusion vers le bas ne peut jamais être statique et ne permet donc pas une économie de temps.

Enfin, les fusions en chaînes sont possibles mais doivent être examinées prudemment, la condition du préfixe doit toujours être vérifiée. Par exemple, dans la figure 5.3, la table de F peut être statiquement fusionnée à celle de E et cette table peut être à son tour fusionnée dynamiquement à celle de G. Avec l'héritage multiple, la condition du préfixe est toujours partiellement vérifiable. L'ordre des méthodes d'une classe étant arbitraire, il est toujours possible de garantir la condition du préfixe sur une de ses super-classe directes — le choix de la super-classe qui sera préfixe est une question d'optimisation, il vaut mieux fusionner les tables les plus grandes ou les tables qui seront utilisées par le plus de sous-classes.

Finalement une combinaison de toutes ces fusions est possible et devient un problème de couplage. La préférence doit être donnée aux fusions statiques, puisqu'elle permet d'économiser du temps en plus de l'espace.

Remarque. En PRM à cause du raffinement de classes, les fusions statiques sont impossibles — car de nouvelles méthodes peuvent être ajoutées dans un autre module. Nous utilisons donc un algorithme simple ne considérant que les cas de fusions dynamiques et où les fusions de haut en bas sont prioritaires sur les fusions de bas en haut — ceci à cause de la racine qui peut être préfixe de toutes les classes.

Cette optimisation simple permet de réduire notablement le surcoût de l'implémentation par sous-objets, au niveau des tables des méthodes, au niveau des instances et au niveau des ajustements de pointeurs. Cette optimisation semble être connue de certains auteurs, d'après les allusions qu'ils y font [Lippman, 1996], mais d'après nos tests, elle ne semble pas être implémentée par les compilateurs C++— de GCC2.2 à 4.3 et SUN5.3 C++. GCC 4.4 semble faire une optimisation mais elle n'est pas systématique car le compilateur ne fait aucun effort pour maintenir la condition du préfixe.

5.1.5 Variation autour de l'implémentation par sous-objets

Il existe de nombreuses variations autour de cette implémentation venant gommer un défaut de l'implémentation au détriment d'une autre caractéristique. Pour les détails des variantes de l'implémentation par sous-objets, le lecteur intéressé est renvoyé à [Ducournau, 2011]

Héritage *virtuel* et *non virtuel*

L'implémentation proposée depuis le début est celle de C++ en supposant que le mot-clef `virtual` annote chaque super-classe. L'absence de `virtual` dans les clauses d'héritage se traduit par une implémentation simplifiée qui entraîne un risque d'héritage répété. Cette implémentation est identique à celle du sous-typage simple— un seul sous-objet — et n'entraîne pas les surcoûts liés à l'héritage multiple et au chargement dynamique. Cependant, c'est au prix d'une sémantique et d'une réutilisabilité limitée — héritage répété en cas de losange. La majorité des programmes C++ utilisent peu le mot-clef *virtual*, ils ne souffrent donc pas vraiment du surcoût entraîné par l'héritage multiple, ce qui fait de C++ un langage efficace.

Moins d'indirections pour les attributs (VBPTR)

Pour éviter un *upcast* lors de l'accès à un attribut, une solution consiste à lier des pointeurs entre les sous-objets directement dans l'objet, ces pointeurs sont appelées *VBPTR* (*Virtual Base Pointer*). Il permettent de réduire le coût d'un accès à $2L$ — au lieu de $3L+1$ — mais au prix d'une augmentation de la mémoire dynamique utilisée. Les statistiques montrent que les besoins mémoire peuvent plus que doubler [Ducournau, 2011]. Il existe des variantes à cette implémentation « standard » des VBPTR, réduisant plus ou moins le nombre de pointeurs nécessaires.

Tables de méthodes compactes

De la même manière que les attributs sont groupés par classe d'introduction, on peut limiter le sous-objet aux méthodes introduites par la classe. A l'instar des attributs, il devient nécessaire d'ajouter un ajustement de pointeur avant l'appel lui même. Ceci rajoute exactement au coût d'un appel celui de l'ajustement — 8 instructions et $4L+B+2$ cycles. Ce coût peut être baissé en liant les tables de méthodes les unes aux autres.

5.2 Coloration

La coloration est une technique d'implémentation par table conservant les invariants du sous-typage simple (Invariants 4.1 et 4.2) dans le cas de l'héritage multiple mais en supposant l'hypothèse du monde clos. Elle permet d'obtenir la même efficacité que le sous-typage simple au prix de *trous* — entrées inutiles — dans les diverses tables.

Un moyen trivial mais inefficace de maintenir les invariants du sous-typage simple en héritage multiple est de donner un identifiant unique à chaque classe et à chaque propriété. Les tables ainsi obtenues n'ont aucune collision mais leur taille est déraisonnable — elles sont le produit du nombre de classes par le nombre de méthodes — et la grande

majorité des entrées sont inutilisées. Cependant, il est possible de réutiliser tout ou partie de ces trous (voir Section 5.11.3).

La coloration est un moyen de minimiser l'espace inutile dans ces tables en améliorant la numérotation des propriétés et en considérant la hiérarchie de classes *in extenso* — sous l'hypothèse du monde clos. La coloration tient son nom du problème de la coloration de graphe [Garey et Johnson, 1979]. Et elle peut être utilisée pour chacun des trois mécanismes que sont l'accès aux attributs (coloration d'attributs), l'invocation de méthodes (coloration de méthodes) et le test de sous-typage (coloration de classes).

Cette technique a été découverte indépendamment dans le cas particulier de chaque mécanisme. Elle a été initialement proposée par Dixon *et al.* [1989a] pour l'invocation de méthodes (sous le nom de « selector coloring »), par Pugh et Weddell [1990]; Ducournau [1991] pour l'accès aux attributs et par Vitek *et al.* [1997] pour le test de sous-typage (sous le nom de « pack-encoding »). Enfin, Ducournau [2010] en fait une synthèse et montre l'identité de toutes ces techniques — en se servant principalement des réductions de mécanismes présentées en section 4.6.3.

5.2.1 Principe

La coloration consiste à associer une *couleur (indice)* unique à une propriété globale et non ambiguë dans toutes les classes possédant cette propriété.

L'invariant 4.2 du sous-typage simple peut s'exprimer ainsi dans le cas de la coloration :

Invariant 5.5 *Chaque propriété globale a une* position (couleur) *unique invariante par spécialisation. Deux propriétés globales de même position (couleur) n'appartiennent pas à la même classe.*

En corollaire direct à cet invariant, deux classes ayant des propriétés globales de même couleur ne peuvent pas avoir de sous-classe commune. La généralisation du test de [Cohen, 1991] (voir Section 4.5.1) se fait sur le même principe.

Invariant 5.6 *Chaque classe a une couleur unique. Deux classes de même couleur n'ont pas de sous-classe commune.*

Maintenir ces invariants est simple grâce à l'hypothèse du monde clos mais minimiser le nombre de couleurs [Dixon *et al.*, 1989b; André et Royer, 1992] est équivalent au problème NP-difficile de la coloration minimale de graphe [Garey et Johnson, 1979]. Au lieu de chercher à minimiser la taille maximale des tables — couleur maximale — une amélioration évidente consiste à minimiser la taille totale des tables [Pugh et Weddell, 1990; Ducournau, 1991; Pugh et Weddell, 1993; Takhedmit, 2003]. Malheureusement ce problème d'optimisation n'étant pas plus simple que le précédent, des heuristiques sont nécessaires pour pouvoir l'utiliser en pratique. Ces heuristiques semblent donner des résultats satisfaisants même sur des grosses hiérarchies de classes [Pugh et Weddell, 1990; Takhedmit,

LISTING 5.3 : Appel de méthode

```
load [object + #tableOffset], table                    2L + B
load [table + #propertyColor], methode
call methode
```

LISTING 5.4 : Accès à un attribut

```
load [object + #propertyColor], value                  lecture : L
store value, [object + #propertyColor]                 écriture : 1
```

2003; Ducournau, 2010] et dans le cas des hiérarchies arborescentes — c'est-à-dire n'utilisant pas d'héritage multiple — ces heuristiques permettent de retrouver exactement l'implémentation du sous-typage simple sans trou.

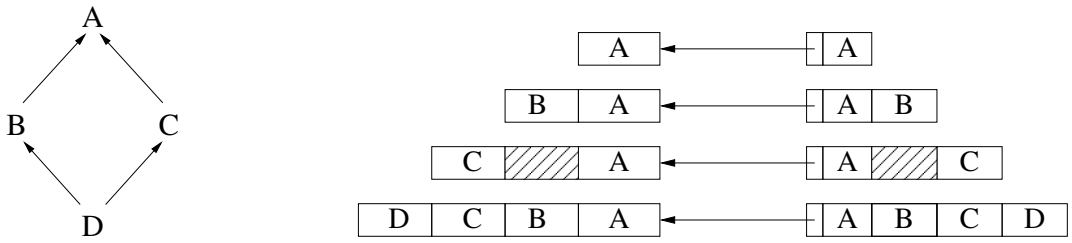


FIGURE 5.4 : Coloration en héritage multiple

Outre l'hypothèse du monde clos, la coloration entraîne l'existence de trous dans les diverses tables. Les trous ne représentent pas un réel problème dans les tables de méthodes, sauf éventuellement pour les systèmes embarqués [Sallenave et Ducournau, 2010]. Mais ces trous peuvent s'avérer désastreux au niveau des instances, ils pourraient en faire grossir certaines jusqu'à 800% [Ducournau, 2010]. Pour remédier au problème des *trous* dans les instances, une implémentation alternative moins compacte statiquement et moins efficace en temps, mais sans le risque de trou, est proposée dans la section 5.5.

Une alternative serait de *profiler* les programmes pour avoir des statiques sur l'instanciation des classes que les heuristiques pourraient utiliser.

Comme la coloration préserve les invariants du sous-typage simple (Invariants 4.1 et 4.2), les séquences de code pour tous les mécanismes sont identiques à celles présentées pour le SST.

5.2.2 Algorithme et heuristiques

La coloration minimale de graphe, en toute généralité, est un problème NP-Complet. Par conséquent, des heuristiques sont nécessaires. En héritage simple ce problème est

LISTING 5.5 : Test de sous-typage

```
load [object + #tableOffset], table                    2L + 2
load [table + #classColor], class
comp #classId, class
bne #checkFailed
```

simple, c'est donc dans la partie en héritage multiple — appelée cœur — qu'il est complexe.

Pour présenter l'algorithme, quelques définitions préalables sont nécessaires. Pour alléger les définitions à venir, nous considérons que la hiérarchie de classe $H = (X, \prec_{\mathrm{d}})$ est restreinte à sa réduction transitive — les arcs de transitivité ont été enlevés.

Définition 5.7 (Cœur de la hiérarchie) *Le cœur de la hiérarchie $H = (X, \prec)$ est le sous-graphe $Core(H)$, constitué par sa restriction aux classes qui ont plus d'une super-classe directe, et à leurs super-classes indirectes : $Core(H) = (X', E')$.*

$$X' = \{x \mid \exists y, z, z' : y \preceq x \wedge (y, z), (y, z') \in E \wedge z \neq z'\}$$

$$E' = \{(x, y) \in E \mid (x, y) \in X' \times X'\}$$

Définition 5.8 (Couronne de la hiérarchie) *La couronne de la hiérarchie $H = (X, E)$ est le sous-graphe $Crown(H) = (X'', E'')$, constitué par sa restriction aux classes qui sont en héritage simple (une seule super-classe directe) et dont les sous-classes sont en héritage simple :*

$$X'' = \{x \mid \exists y, z, z' : y \preceq x, (y, z) \in E \wedge (y, z') \in E \Rightarrow z = z'\}$$

Définition 5.9 (Frontière de la hiérarchie) *La frontière est constituée par les classes minimales du cœur, $Border(H)$:*

$$X''' = \{x \in X' \mid y \prec x \Rightarrow y \in X''\}$$

.

On peut remarquer que le cœur et la couronne forment une partition de l'ensemble des classes ($X = X' \uplus X''$).

Définition 5.10 (Graphe de conflit) *Le graphe de conflit de la coloration de classes de la hiérarchie $H = (X, E)$ est le graphe $CG(H) = (X, \hat{E})$ avec $\hat{E}$ la relation d'incomparabilité de la spécialisation, restreinte aux couples de classes ayant une sous-classe commune :*

$$(x, y) \in \hat{E} \Leftrightarrow x \not\preceq y \wedge y \not\preceq x \wedge \exists z \mid z \prec x \wedge z \prec y$$

La coloration de classes est exactement le problème de coloration du graphe constitué par l'union du graphe de conflit et du graphe de comparabilité de la relation de spécialisation. Néanmoins les critères de minimisation ne sont pas forcément ceux de la coloration de graphe classique — on cherchera plutôt à minimiser le nombre de trous.

En pratique, on peut se restreindre à ne construire que le graphe de conflit du cœur de la hiérarchie — les sommets de X' et X'' sont des sommets isolés dans le graphe de conflit. L'algorithme de la coloration traite séparément les cas du cœur et de la couronne. Une fois le graphe de conflit calculé, la frontière est gérée comme la frontière. La structure générale de l'algorithme est la suivante :

- Calcul des ensembles $Core(H), Crow(H), Border(H)$.
- Calcul du graphe de conflit.
- Coloration du cœur sans la couronne.
- Coloration de la frontière et de la couronne.

Coloration du cœur

La coloration du cœur est la partie délicate de l'algorithme, le principe va être de maintenir deux listes — ordonnées — de classes maximales, à faire un choix parmi ces maximaux et lui affecter une couleur. L'ordre dans les listes et le choix de la couleur étant paramétré par des critères heuristiques.

Une étiquette décrit le statut d'une classe, cette étiquette peut prendre les valeurs suivantes :

- les classes ayant un conflit coloré (`max-cf` et `notmax-cf`).
- les classes n'ayant aucun conflit colorés (`max-ncf` et `notmax-ncf`).
- les classes déjà colorées (`colored`).

Les classes étiquetées par `max-cf` et `max-ncf` correspondent aux éléments présents dans les liste de maximaux du même nom. La distinction entre `cf` et `ncf` est nécessaire pour que la coloration bidirectionnelle soit parfaite quand le graphe de conflit est biparti. Elle est donc inutile si une bipartition est réalisée au préalable.

Au démarrage de l'algorithme 1, toutes les classes sont étiquetées par `notmax-cf`, à l'exception de la racine — si elle est unique, sinon des racines — qui appartient à `max-ncf`.

La table de couleur d'une classe est supposée de taille illimitée et est constituée de couleurs libres, de couleurs occupées et de couleurs *gelées*. Une couleur est dite *gelée* si cette couleur est utilisée par une des classes avec laquelle cette classe est en conflit.

Ce pseudo algorithme se paramètre par trois heuristiques :

- La couleur à affecter à une classe (`choose-color`).
- La classe maximale à choisir (`choose-max-class`).
- Le score de la classe (`compute-score`).

Le lecteur intéressé est renvoyé à [Ducournau, 2010] pour une présentation et une évaluation des heuristiques sur des hiérarchies de classes existantes.

Algorithm 1: Schéma d'un algorithme de coloration de classes

Data: Une copie de la hiérarchie de classe $H = (X', \prec_d)$ restreinte au cœur
Result: Chaque classe a une couleur

```
procedure colorize-core (X', ≺_d)
begin
    Ê ← conflict_graph (X')
    repeat
        list ← Première liste non vide parmi : max-cf-list, max-ncf-list
        c ◁ choose-max-class(list)
        color ← choose-color(c)
        set-color (c) ← color
        tag (c) ← colored
        foreach subclass ← (c, subclass) ∈≺_d do
            Copier ctable → color-table (subclass)
            inherit-color (subclass, c)
            ≺_d ←≺_d \{(c, subclass)}
            if subclass n'a plus de super-classes then
                promote-max (subclass)
        foreach conflict ← (c, conflict) ∈ Ê do
            freeze-color (conflict, c)
            promote-conflict (conflict)
    until max-cf-list = ∅ ∧ max-ncf-list = ∅
end

procedure promote-max (c)
begin
    score ← compute-score(c)
    if tag (c) = notmax-ncf then
        tag (c) ← max-ncf
        insert-ordered (c, score, max-ncf)
    else
        tag (c) ← max-cf
        insert-ordered (c, score, max-ncf-list)
end

procedure promote-conflict (c)
begin
    if tag (c) = notmax-ncf then
        tag (c) ← notmax-cf
    else if tag (c) = max-ncf then
        tag (c) ← max-cf
        remove-from (c, max-ncf-list)
        score ← compute-score(c)
        insert-ordered (c, score, max-cf-list)
end
```

Coloration de la frontière et de la couronne

La coloration de la frontière et de la couronne ne présente aucune difficulté, grâce à une particularité : leur coloration ne peut pas produire de nouveaux trous dans les sous-classes. Il suffit donc de colorer ces classes dans l'ordre de la linéarisation en affectant la première couleur disponible pour une classe.

5.2.3 Coloration bidirectionnelle

Pour réduire le nombre de trous dans les tables, Pugh et Weddell [1990] proposent d'utiliser des couleurs négatives en plus des couleurs positives, on parle donc de coloration bidirectionnelle ce qui réduit, en pratique, le nombre de trous. Les algorithmes de la coloration doivent êtres modifiés pour choisir le premier indice libre, positif ou négatif, et choisir celui qui cause le moins de trous. En cas d'égalité on conservera l'indice positif.

Lorsque que le graphe de conflit est biparti, comme celui de l'exemple du losange (Figure 5.4), tous les trous disparaissent en utilisant la bidirectionnallité (sur l'exemple, il suffit d'affecter à B ou C à une position négative). La coloration est *parfaite*.

D'un point de vue algorithmique, outre le léger changement dans l'énoncé de l'algorithme, il vaut mieux commencer par faire une bipartition de l'ensemble des classes à colorer [Ducournau, 2010].

Enfin, la coloration bidirectionnelle présente un inconvénient pour l'utilisation de ramasse-miettes *mark & sweep* : les zones mémoire allouées doivent être précédées de leur taille. Il est donc nécessaire de trouver le début de la zone. Desnos [2004] propose un ramasse-miettes spécialement adapté à la coloration bidirectionnelle et [Gagnon, 2002] propose un ramasse-miettes copieur tenant compte des des parties négatives.

Généralisation

La coloration bidirectionnelle est une forme de généralisation de la coloration, il est possible de la généraliser à n dimensions. Chaque dimension est représentée par une table et une classe est identifiée par sa couleur et sa table de couleur. Cependant comme une ligne n'a que deux sens, les nouvelles dimensions ne peuvent pas être atteintes autrement que par une — ou des — indirection supplémentaire, ce qui complexifie nettement la structure de données. A part en pure compilation globale, toutes les classes doivent référencer les tables supplémentaires, l'optimisation nécessite surtout un compromis entre le nombre de trous et le nombre de tables — une nouvelle table rajoute des entrées supplémentaires dans toutes les classes. Toutefois, en typage dynamique, des implémentations autour de la généralisation de la coloration ont été proposés [Pugh et Weddell, 1993; Zibin et Gil, 2003].

5.3 Hachage parfait de classes

Le hachage parfait est une optimisation en temps constant des tables de hachage pour des ensembles immutables [Sprugnoli, 1977; Czech *et al.*, 1997]. Pour obtenir des accès en temps constant, le principe est de calculer étant donné un ensemble d'éléments, une fonction de hachage de telle sorte que la table ne contienne pas de collision. Ducournau [2008] propose d'utiliser cette approche pour en déduire un test de sous-typage ainsi qu'un mécanisme d'invocation de méthode efficace. Pour éviter toute confusion avec la proposition originale nous la nommerons *hachage parfait de classes* dans la suite de ce document.

Par souci de clarté nous ne présenterons dans cette section que la problématique générale et l'application du hachage parfait de classes au test de sous-typage. Nous présenterons dans la section 5.4, plus généralement, comment utiliser le test de sous-typage pour réaliser l'invocation de méthode.

5.3.1 Principe

A chaque classe C on associe un identifiant unique et immuable id_C ainsi que l'ensemble I_C des identifiants des super-classes de C, C comprise. L'identifiant peut, par exemple, correspondre à l'ordre de chargement de la classe. Plus formellement, $I_C = \{id_C \mid c \preceq d\}$ et donc $c \preceq d$ si et seulement si $id_D \in I_C$.

Comme l'ensemble I_C est constant, il est possible de le *hacher* parfaitement, c'est-à-dire sans collision, à l'aide d'une fonction de hachage h_c injective sur I_C. En utilisant cette propriété, on a $c \preceq d$ si et seulement si $ht_c[h_c(id_C)] = id_C$ où ht_c représente la table de hachage obtenue en hachant I_C avec h_c.

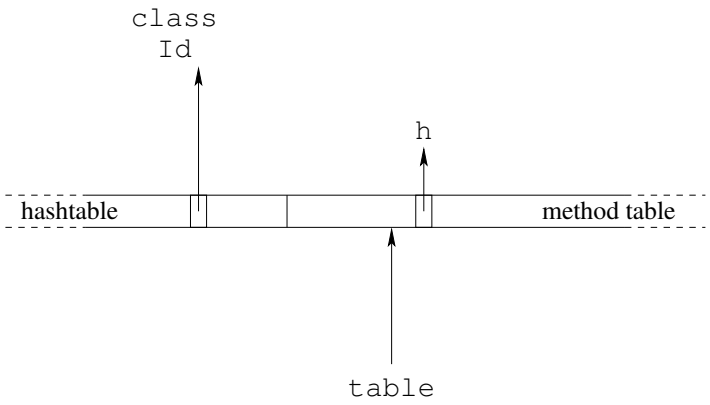
5.3.2 Formalisation

Définition 5.11 (Hachage parfait) *Soit I un ensemble non vide d'entiers, et la fonction $hash : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, telle que $hash(x,y) < y$ et $hash(x,y) \leq x$ pour tout $x, y \in \mathbb{N}$. Le* paramètre de hachage parfait *H de I est le plus petit entier tel que la fonction h qui associe x à $h(x) = hash(x,H)$ soit injective sur I, c'est-à-dire que pour tout $x, y \in I$, $h(x) = h(y) \Rightarrow x = y$.*

Cette définition est étendue à l'ensemble vide par $H = 1$ quand $I = \emptyset$.

Hachage parfait de classes (PH). Soit une hiérarchie de classes $(X, \preceq)$ munie d'une fonction de numérotation de classe injective $id : X \to \mathbb{N}$. Dans ce cas, le hachage parfait peut s'appliquer à chaque classe $c \in X$ en considérant l'ensemble $I_c = \{id_d | c \preceq d\}$. La table de hachage associée à la classe c contient pour chacune de ses super-classe d, l'identifiant id_d à la position $h_c(id_d)$ et à toutes les autres positions j un entier quelconque l tel que $h_c(l) \neq j$. Le paramètre de hachage H_c est la taille de cette table.

La contrainte $hash(x,y) \leq x$ n'est pas strictement nécessaire mais s'avère vérifiée par toutes les fonctions de hachage utilisées en pratique. De plus, cette condition implique



L'objet pointe (par `table`) à l'indice 0 de la table de méthodes, le paramètre `h` est aussi dans la table. Les indices négatifs contiennent la table de hachage. Les entrées de la table sont les identifiants des classes (`classId`) pour le test de sous-typage.

```
; preamble
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable

; subtype testing                                          3L + 4
load [htable +#htOffset], id
comp #classId, id
bne #fail
; succeed
```

FIGURE 5.5 : Hachage parfait de classes

$h_c(0) = 0$ quelque soit c. Cette dernière remarque entraîne que pour les hiérarchies de classes enracinées, 0 est un bon candidat pour l'identifiant de la racine et par la même occasion une valeur possible pour les entrées vides ($j > 0$). Réciproquement, pour les hiérarchies non enracinées, n'importe quel entier non nul peut être utilisé à la position 0, à condition de ne pas être utilisé comme identifiant de classe.

5.3.3 Fonctions de hachage

En toute généralité, le hachage parfait de classes n'impose aucune spécificité à la fonction de hachage utilisée. Néanmoins, une contrainte implicite est nécessaire, la fonction de hachage doit être courte — pour être mise en ligne — et efficace.

Dans un premier temps, nous nous sommes intéressés à des fonctions simples à deux paramètres : le « et » logique (`and`) et le reste de la division entière (`mod`). Ces fonctions pré-

LISTING 5.6 : Algorithme pour calculer le paramètre de hachage de PH-and

```
(defun ph-and (ln) ;; LN is an integer list
 (if (null (cdr ln))
    1
    (let ((mask (logxor (apply #'logior ln) (apply #'logand ln))))
                                ;; MASK consists of all discriminant
                                   bits
        (fill *ht* nil :start 0 :end (1+ mask)) ;; resets *HT*
        (loop for b from (highest-bit mask) by 1 downto 0
            when (logbitp b mask) do    ;; for each 1-bit in MASK
            (let (( new (logxor mask (ash 1 b)))) ;; NEW = MASK with
               switched bit
                (when (ph-and-p ln new) (setf mask new)))
            finally return (1+ mask)))))

(defun ph-and-p (ln mask) ;; check if MASK is a hash parameter for LN
    (loop for i in ln
        for hv = (logand i mask) do
        (if (eql (aref *ht* hv) mask)
            (return nil)
            (setf (aref *ht* hv) mask))
        finally return t))
```

sentent un rapport compacité/efficacité intéressant, la première est très efficace en temps, tandis que la seconde produit des tables plus compactes.

Nous avons essayé ensuite d'identifier une fonction à trois paramètres qui améliorerait la taille des tables produites avec « et » mais plus efficace que le reste de la division entière. Nous avons donc testé la combinaison de deux fonctions efficaces : « et » suivi d'un *décalage* (`and+shift`). Le principe est le même qu'avec le « et » mais le *décalage* permet d'éliminer les bits de poids faible non discriminants. Cette fonction devrait donc engendrer des tables plus compactes. Cependant les expérimentations réalisées sur de grandes hiérarchies (voir Section 5.11.1), ne semblent pas donner de gain significatif [Ducournau et Morandat, 2011] et l'instruction supplémentaire risque de dégrader nettement l'efficacité temporelle de la technique.

Les algorithmes pour calculer le paramètre de hachage sont totalement décrits dans les annexes de [Ducournau et Morandat, 2011] mais pour clarifier les esprits nous rappelons ici un algorithme pour PH-and.

L'idée de cet algorithme est simple, isoler les bits discriminants des identifiants à hacher, puis chercher une combinaison de ces bits qui forme une table de hachage parfaite. Nous construisons donc une table et dès qu'il y a une collision, la valeur de hachage est rejetée. Si tout les identifiants ont pu être hachés sans produire de collision, le masque est valide et il est le paramètre de hachage de cette classe.

5.3.4 Hachage parfait de classes pour les sous-objets

L'implémentation par sous-objets nécessite une table $\Delta_{\Downarrow}$ (voir Section 5.1.3) pour réaliser le test de sous-typage et les *downcasts*. L'utilisation d'une matrice directe ⟨classe, classe⟩ n'est pas vraiment envisageable dès que le nombre de classes devient grand. Le hachage parfait de classes peut être utilisé pour implémenter cette table.

La table de hachage peut être directement embarquée dans chaque table de méthodes. Cependant comme les sous-objets nécessitent une table de méthodes par type dynamique pour chaque type statique, dans le pire des cas elle pourrait être recopiée un nombre quadratique de fois. Cette alternative ne parait donc pas viable et lorsque le hachage parfait de classes est utilisé pour les sous-objets, la table de hachage doit être séparée des tables des méthodes — ce qui nécessite une indirection supplémentaire.

Pour encoder les valeurs des *downcasts*, les entrées doivent être doublées comme expliqué dans la section 5.4 et leur contenu doit être la valeur des décalages $\Delta_{\Downarrow}$.

5.3.5 Numérotation parfaite

La définition du hachage parfait de classes est compatible avec n'importe quelle fonction de numérotation injective, par exemple une numérotation consécutive des classes dans l'ordre de chargement. Une optimisation du hachage parfait consiste à choisir l'identificateur id_c qui minimise le paramètre H_c et donc la taille de la table. Donc, au lieu de numéroter les classes par incrémentation d'un compteur, on peut choisir le paramètre H_c comme le H_c minimum des super-classes qui laisse une place libre, puis l'identifiant id_c qui tiendrait dans la place libre de la table. Nous appellerons cette optimisation *Numérotation parfaite de classes* [Ducournau et Morandat, 2011].

Définition 5.12 (Numérotation parfaite) *Soit l'ensemble d'entiers I' et F un ensemble d'entiers libres disjoints de I'. Le* paramètre numérotation parfaite *H est défini comme le plus petit entier tel que : (i) il existe un identificateur id dans F (ii) H est le paramètre de hachage parfait de l'ensemble $I = I' \uplus \{id\}$*

Numérotation parfaite de classes (PN). On peut appliquer la numérotation parfaite à chacune des classes c de X en considérant $I'_c = \{id_d | c \prec d\}$ et F un ensemble d'entiers ne contenant aucun identificateur de classe déjà chargé. Le paramètre H_c est la taille de la table de hachage c associée et l'identificateur de id_c est le plus petit entier tel que H soit le paramètre de hachage de I_c

Le code généré pour chacun des mécanismes de la numérotation parfaite de classes est identique à celui du hachage parfait de classes, seul le calcul des identifiants diffère. Le hachage parfait de classes et la numérotation parfaite de classes ayant le même code, ces deux implémentations ont la même efficacité temporelle et la numérotation parfaite de classes doit être préférée si l'espace est un critère déterminant.

La numérotation parfaite de classes produit, en pratique, des tables plus compactes que le hachage parfait de classes, bien que ça ne soit pas toujours le cas, comme nous le montrerons les statistiques effectuées sur des grandes librairies [Ducournau et Morandat, 2011].

5.4 Du test de sous-typage à l'appel de méthode

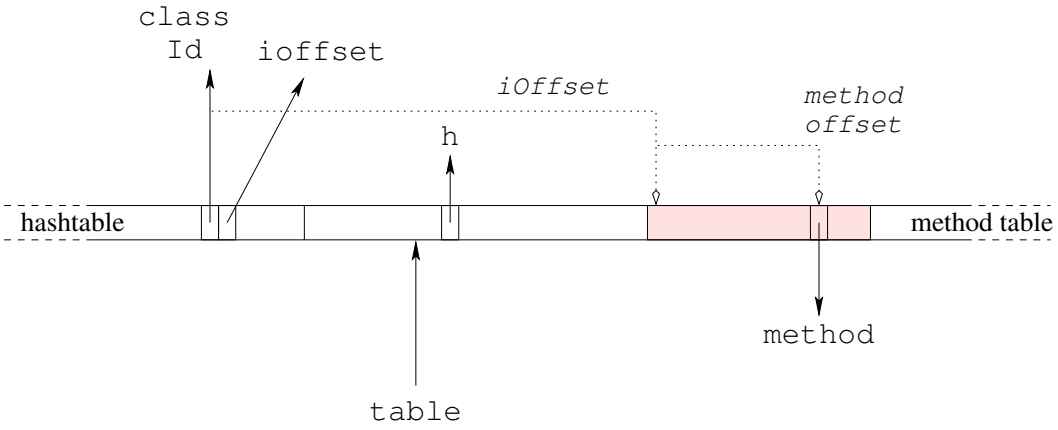
D'après les équivalences de mécanismes proposées pour l'implémentation du sous-typage simple (voir Section 4.6.3), il est possible de déduire un mécanisme d'invocation de méthodes du test de sous-typage. Le principe est de grouper les méthodes par classe d'introduction et d'associer à chaque entrée qui indique une réussite au test pour une classe l'adresse de son groupe de méthodes. Cependant ceci n'est possible que si les méthodes sont groupables par classe d'introduction, ce qui est toujours possible en typage statique.

L'application au hachage parfait de classes se fait directement. Les entrées de la table de hachage sont doublées et la partie rajoutée contient l'offset du groupe de méthode introduit par la classe. Enfin, il suffit d'ajouter au groupe de méthode l'offset de la méthode elle même. Il est possible de remplacer l'offset du groupe de méthode par l'adresse de ce groupe, ce qui réduit la séquence de code d'une instruction. L'offset est à préférer pour les processeurs disposant d'opérations sur les entiers 16 bits. Si ce n'est pas le cas, des entiers de 32 bits sont préférables et l'adresse évite une addition.

5.5 De l'appel de méthode à l'accès à un attribut : la simulation des accesseurs

La simulation des accesseurs n'est pas une technique en tant que telle mais plutôt un moyen de déduire un mécanisme d'accès aux attributs en utilisant une technique d'implémentation d'envoi de message [Myers, 1995; Ducournau, 2011] donc en rajoutant au moins une indirection par la table des méthodes. Pour éviter de faire un appel de fonction, l'idée est de stocker la position de l'attribut (index) dans la table des méthodes à la place de l'adresse de la fonction accesseur qui serait appelée et la position du groupe des attributs. La simulation des accesseurs s'applique à n'importe quelle technique d'implémentation des méthodes — sauf aux sous-objets qui utilisent un équivalent dû à l'invariant 5.2. Elle nécessite juste qu'un attribut soit introduit par une classe unique, ce qui est le cas en typage statique. Pour éviter de rajouter une entrée dans la table des méthodes par attribut, il est conseillé de grouper les attributs par classe d'introduction (voir Section 4.6.1) et de faire une addition supplémentaire [3].

3. Lorsque l'attribut est le premier introduit par une classe, il n'est pas nécessaire de faire cette addition (car le delta vaut 0). Comme le mécanisme de raffinement de classe de PRM complique énormément ce scénario, nous n'avons pas à l'heure actuelle pu implémenter cette optimisation.



L'objet pointe (par `table`) à l'indice 0 de la table de méthodes. Les indices positifs contiennent les adresses des méthodes dans l'implémentation de l'héritage simple, où les méthodes sont groupées par classe d'introduction. Le paramètre de hachage `h` est aussi dans cette table. Les indices négatifs contiennent la table de hachage. Une entrée de la table est formée de l'identifiant de la classe (`classId`) pour le test de sous-typage et de la position relative (`iOffset`) du groupe de méthodes introduites par la classe.

```
; preamble
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable

; method invocation                                          4L + B +3
load [htable + #htOffset + 1], iOffset
add htable, iOffset, itable
load [itable +#methOffset], method
call method
```

FIGURE 5.6 : Appel de méthode simulé avec le hachage parfait de classes (`and`)

LISTING 5.7 : Accesseur simulé avec la coloration

```
load [object + #tableOffset], table
load [table + #classOffset], attrGroupOffset
add object, attrGroupOffset, attrgrp
load [attrgrp + #attrOffset], value
```

LISTING 5.8 : Accès aux attributs en utilisant le hachage parfait de classes

```
; preamble
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #classId, h, hv
sub table, hv, htable

; attribute access
load [htable + #htoffset + 2], attrGroupOffset
add object, attrGroupOffset, attrgrp
load [attrgrp + #attrOffset], value
```

Comme la coloration peut engendrer de nombreux trous, la simulation des accesseurs peut être utilisée pour éviter les trous au prix d'une indirection par la table des méthodes. Les expérimentations du chapitre 8 montrent que son surcoût est réel mais non rédhibitoire. Pour les techniques où le code généré est plus long, comme le hachage parfait de classes ou la coloration incrémentale, le surcoût est nettement plus élevé.

5.5.1 Adaptation des implémentations

L'adaptation de la simulation des accesseurs à la coloration est directe, elle ne rajoute qu'une indirection par la table des méthodes pour récupérer le groupe d'attributs introduit par la classe (Listing 5.7).

Pour le hachage parfait de classes la structure des entrées de la table de hachage doit contenir un champ supplémentaire, celui de la position du groupe d'attributs introduit pour chaque classe (voir Section 5.4). Pour respecter l'alignement des données au mot (ou double mot) il est conseillé de rajouter un quatrième champ — même s'il ne contient aucune information. L'adaptation de la simulation aux autres techniques présentées dans la suite de chapitre, se fait sur le même principe en remplaçant seulement les lignes spécifiques au hachage parfait de classes par la séquence de code de la technique considérée. Bien que le code de la simulation des accesseurs soit quasiment générique, la structure de la table des méthodes doit être reconsidérée pour chaque mécanisme.

5.6 Variantes de la coloration

Nous présentons dans cette section deux variantes de colorations qui ne nécessitent pas l'hypothèse du monde clos.

5.6.1 Coloration incrémentale

Une version incrémentale de la coloration a été proposée par Palacz et Vitek [2003] pour implémenter le test de sous-typage en JAVA sous le nom de *Ranges and Buckets* (R&B). Ce test est basé sur la numérotation de Schubert (voir Section 4.5.2) pour les classes (*Range*) et utilise la coloration pour les interfaces (*Buckets*) Pour un besoin de comparaison, une application à l'invocation de méthode basée sur les mêmes principes que l'appel de méthode du hachage parfait de classes (voir section 5.3) a été proposée dans [Ducournau, 2008; Ducournau *et al.*, 2009].

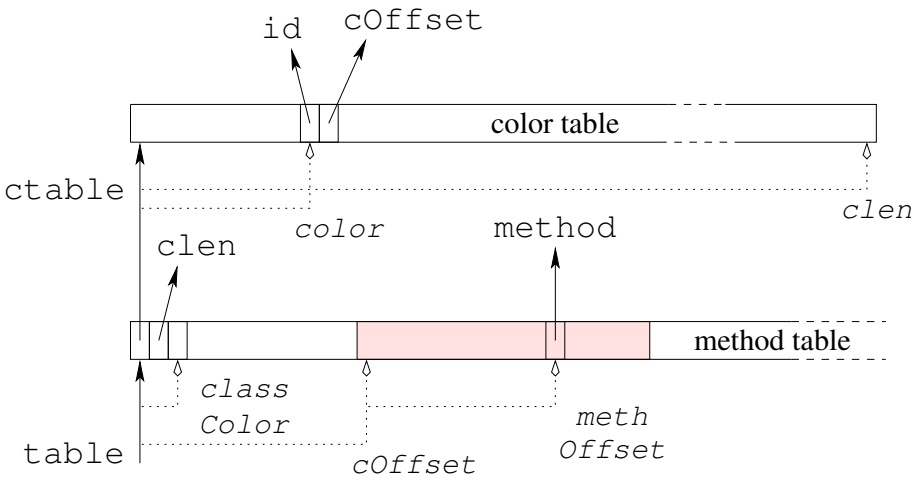
L'implémentation ressemble à celle du hachage parfait de classes sauf que la position d'une interface dépend d'un `load` au lieu d'être le résultat d'une fonction de hachage. La position d'une couleur peut être stockée dans la table des méthodes de la classe cible ou dans une table séparée. De ce fait, la structure de données utilisée ajoute une indirection (`load`) dans une zone mémoire différente, ce qui a pour conséquence d'augmenter le risque de défaut de cache (Figure 5.7). De plus, comme la table des couleurs doit pouvoir être recalculée, elle doit nécessairement être disjointe de la table des méthodes et nécessite donc une indirection supplémentaire ainsi qu'un test de borne sur la valeur `clen` — et donc un `load` supplémentaire — et peut conduire à deux défauts de cache supplémentaires, soit trois au total. Enfin, comme la coloration nécessite l'hypothèse du monde clos (CWA), la coloration incrémentale peut entraîner certains recalculs — ceux des buckets en cas de collision — durant le chargement.

A l'instar du hachage parfait de classes, l'accès aux attributs se fait par simulation des accesseurs (voir Section 5.5). Les entrées de la table des couleurs deviennent donc des entrées triples ⟨`couleur, method_group_offset, attribute_group_offset`⟩. Le lecteur intéressé est renvoyé à [Ducournau, 2011] pour les détails de l'implémentation.

5.6.2 Coloration partagée

La coloration peut être rendue compatible avec les librairies partagées au prix d'une indirection supplémentaire [Privat et Morandat, 2008]. Les classes appartenant à la librairie peuvent être compilées séparément puis partagées par différents programmes (partagées sur le disque et en mémoire).

Cette version de la coloration utilise une table locale à l'unité de code pour stocker les positions des couleurs. L'accès à la couleur proprement dite nécessite un `load` supplémentaire mais comme les deux chargements sont indépendants ils peuvent être quasiment exécutés en parallèle — en fait, avec le modèle de processeurs que nous avons pris comme




```
; preamble
load [object + #tableOffset], table
load [table + #ctableOffset], ctable
load [targetTable + #classColor], color
add ctable, color, entry

; method invocation                                          4L + B + 3
load [entry + #methEntryOffset], cOffset
add table, cOffset, methgrp
load [methgrp + #methOffset], method
call method

; subtype test                                               6L + 6
load [table + #clenOffset], clen
comp clen, color
ble #fail
load [entry], id
comp id, #targetId
bne #fail
; succeed

; Attribute access                                           3L + 3
load [entry + #attrEntryOffset], aOffset
add object, aOffset, attrgrp
load [attrgrp + #attrbOffset], attribute
```

FIGURE 5.7 : Coloration incrémentale

exemple [Driesen, 2001], il y a théoriquement un cycle supplémentaire. Ce qui n'est pas beaucoup, mais qui augment le risque de défauts de cache. Enfin, l'accès aux attributs est réalisé par simulation des accesseurs (voir Section 5.5).

La table locale `localTable_c` pour une classe donnée a la structure suivante :

- les deux premières positions sont réservées au test de sous-typage et contient la position de la couleur et sa valeur pour une classe (`#classColorPos` et `#classIdPos`) ;
- vient ensuite la position du groupe d'attributs introduits par la classe ;
- pour chaque méthode *m* introduite par la classe, une position `#methodColorPos_m` est réservée dans la table.

Afin de réduire encore la taille des tables locales, la table locale peut se contenter de la position du groupe de méthodes introduite par la classe — au lieu de toutes les méthodes — voire se servir de la position de la couleur pour la classe. Cependant cette alternative rajoute une addition au chargement de la couleur et nécessite que la coloration attribue les couleurs des toutes les méthodes introduites par une classe consécutivement.

Remarque. Comme en PRM, l'unité de code est le module et non la classe, les tables locales sont groupées par modules.

```
; Method invocation                                              2L + B + 2
load [object + #colorTableOffset], ctable
load [localTable_c + #methodColorPos_m], methodColor
load [ctable + #methodColor], methode
call methode

; Subtype test                                                   2L + 4
load [object + #colorTableOffset], ctable
load [localTable_c + #classColorPos], classColor
load [localTable_c + #classIdPos], classId
load [ctable + classColor], class
comp #classId, class
bne #checkFailed
```

FIGURE 5.8 : Coloration partagée

5.7 Caches

Une technique d'implémentation souvent appliquée dans les langages à typage dynamique ou aux interfaces de JAVA consiste à coupler une technique d'implémentation — qui est supposée, ici, être plutôt naïve et inefficace — à un cache qui *mémoïse*[4] le résultat de la

4. D'après wikipédia, le terme *mémoïsation* aurait été introduit par Donald Michie en 1968 et est dérivé du mot latin *memorandum* signifiant « qui doit être rappelé ». Il y a donc derrière le terme *mémoïsation* l'idée

dernière recherche [Alpern *et al.*, 2001a; Click et Rose, 2002]. Lors de l'invocation d'un mécanisme, le cache est d'abord consulté puis un test d'égalité de type est effectué. Si le test réussit les données du cache sont valides et peuvent être utilisées. Dans le cas contraire la technique d'implémentation sous-jacente est invoquée et le cache remis à jour. Les caches exploitent l'idée de proximité temporelle, c'est-à-dire qu'une opération sur un type sera généralement suivie d'autres opérations sur ce même type — les boucles sur des collections, dont les éléments sont supposés de même type, en sont une parfaite illustration.

5.7.1 Caches dans la table des méthodes

On peut réserver des entrées directement dans les tables de méthodes pour s'en servir de cache. Chaque fois qu'un mécanisme est déclenché, ce cache est consulté, permettant de court-circuiter l'appel sous-jacent. Comme ce cache est alloué dans la table des méthodes, il est propre à chaque type dynamique — mais il contient des informations relatives à un type statique donné.

Le cache dans les tables de méthodes peut être utilisé conjointement avec toutes les techniques d'implémentation basées sur des tables et pour chacun des trois mécanismes, à condition de *cacher* les trois données que sont : l'identifiant de la classe, l'offset du groupe de méthodes et celui du groupe d'attributs introduits par cette classe. En pratique, on utilisera plutôt un cache à quatre entrées afin de respecter l'alignement, ce qui permet de le transférer avec une seule instruction (de la famille des `mov` sur x86).

Ce type de cache présente plusieurs inconvénients. Tout d'abord la séquence de code pour chacun des mécanismes est relativement plus longue que celle de l'implémentation sous-jacente — elle rajoute la gestion du cache au code de l'appel. Ensuite l'utilisation de ce type de cache nécessite que les tables de méthodes soient accessibles en écriture — donc allouées dans un segment mémoire de données et non plus dans le segment de code.

Enfin en cas de défaut de cache, le temps de mise à jour du cache doit être rajouté au temps d'exécution du mécanisme sous-jacent.

Réduction des défauts de cache

L'amélioration proposée par le cache est une question de statistiques, celles présentées dans [Palacz et Vitek, 2003] montrent qu'en fonction des benchmarks les défauts de cache varient de 0.1% à plus de 50%. L'efficacité des caches étant directement liée à ce taux, réduire le nombre de défauts de cache contribue à améliorer l'efficacité du cache. Nous proposons maintenant deux politiques de gestion de cache, non exclusives, permettant de réduire ce nombre.

---

de résultats de calculs dont il faut se souvenir. Bien que *mémoïsation* évoque « mémorisation », le terme *mémoïsation* a une signification particulière en informatique. La mémoïsation est une façon de diminuer le temps de calcul d'une fonction, au prix d'une occupation mémoire plus importante. La mémoïsation modifie donc la complexité d'un algorithme en temps comme en espace.

LISTING 5.9 : Cache séparé dans la table des méthodes

```
; method invocation                                   cache hit : 4L + B +3
load [object + #tableOffset], table                  cache miss X + 3L + B + 3
load [table + #cacheMethId_i], id
comp id, #targetId
beq #hit
store #targetId, [table + #cacheMethId_i]
; usual sequence of PH or IC (4-5 instructions)
store iOffset, [table + #cacheMethOffset_i]
jump #end
hit:
load [table + #cacheMethOffset_i], iOffset
end:
add table, iOffset, itable
load [itable + #methodOffset], method
call method

; subtype testing
load [object + #tableOffset], table
load [table + #cacheTypeId_i], id
comp id, #targetId
beq #succeed
; usual sequence of PH or IC (6-8 instructions)
store #targetId, [table + #cacheTypeId_i]
succeed:
```

Caches multiples. Le taux de réussite du cache peut être amélioré en utilisant plusieurs (n) caches [Ducournau *et al.*, 2009]. Quand n augmente, le taux de défaut de cache tend asymptotiquement vers 0. La structure du cache est répliquée n fois dans chaque table des méthodes. Les classes (ou les interfaces) sont statiquement partitionnées en n ensembles — par exemple en hachant leur nom. A chaque opération concernant un type statique τ_s, on utilise le cache associé à la partition contenant τ_s.

Si pour une classe C, un des ensembles, restreint aux super-classes de C, est un singleton, on peut initialiser le contenu de ce cache avec les informations de la classe C et omettre ces informations de la structure sous-jacente.

Malgré la diminution évidente du nombre de défauts de cache apportée par cette amélioration, le meilleur et le pire des cas restent inchangés. Le meilleur des cas est borné par une consultation du cache tandis que, dans le pire des cas, la gestion du cache doit être ajoutée au temps du mécanisme sous-jacent.

Caches séparés. Le cache peut être commun aux trois mécanismes ou dédié à chacun d'eux. Dédier un cache à chaque mécanisme consiste à le mettre à jour partiellement et en fonction du mécanisme utilisé — on nommera *sous-cache* la partie associée à un mé-

canisme. La structure de chaque sous-cache doit être modifiée par rapport à la structure proposée initialement (voir Section 5.7.1). Chaque sous-cache doit contenir deux entrées, une pour stocker l'information proprement dite du cache — par exemple la position du groupe de méthodes introduites par une classe. La seconde entrée doit contenir l'identifiant de type associé à l'information contenue dans le cache — pour le test d'égalité. Bien entendu, dans le cas du test de sous-typage, les deux entrées sont identiques. On peut donc se limiter à une entrée simple.

Pour mettre à jour plus efficacement le cache, la structure de données associée au mécanisme sous-jacent peut être légèrement modifiée. Chaque architecture ayant des contraintes différentes, l'idée principale est de pouvoir mettre à jour le cache en une seule instruction. En prenant comme exemple le hachage parfait, on peut changer l'ordre proposé pour les entrées de la table par : ⟨`#methodGroup`,`#classId`,`#attributeGroup`⟩. Dans ce cas, les sous-caches associés à l'envoi de message et à l'accès aux attributs sont inversés. Pour respecter l'alignement, les entrées de la table de hachage sont quadruples et nous dupliquons l'identifiant de type dans l'entrée inutilisée. Une entrée devient donc de la forme : ⟨`#classId`,`#methodGroup`,`#classId`,`#attributeGroup`⟩. Cette deuxième alternative est un peu plus gourmande en espace mais permet d'avoir des opérations et une structure identiques quel que soit le mécanisme considéré.

L'utilisation de caches séparés ne permet pas de garantir une réduction des défauts de cache. Cependant les expériences que nous avons faites (voir Chapitre 8) avec ce type de cache semblent lui donner un avantage par rapport au cache partagé.

Enfin, les caches multiples et les caches séparés peuvent être utilisés conjointement, réduisant considérablement le nombre de défauts de cache par rapport au cache simple mais augmentant sensiblement la taille du code nécessaire pour chaque mécanisme — la mise en ligne du code n'arrangeant rien.

Évaluation

De plus, le pire des cas est plutôt inefficace car il rajoute la gestion du cache au temps nécessaire pour l'appel sous-jacent. Quant au meilleur des cas, il est légèrement plus efficace que PH-`and` pour l'invocation d'une méthode et identique au test de Cohen pour le test de sous-typage. Nous pourrions attendre du cache qu'il dégrade PH-`and` mais qu'il améliore PH-`mod`. De ce fait, nous n'avons testé les caches dans les tables des méthodes que sur les techniques d'implémentation « lentes ».

5.7.2 Caches en ligne

Il est possible de remplacer le cache dans la table des méthodes par un cache spécifique à chaque site d'appel. Cette solution est souvent utilisée dans les langages à typage dynamique, car le coût de la recherche de l'adresse de la méthode à appeler (*lookup*) est généralement important. Comme les programmes objet utilisent peu le polymorphisme, le

LISTING 5.10 : Cas défavorable de cache en ligne

```
for i in [ 1, "a", 1, "a",1, "a", 1, "a" ] do
    println(i.to_s)
end
```

taux de réussite de ce type de cache est, en pratique, très élevé. Cependant, lorsque le site d'appel est véritablement polymorphe, comme dans l'exemple ci-dessous (Listing 5.10), la gestion du cache présente un surcoût non négligeable.

Caches monomorphes

Les caches monomorphes sont les plus simple à implémenter, leur utilisation remonte à l'implémentation de SMALLTALK [Deutsch et Schiffman, 1984]. Le principe général est d'associer à chaque site d'appel une variable *statique* au sens de C (ou globale) pour servir de cache et une seconde pour stocker le type associé au cache. L'appel de méthode est ensuite entouré d'une *garde*[5] qui vérifie l'identité du receveur puis appelle l'adresse stockée dans le cache. Si le test de garde échoue, un appel de méthode classique est utilisé et le cache est mis à jour.

Pour limiter les défauts de cache matériel, il est préférable que les variables qui servent de caches soient dans le même segment mémoire, ce qui est impossible si le code est dans un segment en lecture seule. L'idéal est un code auto-modificateur où les sauts indirects sont remplacés par des sauts directs pour profiter de la prédiction de branchement.

Caches en ligne polymorphes

Les caches en ligne polymorphes (*Polymorphic Inline Cache*) stockent un ensemble de couples ⟨type, méthode⟩ au lieu d'un seul [Hölzle *et al.*, 1991]. A chaque appel de méthode, le cache est d'abord consulté et mis à jour en conséquence. La taille maximale du cache est fixée par une constante. En cas de défaut de cache, comme pour les caches monomorphes, la technique d'implémentation sous-jacente est appelée.

Caches statiques ou dynamiques et profilage

Nous venons de présenter deux types de caches en ligne dynamiques, dans le sens où c'est l'exécution du programme qui détermine leur contenu. Il est possible de les remplacer par des versions statiques, où le contenu du cache est une constante. Le principal avantage d'une version statique est que l'appel de méthode est maintenant direct — ou un branchement conditionnel sur le test de garde — et l'exécution peut donc profiter de la

5. Une garde est une expression de type booléen qui a pour valeur « vrai » si l'exécution du programme doit continuer dans la branche en question.

prédiction de branchement. Cependant pour que cette alternative soit efficace, il est nécessaire d'avoir des statistiques sur l'exécution du programme, rendant le *profilage* d'une — ou plusieurs — exécution du programme obligatoire.

Les caches monomorphes statiques sont souvent utilisés par les compilateurs à la volée qui profitent de l'hypothèse du monde clos temporaire. En effet, cette hypothèse réduit considérablement le nombre de candidats potentiels pour chaque appel de méthode.

La mise à jour du cache polymorphe impose un profilage dynamique qui peut réduire à néant l'efficacité recherchée.

5.8 Arbres binaires de sélection (BTD)

Les arbres binaires de sélection utilisés par SMART EIFFEL [Zendra *et al.*, 1997; Collin *et al.*, 1997] sont une généralisation des *polymorphic inline cache* [Hölzle *et al.*, 1991] (voir section 5.7.2) dans un contexte différent.

5.8.1 Principe

Alors que toutes les techniques présentées jusqu'à présent utilisent des tables pour invoquer les méthodes, les *Binary Tree Dispatch* (BTD) les remplacent par un arbre binaire équilibré de comparaisons de types dont les branches de l'arbre déterminent la propriété locale à appeler et dont les feuilles sont des appels statiques. Cette approche n'est possible que dans le cadre de l'hypothèse du monde clos car c'est seulement dans ce cas que l'ensemble des classes est connu — donc les propriétés locales associées à chaque couple ⟨propriété globale, type statique du receveur⟩ sont connues. Dans l'implémentation de SMART EIFFEL, les comparaisons sont faites directement sur l'identifiant de la classe et le pointeur de l'instance vers la table des méthodes est remplacé par cet identifiant. Il est aussi possible d'utiliser l'adresse de la table des méthodes.

Le temps d'appel d'une méthode est donc logarithmique dans le nombre de méthodes candidates, mais la taille du code nécessaire est exponentielle dans le temps. Pour réduire la taille de l'arbre et donc augmenter l'efficacité de cette technique, l'utilisation d'une analyse de types [Bacon et Sweeney, 1996; Grove et Chambers, 2001] (voir section 6.3.3) permet de réduire l'ensemble des couples ⟨classe, implémentation⟩ à considérer aux ensembles de couples *vivants*. Enfin, pour réduire encore le nombre de tests, on peut ordonner les cibles potentielles par identifiant de classe puis les regrouper par intervalles.

L'efficacité des arbres binaires de sélection vient principalement, du cache d'instructions et des prédictions de branchements des processeurs modernes. Un branchement bien prédit ou un saut direct sont quasiment gratuits si la cible est dans le cache. En pratique, si la profondeur de l'arbre n'excède pas trois niveaux — donc avec huit feuilles au maximum — le nombre moyen de cycles pris pour réaliser les comparaisons est inférieur au temps d'une indirection à la table de méthodes.

LISTING 5.11 : Un arbre binaire de sélection de profondeur 2

```
load [object + #idOffset], id
comp id, id0
bgt #branch1
comp id id1
bgt #branch2
call #method1
jump #end
branch2:
call #method2
jump #end
branch1:
comp id id2
bgt #branch3
call #method4
jump #end
branch3:
call #method3
end:
```

Les arbres binaires de sélection ne sont pas réservés aux appels de méthode, ils peuvent être utilisés aussi pour le test de sous-typage et pour l'accès aux attributs ; c'est le cas de l'implémentation de SMART EIFFEL. Dans le cas des attributs, la plupart ont des positions invariantes car seulles les situations de losanges et d'héritage multiple provoquent des positions différentes. Les arbres qui sont associés à ces sites d'appels sont donc réduits à un seule feuille — une constante, l'accès à un attribut se fait donc comme en sous-typage simple. Au final grâce aux arbres binaires de sélection, le coût des attributs en plein héritage multiple est quasiment annulé — ils représentent plus de 99% des cas dans le compilateur PRMC [Ducournau et Morandat, 2011].

Deux stratégies sont utilisées pour l'implémentation des arbres de sélection. La *mise en ligne* des arbres au niveau du site d'appel ou l'utilisation de *thunks*. La *mise en ligne* est censée profiter de la proximité temporelle et spatiale — ce qui améliore statistiquement les prédictions de branchement — au prix d'un code nettement plus long. De l'autre côté, les *thunks* permettent de factoriser les arbres équivalents en rajoutant un appel de fonction. Dans le cas de l'appel de méthode, cet appel est quasiment gratuit — le contexte est déjà empilé — ce qui n'est pas le cas pour le test de sous-typage ou l'accès à un attribut.

5.8.2 Variations autour des arbres binaires de sélection

De nombreuses variations ont été proposées autour des arbres binaires de sélection, tant au niveau de la forme (structure de peigne ou arbres binaires de sélection bornés),

qu'au niveau du principe (test d'égalité ou de sous-typage). Nous présenterons maintenant quelques unes des principales variations.

Tests d'égalité vs de sous-typage

Les comparaisons de types effectuées dans l'arbre de sélection peuvent se faire selon deux modalités, soit en comparant l'égalité des types, soit en faisant des tests de sous-typage [Queinnec, 1998]. Le première solution autorise des comparaisons rapides mais nécessite de considérer plus de branches dans l'arbre, à l'inverse de la seconde solution. Cependant le test de sous-typage implique généralement un accès à la table des méthodes, ce qui annule le principal bénéfice de cette technique. On pourrait considérer une implémentation hybride qui utiliserait des tests de sous-typage au lieu de tests d'égalité quand le rapport deviendrait avantageux — cependant ce rapport est très dépendant du processeur, ce qui nécessiterait un exécutable par famille de processeurs.

Structure de peigne

Le principe des arbres binaires de sélection n'est pas réservé à la structure d'un arbre binaire équilibré, l'utilisation d'une série de tests en peigne — type `switch` ou `if/elseif` — est envisageable. L'avantage majeur d'un arbre équilibré est que le temps nécessaire pour accéder à une feuille est constant quelle que soit la feuille — pour un arbre donné — ce qui n'est pas le cas des autres structures. Néanmoins, si les méthodes sont ordonnées par probabilités d'appels (ce qui nécessite un profilage statistique), la structure en peigne est plus efficace [Zendra et Driesen, 2002].

Le langage CEYX [Hullot, 1985] utilise cette structure de peigne couplée à un profilage dynamique. Au fur et à mesure de l'exécution du programme, les branches sont réorganisées en fonction de leur taux de déclenchement.

Arbres binaires de sélection bornés

Un compromis intéressant d'utilisation des arbres binaires de sélection peut être fait en ne les considérant que pour les sites d'appel oligomorphes, c'est-à-dire avec une profondeur maximale limitée (par exemple 3 ou 4). Dans le cas contraire, on propose d'utiliser la coloration de méthodes [6]. L'objectif étant d'utiliser les arbres binaires de sélection quand ils sont plus efficaces que la coloration. Avec cette proposition, la table des méthodes peut se restreindre à contenir les méthodes ayant au moins un site d'appel mégamorphe. Comme la coloration peut être utilisée, il est préférable d'utiliser l'adresse du pointeur sur la table des méthodes plutôt que l'identifiant de la classe pour faire les comparaisons.

6. N'importe quelle autre technique d'implémentation est utilisable, cependant dans ce contexte aucune n'est aussi efficace

On notera par la suite cette alternative BTD_i, avec des BTD de profondeur maximale i. Les BTD_0 correspondent aux appels statiques et les BTD_∞ correspondant à la proposition originale. Les arbres binaires de sélection bornés présentent l'intérêt d'être en temps constant.

5.9 Sous-typage multiple

En raison des difficultés d'implémentation de l'héritage multiple présenté dans ce chapitre et de celles liées à sa sémantique — notamment à cause des conflits —, le sous-typage multiple est proposé comme un compromis (voir Section 2.2.7). Il est constitué d'un cœur en héritage simple et d'un système de types en héritage multiple. La dichotomie entre classes et interfaces, apportée par le sous-typage multiple permet de profiter de l'implémentation du sous-typage simple qui est efficace pour les classes et de la puissance de la modélisation de l'héritage multiple, grâce aux interfaces.

Les interfaces étant un genre de classe abstraite et sans attributs, seul le test de sous-typage et l'envoi de message requièrent une implémentation. L'implémentation de ces mécanismes pour les interfaces est généralement considéré comme un problème annexe et peu de soin y est apporté — implémentation en temps non constant, etc.. D'ailleurs, on peut régulièrement lire : « Pour un code efficace, n'utilisez pas les interfaces ».

Nous présenterons dans cette section des implémentations efficaces, pour les deux mécanismes, dans ce contexte. Puis nous décrirons le cas des *mixins*, une autre alternative à l'héritage multiple, à travers l'exemple de SCALA (voir Section 5.9.3)

5.9.1 Tests de sous-typage alternatifs

Dans ce nouveau contexte le test de sous-typage peut être décomposé en deux tests distincts. Le *test primaire* lorsque la cible du test est une classe et le *test secondaire* lorsque la cible est une interface. Dans le cas où la cible est connue statiquement seul un des deux tests est nécessaire mais dans le cas où la cible n'est connue que dynamiquement, une combinaison de ces deux tests doit être faite. Comme les classes sont en héritage simple, le test de Cohen (voir Section 4.5.1) est généralement utilisé pour le test primaire. Le test secondaire est un véritable test en héritage multiple pour lequel, avant le hachage parfait de classes (voir Section 5.3), il n'existait aucune implémentation en temps constant et en espace mémoire raisonnable.

Nous présentons maintenant un ensembles de tests de sous-typage proposé pour le test secondaire — à l'exception de la coloration incrémentale qui à déjà été présentée dans la section 5.6.1.

LISTING 5.12 : Test de sous-typage utilisant des *trits*

```
load [objet + #tableOffset], table
load [table + #tritOffset], table
load [table + #tritSizeOffset], size
comp size, #targetID
ble maybe
load [table + #targetID], implements
comp implements, 1
blt no
bgt maybe
; succeed
```

Trits

Cette technique est une variante de la matrice ⟨classe, interface⟩. Elle repose sur l'utilisation d'un vecteur paresseux de *trits* par classe, vecteur indexé par l'identifiant de l'interface [Alpern *et al.*, 2001b]. À l'instar d'un bit, un *trit* est un mot valise formé par la contraction de *tertiary digit*, il représente donc une valeur à trois états. Chaque *trit* (entrée du vecteur) indique qu'une classe implémente, n'implémente pas ou implémente éventuellement une interface.

Chaque table de méthodes contient une référence à son vecteur de *trits*. Au départ le vecteur associé est partagé par chaque classe et ne contient que « peut-être ». Le test en lui même est implémenté comme une indirection dans le vecteur à la position de l'identifiant de l'interface, précédé d'un test de born. Si la réponse est « peut-être », l'information est recherchée — par un appel à la machine virtuelle non décrit — et stockée dans le vecteur. Le vecteur est réalloué au besoin, s'il est encore partagé ou s'il doit être élargi. Comme ce test est prévu pour une machine virtuelle avec compilateur à la volée, le test de bornes peut être éliminé si l'identifiant de l'interface est statiquement connu et inférieur à la taille du *trit* partagé — les lignes en italiques dans le Listing 5.12 peuvent être omises. Cette partie est appelée *portion rapide*, et l'algorithme d'allocation des identifiants d'interface assure que les interfaces importantes seront allouées dedans — `java.lang.Enumeration`, `java.lang.Serializable`, etc.

Cette technique repose sur le fait que le stockage du vecteur de *trits* est paresseux et que peu de classes seront sujettes au test de sous-typage par une interface. Dans le pire des cas, la taille du vecteur de *trits* est supérieure à celle d'un vecteur de bits — puisqu'il faut stocker trois états au lieu de deux [7] — et l'exécution est plus lente que celle d'un vecteur de bits — à cause de la mise à jour.

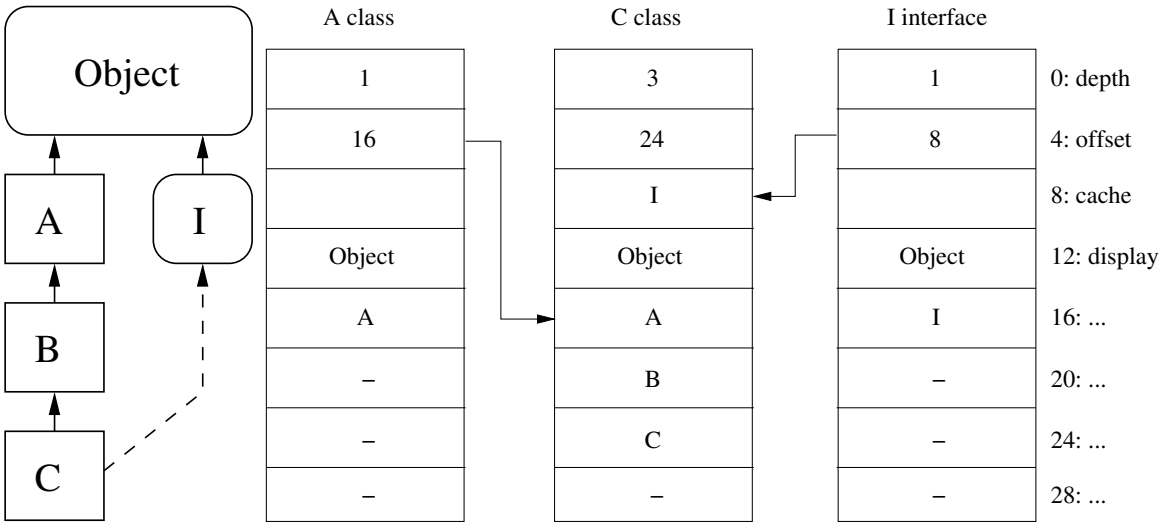
Les *trits* ont été proposés pour l'implémentation de la machine virtuelle JAVA d'IBM JALAPEÑO [8].

7. L'implémentation proposée utilise un octet par *trit* pour éviter le masquage/rotation
8. Depuis 2001, elle est devenue *open source* et a été renommée JIKES RVM— *Research Virtual Machine*.

Test de HOTSPOT

L'implémentation la plus souvent citée pour le test de sous-typage, dans le cadre du sous-typage multiple, est l'implémentation de Click et Rose [2002]. Cette implémentation propose un mécanisme unique valable pour les tests primaire et secondaire. Elle est basée sur une variante du test de Cohen [1991] (voir Section 4.5.1), où la taille de la table Γ_C est bornée par une constante — 5 originellement puis 8 pour des « raisons de robustesse » — et où cette table est fusionnée à la table de méthodes.

Chaque table de méthodes contient, en plus de la table Γ_C, une entrée qui indique la profondeur de la classe courante dans la relation de spécialisation. Une entrée qui indique où trouver l'identifiant de la classe courante — nommée offset — dans les tables des méthodes de ces sous classes — soit $3 * IntSize + Profondeur * IntSize$ si $Profondeur < TailleDeLaTable$, $PositionCache$ sinon. Et enfin, une entrée qui sert de cache.




```
load [objet + #tableOffset], table
load [table + #offsetTargetID], id
comp id, #targetID
beq sucess
comp offset, #cachePosition
beq fail
load [table + #classIdOffset], id
beq success
... ; scan_s_s_array not given in article
bne fail
store #targetID, [table + #cachePosition]
success:
```

FIGURE 5.9 : Test de sous-typage de HOTSPOT

Le test lui même est implémenté comme le test de Cohen [1991]. S'il échoue et que l'offset est différent de l'identifiant du cache — c'est-à-dire qu'il aurait dû être trouvé par le test de Cohen [1991] — une recherche séquentielle est effectuée et le cache est mis à jour.

Remarque. Ce test est présenté comme effectuant *généralement* au maximum deux références à la mémoire sur les benchmarks classiques — à l'époque SPECJVM98 et SPECJBB. Et comme utilisant de huit à onze mots par classes — donc un espace constant par classe.

En réalité, ce test était aussi efficace car les benchmarks de l'époque avaient des hiérarchies de classes très peu profondes — excédant rarement la taille de la table Γ_C — et les classes n'implémentaient que peu d'interfaces. Du point de vue temporel, le test peut réaliser nettement plus de chargements mémoire que ceux annoncés — à cause de la liste secondaire utilisée pour la recherche séquentielle. Quand à l'espace, il n'est clairement pas constant, leurs chiffres ne tiennent jamais compte de la taille de la liste secondaire.

Finalement, ce test repose sur le taux de réussite du cache et sur le faible nombre d'interfaces servant au test de sous-typage. Le passage à l'échelle de ce test est douteux, la taille de la table doit être augmentée en fonction de la profondeur de la hiérarchie de classe (pouvant atteindre 16 de nos jours en JAVA, voir la table 5.4). De plus, comme le nombre d'interfaces implémentées a explosé dans les dernières versions de JAVA, le risque de défauts de cache est augmenté d'autant.

5.9.2 Appel de méthode typé par une interface

Comme pour le test secondaire, l'envoi de message sur un receveur typé par une interface relève exactement des mêmes problèmes que l'héritage multiple.

Cependant comme le problème est limité aux méthodes introduites par des interfaces — et que ce nombre est nettement plus faible que celui du nombre de méthodes —, les stratégies peuvent être différentes. En toute généralité, on peut résoudre la liaison tardive en ayant une structure adéquate soit pour toutes les méthodes d'un coup, soit pour toutes les interfaces — en regroupant les méthodes par interface d'introduction. Dans le premier cas, cette structure d'association peut être une réalisé par une matrice directe — voir ci-après l'implémentation par *tables clairsemées* —, par une liste et une recherche séquentielle ou dichotomique ou par hachage parfait de méthodes. La recherche dans une liste ne semble pas être une alternative viable à la vue du nombre de méthodes concernés. L'expérience sur le hachage parfait de méthodes à montrée que les tables sont essentiellement vides [Ducournau et Morandat, 2011], ce qui entraînerait une perte d'espace non négligeable — toutefois les arguments en faveur des tables de méthodes clairsemées sont aussi valides dans ce cas (voir Section 5.11.3). Dans le second cas, le problème de l'invocation de méthode se ramène à trouver la position de l'interface dans une structure référencée par les classes. Comme dans le cas précédent, cette recherche peut être séquentielle (ou dichotomique). Néanmoins le nombre d'interfaces implémentées par une classe est bien plus faible que le nombre de méthodes qu'elles introduisent, la recherche est nettement

plus efficace. C'est une des techniques les plus utilisée pour l'implémentation des machines virtuelles, elle est appelée « implémentation classique » et sert souvent de référence aux auteurs pour leurs propositions d'implémentation de l'envoi de message typé par une interface. L'utilisation d'une table de hachage pour réaliser cette association revient exactement au hachage parfait de classes.

Dans le cas spécifique de JAVA, comme les méthodes introduites par plusieurs interfaces différentes sont unifiées. Lorsque l'association est réalisée par interfaces d'introduction, les tables associées aux interfaces peuvent contenir des doublons. Alors que nombreuses personnes crient au scandale à cause de ces doublons, nous pensons qu'ils sont négligeables — voire nécessaire. Les solutions proposées pour éliminer ces doublons reposent sur des optimisations plutôt que sur des implémentations générales. Lorsque ce n'est pas le cas, elles complexifient énormément le modèle sous-jacent.

Tables clairsemées

L'implémentation de la machine virtuelle SABLEVM [Gagnon et Hendren, 2001] utilise une matrice ⟨classes, méthodes⟩ pour l'envoi de message typé par une interface — les lignes de la matrice sont directement encodées dans les parties négatives des tables de méthode de chaque classe. Chaque entrée de colonne de la matrice est indexée par l'identifiant de la méthode. À cause de l'unification, l'identifiant d'une méthode n'est pas dépendant de la propriété globale mais il est construit à partir du nom de la méthode couplé à sa signature. Comme cette matrice est essentiellement creuse [9], les trous sont réutilisés par le gestionnaire de mémoire (voir Section 5.11.3). Cependant, la réutilisation des trous empêche de déduire un test de sous-typage de ce mécanisme d'invocation de méthodes.

Le principal avantage de cette technique est que le coût d'appel d'une méthode typée par une interface est identique au coût d'appel d'une méthode typée par une classe. Cependant, ceci n'est possible que grâce à l'unification et au fait qu'à l'origine les interfaces étaient très peu utilisées en JAVA.

Table de méthodes d'interfaces

Pour l'appel de méthode typé par une interface, la machine virtuelle JIKES RVM utilise des tables de hachage avec chaînage séparé compilé — cette technique est proposée sous le nom d'*Interface Method Table* (IMT) dans [Alpern *et al.*, 2001a]. Une table de hachage de taille fixe [10] est associée à chaque classe. Les identifiants de méthodes — noms et signatures des méthodes — sont ensuite hachés dans la table. En cas de collision dans la table de hachage, le chaînage est implémenté par *thunk* généré à la volée. Le contenu du *thunk* est un arbre binaire de sélection, où la clé est l'identifiant de la méthode.

9. Le taux de trous rapporté par Gagnon dans sa thèse est voisin des 80%
10. Initialement l'article proposait 5 et 40 comme taille, en pratique 29 est utilisé par la machine virtuelle.

De la même manière que les tables clairsemées, ceci fonctionne bien en pratique, sur les programmes où les appels de méthode typés par des interfaces sont peu utilisés ou pour les classes qui implementent peu d'interfaces. Sur les classes extrêmes des hiérarchies de la table 5.4, ce serait complètement inefficace. Comme, la taille des tables de hachage est fixé par une constante, ces tables sont essentiellement vides pour les classes où il y a peu de méthodes introduites par des interfaces, tandis que pour les autres il y aurait beaucoup de collisions. Finalement, l'alternative qui consisterait à faire varier la taille de ces tables revient au hachage parfait de méthodes et dont l'expérimentation à montrée que la taille des tables est déraisonnable [Ducournau et Morandat, 2011].

5.9.3 Mixins : le cas SCALA

Le principe du sous-typage multiple peut être généralisé à toutes les hiérarchies de classes partitionnées en deux catégories. Les classes *primaires* qui sont des classes standard mais en héritage simple et des classes *secondaires* en héritage multiple avec des restrictions.

Les *mixins* — ou *traits* — ont été proposés par de nombreux auteurs avec des définitions légèrement différentes, allant des définitions formelles aux définitions informelles [Stefik et Bobrow, 1986; Bracha et Cook, 1990; Ancona *et al.*, 2002; Ernst, 2002; Ducasse *et al.*, 2005]. On peut cependant résumer le concept comme des *classes abstraites* moins abstraites que les interfaces de JAVA— c'est-à-dire contenant du code.

Nous présentons ici seulement la définition utilisée par le langage SCALA [Odersky *et al.*, 2008]. Les classes sont en héritage simple mais peuvent avoir un nombre quelconque de *mixins* . Un *mixin* peut spécialiser un nombre quelconque de *mixins* mais ne peut avoir qu'une super-classe. Si une classe C spécialise une classe B avec le *mixin* M, pour respecter la relation d'héritage simple les super-classes de M doivent être des super-classes de B (Figure 5.10). En plus, par rapport aux interfaces les *mixins* peuvent définir du code pour les méthodes et introduire des attributs. Par opposition à une classe abstraite, qu'un *mixin* soit instanciable ou pas est relativement sans objet dans ce contexte, il est toujours possible de définir une classe M' utilisant le *mixin* M et ne rajoutant rien. Cependant, il semblerait préférable que le compilateur autorise l'instanciation du *mixin* et qu'il génère automatiquement cette classe M', évitant de ce fait de multiples définitions de cette classe par différents programmeurs.

La compilation d'un *mixin* est une projection sur l'espace du sous-typage multiple — à la JAVA. Le *mixin* M est compilé en une classe abstraite C_M pour le corps des méthodes et une interface I_M pour les signatures des méthodes. Les méthodes de M sont définies en tant que méthodes statiques de la classe C_M avec un paramètre supplémentaire, le receveur. La classe C qui utilise le *mixin* M, implémente en fait l'interface I_M et pour toutes les méthodes définies par M, une méthode du même nom est définie dans C — le corps de la méthode faisant un appel à la méthode statique de C_M avec comme premier paramètre `self`.

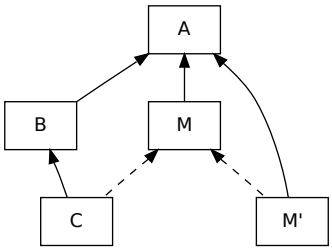


FIGURE 5.10 : Un *mixin* en SCALA

Pour les attributs introduits par le *mixin* M, deux signatures d'accesseurs sont introduites dans l'interface I_M. Dans la classe C, les attributs de M sont recopiés — en fait ils sont introduits, puisqu'il n'y a pas d'attribut dans la classe C_M — et les accesseurs sont générés dans C. Enfin, les appels à un attribut sont faits au travers des accesseurs si le receveur est typé par le *mixin*, dans les autres cas un accès direct est possible.

D'autres implémentations sont possibles, par exemple en utilisant l'implémentation *hétérogène* des génériques (voir Section 5.10.3).

5.10 Autres mécanismes

En plus des trois mécanismes de base, la compilation des langages objets fait intervenir d'autres mécanismes.

Sans être exhaustif, nous présentons dans cette section des implémentations pour certains d'entre eux. Ces mécanismes n'étant pas le cœur de cette thèse, ils seront traités moins en profondeur.

5.10.1 Types virtuels

Les types virtuels sont une alternative élégante à la covariance (voir Section 2.2.3). Ce *type* de propriétés ne rajoute pas de nouveau mécanisme, il nécessite seulement un test de sous-typage dynamique — la cible du test est inconnue.

Pour la majorité des techniques d'implémentations, un type virtuel est une pseudo-méthode — c'est-à-dire possède une entrée dans la table des méthodes. La valeur de cette pseudo-méthode est l'identifiant de type associé au type virtuel. Ensuite, un test de sous-typage dynamique, avec comme cible l'identifiant, est réalisé.

Le cas de la coloration est un peu différent, les entrées doivent être doublées (éventuellement elles peuvent être compressées s'il s'agit de petites entiers). Une des deux entrées détermine la position où trouver la couleur et la seconde contient la valeur couleur à trou-

ver. Il faut cependant faire attention au codage des valeurs pour ne pas entrer en conflit avec le test de Cohen (voir Section 4.5.1). Pour cela, les valeurs peuvent être systématiquement rendues paires à l'aide d'un décalage (*shift*). Une alternative consisterais à s'assurer que la position soit paire, en rajoutant une entrée vide au besoin.

5.10.2 Types primitifs polymorphes

L'intégration des types primitifs dans la hiérarchie de classes augmente le pouvoir expressif des langages au prix d'une certaine complexification pour le compilateur ainsi que d'une perte d'efficacité.

Boxing et *Tagging*

Étant donné que les types primitifs sont sous-classes d'`Any`, les collections peuvent être entièrement génériques (paramétrées par `Any` qui est valable pour les types primitifs comme pour les « vrais » objets). Cela impose au compilateur de rajouter des boîtes pour les méthodes utilisant des primitifs. De l'autre côté, quand on récupère une valeur primitive d'une collection générique, le compilateur doit déboiter les valeurs.

Pour améliorer les performances, c'est-à-dire éviter un boitage/déboitage inutile, certains types sont *taggués*. Cette astuce empruntée aux langages à typage dynamique consiste à se servir de certains bits pour marquer les types primitifs (et donc se passer des boîtes). Cette technique a un coût, car elle impose de vérifier si le type d'une référence vers un objet est un type taggué avant d'envoyer un message. La question se pose aussi pour les comparaisons de références, avec une difficulté supplémentaire : deux boîtes de même contenu sont égales (voir section suivante).

Remarque. Le `tagging` est activé par l'option `-tagging` (par défaut) du compilateur. Les types taggués du langage sont : `Int`, `Bool`, `Char`. Ce qui nécessite deux bits ; les deux bits utilisés sont les bits de poids faibles car les références sont alignées au double mot (donc multiples de quatre).

Égalités

L'implémentation du test d'égalité est particulièrement simple quand on a seulement affaire à des références ou seulement à des types primitifs. Dès lors que les types primitifs sont intégrés à la hiérarchie de classes, cette simplicité disparaît, car deux boites même différentes sont identiques si elles ont le même contenu. De plus, avec le *tagging* la combinatoire en est encore augmentée.

La compilation des comparaisons est une étude de cas sur le type des paramètres. Nous traiterons ici seulement le cas *tagging* et *boxing* simultanés, le cas sans *tagging* se déduisant aisément.

LISTING 5.13 : Comparaison entre un deux objets quelconques avec invariant de référence

```
(defun equal-any-any (a b)
    (cond ((eq a b))
          ((or (tagp a) (tagp b)) ())
          ((and (boxp a) (eq-box a b)))))
```

LISTING 5.14 : Comparaison entre un deux objets quelconques sans invariant de référence avec le même type statique

```
(defun (equal-any-any a b)
    (cond ((eq a b))
          ((or (tagp a) (tagp b)) ())
          ((boxp a) ((eq-box (downcast a) (downcast b))))))
```

LISTING 5.15 : Comparaison entre un deux objets quelconques sans invariant de référence avec des types statiques différents

```
; Un des deux a forcement un type statique different de any
; nous supposerons qu'il s'agit de b.
(defun equal-any-other (a b)
    (cond ((tagp a) (eq a b))
          ((tagp b) ())
          (t (let ((aa (downcast a))
                   (bb (downcast b)))
                   (or (eq aa bb)
                       (and (boxp aa) (eqbox aa bb)))))))
```

Le cas des implémentations sans invariant de référence 4.1, comme les sous-objets, rajoute encore de la combinatoire. Il faut différencier le cas où les deux objets ne pointent pas sur le même sous-objet (voir Section 5.1.2), en faisant attention chaque fois à ne pas faire d'indirection sur un non objet — soit un élément taggué.

5.10.3 Généricité

Il existe deux grandes familles d'implémentations de la généricité (voir Section 2.2.4). L'implémentation homogène — utilisée en JAVA ou PRM— et l'implémentation hétérogène — utilisée en C++. Un compromis des deux implémentations est utilisé par C#, nous la nommerons implémentation semi-hétérogène.

Implémentation homogène

L'implémentation homogène repose sur le fait que les types formels sont bornés, c'est-à-dire qu'ils sont contraints à être sous-types d'un type donné — `Any` est utilisé par défaut. Pour que la classe soit valide, il faut que les appels sur les types formels soient compatibles avec le type de la borne. La compilation d'une classe paramétrée procède alors par *effacement de types*, c'est-à-dire que toutes les annotations de types formels dans le corps de la classe paramétrée sont remplacées par le type la borne. Lors de l'utilisation de la classe, le type des paramètres formels est remplacé par le type concret — fourni à l'instanciation.

Cette implémentation a de nombreux avantages. Elle est compatible avec la compilation séparée (voir Sections 6.4.1, 6.4.2) et est particulièrement simple à implémenter. C'est d'ailleurs la principale raison de l'utilisation de cette implémentation en JAVA. En effet, le code fonctionnant sans généricité est toujours fonctionnel avec les versions génériques des classes — en ignorant les paramètres génériques. De fait, cette implémentation ne nécessite aucun changement de la machine virtuelle JAVA— aucune instruction n'est ajoutée au niveau du *bytecode*. Enfin, la classe paramétrée n'est compilée qu'une seule fois, indépendamment de son nombre d'instanciations, et la taille du code nécessaire pour la classe est identique à une version non paramétrée — c'est le même code.

Cependant, quand la classe paramétrée est instanciée avec des types primitifs, le corps de la classe utilise systématiquement des boites (voir Section 5.10.2). Ceci provoque une dégradation non négligeable de l'efficacité. Cet argument est particulièrement flagrant en PRM où même les tableaux sont implémentés de cette manière — le nombre de boites est très élevé et, sans *tagging*, les temps d'exécution deviennent déraisonnables.

Implémentation hétérogène

L'implémentation hétérogène s'apparente plus à de la transformation source-à-source. Le code de la classe paramétrée n'est pas compilé lors de sa définition. Lors de l'instanciation de la classe paramètrée par des types concrets, le code de la classe est recopié et les paramètres concrets sont substitués aux paramètres formels. Le programme final contiendra donc une version de la classe pour chaque ensemble de paramètres. C'est exactement le cas pour les `templates` [11] de C++.

Cette implémentation est particulièrement efficace si les paramètres sont instanciés par des types primitifs. De plus, comme elle permet de faire du code spécifique, cette implémentation peut servir à faire de la *customization* et éviter un grand nombre de délégations. Elle autorise aussi une implémentation des *mixins* comme des *parametrized heirs classes* [Bracha et Cook, 1990] — où le code du *mixin* est recopié dans chaque classe qui l'utilise. La librairie STL de C++ l'utilise intensément pour les *allocateurs*, les conteneurs ou les algorithmes.

11. Classe paramétrée dans le jargon C++.

Cependant, cette alternative présente aussi un grand nombre d'inconvénients. Tout d'abord le code est dupliqué pour chaque instanciation, ce qui n'améliore pas les performances des types véritablement objets mais augmente sensiblement la taille du code. De plus, le code ne peut pas être vérifié lors de la compilation — une alternative consisterait à borner les paramètres formels et à vérifier les types comme en effacement de types, mais cela resterait une pseudo-compilation puisqu'aucune génération de code ne peut être faite.. Enfin, ce type d'implémentation est peu compatible avec la compilation séparée, il faut obligatoirement disposer du source des classes paramétrées pour pouvoir le recopier — une alternative serait de considérer un format de représentation interne.

Implémentation semi-hétérogène

Le langage C# propose une implémentation alternative intéressante. Les classes paramétrées sont compilées en *bytecode CIL* comme dans l'implémentation homogène en utilisant l'effacement de type. Lors de leurs instanciations une table des méthodes sur mesure est générée pour chacune et le *bytecode* est compilé en langage machine.

L'implémentation retenue pour C# rajoute le préfixe `constrained` aux *bytecodes* d'invocation de méthodes. Ce préfixe est traité différemment, selon que le receveur est un type primitif ou un type normal. Pour être efficace, le code de la classe doit être traité spécialement (*customize*) par le compilateur à la volée quand le préfixe est typé par un type primitif. Cependant ceci n'est réalisable que grâce au niveau de traduction supplémentaire du *bytecode* au code machine. Dans le cas où le type est une classe normale, le code est traité comme dans l'implémentation homogène.

5.11 Évaluations abstraites

Cette section propose un résumé de l'évaluation des principales techniques d'implémentation décrites dans ce chapitre. Les techniques sont évaluées en fonction de leur efficacité temporelle et de l'occupation mémoire.

L'efficacité temporelle est considérée d'après le modèle de calcul de Driesen [2001] (voir Section 4.2.1) et des séquences de codes présentés dans ce chapitre. Le résultat est cette évaluation *a priori* est résumé par la table 5.1.

5.11.1 Évaluation du coût spatial

L'espace est considéré d'un point de vue statique, c'est-à-dire la taille des structures et la longueur du code supportant les mécanismes élémentaires, et d'un point de vue dynamique, c'est-à-dire la taille des instances.

TABLE 5.1 : Nombre de cycles pour les différentes techniques et pour chaque mécanismes

technique	appel de méthode		test de sous-typage		accès à un attribut	
Sous-typage simple	$2L+B$	16	$2L+2$	8	L	3
Coloration	$2L+B$	16	$2L+2$	8	L	3
avec simulation					$3L+1$	10
Sous-objets	$2L+B+2$	18	$4L+4$	16	L	3
avec ajustement					$3L+1$	10
Coloration incrémentale	$4L+B+3$	25	$6L+6$	24	$4L+3$	15
Coloration partagée	$2L+B+2$	18	$2L+4$	10	$3L+3$	12
BTD_0	1	1				
BTD_n	$L+\lceil\log(n)\rceil+2$	$5+\lceil\log(n)\rceil$				
PH-`and`	$4L+B+3$	25	$3L+5$	14	$4L+3$	15
PH-`and+shift`	$4L+B+4$	26	$3L+6$	15	$4L+4$	16
PH-`mod`	$4L+B+D+2$	49	$3L+D+4$	38	$4L+D+2$	39
Cache (réussite)	$4L+B+3$	25				
Cache (échec)	$X+3L+B+3$	$X+22$				
Cache PH-`mod` (échec)	$+3L+B+3$	$+22$				

Cette table résume le nombre de cycles et la longueur du code nécessaire pour chaque technique d'implémentation en fonction du mécanise considéré. Le nombre de cycles est compté avec le modèle de calcul de Driesen [2001]. L, B et D représentent les latences respectives d'un chargement mémoire, d'un branchement indirect ou mal prédit et de la division entière. Les valeurs considérés ici sont $L = 3$, $B = 10$ et $D = 25$.

Espace dynamique

L'espace occupé dynamiquement dépend du nombre d'instances allouées par un programme. Pour toutes les techniques d'implémentation, à l'exception des sous-objets et de la coloration d'attributs, la taille d'une instance est identique à la taille d'une instance en sous-typage simple, c'est-à-dire un pointeur vers la table des méthodes plus tous les attributs connus par la classe. Dans le cas d'une implémentation par arbres binaires de sélection pure, comme SMART EIFFEL, le pointeur est remplacé par l'identifiant de la classe.

Sans l'utilisation de la simulation des accesseurs, la coloration peut rajouter des trous dans les instances. Ce nombre de trous peut être grand — d'après [Ducournau, 2010], il peut atteindre 800% sur certaines instances — mais en pratique le nombre de trous est modéré.

Pour l'implémentation par sous-objets de C++, la taille d'une instance est la somme de tous ses sous-objets. La taille d'un sous-objet est la somme de tous les attributs qu'il introduit plus un pointeur. Le nombre de sous-objets pour une classe est le nombre de ses super-classes plus elle même. Les statistiques de [Ducournau, 2011] montrent que le nombre de pointeurs pour une classe peut être nettement plus grand que le nombre d'attributs de la classe. Il est possible de réduire ce nombre grâce à l'optimisation des sous-

objets vides (voir Section 5.1.4), qui permet de limiter approximativement le nombre de pointeurs au nombre de super-classes qui introduisent au moins un attribut.

Espace statique

Hormis les séquences de code, résumées dans la table 5.2, l'espace statique nécessaire pour une technique d'implémentation donnée est constitué des structures supportant les mécanismes élémentaires, soit principalement les tables de méthodes. Pour toutes les implémentations utilisant un groupe de méthodes, soit l'implémentation du sous-typage simple, le hachage parfait de classes et la coloration (et ses variantes), la taille d'une table de méthodes pour une classe est limitée au nombre de méthodes connues par la classe et le nombre de tables est le nombre de classes. Si on suppose que les méthodes sont équitablement introduites par classe, la taille totale des tables est donc quadratique dans le nombre de classes.

Cependant pour le hachage parfait de classes, il faut ajouter à chaque table de méthode une table de hachage. Dans le pire des cas, la taille de cette table serait quadratique dans la relation de spécialisation mais les statistiques effectuées sur de nombreuses hiérarchies de classes montrent que la taille de ces tables est proportionnelle à la relation de spécialisation — entre 1.5 et 3 fois avec PN-`and` [Ducournau et Morandat, 2011]. Pour comparaison, un taux d'occupation de 2 donne une table de hachage avec un risque de collision très faible [Knuth, 1973].

Les sous-objets utilisent une table de méthodes pour chaque type statique par type dynamique, le nombre de tables n'est donc plus le nombre de classes mais le cardinal de la relation de spécialisation — donc dans le pire des cas quadratique dans le nombre de classes. Pour une classe, ce cas correspond à une chaîne de la racine à une classe. Dans le pire des cas la taille totale des tables est donc cubique dans le nombre de classes. Ce cas correspond à une hiérarchie dégénéré de classes qui ne serait qu'une chaîne. De plus, il faut rajouter le coût du test de sous-typage. Comme nous utilisons le hachage parfait de classes, il s'agit de la taille des tables de hachage.

La taille d'une table de méthodes en utilisant la coloration est, dans le pire des cas, quadratique dans le nombre de classes. Ce qui correspond à une hiérarchie de classes fort peu réaliste dans laquelle il n'y aurait une racine, une unique feuille et toutes les autres classes entre les deux au même niveau. En pratique, les statistiques ne montrent qu'un surcoût faible par rapport au sous-typage simple. Il en va de même pour la table des couleurs de la coloration incrémentale.

Les arbres binaires de sélection purs n'utilisant pas de table de méthodes, nous ne les considérons pas dans cette évaluation — une comparaison nécessiterait d'inclure le code.

TABLE 5.2 : Espace statique

technique	séquence de code méth.	st.	attr.	taille des tables	algo.
Sous-typage simple	3	4	1	$\sum_C M_c$	-
Coloration	3	4	1	$\sum_C M_c$ + trous	$\mathcal{O}(N^3)$
avec simulation			4		
Sous-objets	5	8	1	$\sum_C \sum_{C \preceq D} M_c$	PH-and
avec ajustement			4		
Coloration incrémentale	8	10	7	$\sum_C M_c + 3 * \text{trous}$	$\mathcal{O}(N^3)$
Coloration partagée	5	7	7	Coloration + tables locales	$\mathcal{O}(N^3)$
PH-and	8	7	7	$\sum_C M_c$ + table de hachage	$\mathcal{O}(N^2 \log N)$
PH-and+shift	10	10	10	$\sum_C M_c$ + table de hachage	
PH-mod	8	7	7	$\sum_C M_c$ + table de hachage	
BTD_0	1			Complété par coloration	
BTD_n	$3 \times 2^n - 1 + L$				
$Cache_n$	$X+11$	$X+11$	$X+11$	$+3 * \sum_C n$	-
$Cache_n$+ PH-mod	16	15	15	PH-mod $+3 * \sum_C n$	-

Cette table résume la longueur du code nécessaire pour chaque technique d'implémentation en fonction du mécanise considéré. La dernière colonne donne la taille totale des tables de méthodes avec M_c le nombre de méthodes introduites par classes. Le nombre de trous entre la coloration et la coloration incrémentale ne sont pas calculés sur les mêmes ensembles.

Statistiques sur les hiérarchies de classes

Les statistiques abstraites citées dans ce chapitre sont calculées sur divers *benchmarks* objet classiques utilisant l'héritage ou le sous-typage multiple. Les *benchmarks* sur l'héritage multiple sont complétés par des *benchmarks* sur le sous-typage multiple (voir Section 2.2.7) — limité à JAVA. Des statistiques sur tous ces *benchmarks* sont données dans les tables 5.3 et 5.4.

Ces statistiques sont présentées pour plusieurs raisons : tout d'abord parce que l'héritage multiple est en pratique utilisé, puis pour montrer l'augmentation de la taille des hiérarchies de classes et enfin car ces statistiques représentent le corpus des hiérarchies de classes utilisées pour calculer les statistiques citées.

Ces statistiques justifient d'abord le fait que l'héritage multiple, même sous sa forme dégradée, est effectivement utilisé. Il est par conséquent nécessaire de proposer une implémentation efficace dans le pire des cas.

On peut constater, de plus, que la hiérarchie de classes standard de JAVA a littéralement explosé, elle passe de 604 classes dans sa version 1.0.2 à 7401 classes dans sa version 1.3.1. Un implémentation réelle de l'API complète peut même atteindre les 17000 classes, auxquelles se rajoutent les classes des applications — ECLIPSE, par exemple, et ses presque

TABLE 5.3 : Statistiques sur le nombre de classes et de super-classes, d'après [Ducournau et Morandat, 2011]

	class number	avg n_c
IBM-SF	8793	9.2
JDK1.3.1	7401	4.4
Java.1.6	5933	4.3
Orbix	2716	2.8
Corba	1699	3.9
Orbacus	1379	4.5
HotJava	736	5.1
JDK.1.0.2	604	4.6
Self	1802	30.9
Geode	1318	14.0
Vortex3	1954	7.2
Cecil	932	6.5
Dylan	925	5.5
Harlequin	666	6.7
Unidraw	614	4.0
PRMcl	479	4.6
Lov-obj-ed	436	8.5
SMART EIFFEL	397	8.6
Total	38784	7.3

Les colonnes représentent dans l'ordre, le nom du *benchmark*, le nombre de classes totales et le nombre moyen de super-classes par classe.

60000 classes [12], (Table 5.4). Ceci confirme l'importance de la prise en compte du passage à l'échelle pour une technique donnée.

Les informations sur les hiérarchies JAVA sont extraites des SPECJVM 2008 et de DACAPO [13]. Les extractions sont obtenues à travers l'API `reflect` de JAVA par l'outil SCHEMEBUILDER [14].

Les différentes statistiques présentées ici proviennent de [Ducournau et Morandat, 2011].

12. Toutefois, dans sa version *benchmark*, ECLIPSE ne compte que 870 classes. Cette version impressionnante d'ECLIPSE est extraite d'une installation ayant beaucoup de *plugins*.

13. http://www.spec.org/jvm2008/ et http://www.dacapobench.org/

14. Cet outil génère des hiérarchies de classes en utilisant l'introspection de JAVA et fournit des résultats dans une variétées de formats (http://www.lirmm.fr/~morandat/index.php/Benchmarks/Benchmarks).

TABLE 5.4 : Statistiques sur le nombre de classes et d'interfaces pour les *benchmarks* JAVA, d'après [Ducournau et Morandat, 2011]

	class numbers			interface	total		extends		implements	
	all	concrete	abstract	number	avg	max	avg	max	avg	max
eclipse	59690	56117	3573	8122	5.6	36	3.5	16	2.1	28
java-6-sun	17213	15777	1436	2257	4.5	29	3.1	10	1.4	20
Total	76903	71894	5009	10379	5.4	36	3.4	16	2.0	28
derby	1764	1528	236	453	5.2	20	3.1	8	2.0	14
jfreechart-1.0.5	1393	1215	178	382	5.3	19	3.1	8	2.2	13
jess	1289	1137	152	329	4.3	16	2.8	7	1.6	9
sunflow	1140	982	158	283	4.3	15	2.8	7	1.5	9
ant	1022	895	127	187	4.2	14	3.2	8	1.1	8
xom-1.1	898	796	102	262	4.2	12	3.0	9	1.3	7
javac	864	770	94	207	4.3	12	3.0	6	1.4	8
Tidy	793	679	114	230	4.1	11	2.8	7	1.3	7
jl1.0	778	655	123	215	4.1	12	2.8	7	1.3	8
janino	783	668	115	132	4.4	13	3.2	7	1.2	9
scheme-builder	340	275	65	82	4.4	12	3.0	6	1.4	8
check	326	266	60	75	4.3	11	3.0	6	1.4	7
Total	11390	9866	1524	2837	4.5	20	3.0	9	1.6	14
jython	3715	3494	221	347	6.3	14	4.4	8	1.8	8
fop	2669	2338	331	633	4.3	27	3.1	9	1.3	21
xalan	2562	2309	253	490	5.2	18	3.2	9	2.0	13
pmd	1747	1548	199	397	4.7	18	3.2	8	1.5	12
antlr	1417	1239	178	302	4.2	15	3.0	7	1.2	10
eclipse-dacapo	872	759	113	226	3.9	14	2.7	6	1.2	8
luindex	452	356	96	89	4.1	12	2.9	6	1.2	7
lusearch	396	316	80	86	4.2	11	2.9	6	1.3	7
Total	13830	12359	1471	2570	5.0	27	3.5	9	1.6	21

Ces *benchmarks* sont regroupés, de haut en bas, en trois catégories : les librairies complètes, SPEC-JVM 2000 et DACAPO. Les premières colonnes représentent le nombre de classes et d'interfaces contenues dans le *benchmark*. La section donne les moyennes et les maximaux du nombre de classes spécialisées et du nombre d'interfaces implémentées. La colonne total ne distingue pas les classes des interfaces et représente la hiérarchie comme si elle était seulement en héritage multiple.

5.11.2 Coût de l'héritage multiple

L'utilisation de l'héritage multiple, n'est pas anodine. Elle rajoute un pouvoir expressif mais souvent au détriment de l'efficacité spatiale ou temporelle.

La coloration a la même efficacité temporelle que l'implémentation du sous-typage simple mais ne permet pas l'hypothèse du monde ouvert. L'efficacité temporelle de cette implémentation est obtenue en utilisant plus d'espace si et seulement si l'héritage multiple est utilisé. La coloration bidirectionnelle limite encore cette perte, elle annule totalement le coût de l'héritage multiple si le graphe de conflit est biparti.

Les implémentations comme le hachage parfait de classes ou la coloration incrémentale, rajoutent un surcoût temporel constant à tous les mécanismes. Cependant ce coût est modéré — autour de 2 fois le coût de l'implémentation du sous-typage simple— pour l'appel de méthode et le test de sous-typage. L'accès aux attributs est quand à lui très coûteux — 5 fois l'implémentation du sous-typage simple. De plus, ces implémentations nécessitent des structures annexes — table de hachage ou de couleurs.

L'implémentation par sous-objets rajoute un coût systématique pour l'héritage multiple, même lorsque celui ci n'est pas utilisé. Cependant, par rapport à toutes les autres implémentations, elle rajoute en plus un surcoût à l'abstraction. En effet, le simple fait de couper une classe en deux rajoute des ajustements de pointeurs ainsi qu'une table de méthodes pour le nouveau type statique (voir Section 4.7.3).

5.11.3 Réutilisation de l'espace

Certaines des techniques présentées introduisent un gaspillage d'espace — la coloration (voir Section 5.2), le hachage parfait de classes (voir Section 5.3) ou la coloration incrémentale (voir Section 5.6.1). Cependant cet espace peut être réutilisé en prenant certaines précautions.

Si la technique est utilisée pour l'appel de méthode, comme les appels sont sûrs en typage statique, les entrées inoccupées peuvent être réutilisées pour stocker d'autres données statiques — code, références statiques, etc. C'est d'ailleurs la justification donnée par Gagnon et Hendren [2001] pour l'utilisation des tables clairsemées de l'implémentation de SABLEVM (voir Section 5.9.2).

Dans le cas du test de sous-typage, une valeur est nécessaire pour représenter l'échec. La réutilisation de l'espace perdu est donc une question d'encodage. Sans précautions particulières sur l'encodage des identifiants, toutes les entrées sont nécessaires et sont réellement perdues. En considérant que les identifiants utilisés avec la coloration soient codés par des nombres impairs, n'importe quel nombre pair peut représenter un échec. De ce fait, on peut utiliser les trous pour stocker n'importe qu'elle information paire. Comme les références à des objets sont toujours multiples de quatre — alignement au double-mot — elles peuvent y être stockées. Les attributs statiques de classes — en JAVA ou C#— sont donc de bons candidats à la réutilisation de l'espace [Sallenave et Ducournau, 2010].

Le cas de hachage parfait de classes avec la fonction de hachage and est particulier. Par définition de cette fonction de hachage, certaines plages sont réellement inaccessibles, les entrées qui y sont associées peuvent donc être utilisées pour stocker n'importe quelle donnée statique. Une estimation de l'espace qui peut être ainsi réutilisé est donnée par la proposition 3.11 de [Ducournau et Morandat, 2011].

De plus quand le hachage parfait de classes et la coloration incrémentale servent à l'invocation de méthodes (resp. à l'accès aux attributs) la partie des entrées associées aux groupes de méthodes (resp. d'attributs) est réutilisable pour stocker n'importe quelle donnée statique.

5.12 Conclusions

Nous avons présenté dans ce chapitre les principales implémentations, en temps constant, de l'héritage multiple — et du sous-typage multiple — en typage statique. L'implémentation de l'héritage multiple n'est pas un problème trivial et nécessite des structures de données et des algorithmes adéquats. Le maintien strict des deux invariants (Invariants 4.1 et 4.2) nécessite la connaissance entière de la hiérarchie de classes mais il offre une implémentation aussi efficace que celle du sous-typage simple (voir Chapitre 4) avec la coloration (voir Section 5.2). Avec l'hypothèse du monde clos et en utilisant des arbres binaires de sélection (voir Section 5.8), il est même possible d'être plus efficace que le seul appel indirect utilisé par la coloration, en se servant des prédictions de branchement.

L'hypothèse du monde ouvert complique la situation et alourdit invariablement les séquences de code des mécanismes, qui doivent faire appel à des structures auxiliaires — qu'il s'agisse d'une table de hachage pour le hachage parfait de classes (voir Section 5.3) ou de couleur pour la coloration incrémentale (voir Section 5.6.1), ou encore d'une multiplication des tables de méthodes pour les sous-objets (voir Section 5.1). De plus, ce surcoût est systématique, que l'héritage multiple soit utilisé ou pas. L'abandon de l'invariant de référence (Invariant 4.1) par les sous-objets rajoute des ajustements de pointeurs à cette complexification. Les autres implémentations le conservent mais doivent relaxer l'invariant strict de position (Invariant 4.2) et les positions ne sont invariantes que dans leur groupe.

Le cas du sous-typage multiple (voir Section 5.9) est équivalent à celui de l'héritage multiple, bien qu'il en limite l'usage aux seules interfaces pour l'invocation de méthode et le test de sous-typage. Les préoccupations dans ce contexte ne sont pas différentes, à l'accès aux attributs près, et les techniques d'implémentation compatibles avec l'héritage multiple le sont aussi dans ce contexte limité. Une évaluation d'une implémentation du sous-typage multiple peut être extrapolée de celle de l'héritage multiple en utilisant à la simulation des accesseurs (voir Section 5.5). Bien que, dans ce cas, les interfaces sont nettement plus utilisées qu'en vrai sous-typage multiple.

Le caractère incrémental ou non incrémental des techniques considérées est un élé-

ment essentiel de leur évaluation. Il apparait que chaque technique peut être utilisée dans un cadre de compilation bien spécifique, dont l'analyse fait l'objet du chapitre suivant.

CHAPITRE

6 Schémas de compilation

Les schémas de compilation représentent les différentes possibilités pour réaliser la chaîne de production d'un programme exécutable*, du code source à l'exécution du programme. Elle est composée de plusieurs* tâches *distinctes qui incluent, de façon non limitative, la compilation, le chargement des unités de code ainsi que l'édition de liens. Les schémas de compilation sont étroitement reliés aux hypothèses faites sur le monde, de l'hypothèse du monde ouvert à l'hypothèse du monde clos.*

Sommaire

6.1 Compilation et schéma de compilation

En toute généralité, la *compilation* consiste à transformer tout ou partie d'un programme d'un langage source dans un langage cible tout en préservant la sémantique du programme initial. En général, le langage source est de plus haut niveau que le langage cible — c'est-à-dire d'un niveau d'abstraction supérieur.

Attention. Le mot *compilation* représente à la fois le processus de compilation — c'est-à-dire celui qui transforme un programme d'un langage source vers un langage cible tout en respectant la sémantique du code source — ainsi qu'une des tâches du processus de compilation.

Le mot *compilation* vient probablement du latin *compilare* et désigne le fait de regrouper et combiner ensemble des éléments d'origines diverses (par exemple *compiler une encyclopédie*).

Pour éviter les ambiguïtés, dans ce chapitre, nous utiliserons le terme non qualifié *compilation* pour se référer au processus global. Tandis que nous utiliserons plutôt le terme en *génération de code* pour parler de l'étape au sein du processus global.

Les *schémas de compilation* constituent la chaîne de production d'un exécutable. Cette chaîne de production met en jeu différents outils tels que les compilateurs, les éditeurs de liens, les machines virtuelles, les compilateurs *just-in-time*, les interprètes, etc.

Le choix d'un schéma de compilation particulier pour un programme exécutable donné est conditionné par les exigences fonctionnelles souhaitées — par exemple le chargement dynamique, modularité du code, l'espace mémoire disponible, etc. D'un côté la modularité prônée par le génie logiciel est mieux exprimé sous l'hypothèse du monde ouvert qui permet des recompilations plus rapides, des vérifications modulaires et le chargement dynamique. Tandis que de l'autre côté, les techniques les plus efficace ne sont disponibles que dans les schémas contraints par l'hypothèse du monde clos. Le choix d'un schéma de compilation restreint donc l'ensemble des techniques d'implémentation utilisables (voir Section 6.5). Entre les deux extrêmes — hypothèse du monde ouvert stricte ou hypothèse du monde clos permanente — se situent un ensemble de schémas proposant des compromis entre efficacité et modularité du code produit.

Remarque. Bien que les spécifications des langages de programmation soient en principe indépendantes de leur implémentation et de leur schéma de compilation, de nombreux langages sont en fait indissociables de ceux-ci. Par exemple, la covariance non contrainte d'EIFFEL n'est pas implémentable efficacement sans compilation globale et la règle des *catcalls*[1] n'est même pas vérifiable en compilation séparée [Meyer, 1997]. Il en va de même pour JAVA et le chargement dynamique.

Dans cette thèse nous nous intéressons seulement aux schémas de compilation ne nécessitant aucune recompilation lors de l'exécution d'un programme, nous qualifierons ces schémas de *schémas de compilation statique*.

6.2 Phase locale et phase globale

Cette section décrit comment les différentes tâches du processus de compilation sont organisées dans le temps et dans l'espace du programme. Sans perte de généralité, on retrouve deux types de phases, une phase *locale* et une phase *globale* — l'une ou l'autre n'étant pas obligatoire. Ces deux phases étant complexes, il est naturel de les séparer — au moins conceptuellement, mais aussi en pratique — en plusieurs étapes. La section 6.3

1. Changing Availability or Type of CALLs.

décrit non exhaustivement les principales tâches et sous-tâches de ces phases que nous nommerons *ingrédients de la compilation*.

6.2.1 Phase locale

Classiquement, la phase locale a pour but de compiler une unité de code indépendamment de tout programme l'utilisant, c'est-à-dire de toutes ses utilisations futures. Cette phase est donc essentiellement caractérisée par l'hypothèse du monde ouvert qui permet d'assurer cette notion « d'indépendance ». Le résultat de cette phase est dit *fichier binaire* (ou *fichier objet*[2]). Ces fichiers ne sont pas fonctionnels en l'état car il leur manque toutes les informations et le contenu des autres unités de code. Cette phase implique nécessairement une modularité du processus de compilation, puisqu'elle s'effectue de façon indépendante des programmes finaux. Cette modularité offre plusieurs avantages immédiats. Les unités ainsi compilées peuvent prendre part à la constitution de plusieurs programmes distincts, assurant ainsi une meilleure réutilisabilité. De plus, la reconstruction (ou recompilation) d'un programme peut réutiliser les fichiers déjà produits si le code source n'est pas modifié. Enfin, la vérification de la validité et la cohérence de l'unité de code se trouve isolée à l'unité considérée, ce qui permet une vérification plus rapide ainsi qu'une meilleure localisation des erreurs. En effet, les erreurs sont découvertes directement sur l'unité de code et ne se propagent pas à tout le programme final.

Les fichiers binaires produits par cette phase peuvent être destinés, soit à une vraie machine, soit à une machine virtuelle. Le code généré pour une machine virtuelle fait abstraction de l'implémentation réelle de la machine — registres, pile, ...— et peut intégrer directement des primitives objet, délégant de ce fait l'implémentation des objets à l'implémentation de la machine virtuelle. Si le code est destiné à une machine réelle, la génération n'est cependant que rarement directe. Les mécanismes objet peuvent être projetés dans un langage sans objet, le programme ainsi obtenu pouvant à son tour être aussi transformé vers un langage de plus bas niveau. En pratique, même les fichiers binaires directement destinés à une machine réelle contiennent encore des abstractions.

6.2.2 Phase globale

La phase globale, est bien connue sous le nom d'*édition de liens* ou *reliure*. Elle a pour but de construire une version exécutable d'un programme à partir des différentes unités de codes (les fichiers binaires) qui le composent. Cette phase, à l'inverse de la phase locale, est donc essentiellement caractérisée par l'hypothèse du monde clos puisque l'ensemble des unités de code qui constitue le *fichier exécutable* est connu. Le cas du chargement dynamique y fait exception et les unités sont chargées par le système d'exécution qui considère l'hypothèse du monde ouvert. Ce système d'exécution peut être matérialisé par une

2. Pour une raison évidente d'ambiguïté dans une thèse sur la programmation objet, nous n'utiliserons pas ce terme.

machine virtuelle. Dans le cas contraire, où une machine réelle exécute le programme, le système d'exécution doit être directement intégré au code du programme.

Comme nous allons le voir, la phase globale peut aussi comporter des ingrédients qui n'existent généralement pas dans les éditeurs de liens habituels. La diversité des spécificités des phases globales entraîne que, pour chaque schéma, cette phase est bien distincte et le caractérise. C'est à cause de tout cela que nous préférons le terme de phase globale à celui d'édition de liens.

6.3 Ingrédients de la compilation

La compilation d'un programme met en jeu plusieurs ingrédients qui peuvent être combinés de différentes manières. L'ensemble des ingrédients utilisés, ainsi que leur ordre forme un *schéma de compilation*. La teneur exacte du contenu d'un ingrédient dépend du schéma en question et de sa position dans la chaîne de compilation. Cette section présente ces *ingrédients* dans leur contexte d'utilisation normale.

6.3.1 Analyse syntaxique

Pour commencer, le processus de compilation nécessite de lire des fichiers sources et doit vérifier s'ils sont valides afin d'en interpréter le contenu. Généralement, le résultat de toute cette phase d'analyse est un *arbre syntaxique abstrait*.

Recherche de l'unité de code

La première étape de cette tâche consiste à rechercher l'unité de code à traiter — à moins que le chemin vers celle-ci ne soit passé en paramètre. Cette recherche se fait généralement dans l'ordre suivant :

- Liste des chemins fournis au compilateur (option `-I` de GCC ou PRMC)
- Variables d'environnement (souvent `$PATH`, ou plus particulière par ex : `$PRM_PATH`)
- Liste des chemins système (souvent codés en dur dans le compilateur)

Lorsqu'il existe plusieurs sources pour une unité le compilateur peut choisir parmi celles-ci les plus adaptées. Ce choix peut être fait selon la date des unités trouvées — seule la plus récente est utilisé —, selon l'ordre de recherche — seule la première trouvé est utilisé, c'est le cas de PRMC. Éventuellement si le compilateur gère plusieurs types d'unités, par exemple des unités pré-compilées, le compilateur peut choisir celle qui lui demandera le moins de travail.

Analyse lexicale

L'analyse lexicale segmente le texte du code source en petits morceaux appelés jetons (*tokens*) ou lexèmes — cette phase est aussi appelée *tokenization* ou balayage. Chaque je-

ton est une unité atomique unique de la langue (unités lexicales ou lexèmes), comme par exemple un mot-clé, un identifiant ou un nombre. La syntaxe des jetons est généralement décrite par un langage régulier (ou rationnel) — pouvant donc être reconnu par un automate déterministe à états finis — qui possède des implémentations très efficaces en temps comme en espace. Cette analyse est réalisée par un *analyseur lexical* ou *scanner*. Cet analyseur lexical peut être un logiciel à part entière ou un bout de programme généré par un *générateur d'analyseur lexical*[3] (LEX, FLEX...).

Analyse syntaxique

L'analyse syntaxique consiste à mettre en évidence la structure d'une unité de code et à vérifier qu'elle est correcte — c'est-à-dire qu'elle vérifie la *grammaire* du langage. Cette grammaire formelle est exprimée par des règles de syntaxe — souvent décrites sous la *forme de Backus-Naur* (BNF) — qui seront appliquées par un *analyseur syntaxique* (en anglais *parser*). La structure révélée par l'analyse donne une hiérarchie de syntagmes, représentable par un arbre syntaxique.

Les langages informatiques sont souvent décrits par une grammaire non contextuelle et par conséquent sont reconnaissables par un automate à pile. Il existe principalement deux stratégies pour appliquer les règles de syntaxe :

- l'analyse descendante qui retrace cette dérivation en partant de l'axiome et en essayant d'appliquer les règles pour retrouver le texte. Cette analyse procède en morcelant la phrase en éléments de plus en plus réduits jusqu'à atteindre les unités lexicales. L'analyse LL est un exemple d'analyse descendante.
- L'analyse ascendante qui retrouve ce cheminement en partant du texte, en tentant d'associer des lexèmes en syntagmes de plus en plus larges jusqu'à ce qu'on retrouve un axiome. Les analyses LR et LALR sont des exemples d'analyse ascendante.

Le générateur d'analyseur syntaxique PRMCC est un analyseur de type LALR.

L'arbre syntaxique obtenu après analyse syntaxique est généralement trop précis, car il contient tous les lexèmes. Pour simplifier les étapes suivantes mais aussi pour abstraire les concepts du langage de leur expression syntaxique, on transforme (éventuellement dès la construction) l'arbre syntaxique obtenu en *arbre syntaxique abstrait*.

Transformation source-à-source

Les transformations source-à-source permettent d'ajouter au langage de nouvelles constructions (sucre syntaxique) sans pour autant rajouter de nouveaux concepts et sans changer le compilateur.

Il existe principalement deux manières de faire des transformations source-à-source, soit en les faisant avant l'analyse syntaxique sur le texte, soit en les faisant ultérieurement sur l'arbre.

3. Cette dernière alternative est généralement la solution utilisée par la majorités des compilateurs.

LISTING 6.1 : Une boucle `for`

```
for i in col do
...
end
```

LISTING 6.2 : La même boucle en utilisant seulement un `while`

```
do
# Ce nouveau bloc permet de masquer la variable it
# pour permettre d'avoir des boucles for imbriquees
    let it := col.iterator
    while it.is_ok do
        let i := it.item
        ...
        it.next
    end
end
```

Substitution d'expression ou expansion des macros. La substitution d'expression consiste à remplacer un type d'expression par une forme ne contenant que des expressions plus simples. Cette substitution peut être faite par une pré-analyse du code source (c'est le cas du pré-processeur C ANSI) ou pendant l'analyse du code source (c'est le cas de l'implémentation du pré-processeur C par GCC).

Ce style de transformation source-à-source est particulièrement simple à mettre en œuvre. Elle peut être implémentée directement par le compilateur (solution de GCC), ou être rajoutée comme un filtre de pré-traitement sur le code source (solution C ANSI).

Exemple. Bien que ce ne soit pas la solution utilisée par le compilateur PRMC pour réaliser des transformations source-à-source, les listings suivants montrent comment une itération de type `for` (Listing 6.1) peut être remplacée par une boucle `while` (Listing 6.2).

Réécriture d'arbre. On peut réaliser des transformations source-à-source après analyse syntaxique en réalisant des transformations directement sur l'arbre syntaxique — c'est la solution utilisée par le compilateur PRMC. Il peut être judicieux de conserver dans le nœud de l'arbre transformé la forme originale de l'arbre — principalement pour l'affichage.

La transformation de l'arbre syntaxique en arbre syntaxique abstrait est un très bon exemple de transformation d'arbre.

6.3.2 Vérification sémantique

L'analyse sémantique est la phase intervenant entre l'analyse syntaxique et la génération de code. Elle a pour but d'effectuer les vérifications nécessaires à la sémantique du langage considéré et peut éventuellement annoter l'arbre syntaxique abstrait.

Instanciation du méta-modèle

Si l'on considère que la sémantique du langage peut être décrite par un méta-modèle, alors son instanciation constitue un certificat de validité de l'unité considérée. En effet, en cas de conflit, il est impossible d'associer tous les éléments du méta-modèle sans ambiguïté.

Le modèle obtenu est de plus haut niveau conceptuel que l'arbre syntaxique abstrait. Ceci permet une meilleure navigabilité entre les différentes entités du programme, ce qui simplifie grandement l'étape de vérification de type.

Les instances du méta-modèle constituent de plus la base des structures de données sur lesquelles vont s'appuyer l'implémentation des objets.

Vérification des types

La vérification de type est une tâche spécifique aux langages de programmation statiquement typés, dont le but est de vérifier que le programme ne contient pas d'erreur de type. En pratique, la vérification de type consiste à vérifier que le receveur connaît bien la méthode appelée — ce qui est déjà fait par l'instanciation du méta-modèle — et que les types des expressions passées en arguments à un appel de méthode sont compatibles (c'est-à-dire sont des sous-types) avec le type des paramètres de la dite méthode (c'est-à-dire la signature de la méthode). Grâce au méta-modèle, il est trivial de récupérer les signatures d'une méthode en fonction du type du receveur. En toute généralité elle vérifie qu'un membre droit (*R-value*) est compatible avec un membre gauche (*L-value*), ce qui correspond aussi aux affectations, formes conditionnelles, etc. Un système de types est vérifié par un algorithme de vérification de types (*type checker*).

Si le système de types est sûr, une fois la vérification de type effectuée, tous les tests de types dynamiques implicites peuvent être enlevés de l'exécutable — les tests explicites (*cast*, etc.) doivent être conservés.

6.3.3 Analyses statiques

Il est possible d'analyser statiquement le programme afin d'améliorer la compréhension que le compilateur a du code. Pour les langages statiquement typés, une grande majorité de ces analyses consiste à affiner la connaissance que le compilateur a sur le type des expressions afin d'éliminer un maximum de tests inutiles et le *code mort*, c'est-à-dire le code jamais appelé par aucune exécution du programme [Srivastava, 1992].

LISTING 6.3 : Analyse intra-procédurale

```
let foo := new Foo
println(foo.to_s)    # Pas de test a null
                     # Appel statique sur Foo::to_s
...
let bar := some_method
bar.do_something
bar.do_something_else    # Pas de test null
```

Analyse intraprocédurale

L'analyse intraprocédurale est une analyse locale qui consiste à regarder seulement le contenu d'une méthode (procédure). Bien que cette analyse soit très légère, elle permet tout de même de détecter un grand nombre d'opportunités d'optimisation.

Variables monomorphes : Lors de l'instanciation d'une classe, le type statique est identique au type dynamique. Cette information peut être propagée à travers la méthode pour remplacer les appels polymorphes par des appels statiques.

Élimination des tests à `null` : On peut éliminer les tests à `null` de deux manières différentes. Lors de l'affectation d'une variable par une valeur non nulle, les tests à `null` sur cette variable peuvent être supprimés. De plus, tant qu'une variable n'est pas réaffectée, seule la première utilisation de la variable nécessite d'être testée.

Élimination des tests de sous-typage : En l'absence d'une construction comme `typecase`, les tests de sous-typage explicite (`isa`) sont généralement suivis d'un *cast*. Ce *cast* est rendu sûr par la garde (`isa`) et ne nécessite donc pas de test de sous-typage supplémentaire. Cet avantage est perdu avec les sous-objets puisqu'il est nécessaire de trouver le décalage.

Analyses de types

Les analyses de types inter-classes sont couramment utilisées avec le schéma de compilation globale (voir Section 6.4.3) pour calculer l'ensemble des types dynamiques que peuvent prendre les expressions, ce qu'on appelle le *type concret* (il est noté $[e]$ pour une expression e) [Ryder, 1979; Grove *et al.*, 1997; Grove et Chambers, 2001; Grove, 1995; Wang et Smith, 2001; Privat, 2002]. Le problème est indécidable puisque sa solution dépend du test d'arrêt d'un algorithme. Il existe donc un grand nombre d'algorithmes d'analyses de types plus ou moins complexes, dont le principe général est d'approximer le graphe d'appel d'un programme afin de déterminer, plus ou moins grossièrement, les types concrets.

Les paragraphes qui suivent présentent un bref aperçu des quatre analyses de type classique utilisées par PRMC.

CHA. En considérant uniquement la hiérarchie de classes, il est possible de déterminer l'ensemble des propriétés locales accessibles pour un couple ⟨propriété globale, type statique⟩. Cette analyse est appelée CHA (*Class Hierarchy Analysis*) [Dean *et al.*, 1995; Dean et Chambers, 1994; Gil et Itai, 1998]. Ce n'est pas réellement une analyse de types mais plutôt une préanalyse, dans la mesure où toutes les informations sont contenues dans la définition de la hiérarchie de classe et que l'analyse est indépendante du code des méthodes. Avec le méta-modèle, l'implémentation de CHA est parfaitement triviale et c'est l'analyse de type par défaut du compilateur PRMC.

Malgré la simplicité de cette analyse, elle est souvent utilisée car en plus d'être rapide, elle donne de très bons résultats en pratique [Tip et Palsberg, 2000; Qian et Hendren, 2005].

RTA. RTA (*Rapid Type Analysis*) est l'analyse de types la plus simple qui considère en plus de la hiérarchie de classes les instanciations faites par le programme et en déduit un graphe d'appel [Bacon *et al.*, 1996; Bacon et Sweeney, 1996; Bacon, 1997]. Le principe est de construire itérativement l'ensemble des classes instanciées ainsi que l'ensemble des méthodes appelées par le programme, en partant du point d'entrée de celui-ci. Cette analyse produit le résultat de CHA restreint aux classes vivantes. Elle a comme principal avantage, de permettre d'éliminer une partie du code mort — l'ensemble des classes et des méthodes jamais utilisées par un programme. Cet avantage est minime si l'ensemble du code est *ad hoc* mais si le programme utilise des librairies les gains peuvent être considérables.

XTA. Alors que RTA utilise un seul ensemble de classes/méthodes vivantes pour tout le programme, XTA (*Separate sets for methods and fields*) en utilise un pour chaque méthode et pour chaque attribut [Tip et Palsberg, 2000]. En pratique, l'algorithme de XTA est nettement plus coûteux en temps que RTA (5 fois plus long en moyenne d'après les chiffres rapportés par les auteurs) et sa précision n'est guère meilleure que RTA.

CFA (k-CFA). Les algorithmes présentés jusqu'à présent ne considéraient pas le flot de contrôle du programme. L'idée des algorithmes de la famille CFA (*Control Flow Analysis*) est de simuler le flot de circulation des types dans le programme pour calculer les types concrets de chaque expression [Chambers *et al.*, 1997; Shivers, 1988, 1991]. Cette famille d'algorithme construit des graphes d'appels très précis, mais le passage à l'échelle (même pour 0-CFA) est douteux [Tip et Palsberg, 2000] aussi bien du point de vue espace — l'algorithme utilise un ensemble de types concrets par expression — que du point de vue temps — les algorithmes ne sont pas polynomiaux. Le compilateur PRMC n'implémente que la version 0-CFA.

6.3.4 Génération

Les étapes de génération sont l'aboutissement du processus de compilation — bien qu'elles n'en soient pas les dernières étapes — et correspondent à la production du compilateur. Nous avons découpé cette étape en trois sous-étapes bien distinctes, même si la plupart des compilateurs les réalisent en même temps, voire en omettent certaines.

Génération de code

La génération de code est l'étape clé du processus de compilation. Elle consiste à produire à partir de la représentation interne (généralement l'arbre syntaxique abstrait) son équivalent en syntaxe concrète dans le langage cible. Afin que la génération de code se passe correctement, c'est-à-dire que le code produit soit consistant, il est nécessaire que toutes les étapes de vérification (analyses) soient effectuées auparavant avec succès.

Perte d'information. En général, les premières phases de la compilation (analyse syntaxique, analyse sémantique) permettent d'enrichir un code source en y ajoutant des informations (structurelles et/ou sémantiques) — par exemple l'inférence de certains types, etc. A l'inverse, la génération de code se sert de ces informations pour produire du code (en général avec un niveau d'abstraction inférieur) et perd donc une partie des informations connues a priori. Il n'est donc plus forcément possible de connaître les types, les numéros de lignes, voire le nom des fichiers originaux, rendant de ce fait le débogage plus complexe (voire impossible).

Afin de conserver tout ou partie de ces informations, la plupart des générateurs de code rajoutent dans le code produit des annotations (par exemple sous la forme de commentaires ou de fichiers auxiliaires) [4].

Écorcher les noms (*name mangling*). Un même nom, suivant le contexte, peut désigner plusieurs éléments distincts — en fonction de l'espace de nom par exemple. N'étant pas unique, l'utilisation du nom de l'élément comme symbole est alors impossible. Écorcher les noms consiste à les annoter de manière à les rendre non ambigus — et uniques.

Remarque. Le terme « écorcher » est relativement impropre mais c'est la traduction directe de *mangle* qui est utilisé en C++ et partout ailleurs par extension. En fait, il s'agit plus de décorer ou d'annoter le nom que de l'écorcher.

Au départ, le *name mangling* était utilisé pour séparer les routines systèmes de routines de l'utilisateur. Ensuite, la convention adoptée par les systèmes UNIX était d'utiliser le *name mangling* pour distinguer les routines C et FORTRAN et éviter ainsi des collisions par inadvertance. Certains systèmes utilisent des caractères interdits en C pour les symboles réservés (`$`, `@`, `.`).

4. GCC se sert par exemple du `#` et des `pragma` du préprocesseur pour stocker le nom du fichier et le numéro de la ligne qui a engendré un bout de code.

En C++, les règles de *mangling* sont complexes et dépendantes du compilateur utilisé. Globalement, un symbole est obtenu par concaténation de l'espace de nom (classe comprise), du nom de la propriété — éventuellement transformé, s'il s'agit d'un opérateur — et du type des paramètres, eux mêmes écorchés — s'il y en a.

PRMC utilise le même principe, en les préfixant par le type de propriété et en utilisant des caractères interdits en PRM comme séparateurs. Enfin, comme la seule surcharge statique en PRM est basée sur le nombre de paramètres, les types des paramètres sont remplacés par leur nombre — ce qui réduit un peu les noms écorchés.

Le *mangling* permet d'éviter un certain nombre de conflits de nom à l'intérieur des programmes générés, au détriment de leur lisibilité. De plus, les noms écorchés sont généralement très longs et leur lecture à l'intérieur d'un code source généré est particulièrement cryptique pour un utilisateur humain.

Génération de l'implémentation

La génération de l'implémentation consiste à créer les structures de données relatives à la technique d'implémentation sous-jacente. Cette étape est spécifique aux langages de programmation objet et ces structures correspondent au support des trois mécanismes objet de base.

- Les tables de méthodes pour l'invocation de méthode (représentation des classes).
- Les patrons d'instances pour la génération et l'initialisation d'objets (non nécessaire mais simplifie les constructions d'objets).
- L'encodage de la relation de spécialisation pour le test de sous-typage.

La construction de ces structures dépend du modèle externe de l'application considérée ainsi que de l'implémentation choisie — par exemple le hachage parfait nécessite le calcul d'une table de hachage (voir Section 5.3), la coloration nécessite un calcul compliqué et entraîne certains trous dans les tables de méthodes (voir Section 5.2), etc.

Génération des modèles

Un compilateur peut produire, en plus du code cible, des modèles de l'unité de code considérée. On peut distinguer deux types de modèles.

Le modèle externe d'une classe (ou d'une unité de code) contient des informations sur la classe (ou les classes) et l'ensemble des propriétés connues par chaque classe. Ces informations sont nécessaires pour qu'un client[5] puisse réutiliser l'unité en question. Les langages à base de *bytecode* extraient souvent ces informations des fichiers compilés (c'est le cas de JAVA ou C#). Dans le cas de C++, ce modèle est fourni par le programmeur sous la forme de fichier d'en-tête (`.h`). Dans ce dernier cas, il est à noter que, par rapport au compilateur, le programmeur peut se tromper, fournir des

5. Notion de clientèle à la EIFFEL et non d'acheteur (humain).

informations obsolètes et même fournir des informations sans aucun rapport avec l'unité considérée. À notre avis, plus aucun langage ne devrait fonctionner sur ces modèles manuels. Dans notre cas, ce modèle externe est une transcription textuelle des instances du méta-modèle impliquées.

Le modèle interne contient les informations relatives à la circulation des types à l'intérieur des méthodes, c'est-à-dire les instanciations et les appels faits par chaque méthode, etc. Ce modèle est principalement utilisé pour réaliser des optimisations plus poussées (par exemple des analyses de types contextuelles comme RTA, XTA ou CFA) et sont à la base du schéma de compilation optimisé (voir Section 6.4.4). A notre connaissance, aucun langage grand public n'utilise de modèle interne.

6.3.5 Recoller les morceaux

L'assemblage

C'est la compilation d'un code assembleur en fichier binaire, c'est-à-dire directement utilisable par le processeur. Cette phase est spécifique à chaque architecture matérielle. L'assemblage, par rapport à la compilation en général, est une traduction littérale des mnémoniques en instructions machine, c'est-à-dire que le compilateur ne réalise aucune optimisation pas plus qu'il ne réorganise les instructions.

L'édition de liens

L'édition de liens regroupe différentes unités de code en un même fichier afin d'en produire un fichier exécutable — ou une bibliothèque. Les références symboliques — adresses de saut, adresses de variables, noms de fonctions — sont résolues par leur valeur réelle — ou relative au début du programme (code translatable).

Remarque. Malgré leur nom, les *fichiers exécutables* ne le sont généralement plus directement. Certains anciens systèmes d'exploitation disposaient de programmes directement exécutables (par exemple les `.COM` de MS-DOS). De nos jours, tous les systèmes d'exploitation récents requièrent l'utilisation d'un *chargeur* (ou *loader*) faisant aussi office d'éditeur de liens.

Le chargement de l'exécutable

Le chargement d'un exécutable est une étape importante dans la création d'un processus. Elle consiste à mettre en mémoire vive un programme enregistré sur un support de stockage. Le *chargeur* doit traiter les spécificités du système d'exploitation sous-jacent — position du début du programme, alignement du programme [6], ... Enfin, dans la plupart

6. Généralement, les programmes stockés sont découpés en segments, alors qu'en mémoire ils sont découpés en pages.

des systèmes d'exploitation, le chargeur doit remplir partiellement le rôle de l'éditeur de liens en additionnant aux adresses la position du début du programme, ainsi qu'en résolvant les derniers symboles manquants — ceux des appels systèmes entre autres.

Chargement dynamique

Machine virtuelle Une machine virtuelle désigne un logiciel ou interpréteur qui isole l'application utilisée par l'utilisateur des spécificités de l'ordinateur, c'est-à-dire de celles de son architecture ou de son système d'exploitation. Cette indirection permet au concepteur d'une application de la rendre disponible sur un grand nombre d'ordinateurs sans les contraintes habituelles à la rédaction d'un logiciel portable tournant directement sur l'ordinateur. Les machines virtuelles exécutent généralement du *bytecode*, un genre d'assembleur de plus haut niveau.

Remarque. Le terme machine virtuelle peut aussi désigner la création de plusieurs environnements d'exécution sur un seul ordinateur, où chaque environnement émule un ordinateur hôte complet et dédié. Cela fournit à chaque utilisateur l'illusion de disposer d'un ordinateur complet alors que chaque machine virtuelle est isolée des autres.

La compilation à la volée (*Just-In-Time compilation* ou JIT compilation) consiste à compiler des fragments de programmes pendant son exécution. Dans le cas des langages objets, la compilation à la volée compile le corps des méthodes au moment de leur premier appel — ou de leur chargement. La compilation étant une tâche coûteuse, pour simplifier le travail, les compilateurs à la volée traduisent généralement un langage intermédiaire — généralement du *bytecode* — vers du code machine natif.

Compilation à l'avance (*ahead of time compilation* ou AOT compilation). Dans le cadre des systèmes d'exécution JAVA ou .NET, le terme semble désigner la compilation « classique », c'est-à-dire vers le langage machine (par opposition au *bytecode*), et non adaptative (par opposition au JIT), qu'elle soit séparée ou globale.

6.3.6 Analyses dynamiques

Enfin, il est possible de profiler l'exécution du programme, puis d'utiliser les informations collectées pour en améliorer sa compilation. Les profils ainsi obtenus peuvent être utilisés de différentes manières. Le nombre d'instanciations de chaque classe peut servir les heuristiques de la coloration d'attributs. Les statistiques sur les sites d'appels peuvent conditionner l'utilisation de caches en ligne (voir Section 5.7.2) ou servir à ordonner les branches d'un arbre de sélection non équilibré (voir Section 5.8.2).

Ces analyses peuvent aussi être utilisées durant l'exécution du programme par un compilateur à la volée, pour choisir le « niveau » de compilation qu'il doit appliquer à une méthode.

Cependant ces informations sont très dépendantes du flot d'exécution du programme et biaise l'évaluation des mécanismes élémentaires — puisqu'ils pourraient être court-circuités. Nous ne considérerons donc plus les retours d'informations dynamiques pour conditionner la compilation dans cette thèse.

6.4 Différents schémas de compilation

Les différents schémas de compilations peuvent être classés en fonction de l'hypothèse faite sur le *monde*, de l'hypothèse du monde ouvert à l'hypothèse du monde clos. Les schémas les plus modulaires, donc compatibles avec l'hypothèse du monde ouvert, sont les moins efficaces (à gauche dans la figure 6.1) tandis que les plus efficaces (à droite) utilisent intensivement les hypothèses faites sur le monde fermé au détriment de la modularité.

Les schémas de compilations peuvent être considérés en termes d'ingrédients répartis entre la phase locale et la phase globale. Bien entendu la combinatoire de l'ensemble des ingrédients répartis dans les deux phases peut être considérée mais la plupart des combinaisons n'auraient que peu d'intérêt voire n'auraient aucun sens. Nous présentons maintenant quelques unes des combinaisons intéressantes, parmi ces combinaisons on peut retrouver les schémas de compilation classiques.

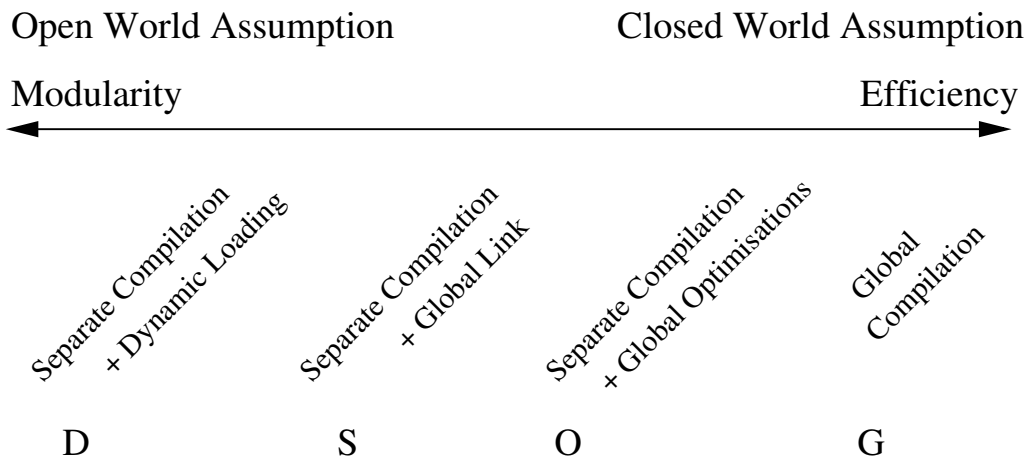


FIGURE 6.1 : Divers schémas de compilations, du plus modulaire au plus efficace

6.4.1 Compilation séparée et édition de liens dynamique (D)

La compilation séparée, associée au chargement dynamique, est un schéma de compilation classique illustré par de nombreux langages LISP, ADA, SMALLTALK, C++, JAVA.

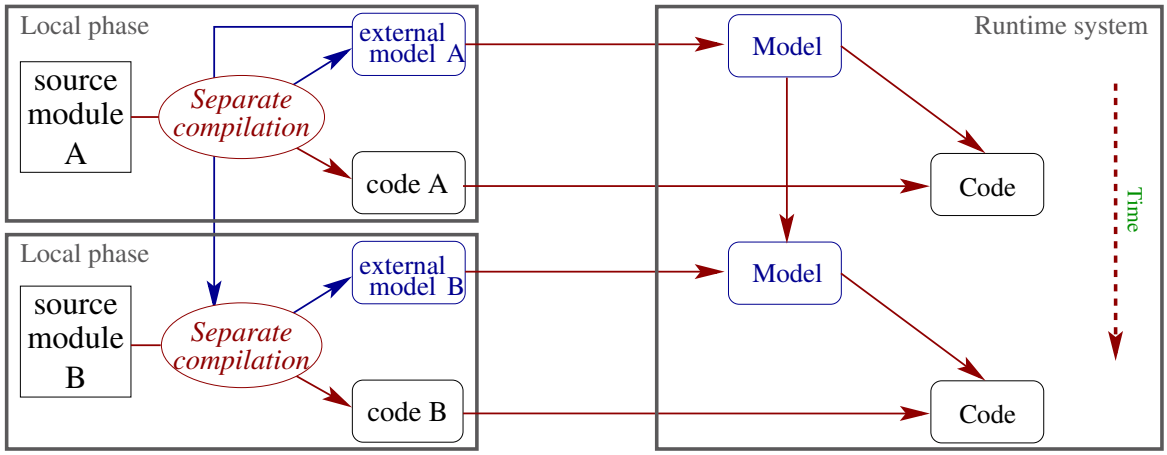


FIGURE 6.2 : Compilation séparée et édition de liens dynamique

Plus que tout autre schéma de compilation séparée, la compilation séparée avec chargement dynamique opère principalement sur la phase locale. Chaque unité de code est compilée séparément et indépendamment de toute utilisation durant la phase locale. Cette compilation doit, au moins, produire un fichier binaire ainsi que le modèle des classes qu'elle définit (modèle externe).

Remarque. En JAVA le modèle externe n'est pas produit durant la compilation mais extrait du *bytecode* (`.class`).

Si l'unité à compiler comporte une dépendance vis-à-vis d'une autre unité, le modèle externe de cette autre unité est utilisé durant la compilation — principalement afin de vérifier l'existence des entités utilisées et la validité des types. Dans ce schéma, la phase globale est représentée par le système d'exécution. Il charge directement le fichier binaire, substitue les symboles manquants et lance l'exécution du programme. Durant l'exécution du programme, il peut faire appel à des modules ou à des classes non chargés. C'est là que le chargement dynamique intervient. Le système d'exécution charge la nouvelle unité, met à jour son modèle, ce qui consiste essentiellement en C++ à maintenir la liste des éléments — classes, modules — chargés et des symboles résolus, et à substituer les symboles. En JAVA, le modèle de la hiérarchie de classes au complet est maintenu. Durant cette étape de chargement de nouvelles unités, les unités précédemment chargées ne sont pas modifiées.

Le cas de C++ diffère sur au moins deux points par rapport à celui de JAVA. Tout d'abord l'unité de code à charger n'est pas limitée à une seule classe mais peut être constituée d'un ensemble de classes. Enfin, son système d'exécution est une machine réelle et C++ utilise un système de *thunks* pour déclencher paresseusement le chargement des librairies. Ceci aurait été réalisé par le chargeur de classes (*classloader*) dans une machine virtuelle.

Pour rendre ce schéma de compilation opérationnel, en l'absence de recompilation

dynamique, les techniques d'implémentation utilisées doivent être incrémentales (compatibles avec l'hypothèse du monde ouvert). Ce qui limite le nombre des techniques d'implémentation compatibles avec l'héritage multiple, seuls les sous-objets de C++ et le hachage parfait sont compatibles. Dans le cas particulier de JAVA et du test de sous-typage, la coloration incrémentale a été essayée [Palacz et Vitek, 2003], mais le chargement dynamique impose un recalcul et des indirections qui sont coûteuses, aussi bien au chargement qu'à l'exécution.

D'un point de vue génie logiciel, ce schéma présente plusieurs avantages. Il permet de distribuer des modules déjà compilés sous forme de boîtes noires. Comme les unités sont traitées de manière indépendante, la recompilation des programmes peut se restreindre aux unités modifiées directement ou indirectement.

Malheureusement, si l'on exclut des recompilations dynamiques, ce schéma ne permet aucune optimisation des mécanismes objet sur le code produit — librairies, programmes.

6.4.2 Compilation séparée et édition de liens globale (S)

La compilation séparée avec édition de liens globale est l'un des premiers schémas de compilation inventés, le premier exemple est très certainement FORTRAN en 57. Les premiers schémas de compilations séparées sont apparus pour des raison pratiques plus que théoriques, les ordinateurs de l'époque n'avaient pas assez de mémoire pour compiler de gros programme en une seule fois.

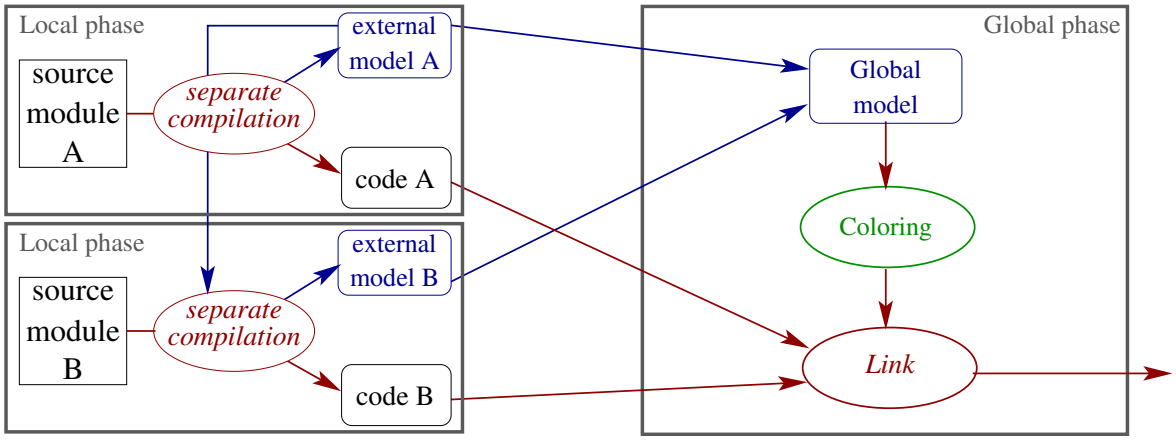


FIGURE 6.3 : Compilation séparée et édition de liens globale

Ce schéma est très proche de la compilation séparée avec chargement dynamique, mis à part la phase globale. Avec cette approche le programme final est constitué d'un ensemble d'unités de code connues lors de la phase globale.

Les unités sont d'abord compilées indépendamment les unes des autres et le code produit est identique au code produit par le schéma (D). Pour donner un sens à ce schéma,

l'étape de génération des implémentations ne peut pas être faite durant la phase locale et elle doit être retardée à la phase globale. Lors de cette phase, comme le modèle global de l'application est connu (hypothèse du monde clos), les structures peuvent être créées sur mesure pour ce programme. Il devient possible de procéder au calcul de la coloration, par exemple, puis de générer des tables de méthodes contenant les *trous* nécessaires. Comme la coloration est une technique intrinsèquement globale, une fois l'édition de liens effectuée, aucune nouvelle classe ni aucune nouvelle méthode ne peuvent être ajoutées sans recalculer tout ou partie de la coloration. Si la représentation des objets utilise aussi la coloration, il serait alors nécessaire de remplacer les instances vivantes par des nouvelles. Ceci est appelé *migration d'instance* et est hors du cadre de cette thèse.

Cette version de la compilation séparée est donc incompatible avec le chargement dynamique. Cependant elle en conserve les nombreuses qualités — distribution du code compilé, recompilation partielle — tout en annulant le surcoût ajouté par l'héritage multiple — la coloration a la même efficacité que l'implémentation du sous-typage simple.

6.4.3 Compilation globale (G)

Ce schéma de compilation repose sur l'hypothèse du monde clos et constitue le cadre d'une grande partie de la littérature sur l'optimisation des programmes objet autour des langages SELF [Chambers *et al.*, 1989], CECIL [Chambers, 1993], EIFFEL [Zendra *et al.*, 1997; Collin *et al.*, 1997]. Ce schéma de compilation permet d'obtenir le code le plus efficace de tous les schémas de compilation statique.

Comme son nom l'indique la compilation globale n'est constituée que d'une phase globale et suppose que l'intrégralité du code source des unités soit connue lors de l'étape de compilation (hypothèse du monde clos).

Le code de toutes les unités étant disponible, le point d'entrée du programme y compris, il est possible d'effectuer une *analyse de types* pour déterminer l'ensemble des *classes vivantes* (effectivement instanciées par le programme) ainsi que le *code mort*. Ceci permet de réduire la taille de l'exécutable, car seul le code utilisé est présent dans le code généré. À la différence du schéma (S) où seule la génération de l'implémentation peut être faite sur mesure, en compilation globale l'intégralité de la génération de code peut être faite sur mesure. Donc plus particulièrement pour chaque site d'appel, les arbres binaires de sélection peuvent être utilisés et *expansés en ligne*. De plus, les branches des arbres peuvent se limiter à contenir les branches utiles. La coloration peut alors s'utiliser en complément lorsqu'elle est plus efficace que les arbres binaires de sélection, c'est-à-dire pour les sites d'appels dont le degré de polymorphisme reste grand.

Bien que ce schéma permette de générer le code le plus efficace, il présente de nombreux inconvénients. L'ensemble des sources doit être disponible, ce qui empêche la distribution de *modules* pré-compilés. De plus, chaque modification sur le code, même mineure, entraîne une recompilation totale. Inversement, si la modification porte sur une partie de code mort, elle peut n'être même pas vérifiée par le compilateur.

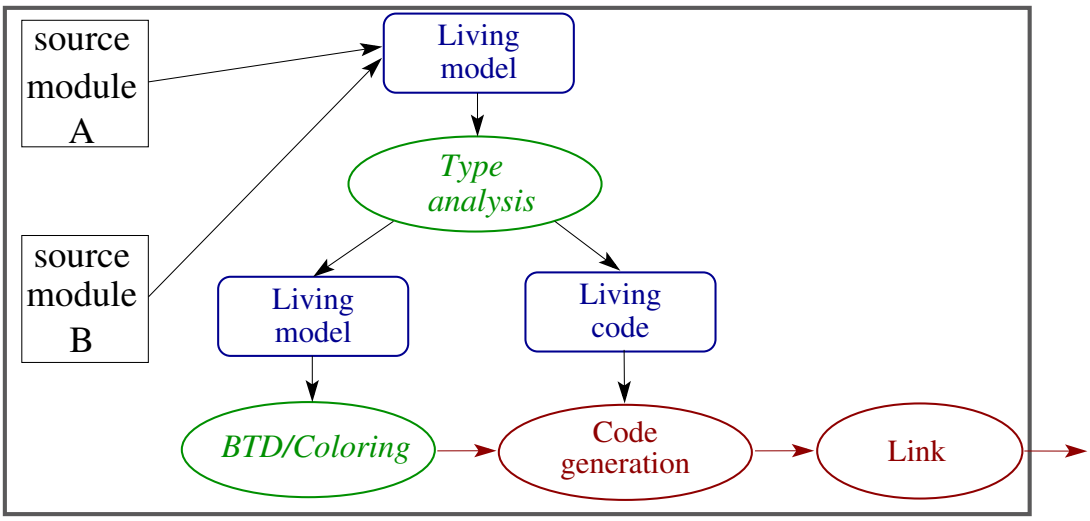


FIGURE 6.4 : Compilation globale

Remarque. Le compilateur SMART EIFFEL, n'ayant pas de phase locale, signale les erreurs seulement lorsqu'elles affectent une partie du programme vivant — effectivement utilisé (voir Section 6.3.3). De ce fait, il est possible de rajouter du code erroné sans en être prévenu par le compilateur — car pour une raison ou un autre il est considéré comme mort, par exemple lors de l'écriture d'une librairie — et de s'en rendre compte seulement beaucoup plus tard, lorsqu'on finit par appeler effectivement ce code. Ce qui lui à valu à juste titre le petit nom de *compilateur qui rend fou*.

Mise en ligne des mécanismes objet et *customization* du code

La *customization* [Chambers et Ungar, 1989; Detlefs et Agesen, 1999] consiste à créer une propriété locale adaptée pour chaque type concret. Sans compilation globale, la *customization* n'est pas envisageable à cause de l'explosion combinatoire.

6.4.4 Compilation séparée et édition de liens optimisée (O)

Privat et Ducournau [2004, 2005] proposent un schéma de compilation séparée, permettant l'utilisation d'optimisations globales lors de la phase globale — initialement nommé édition de liens dans [Privat et Ducournau, 2005]. Un schéma voisin a été proposé par Boucher [2000] dans un cadre de programmation fonctionnelle et dans une moindre mesure pour la programmation objet dans [Fernandez, 1995; Findler et Flatt, 1999].

Ce schéma est une généralisation du schéma de compilation séparée avec édition de liens globale mais qui nécessite des artifices supplémentaires. Tout d'abord lorsqu'une unité de code est compilée, le compilateur produit, en plus du code compilé et du *mo-*

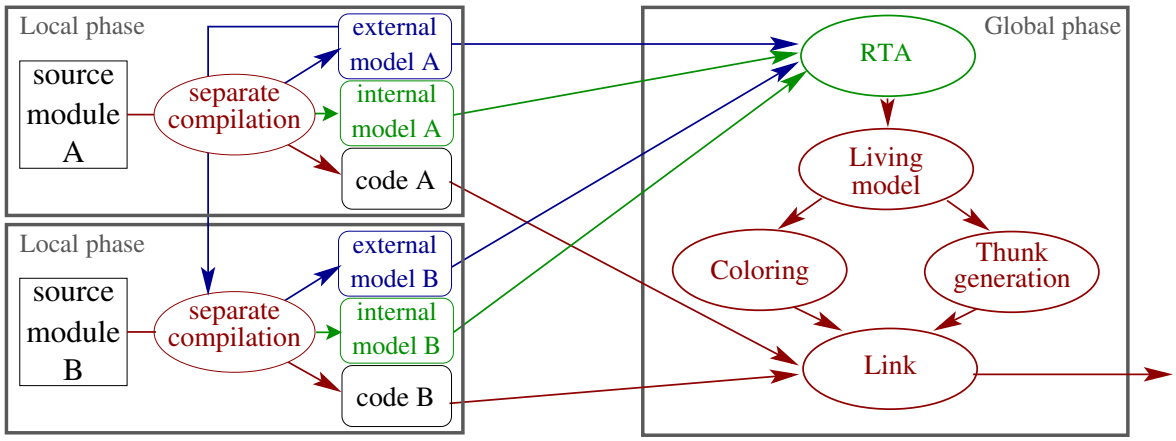


FIGURE 6.5 : Compilation séparée et édition de liens optimisante

dèle externe, un *modèle interne* (voir Section 6.3.4) qui contient les informations relatives à la circulation des types dans l'unité compilée. Cependant, le code compilé par cette phase locale est différent de celui produit dans le schéma (S). Le code de chaque site d'appel est compilé comme un appel de fonction à un symbole unique et inexistant.

L'adjonction des modèles internes aux modèles externes, permet lors de la phase globale, de reconstruire en partie le graphe d'appel du programme, grâce aux flux de types qui y sont contenus, puis d'effectuer une analyse de type comme RTA. Une fois l'analyse de type effectuée on peut regrouper tout les sites d'appels identiques, c'est-à-dire avec même méthode appelée et même type concret du receveur, et générer le bout de code réalisant l'appel. Ce bout de code est appelé *thunk*, comme dans l'implémentation de C++. Enfin, on termine l'édition de liens en substituant les symboles uniques par le nom des *thunks* générés. L'avantage principal de ce schéma est que le code généré pour chaque *thunk* peut profiter des spécificités du site d'appel qui lui est associé — c'est-à-dire en faisant un appel statique pour un site monomorphe, ou un arbre binaire de sélection pour un site oligomorphe ou enfin en utilisant la coloration quand le site est mégamorphe. Dans le cas des appels monomorphes, le *thunk* peut être court-circuité, ce qui revient à annuler totalement le surcoût de la liaison tardive.

Dans ce schéma, il est aussi possible d'éliminer une partie du code mort. Cependant cette élimination est nettement moins triviale et moins efficace qu'en compilation globale. En effet, en compilation globale il suffit de ne pas générer le code associé au code mort, tandis que dans ce cas le code est déjà généré et il faut pouvoir en extraire les méthodes ou les classes inutiles — seule l'élimination de module entier est simple à réaliser mais ne sert quasiment à rien [7].

7. Dans le cas du compilateur PRMC et de la bibliothèque standard du langage, tout les modules sont utilisés, y compris le module `math` pour la génération de nom de fichier aléatoire.

Dans une version simplifiée, l'analyse de types repose sur CHA et la génération des schémas internes n'est plus nécessaire. Par contre, l'élimination du code mort, même au niveau du module n'est plus possible (Figure 6.6).

Comme la phase globale de ce schéma travaille toujours en utilisant l'hypothèse du monde clos, le chargement dynamique reste exclu. Cependant il est possible de réutiliser une partie des concepts utilisés par ce schéma dans le cadre de la compilation adaptative (voir Section 6.4.6).

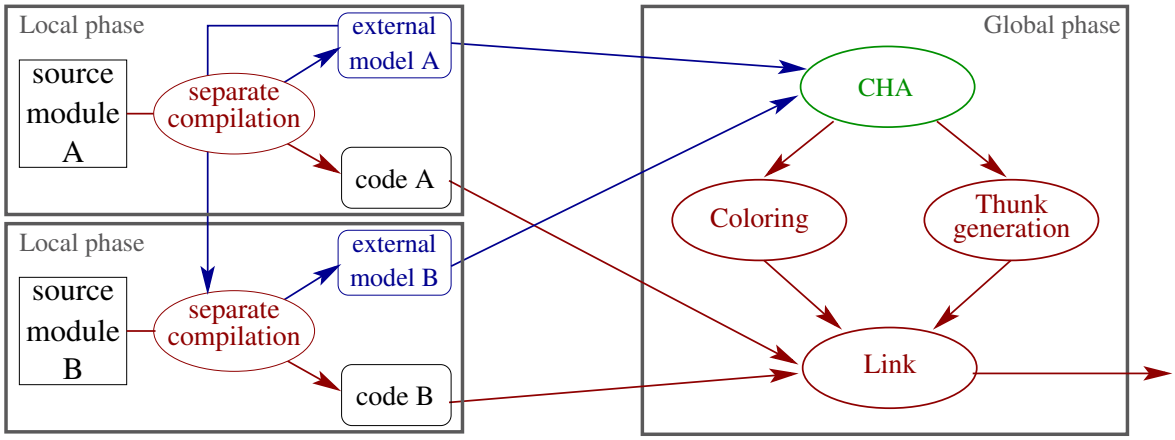


FIGURE 6.6 : Compilation séparée et édition de liens optimisante version simplifiée

6.4.5 Compilation hybride (H)

Une approche intermédiaire consiste à compiler les bibliothèques de manière séparée comme dans les schémas séparées (S ou O). Le programme lui-même est compilé de manière globale (G) en effectuant toutes les optimisations possibles. Le schéma hybride (H) combine donc la compilation séparée des librairies et la compilation globale du programme. Il permet un bon compromis entre la flexibilité — recompilations rapides — et l'efficacité du code produit.

6.4.6 Compilation adaptative (A)

La compilation adaptative est une forme de compilation séparée avec chargement dynamique mais qui repose sur l'hypothèse du monde clos temporaire. Le système d'exécution fait appel à des techniques d'optimisation globales (voir Section 6.4.3) pour le code déjà chargé au prix d'une recompilation dynamique lorsque les hypothèses initiales se trouvent infirmées. En comparaison avec le schéma (D), le chargement d'une nouvelle unité de code entraîne des modifications sur le code généré pour les unités déjà chargées.

En pratique, comme le le code généré pour chaque unité doit pouvoir être changé lorsque les hypothèses initiales se trouvent infirmées les méthodes doivent pouvoir être recompilées et les références à celles ci doivent pouvoir être changées. L'utilisation de *thunks* pourrait simplifier le travail des compilateurs adaptatifs, qui n'auraient qu'a en remplacer un *thunk* par un autre au lieu de recompiler toutes les méthodes impactées.

La plupart des systèmes utilisant la compilation adaptative sont basés sur des machines virtuelles et utilisent un *compilateur à la volée* pour prendre en compte plus d'hypothèses lors de la génération du code, donc mieux optimiser le code en fonction des besoins. A l'heure actuelle, l'efficacité de nombreux langages *mainstream* — JAVA, C#, SMALLTALK, ... — vient principalement de l'utilisation des compilateurs adaptatifs. En effet, la plupart des hypothèses d'optimisation faites par le compilateur ne seront pas remises en question par la suite : dans l'exécution réelle du programme, les appels de méthodes sont principalement monomorphes, les interfaces sont généralement implémentées par une seule classe, etc.

Par ailleurs, le modèle des classes qui est maintenu à l'exécution doit inclure les dépendances entre les éléments du programme qui permettent de décider effectivement des recompilations à effectuer. [Ducournau et Morandat, 2010] en donne un exemple dans le cadre d'une machine virtuelle, à concevoir, en héritage multiple.

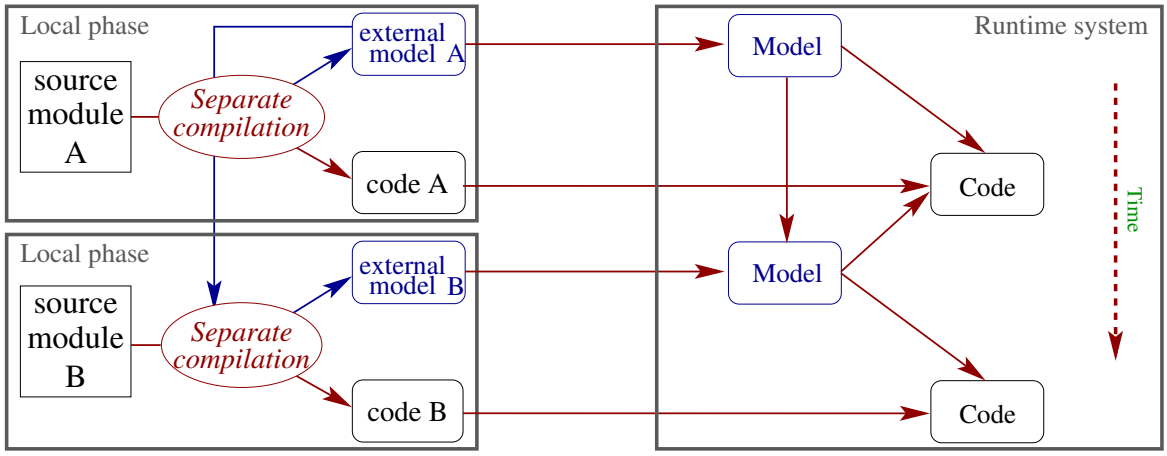


FIGURE 6.7 : Compilation adaptative

Comme ce schéma fait appel à des recompilations dynamiques durant l'exécution, nous ne le considérons pas d'avantage dans cette thèse. Le lecteur intéressé par ce schéma de compilation et les optimisations qu'il permet est renvoyé à [Arnold *et al.*, 2005] pour une étude détaillée.

Invocation de méthode typée par une interface. A cause de l'unification des propriétés (voir Section 3.2.4) en JAVA, il arrive régulièrement que l'on considère dans ce schéma

ces appels en deux catégories, ceux qui peuvent être court-circuités et les autres. Dans le cas, où la méthode est introduite par une classe — sans relation avec l'interface — et que toutes les classes implémentant l'interface sont sous-classes de la classe introduisant la méthode, le *bytecode* de `invokeInterface`[8] est remplacé par celui de `invokeVirtual`[9]. Dans l'autre cas, l'appel est un véritable appel en héritage multiple auquel les implémenteurs de machines virtuelles donnent le nom de *miranda*[10]. Dans ce contexte (A), seuls les appels *miranda* posent un problème et étrangement ils sont généralement traités de manière inefficace.

6.5 Conclusion

Ce chapitre a présenté le concept de schéma de compilation, qui représente la chaîne de production d'un exécutable, et en a exhibé les principaux représentants. Ces schémas sont extrêmement variés et cette variation permet, essentiellement, de graduer l'hypothèse du monde ouvert/fermé, ou plutôt le moment où l'hypothèse du monde fermé est considérée — tout de suite (avec ou sans révision), à l'édition de liens ou jamais.

Les différents schémas présentés ne permettent cependant pas tous d'utiliser toutes les techniques d'implémentation présentées au chapitre 5. Les techniques d'implémentation ne sont utilisables qu'en fonction de l'hypothèse faite sur le monde (ouvert ou fermé). Lorsqu'une technique est disponible pour un schéma de compilation donné, elle l'est aussi pour les schémas plus contraints. Inversement, lorsqu'un schéma est suffisamment contraint pour utiliser une technique, il n'y a aucun intérêt à utiliser une technique moins efficace. Par exemple, le hachage parfait de classes est compatible avec le schéma de compilation séparée avec chargement dynamique. Il est inefficace en compilation globale où on lui préférera des arbres binaires de sélection complétés par la coloration. La table 6.1 résume les compatibilités entre les diverses techniques d'implémentation et les schémas présentés dans cette thèse.

Le choix imposé par les schémas de compilation entre modularité et efficacité ne semble pas avoir de solution intrinsèque, malgré les compromis proposés par certains schémas.

Une partie de ces schémas sont implémentés par le compilateur PRMC qui sera présenté au chapitre suivant. Ce compilateur sera utilisé au chapitre 8 pour examiner l'impact du schéma de compilation sur l'efficacité du code.

8. Appel de méthode typé par une interface.
9. Appel de méthode typé par une classe.
10. Du latin *mirandus* « admirable », nom de la fille de Prospéro dans *The Tempest* de William Shakespeare.

TABLE 6.1 : Compatibilité avec les schémas et efficacité des techniques d'implémentation

Technique d'implémentation		Schéma de compilation				
		Dynamique	**S**éparé	**O**ptimisé	**H**ybride	**G**lobal
sous-objets	SO	∘	•	∗	∗	∗
hachage parfait	PH	∘	•	∗	∗	∗
coloration incrémentale	IC	∘	•	∗	∗	∗
caching	CA	∘	•	∗	∗	∗
method coloring	MC	×	•	•	◇	•
binary tree dispatch	BTD	×	×	•	◇	•
attribute coloring	AC	×	•	•	◇	•
accessor simulation	AS	∘	•	•	◇	•

• : Testé, ∘ : Extrapolé, ◇ : Compatible mais non testé, ∗ : inintéressant, × : Incompatible

CHAPITRE

7 Le compilateur PRMC

Ce chapitre décrit l'architecture modulaire du compilateur PRMC. *Il y sera aussi présenté les grands points du méta-modèle concret et les choix faits pour la génération de code. Nous expliquerons comment les différents schémas de compilation (voir Chapitre 6) et les diverses techniques d'implémentation alternatives (voir Chapitre 5) sont mis en œuvre dans le compilateur. Finalement, nous parlerons du problème du* bootstrap *et de la manière dont il est résolu pour ce compilateur.*

Sommaire

Dans notre travail le compilateur est central. C'est d'abord un objectif de produire un compilateur pour expérimenter un nouveau langage (thèse de Jean Privat [2006]), puis de le *bootstrapper* pour produire un compilateur écrit dans ce nouveau langage (voir Section 7.4). Le compilateur peut alors servir d'outil pour construire des expérimentations où il servira à la fois de programme et de donnée (voir Chapitre 8)

7.1 Architecture du compilateur

Le compilateur PRMC est un compilateur du langage PRM (voir Section 3.4) produisant des programmes exécutables — via le langage C (voir Section 7.3). Il est écrit en PRM, on

parle dans ce cas de compilateur autogène [1]. Une des grandes difficultés dans un compilateur auto-gène est son initialisation, ce problème est connu sous le nom de *bootstrap* et il est détaillé dans la section 7.4. Dans notre cas, il a été résolu par l'utilisation d'un prototype écrit en RUBY par Jean Privat [2006], bien que PRMC soit maintenant incompatible avec ce premier prototype.

Le module est l'unité de code du langage et il est matérialisé par un fichier source. Un programme PRM est constitué de modules, dont un principal (*bottom module*) qui dépend de super-modules (voir Section 3.3.3, 7.5.2).

L'objectif de ce compilateur est de tester les techniques d'implémentation et les schémas de compilation présentés dans cette thèse, *ceteris paribus* (toutes choses égales par ailleurs). L'efficacité du compilateur en lui-même n'est pas une priorité, seule sa modularité en est une. Pour ce faire, il utilise le mécanisme des modules et du raffinement de classes (voir Section 3.3) et il donc constitué d'un ensemble de modules gérant chacun une préoccupation (Figure 7.1). Ce découpage en modules permet d'avoir des unités de code courtes mais cohérentes.

7.1.1 Groupe de fonctionnalités

Les modules du compilateur peuvent se classer en grandes catégories, qui correspondent à des préoccupations connexes. Pour simplifier les explications, nous ne décrirons pas les modules à l'unité et nous les regrouperons en groupes de fonctionnalités, que nous appellerons *cluster* — bien que ni ce terme ni ce concept n'aient de signification, actuellement [2], dans le langage.

Les principaux groupes de fonctionnalités sont les suivants :
- les modules gérant le méta-modèle (*cluster* 1).
- les modules d'analyse syntaxique (*cluster* 2).
- les modules d'analyses de types (*cluster* 3).
- les modules gérant les techniques d'implémentation (*cluster* 4).
- les modules de génération de code C (*cluster* 5).
- les modules d'outils concrets (*cluster* 6).
- les modules servant à modéliser les grandes tâches des outils de la plate-forme (hors *cluster*).

Comme ni ce chapitre ni cette thèse n'aient vocation d'entrer dans les détails de l'implémentation du compilateur, nous ne la présenterons que dans ses grandes lignes. Les sections 7.2, 7.3 et 7.5 traiterons, respectivement, des *clusters* relatifs au méta-modèle, à la génération de code et à l'intégration des techniques d'implémentation et des schémas de compilation dans le compilateur. Les autres *clusters* seront rapidement décrits ci-dessous.

1. Compilateur écrit dans son propre langage.
2. Ce point a cependant été reconsidéré par le langage NIT, petit frère du langage PRM.

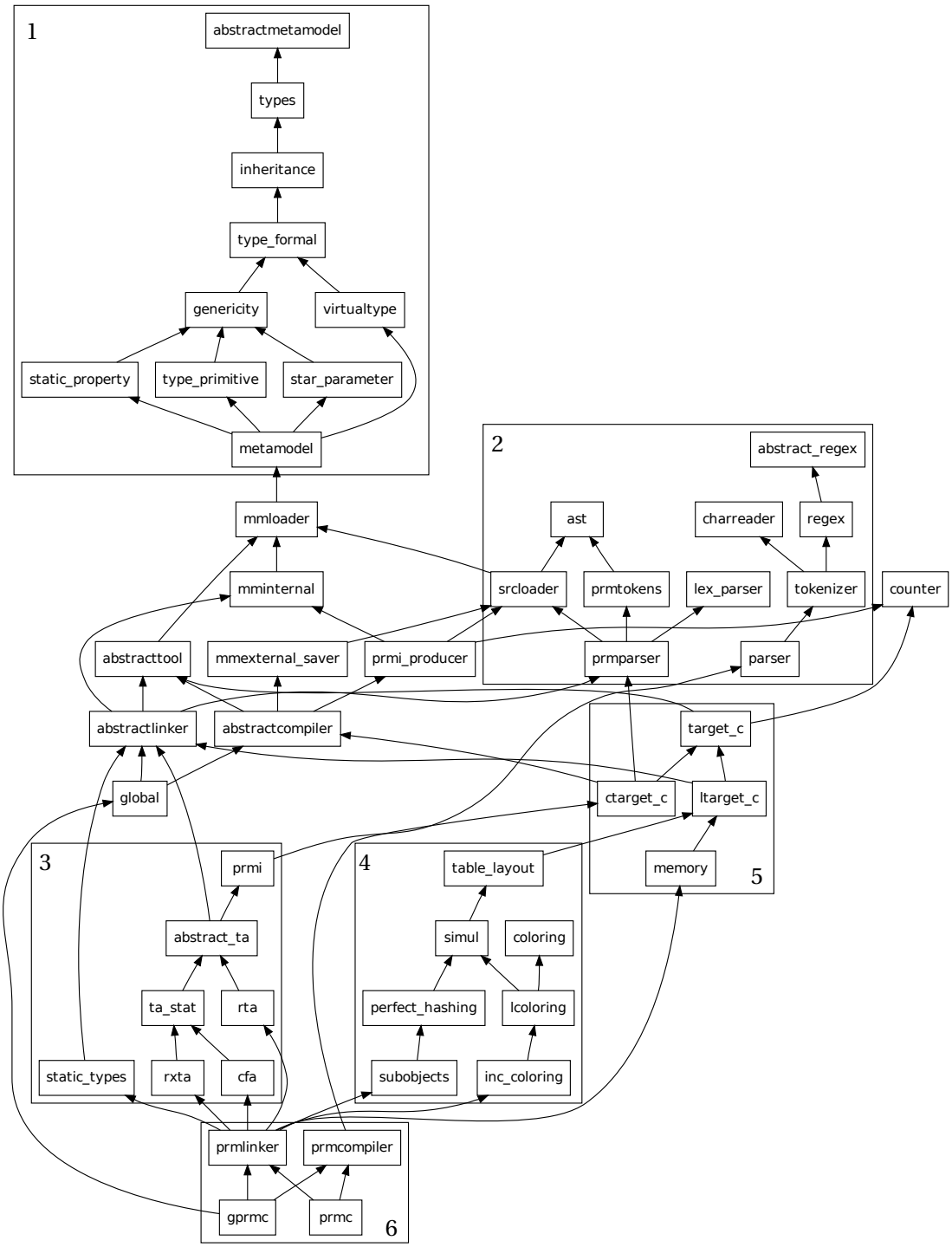


FIGURE 7.1 : Les modules du compilateur

Analyse syntaxique

Pour des raisons historiques, l'analyse syntaxique dans PRMC est réalisée par deux analyseurs différents : FLEX et PRMCC. Ceci explique le nombre élevé de modules dans son *cluster*.

Le premier a été utilisé pour l'analyse du code source PRM. Comme FLEX ne génère que du C, l'intégration à PRM est réalisée au travers de *bindings* (voir Section 7.3.3). Le générateur d'analyseur syntaxique PRMCC a ensuite été développé par des étudiants. Il doit à terme remplacer FLEX, cependant ceci n'a pas été fait en raison de ses piètres performances. Néanmoins PRMCC est tout de même utilisé pour la lecture des modèles internes.

Analyses de types

A l'instar de l'analyse syntaxique, les analyses de types existent aussi en deux variantes dans le compilateur. On retrouve d'un côté les modules `static_types` et `rta`, le premier étant une implémentation triviale de CHA basé uniquement sur le méta-modèle et le second une implémentation allégée de RTA. Les autres modules (`rxta` et `cfa`) proposent une implémentation de CHA, RTA et XTA — pour le premier module — ainsi qu'une implémentation de 0-CFA — pour le second. Ces dernières implémentations d'analyses de types nécessitent les modèles internes complets des modules, ceux-ci étant lus par l'analyseur syntaxique PRMCC. Comme cet analyseur est lent, le compilateur utilise les deux premières implémentations d'analyses de types — cette version RTA se contente de modèles simplifiés lus de manière *ad hoc*.

7.2 Méta-modèle

Remarque. Tous les éléments du méta-modèle sont préfixés dans le code source par `MM`. Pour des raisons de lisibilité, le préfixe est sous-entendu dans les figures comme dans le discours.

Cette section présente le méta-modèle du langage PRM tel qu'il est implémenté dans le compilateur PRMC (figure 7.2). L'implémentation du méta-modèle de PRM est conforme à celui présenté dans la section 3.3, le compilateur l'étend principalement en rajoutant les entités concrètes. Aux entités déjà présentées, nous rajoutons le système de types du langage, un *contexte* et un ensemble de classes *ancêtres*. Le *contexte* est une classe singleton [3] qui connaît tous les modules ainsi que la hiérarchie de classes. Elle sert de point d'entrée aux différentes requêtes sur le modèle. Les classes *ancêtres* réifient les relations de spécialisation et de raffinement ainsi que les instanciations des classes génériques.

Le protocole d'instanciation du méta-modèle est réalisé par le module `inheritance`, il est lui aussi conforme à celui présenté dans la section 3.3.5.

3. Une classe singleton est une classe qui n'a qu'une seule instance.

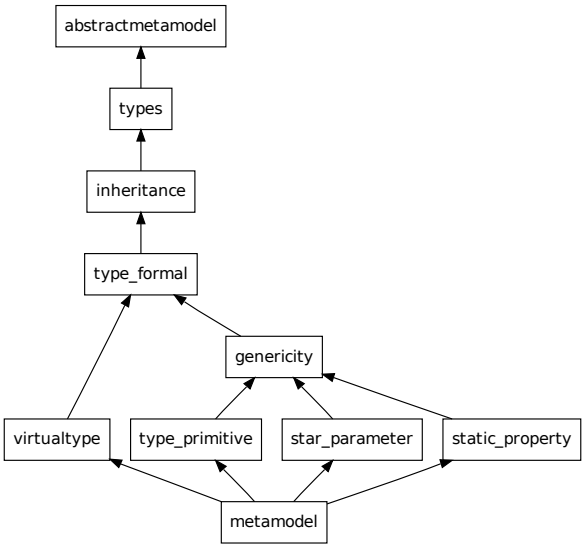


FIGURE 7.2 : Les modules implémentant le méta-modèle

7.2.1 Système de types

Dans l'implémentation complète, le méta-modèle de PRM est complété par un système de types. Les types sont implémentés parallèlement aux classes, bien qu'il existe certaines connexions — par exemple, une classe peut spécialiser une classe générique avec certains types instanciés. Toutes les opérations relatives aux types sont réalisés après l'instanciation du méta-module, il s'agit principalement de la phase de vérification de types (voir Section 6.3.2).

Comme PRM est un langage statiquement typé, toutes les entités du programme sont annotées par des types. Il existe dans ce langage deux catégories [4] de types : les types associées à un classe et les types formels.

Type classe Ces types représentent tous les types qui peuvent être directement associés à une classe, qu'elle soit, ou non, paramétrée — nous appelons *types simples* les types associés à des classes non paramétrées. Les types génériques sont associés à des classes paramétrées, instanciées ou non. Néanmoins, l'instanciation d'une classe générique nécessite que les paramètres du type générique soient aussi instanciés (voir Section 2.2.4).

Remarque. Cette distinction entre type simple et type générique est somme toute

4. La catégorie spéciale `TypeNone` sert uniquement à typer `nil` et devrait disparaître dans une prochaine version.

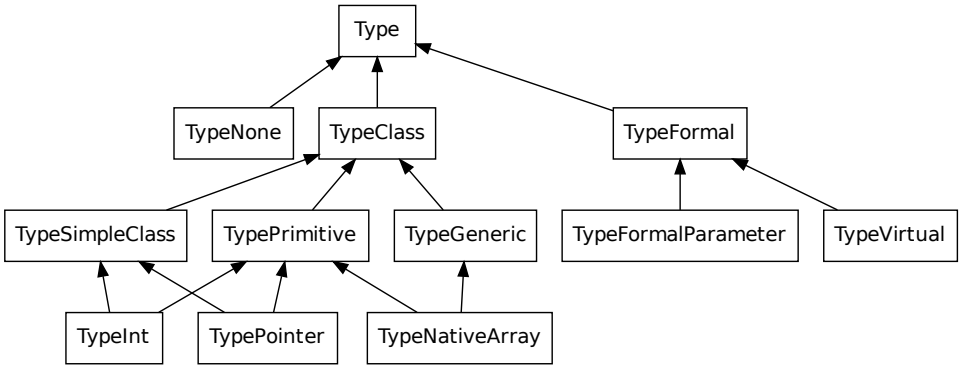


FIGURE 7.3 : Hiérarchie des classe de type

relativement futile, les types simples ne sont que des types associés à des classes génériques sans paramètre. L'implémentation des classes ne fait d'ailleurs aucune distinction, cependant pour des raisons de modularité et d'efficacité, ce choix n'a pas été retenu pour l'implémentation des types.

Types formels Les types formels représentent les types qui ne peuvent pas être directement associés à une classe. En PRM, il existe deux catégories de types formels, les paramètres des types génériques et les types virtuels. Ces deux catégories de types imposent tout de même une restriction sur leur valeur en étant bornées.

Comme l'implémentation des génériques est homogène, tous les appels faits sur des types génériques, durant la compilation d'une classe, sont implémentés comme des appels faits sur le type de la borne, c'est-à-dire qu'il n'y a que la vérification de type qui tienne compte de la valeur instanciée par un type formel.

Types primitifs Dans le compilateur, les types primitifs sont implémentés par le module `type_primitive` et sont sous-classe de `TypePrimitive`. Cette classe gère entre autres la gestion des boites — initialisation, mise ou sortie de boites — et le court-circuit de certaines méthodes.

Les types primitifs de PRM sont restreints aux classes suivantes :
- `Int` : pour les entiers signés (32 bits).
- `Real` : pour les nombres flottants (32 bits seulement).
- `Bool` : pour les valeurs booléennes (`true`, `false`).
- `Char` : pour les caractères (8 bits, alignés sur 32 bits).
- `NativeArray` : pour les tableaux natifs (alignés sur 32 bits).
- `NativeString` : pour les chaînes de caractères. L'utilisation de `NativeString` n'aligne pas les caractères sur 32 bits.

– `Pointer` : pour l'interface avec les types externes au langage.

La classe `Pointer`. La classe `Pointer` est une classe particulière du langage. Elle sert à représenter des pointeurs externes à PRM. À la différence des autres classes primitives, elle peut être spécialisée — ses spécialisations sont aussi primitives. Comme ce n'est pas une vraie classe du langage, elle n'est pas directement instanciable — son instanciation doit être faite dans le langage sous-jacent, donc au travers d'une méthode externe (voir Section 7.2.2).

Littéraux

Les littéraux de PRM sont majoritairement des valeurs immédiates associées aux types primitifs, il s'agit des booléens, des caractères et des nombres. Le compilateur traitant énormément de chaînes de caractères, nous avons traité les chaînes immutables littérales. Durant la compilation d'un module, chaque chaîne qu'il contient est associée à une variable et une méthode d'initialisation est générée pour le module. Cette méthode instancie toutes les chaînes et les affecte aux variables.

Pour accélérer la comparaison de ces chaînes constantes et limiter la consommation mémoire qu'elles requièrent, un dictionnaire — unique et commun à tout le programme — les référence. Ce dictionnaire est consulté avant l'initialisation d'une chaîne et, lorsqu'elle est trouvée, la chaîne déjà initialisée est réutilisée. De ce fait, deux chaînes littérales peuvent se comparer en comparant seulement leur référence. Ce genre d'implémentation des littéraux a déjà été utilisée par SMART EIFFEL [Zendra et Colnet, 2000]. Néanmoins, par rapport à leur proposition, nous avons rajouté des classes spécifiques aux chaînes mutables, immutables et littérales.

L'utilisation du dictionnaire a un coût : les chaînes qu'il référence ne seront jamais libérées par le ramasse-miettes. Cependant le nombre de chaînes utilisées par un programme est constant et la mémoire ainsi occupée est limitée.

7.2.2 Propriétés locales

Dans le compilateur, il a été fait le choix de ne spécialiser que les propriétés locales, la classe `GlobalProperty` est la même pour tous les types de propriétés. Cependant quand une instance de `GlobalProperty` est associée à un type de propriétés, elle n'en change pas.

Les trois types de propriétés locales présents en PRM sont chacun une sous-classe de `LocalProperty` (Figure 7.4). Les méthodes sont associées à une `Implémentation` qui associe la méthode à son corps et tient compte des spécificités de la méthode.

Nous ne détaillerons que les catégories de méthodes que le compilateur doit traiter différemment.

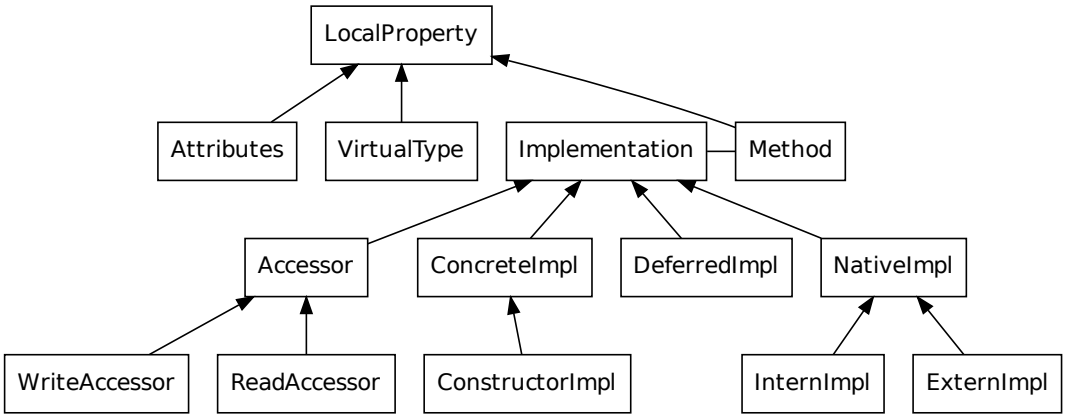


FIGURE 7.4 : Hiérarchie des propriétés locales

Méthodes abstraites

Les méthodes *abstraites* (`abstract`) et *différées* (`deferred`) sont des méthodes sans implémentation associée. Les méthodes abstraites — relativement classiques dans les langages de programmation objet [5] — servent à introduire des méthodes sans y associer de corps. Si une classe possède (introduit ou hérite) une méthode abstraite alors cette classe doit être abstraite. Les méthodes différées — rendues nécessaires par le raffinement de classe — ont la même utilité mais n'imposent pas à la classe de définition d'être abstraite. Elles servent à dire au programmeur (et au compilateur) qu'il existera une implémentation pour cette méthode dans un module plus spécifique. L'absence d'implémentation pour une méthode *différée* ne produit pas d'erreur, sauf si elle est appelée. Cependant, si l'option `-no-deferred` est spécifiée, le compilateur provoquera une erreur lors de l'édition de liens signalant le problème.

Accesseurs

Les *accesseurs* sont des méthodes générées par le compilateur si l'attribut est annoté par les mots-clefs `def_read` et `def_write` [6]. A l'origine, ces mots-clefs auraient simplement dû être du *sucre syntaxique* mais, pour permettre une optimisation des accesseurs, il est préférable d'en faire une catégorie spécifique.

5. En C++, ces méthodes sont appelées *virtuelles pures*.
6. Ces mots-clefs ne sont pas définitifs.

Méthodes natives

Les méthodes *internes* et *externes* sont des méthodes natives qui servent à interfacer PRM avec son implémentation sous-jacente en C. Les méthodes *externes* sont de « vraies » méthodes, seul leur code n'est pas disponible dans le langage — elles servent principalement à communiquer avec les fonctions du système d'exploitation ainsi qu'à rajouter au langage des bibliothèque externes. Les méthodes internes sont nettement plus particulières, elles servent à représenter les courts-circuits que peut faire le compilateur. Ce ne sont pas des « vraies » méthodes, dans le sens où elles ne sont pas forcément soumises à la liaison tardive. Si l'appel d'une méthode est statiquement reconnu comme *interne* — par exemple, l'appel à la méthode + dans le code suivant : **let** `a := 2 + 3` —, le compilateur court-circuite l'appel et lui substitue le code natif correspondant.

Remarque. Comme il n'y a plus de polymorphisme, les méthodes internes ne peuvent pas être redéfinies (ni raffinées).

Dans le cas où la méthode est appelée de façon polymorphe — par exemple, **let** `a: Number...a := a + b` — le court-circuit n'est plus possible et l'appel à la méthode est fait de manière classique.

Remarque. A cause des *boites* sur les types primitifs, l'appel est plus coûteux qu'un appel de méthode classique.

7.3 Génération de code

Nous ne présenterons pas dans cette section comment le code est en pratique généré mais plutôt les choix qui ont été faits pour la génération de code. La section 7.5 présentera plus en détails la problématique des techniques d'implémentation alternatives.

7.3.1 Génération vers C

Le compilateur PRMC sert à produire des exécutables écrits en langage machine, cependant il ne génère pas directement de l'assembleur. Nous avons fait le choix de produire exclusivement du C, un compilateur C (GCC) générant l'exécutable final.

L'utilisation de langages intermédiaires n'est pas rare dans la réalisation de compilateurs de langage de haut niveau. Le choix de C en tant que langage intermédiaire a déjà été utilisé par de nombreux compilateurs dont SMART EIFFEL [Colnet et Zendra, 1999] et SCHEME BIGLOO [Serrano, 1994].

L'utilisation de C comme langage intermédiaire a de nombreux avantages et bien sûr certains défauts.

Tout d'abord, le code généré par PRMC est nettement plus *portable* — comme le compilateur est autogène, il est lui aussi portable et seul un compilateur C est nécessaire. Ceci vient du fait que C est un langage standardisé (ISO/IEC 9899) et il existe un compilateur

compatible avec quasiment toutes les architectures. Ce qui n'est pas le cas de l'assembleur qui, lui, est dépendant du jeu d'instructions du processeur ainsi que de la syntaxe d'assemblage — assembleur INTEL, AT&T, MOTOROLA, etc. Malgré nos efforts pour générer un code C parfaitement standard, nous utilisons une syntaxe propre à GCC pour certaines macros — il s'agit des blocs comme expressions `({ ... })` qui nous permettent d'éviter les évaluations multiples.

Les compilateurs C modernes produisent un code machine très efficace, notamment grâce à l'utilisation de nombreuses techniques d'optimisation — allocation des registres, évaluations statiques, etc. Le code produit est quasiment aussi efficace que s'il avait été écrit à la main.

Implémenter l'interface extérieure d'un langage est particulièrement simple en C. Les appels système sous UNIX sont disponibles en tant que fonctions C et leur convention d'appel — passage de paramètre et valeur de retour — sont ceux de la norme C. C'est pour cette raison que quasiment tous les langages de programmation permettent d'interopérer avec du C. Enfin, il existe un grand nombre de bibliothèques écrites en C, donc facilement intégrable au langage. En plus des appels système basiques, nous avons *bindés* CDB, FLEX, SQLLITE et une partie de SDLIMAGE — cependant ces *bindings* ne sont pas tous maintenus.

A part les concepts objets, les concepts utilisés en C sont proches de ceux de PRM— c'est un langage procédural, avec des variables, des fonctions, des boucles, des tests, etc. Cette proximité entre les langages nous permet de générer du code dit *orthodoxe*, c'est-à-dire proche du code d'origine [Serrano, 1994]. Le code généré est directement lisible par un humain et ressemble à un code qui aurait pu être écrit à la main [7].

Enfin, l'utilisation de C nous permet de disposer d'un environnement de développement complet. En effet, il existe un grand nombre d'outils de développement pour C. Sans être exhaustif, on peut citer les *débogueurs* (GDB), les profileurs (GPROF) et les outils de vérification de la suite VALGRIND (fuite mémoire, défauts de cache, etc).

En dépis de tous ces avantages, l'utilisation de C présente quelques inconvénients pour l'usage que nous en faisons.

Toutes les structures bidirectionnelles — qu'il s'agisse de la coloration, des tables de hachage ou même de l'allocation des sous-objets — nécessitent des ajustements car il n'existe aucune syntaxe pour les exprimer en C. En général, pour les structures statiques, la solution que nous avons adoptée consiste à utiliser un symbole fictif et à le rajouter dans la liste des symboles résolus. Pour les structures dynamiques, nous ajoutons une addition à l'exécution.

Il n'existe pas de structure de contrôle complexe comme les *exceptions*. C ne propose aucun mécanisme standard de détection des erreurs arithmétiques. Il est cependant possible, sous UNIX, de rajouter un gestionnaire d'erreurs pour certains signaux — division

7. Bien que ce code soit verbeux, que tous les noms soient écorchés (voir Section 6.3.4), que les mécanismes objet soient remplacés par des macros à plusieurs paramètres, etc.

par zéro, etc. Dans tous les cas, la gestion des signaux est une opération lourde — et lente — et ne doit pas être utilisée en excès — c'est-à-dire pour les calculs standards.

L'utilisation des symboles est limitée aux adresses — de fonctions ou de variables globales. Cette restriction est principalement due à l'éditeur de liens. En effet, celui-ci n'est capable de substituer les symboles que dans certaines instructions — `load`, `call`, `jump`, etc.

Ce dernier point complexifie sérieusement la mise en place des schémas de compilation séparée (S) et de compilation séparée avec édition de liens optimisée (O).

7.3.2 Gestion de la mémoire

PRM est spécifié pour être utilisé avec un ramasse-miettes, c'est-à-dire qu'il ne propose pas d'instruction de désallocation explicite (voir Section 3.4.4). Le compilateur PRMC n'impose pas de ramasse-miettes particulier, cependant il ne propose actuellement que peu de choix. Le module `memory` et la classe `MemoryManager` — et ses sous-classes — sont responsables de la politique de gestion de la mémoire. Par rapport aux premières versions de PRMC, nous avons utilisé plusieurs allocateurs différents pour les objets *atomiques* — c'est-à-dire ne contenant pas de référence — et pour les objets non collectables — à cause des librairies externes. Le choix du ramasse-miettes est contrôlé par l'option `-gc` du compilateur, qui peut prendre comme valeur `none`, `large` ou `boehm`.

Le ramasse-miettes utilisé par défaut est celui de Boehm [1993], qui présente au moins deux avantages : il est très simple à utiliser et il est portable. C'est un ramasse-miettes *conservatif* qui ne nécessite aucun pré-requis sur le programme pas même sur la forme des références manipulées. Il remplace la fonction `malloc` de la librairie standard C.

Remarque. Il s'agit en fait d'une fonction `alloc`, donc `malloc` est toujours disponible. Cependant la documentation en déconseille l'usage conjoint — il ne doit pas y avoir de référence entre les éléments alloués par les deux fonctions.

Les autres politiques de gestion mémoire provoquent des fuites mémoire importantes, car la mémoire n'est jamais libérée.

L'impact temporel du ramasse-miettes sur le compilateur n'est pas négligeable, les expérimentations lui attribuent 50% du temps de compilation (Table 8.5 et 8.9). Pour réduire cet impact, il faudrait limiter l'utilisation des objets inutiles, aider le ramasse-miettes en lui donnant plus d'informations sur les instances manipulées, ou tout simplement utiliser un ramasse-miettes adapté au langage. A terme, le compilateur devrait utiliser le ramasse-miettes de Desnos [2004] qui est spécialement adapté à la coloration bidirectionelle (voir Section 5.2).

7.3.3 Interopérabilité

L'interopérabilité de PRM est pour le moment limitée au langage C. Elle est réalisée au travers de deux artifices, les sous-classes de `Pointer` et les méthodes *natives*.

Les sous-classes de `Pointer` sont des références directes à des objets externes (voir Section 7.2.1) et ne peuvent pas être instanciées directement en PRM. Du point de vue du langage, ce sont des types primitifs, ces instances sont donc mises en boites dès qu'elles sont utilisées de façon polymorphe.

Les méthodes natives — c'est-à-dire définies `extern` ou `intern` — n'ont pas de code en PRM (voir Section 7.2.2). Les méthodes externes font des appels de fonction et sont soumises au polymorphisme tandis que les méthodes internes ne le sont pas — et sont généralement des macros. Dans les deux cas, elles doivent être déclarées dans un fichier d'entête C (`.h`) associé au module qui l'a introduit.

Un module (fichier `.prm`) peut s'accompagner d'un fichier de configuration externe (fichier `.extern`) du même nom. La syntaxe de ces fichiers est très simple, il s'agit d'un fichier texte où chaque ligne est un couple `clé=valeur`. La clé peut prendre une des six valeurs suivantes :

cheader nom d'un fichier d'entête C a inclure dans le module et les modules qui en dépendent.

cbody nom d'un fichier C à compiler et à lier à l'exécutable final.

cflags paramètres supplémentaires à donner au compilateur pour compiler le module et les modules qui en dépendent.

cflags_exec comme `cflags`, sauf que les paramètres donnés au compilateur sont le résultat de l'évaluation de `valeur` par l'interpréteur de commande (*shell*).

clibs librairies supplémentaires à rajouter lors de l'édition de liens finale.

clibs_exec comme `clibs` mais avec le comportement de `cflags_exec`.

7.4 Bootstrap

Le compilateur PRM est écrit en PRM et, pour pouvoir le compiler, il est donc nécessaire de disposer d'un compilateur PRM. Ce problème est connu sous le nom de *bootstrap*— ou amorçage —, c'est une variante de l'éternel problème de « la poule et de l'œuf ».

Remarque. Le terme *bootstrap* est censé faire référence aux aventures du baron de Münchhausen, dans lesquelles il se serait sorti d'un marécage où il était embourbé, ceci juste en se tirant par sa queue de cheval — et non pas en tirant sur sa boucle de chaussure comme le terme semblerait l'indiquer[8]. D'après wikipédia, le terme serait, en réalité, dérivé d'une

8. D'après wiktionnary : *bootstrap* est une languette fabriquée en cuir (ou dans une autre matière) qui est cousue sur certaines bottines pour pouvoir les enfiler plus facilement.

histoire toute aussi invraisemblable du début du XIX° siècle aux États-Unis : « *pull oneself over a fence by one's bootstraps* ».

Dans tous les cas, le terme *bootstrap* fait référence, de façon humoristique, au fait qu'un système puisse s'auto-amorcer depuis un état indéfini.

Le fait d'écrire le compilateur d'un langage dans ce même langage peut, de prime abord, sembler un peu farfelu. Néanmoins, si l'on ne considère pas le problème du *bootstrap*, ceci présente de nombreux avantages :

- Écrire un compilateur constitue un test non trivial de la viabilité du langage. Le langage est au moins capable d'exprimer son propre compilateur.
- Une fois résolu le problème du *bootstrap*, le langage et son compilateur peuvent être étendus sans disposer d'un autre compilateur. De plus, les programmeurs n'ont qu'un seul langage à connaître.
- Les extensions apportées au compilateur permettent d'améliorer l'efficacité ou l'expressivité des programmes de ce langage. Tout en améliorent le compilateur lui même — en le rendant plus efficace ou plus simple à maintenir.
- Enfin, ceci constitue un très bon test de non régression du compilateur lui-même.

7.4.1 Le *bootstrap* initial

Le *bootstrap* d'un nouveau langage est réalisé en deux étapes. Tout d'abord il faut pou-

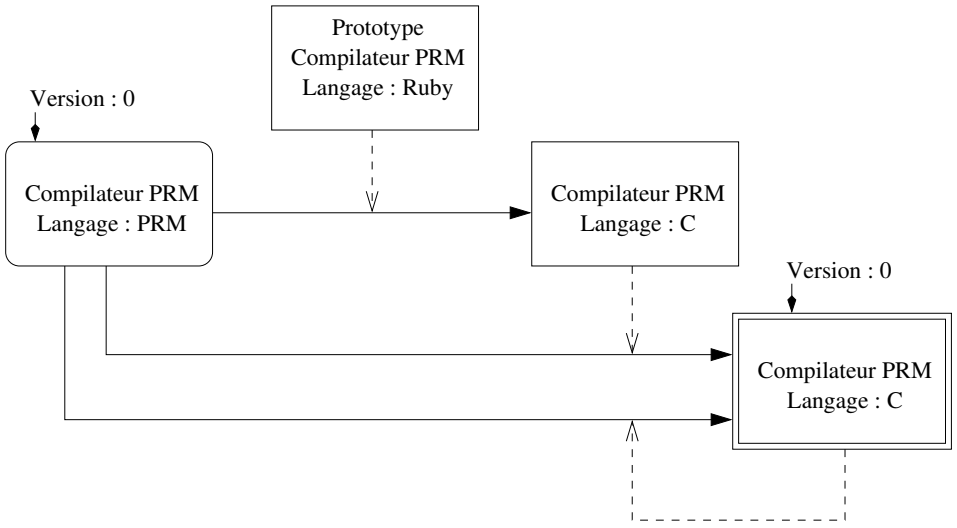


FIGURE 7.5 : *bootstrap* du compilateur PRM

voir compiler le compilateur initial. Il faut pour cela, soit disposer d'un compilateur pour ce langage, soit réaliser cette compilation « à la main ». La deuxième alternative est éventuellement envisageable avec des langages simples, comme l'assembleur, mais avec des

langages riches [9], elle ne l'est plus. La solution la plus utilisée de nos jours consiste à disposer d'un prototype de compilateur écrit dans un autre langage afin de pouvoir compiler une première version d'un compilateur autogène. Cependant, il n'est pas nécessaire que le prototype implémente la totalité du langage, il peut se restreindre au sous-ensemble que le compilateur auto-gène utilise — ou inversement, le compilateur auto-gène peut se restreindre à la partie implémentée par le prototype. Cette solution avait déjà été utilisée pour PASCAL, dont le prototype était écrit en FORTRAN. Dans notre cas, le prototype du compilateur a été écrit en RUBY par Jean Privat [2006] durant sa thèse.

Une fois cette première amorce réalisée, le compilateur est en mesure de se recompiler lui-même. De plus, si le processus de compilation est déterministe, ce qui est souhaitable, le résultat de cette recompilation est un point fixe, c'est-à-dire que si on le recompile une fois de plus l'exécutable obtenu sera le même (voir Section 8.3.3).

En général ce processus est représenté par des *T-diagram* (*Tombstone diagram*) mais nous trouvons qu'il ne fait pas apparaître la « boucle » inhérente à ce processus. C'est pour quoi, nous les avons remplacés par un schéma de notre cru (Figure 7.5). Dans ce schéma, la flèche en trait plein représente la production (partie supérieure du T) et la flèche en pointillé représente l'exécution (barre verticale du T). Les boites carrées sont considérées comme exécutables, bien que la boite RUBY nécessite un interpréteur et les boites C un compilateur — qui ne sont pas représentés sur le diagramme pour ne pas le surcharger. Enfin, la boite arrondie représente le code source tandis que la double boite représente le point fixe.

7.4.2 Ajout de fonctionnalités

Depuis son premier *bootstrap*, le langage a évolué (ajout des types virtuels, retrait des variables statiques, changement de syntaxe, ...) à tel point que le compilateur n'est plus compatible avec cette ancienne version.

L'ajout d'une fonctionnalité au langage présente le même genre de problèmes que le *bootstrap* : le compilateur implémentant et utilisant une nouvelle fonctionnalité doit pouvoir se compiler. Pour réaliser la migration, il est à nouveau nécessaire de procéder en deux étapes (Figure 7.6). La nouvelle fonctionnalité doit d'abord être implémentée dans un compilateur sans être utilisée. Ce compilateur doit ensuite être *bootstrapé* à son tour. Enfin, le compilateur — ou la librairie standard du langage — peuvent être réécrits en utilisant la nouvelle fonctionnalité. Finalement, une version stable — ou du moins complète — est identifiée par un numéro de version pair ($2k$, $2k+2$, ...), les versions impaires ne servant que de transitions.

Considérons, par exemple, l'ajout des types virtuels dans le langage. Pour cela, nous partons d'un compilateur fonctionnant mais ne gérant pas les types virtuels (version $2k$). La première étape consiste à implémenter leur support dans le compilateur, ceci néces-

9. En général, si le langage est riche, son compilateur est plus gros.

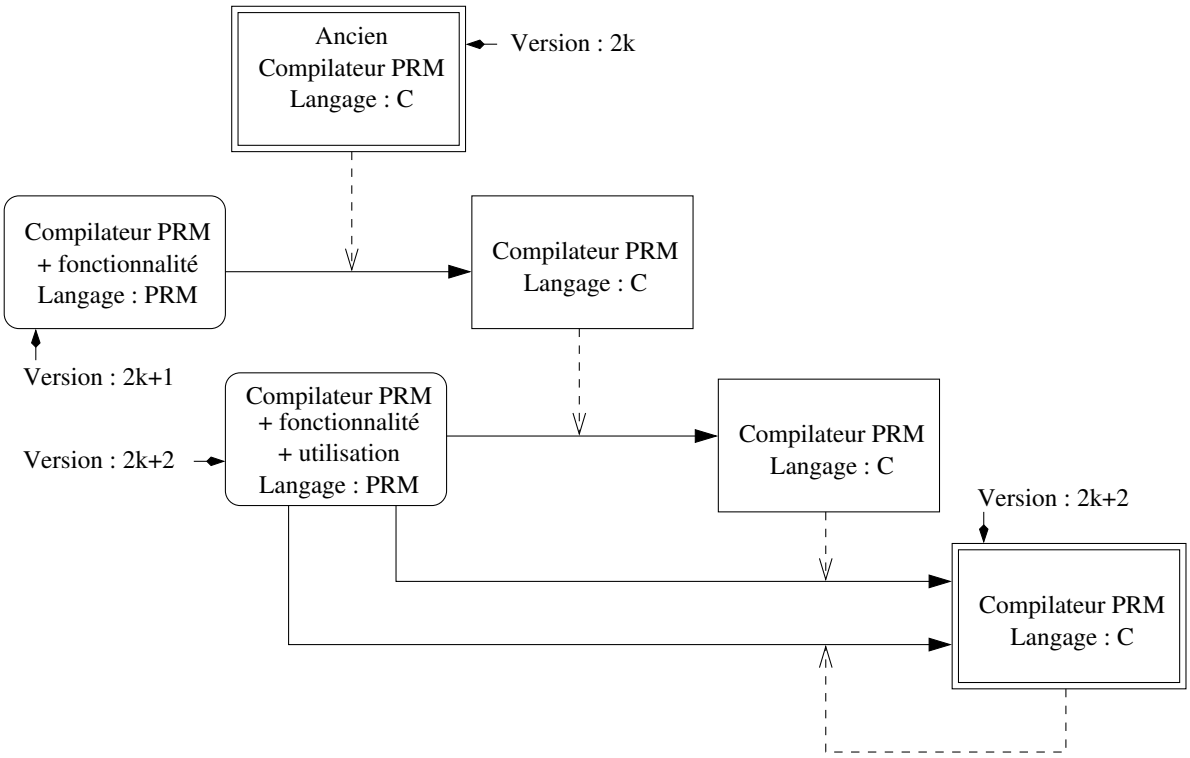


FIGURE 7.6 : Ajout d'une nouvelle fonctionnalité au compilateur PRM

site d'ajouter des règles de syntaxe à l'analyseur syntaxique, d'ajouter un nouveau type de propriétés locales dans le méta-modèle complet et enfin de rajouter le support pour la génération de code (version $2k+1$). Cette nouvelle version est ensuite compilée par le compilateur de la version $2k$. Le compilateur ainsi obtenu peut être testé — par exemple en ajoutant des tests de non régression. Une fois qu'il est considéré comme opérationnel, son code peut être simplifié en utilisant les types virtuels (version $2k+2$). Enfin, cette dernière version est compilée successivement par la version $2k+1$, puis par elle même. S'il n'y a pas d'erreur, un nouveau point fixe est atteint par la dernière compilation, le nouveau compilateur peut être considéré comme utilisable et tous les anciens compilateurs n'ont plus de raison d'être.

Gestion des versions

La gestion de ces fichiers d'entête a soulevé de nombreux problèmes lors des évolutions du compilateur, et un système de version a donc été ajouté au compilateur. Ce système est directement géré par le gestionnaire de code source (*Source Code Management*)

GIT. A chaque *commit*[10], la classe `Sys` est raffinée par un module `version` généré pour l'occasion. Les modifications apportées aux différents entêtes sont sauvegardées dans un répertoire unique qui lui est sous contrôle de version (`prm/lib/intern`). Enfin, un nouveau répertoire dédié à la version à venir, contenant une copie de tous les entêtes, est créé. Grâce à cette solution, nous pouvons utiliser en même temps plusieurs versions du compilateur qui utilisent chacune des macros différentes.

Cette tâche est automatisée par un script appelé par différents *hooks* de GIT (`prm/misc/scripts/git-hook`). L'installation de ces *hooks* est faite automatiquement par l'option `configure` de ce même script.

7.4.3 Distribution du compilateur compilé

A l'instar de SMART EIFFEL, le code source du compilateur est fourni[11] avec une image de lui-même en C. Cette solution permet de rendre portable l'installation du compilateur, il suffit de disposer d'un compilateur C ainsi que des librairies utilisées[12]. L'image du compilateur fournie est une version sans *thunk* afin d'être compatible avec un plus grand nombre de systèmes — en particulier sur les systèmes ne disposant pas d'un `ld` convenable (voir Section 8.3.1).

Remarque. L'image du compilateur est située dans le répertoire `prm/src/c_src` dans lequel on retrouve, entre autre, un script (`compil-Xprmc`) pour compiler l'image (en faisant appel à GCC).

La production d'une image est assurée via la règle `image` du `Makefile` situé dans le répertoire des sources du compilateur. Cette règle, génère une compilation de PRMC, puis par analyse du journal (*log*) de compilation le script est généré et les sources sont récoltés..

7.5 Schémas de compilation et techniques d'implémentation alternatives

La mise en place d'une technique d'implémentation est fortement liée à son schéma de compilation. Cependant l'implémentation tente de s'en abstraire au maximum.

La compilation des modules est faite séparément, elle est ensuite suivie de la phase globale (voir Section 6.2). La compilation purement globale (G), n'est qu'un cas particulier où la phase de compilation des modules ne génère rien et la génération de code est reportée à la phase globale (voir Section 6.4.3).

10. Un *commit* est l'action d'envoyer ses modifications locales vers le dépôt. C'est le vocabulaire d'une transaction et il n'existe pas de terme francophone adapté pour le désigner.

11. Uniquement dans les versions stables. Pour les version de développent, elle sont disponibles en ligne.

12. FLEX et suivant les versions `libgc` (Boehm [1993]).

7.5.1 Compilation séparée

Lors de la compilation séparée — qui correspond à la phase locale —, le code des modules ainsi que les modèles internes et externes doivent être générés. En pratique, durant la compilation d'un module, seules les méthodes qui y sont définies sont compilées. On génère aussi une fonction d'initialisation pour le module. Actuellement, l'utilité de cette fonction d'initialisation est limitée aux littéraux — par exemple, les chaînes de caractères — puisqu'il n'y a pas d'éléments statiques dans le langage.

Dans le corps des méthodes compilées, tous les appels aux mécanismes objet sont réalisés à travers des macros paramétrées par des symboles. Grâce à cela, le code C généré pour un module est indépendant de la technique d'implémentation utilisé, avec deux exceptions :

- pour les arbres binaires de sélection qui nécessitent un code *ad hoc*.
- pour les *thunks*, dans le cas du schéma de compilation séparée avec édition de liens optimisée (O). Lorsque l'on utilise des *thunks*, les appels de macros sont remplacés par des appels de fonction et les macros ne serons utilisées que dans une phase ultérieure (voir Section 7.5.2).

Macros

Tous les mécanismes objet sont implémentés par des macros définies dans un fichier d'entête C— et ce sans intervention d'un module spécifique à la technique sous-jacente. Ce choix peut sembler étrange alors que nous prônons l'utilisation des modules. Néanmoins, il trouve sa justification par le fait que :

- le code produit est plus lisible, puisqu'il est court et toujours identique.
- le code produit est plus simple à tester, puisqu'il suffit de changer un fichier d'entête et de recompiler le code C déjà généré — qui peut avoir été modifié à *la main* pour l'occasion.
- des modules spécifiques n'auraient rajouté qu'une ou deux méthodes utiles pour la phase de compilation séparée et auraient alourdi l'architecture générale — qui est déjà assez complexe.

Les macros sont définies par un fichier d'entête spécifique à chaque technique.

Remarque. En pratique, tous les fichiers d'entête sont inclus par `prm.h` mais leur activation est contrôlée par la définition d'une macro [13]. Ceci permet, par exemple, d'activer seulement certaines parties des définitions du hachage parfait de classes avec les sous-objets. Tout ces fichiers sont situés dans le répertoire `prm/lib/intern-XXXX` — où XXXX représente le numéro de version (voir Section 7.4.2).

Les macros utilisées pour l'implémentation des mécanismes de base sont les suivantes :

- `ATTR_ADDR` : pour l'accès aux attributs.

13. `#ifdef` du préprocesseur C.

- `ISA` et `CAST` : pour le test de sous-typage et le *cast*. En général, une de ces deux macros appelle l'autre.
- `METHOD_CALL0` et `METHOD_CALL2` : pour l'invocation de méthode. La duplication de ces macros est due à une erreur de spécification du préprocesseur C quand il est utilisé avec des macros qui ont un nombre de paramètres variables. La première macro est appelée quand la méthode n'a pas de paramètre, sinon c'est l'autre qui est appelée.

Boites et *tags*. Lorsque ces macros sont appelées sur de *vrais* objets — les boites en font partie —, la table des méthodes est toujours accessible depuis le receveur. Il suffit de suivre le pointeur en tête d'objet. Quand il s'agit de types primitifs *taggués*, ceci n'est plus vrai.

Remarque. Nous ne considérons, ici, que les opérations polymorphes. Lorsque le type statique est connu pour être primitif, tous les appels sont court-circuités (voir Section 7.2.2).

Pour répondre à ce problème, nous utilisons une table de pointeurs vers les tables de méthodes des types *taggués*. Cette table est indexée par la valeur du *tag* — 2 bits de poids faible —, la première entrée de cette table est libre puisque le *tag* 0 correspond à un *vrai* objet. Ensuite, pour passer d'un objet à sa table de méthodes, nous utilisons la macro `VAL2VFT`. Cette macro utilise au besoin cette table pour accéder à la table des méthodes.

Pour éviter de pénaliser les sites où l'objet ne sera jamais *taggués* et ne pas dupliquer toutes les macros qui implémentent les mécanismes objet, nous n'utilisons pas cette macro directement dans le code des macros précédentes. À la place, nous rajoutons un paramètre à toutes les macros précédentes, ce paramètre vaut soit la macro `VAL2VFT` soit sa version simplifiée `_VAL2VFT` — qui se limite à faire l'indirection vers la table de méthodes.

Comparaison et sous-objets. Pour simplifier la lecture du code, les comparaisons entre objets sont aussi implémentées par des macros — à l'exception de certains cas triviaux qui sont directement exprimés par l'opérateur == de C. Néanmoins comme nous l'avons expliqué précédemment (Section 5.10.2), la combinatoire peut être complexe et les sous-objets l'augmentent encore.

Les macros utilisés pour les comparaisons sont les suivantes :

- `EQUAL_OO` : lorsque les deux objets sont de type statique incomparable mais qu'il ne peuvent pas être des boites.
- `EQUAL_XX` : identique à la précédente, sauf que les objets peuvent correspondre à des éléments mis en boites.
- `EQUAL_SO` : lorsque les deux objets sont de type statique différent mais que l'un des deux est une spécialisation de l'autre et qu'ils ne peuvent pas correspondre à des boites.
- `EQUAL_SX` : identique à la précédente, sauf qu'ils peuvent correspondre à des éléments mis en boites.

Dans le cas des sous-objets, les deux dernières macros (celles contenant des S) ne font qu'un seul ajustement de pointeur, les autres en font deux. Pour toutes les autres techniques d'implémentation, les deux dernières macros sont identiques aux deux premières.

7.5.2 Phase globale

Durant la phase globale, le modèle complet du programme est construit, avant de construire les structures de données supportant les mécanismes objet. L'édition de liens finale peut enfin avoir lieu.

Sémantique du programme et *bottom* module

La sémantique d'un programme PRM est donnée par une sémantique de mise à plat (*flattering*) qui correspond à la projection de la hiérarchie de modules sur le *bottom* module. Le *bottom* module est le module unique qui dépend de tous les modules utilisés par le programme. Si ce module n'existe pas, il peut être créé pour l'occasion — bien qu'actuellement le compilateur ne le fasse pas tout seul. Le modèle complet de l'application correspond donc à l'aplatissement des classes locales sur le *bottom* module, c'est-à-dire qu'il ne contient qu'une seule classe locale par classe globale — la plus spécifique — et toutes les propriétés de ces classes sont les plus spécifiques — aux appels à super près.

Après aplatissement, le *bottom* module est une hiérarchie de classes « normale ». Toutes les opérations suivantes sont donc réalisées sur ce module, qu'il s'agisse de l'analyse de types ou du calcul des structures de données.

Calcul des structures de données

Une fois l'aplatissement réalisé, le modèle de l'application est connu et la construction des diverses structures supportant les mécanismes objet peut avoir lieu. Les structures sont construites en deux temps, d'abord le calcul des différentes tables, suivi de leur génération. Ceci est réalisé par un module spécifique à chaque technique d'implémentation (Figure 7.7) par des sous classes de `LayoutGenerator` — introduite dans le module `abstractlinker`.

Le calcul peut nécessiter des algorithmes *ad hoc* — calcul des tables de hachage, coloration, etc —, qui peuvent être fait sur chaque classe unitairement — comme pour le hachage parfait de classes— ou sur la totalité des classes — comme pour la coloration. Parallèlement à la construction des tables, les positions des diverses propriétés sont affectées aux symboles correspondants. Une fois le calcul réalisé, les tables peuvent être effectivement générées. Elles contiennent pour chaque méthode l'adresse de la propriété locale la plus spécifique par rapport à la sémantique du *bottom* module — ou l'adresse d'un *thunk*. En pratique, à la place des adresses ou des valeurs réelles des identifiants de classe, on

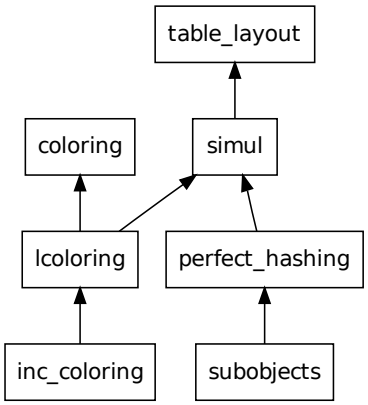


FIGURE 7.7 : Modules gérant les techniques d'implémentations

utilise à nouveau des symboles — nom de la fonction appelée (ou nom du *thunk*) ou un symbole spécifique (identifiant de classe).

De plus, comme la création d'une instance est implémentée par recopie d'un *patron*, celui-ci est aussi calculé et généré en même temps et de la même façon que les tables de méthodes. Ce patron contient un pointeur vers la table de méthodes de la classe ainsi que la valeur par défaut des différents attributs. L'utilisation d'un patron permet d'initialiser simplement et efficacement les attributs avec des valeurs constantes [14]. Les boites font exception à ce mécanisme et sont créées de façon *ad hoc*.

Symboles

Une fois la position d'une propriété connue, elle est associée à son symbole. L'association en tant que telle est faite par utilisation d'un script fourni à l'éditeur de liens — pour (S) et (O). La syntaxe de ce script pour GNU `ld` est très simple, il s'agit d'entrées dans un fichier texte de la forme : `symbole = valeur;`, où valeur peut être une constante (en hexadécimal), un autre symbole ou une expression arithmétique simple (+, −, ∗, ≪, etc) contenant des constantes ou des symboles.

Les symboles ne sont normalement utilisables que pour des adresses — fonctions, variables globales, etc. Pour pouvoir les utiliser comme des positions, il nous faut « tromper » l'éditeur de liens. Ceci est fait en C en demandant l'adresse d'une fonction fictive (donc en utilisant `&symbole`, au lieu de `symbole`), le nom de cette fonction étant le nom du symbole.

14. Cependant, les initialisations constantes ne sont actuellement gérées ni par le langage ni par le compilateur.

A cause du raffinement de classes, la propriété locale appelée par un `super` ou celle appelée sur un type primitif — à l'exception des méthodes internes — est inconnue durant la phase de compilation séparée. On utilise donc aussi des symboles, et la même mécanique est utilisée pour les associer.

Thunks

Lorsque l'implémentation nécessite des *thunks*— (O) principalement — ils doivent être générés et associés à des symboles. Suivant le degré de polymorphisme des sites d'appel, le contenu du *thunk* qui lui est associé est différent. Quand il est monomorphe, le *thunk* est tout simplement court-circuité par identification des deux symboles — celui du *thunk* et celui de la bonne fonction à appeler. Lorsqu'il est oligomorphe, un arbre binaire de sélection spécifique est généré. Enfin, s'il est mégamorphe la macro `METHOD_CALLX` est utilisée — il s'agit de la même macro qui est utilisée dans les implémentation sans *thunk* (voir Section 7.5.1).

Pour limiter le nombre de *thunks* à créer, à chaque fois qu'un *thunk* est généré, son nom est stocké dans une table qui l'associe à la propriété globale initialement demandée et au type statique du receveur. Ensuite, avant chaque génération de *thunk*, on consulte la table pour savoir s'il est vraiment nécessaire de le générer ou si on peut réutiliser celui stocké. Actuellement, les thunks ne tiennent pas compte du fait que certains tests peuvent être évités — test à `null`, test de sous-typage — mais s'ils devaient en tenir compte, il faudrait stocker aussi cette information dans la table.

L'association du symbole au *thunk* est réalisée de la même manière que pour les autres symboles.

Cas particulier de la compilation globale

La compilation globale présente certaines spécificités. Tout d'abord, elle n'utilise pas de phase locale et commence par construire le modèle de l'application aplatie (*bottom* module). Elle applique ensuite l'analyse de types à ce *bottom* module, en ne considérant donc que les propriétés les plus spécifiques — modulo les appels à `super`. Grâce à cette analyse, elle peut marquer les propriétés locales vivantes et limiter ainsi la génération de code à venir à ces seules propriétés. Les appels se passent de *thunks* et, s'ils sont oligomorphes, un arbre binaire de sélection spécifique peut être généré en ligne — c'est le seul cas où les macros ne sont pas utilisées pour l'invocation de méthode. Enfin, pour limiter les modifications à apporter au compilateur déjà existant (séparé), la plupart des symboles est quand même utilisée à la place de leur valeur — qui est pourtant souvent connue — et ils sont définis comme des macros C (`#define`).

7.5.3 Les outils

Bien que les modules permettent de réaliser simplement un ensemble d'exécutables différents, nous avons choisi de réaliser un seul compilateur proposant, par l'activation de drapeaux, toutes les techniques. Ce choix complexifie nettement l'architecture simple que les modules auraient fournis.

Les différents schémas de compilation sont mis en œuvre dans la plate-forme par l'intermédiaire de quatre outils concrets, dont deux compilateurs complets différents : `prmc` et `gprmc`. Le premier permet de faire de la compilation séparée avec édition de liens globale (S) avec l'option `-inline-call` ou édition de liens optimisée (O) — par défaut. Le second permet de réaliser de la compilation globale (G) — par défaut — ou séparée avec édition de liens globale (S) avec l'option `-vft-only`.

Pour des raisons de simplicité d'utilisation, le *bottom* module `prmc` réunit en seul programme le compilateur séparé (`prmcompiler`) et l'éditeur de liens (`prmlinker`). Sans rentrer dans le détail du code de ces deux outils, le module `prmc` est très intéressant. C'est un module très court qui, pour l'essentiel, rajoute une classe en héritage multiple `PRMCompilerLinker`, sous-classe de `PRMCompiler` et `PRMLinker`. Le code de cette classe se limite à régler un conflit de propriété locale sur la méthode `perform_work`, en appelant les deux `super` consécutivement. Qui a dit que l'héritage multiple était si compliqué ?

7.6 Conclusion

Nous avons présenté dans ce chapitre PRMC, le compilateur du langage PRM (voir Section 3.4). Ce compilateur modulaire, grâce aux modules et au raffinement de classes (voir Section 3.3), met en œuvre de nombreuses techniques d'implémentation (voir Chapitre 5) dans le contexte de différents schémas de compilation (voir Chapitre 6).

Le schéma de compilation séparée avec édition de liens globale (S) à été suggéré par Pugh et Weddell [1990] pour la coloration. Mais, à notre connaissance, aucun langage de programmation n'a utilisé cette possibilité avant PRM. Au final, le compilateur PRMC est le premier compilateur à utiliser un certain nombre de techniques :

- Coloration à l'édition de liens.
- Test de Cohen optimisé (sans test de bornes, ni tables de taille fixe).
- Arbres binaires de sélection bornés.
- Hachage parfait de classes.

Cependant si ce travail était à refaire, nous changerions certains des choix qui ont été fait. Tout d'abord, bien que PRM soit un langage récent, une nouvelle version présentant de nombreuses différences, tant syntaxiques que conceptuelles, a été spécifiée et aurait dû être utilisée [15]. Ce nouveau langage se nomme NIT, il propose une meilleure gestion de la visibilité, des blocs (fermetures limitées), de multiples valeurs de retour, de nouveaux

15. Toutefois elle n'existait pas au moment ou se projet a commencé.

mots-clefs pour la redéfinition ou le raffinement, les constructeurs ont été revus, ... De plus, dans ce langage les quelques aberrations de PRM ont été supprimés, absence de mot-clefs pour le raffinement, covariance, ... Son compilateur est aussi autogène mais ne propose que la coloration (normale ou partagée) dans le schéma de compilation séparée avec édition de liens optimisée. Enfin, ce compilateur utilise une représentation du code intermédiaire (IR) lors de la génération de code pour s'abstraire un peu plus du C— bien qu'au final ce compilateur génère toujours du C. Cette représentation manque cruellement au compilateur PRMC et permettrait de nombreuses optimisations sur le code produit.

Toujours dans l'optique de s'abstraire du C, le projet `llvm`, propose une infrastructure, abstraite et assez souple, pour le développement de compilateur, de compilateur à la volée voire même de machine virtuelle. Ce projet est de plus en plus utilisé tant dans le monde académique que libre. `llvm` utilise un *bytecode* pour sa représentation intermédiaire (IR) et propose de très nombreuses optimisations, tant sur le *bytecode* lui-même que sur la projection du *bytecode* vers une machine réelle — allocations des registres, etc. Malheureusement ce projet est entièrement développé en C++, ce qui nécessiterait de s'arrêter à la production du *bytecode* ou de créer de nombreux *bindings*.

Enfin, nous n'aurions jamais dû nous lancer dans le développement de notre propre générateur d'analyseur syntaxique PRMCC. Ceci nous a pris un certain temps, pour au final avoir un générateur plutôt inefficace en terme de performances et ne proposant pas tant de nouvelles fonctionnalités par rapport aux générateurs existants. L'utilisation de SABLECC, surtout depuis sa modification pour NIT aurait été un bien meilleur choix.

Remarque. Le compilateur PRMC n'est qu'un prototype, cependant il n'en demeure pas moins utilisable. Il est disponible en ligne (http://www.lirmm.fr/prm) et son développement est réalisé via un dépôt GIT [16] (http://www.lirmm.fr/~morandat/git/prm-git).

16. http://git-scm.com/

CHAPITRE

8 Expérimentations

Ce chapitre présente les expérimentations qui ont été réalisées avec et grâce au compilateur PRMC *(voir Chapitre 7). S'il n'est pas le plus long, il n'en constitue pas moins l'objectif initial de cette thèse : comparer empiriquement l'efficacité des techniques d'implémentation (voir Chapitre 5) dans le contexte de différents schémas de compilation (voir Chapitre 6), et en vérifier les évaluations a priori. Nous débuterons pour cela par un rappel des estimations a priori, puis nous dresserons le protocole d'expérimentation. Nous nous intéresserons ensuite aux statistiques du programme de test et nous les commenterons. Enfin nous présenterons les différents tests mis en œuvre et leurs résultats seront ensuite discutés.*

Sommaire

8.1 Techniques d'implémentation et schémas de compilation testés

Nous présentons maintenant les schémas de compilation et les techniques d'implémentation que nous avons testés et nous rappellerons leur compatibilité respective. Les

techniques d'implémentation présentées dans cette thèse ne sont pas toutes compatibles avec tous les schémas de compilation.

8.1.1 Schémas de compilation

Actuellement, le compilateur PRMC propose trois schémas de compilation, la compilation globale (G), la compilation séparée avec édition de liens globale (S) et avec édition de liens optimisée (O). La compilation séparée avec édition de liens optimisée utilise des *thunks* pour réaliser les appels de méthodes, *thunks* qui peuvent être court-circuités en cas d'appel monomorphe.

À cause des modules et du raffinement de classes, PRM exclut toute forme de chargement dynamique et le schéma de compilation séparée avec chargement dynamique (D) est extrapolé du schéma de compilation séparée avec édition de liens globale (S). Cependant le code généré avec le schéma (S) est exactement le même que celui qui l'aurait été pour (D), seuls les défauts de cache étant probablement sous-estimés.

8.1.2 Techniques d'implémentation

Les techniques d'implémentation pleinement compatible avec l'héritage multiple proposées au chapitre 5 sont testées par les expériences à venir. Les différentes techniques testées sont :

Coloration de méthodes (MC). Il s'agit de l'implémentation de référence car elle est identique à l'implémentation du sous-typage simple, aux trous près. Elle est utilisée ici dans sa version uni-directionnelle (pas de partie négative) (voir Section 5.2). Cette technique est compatible avec la compilation séparée avec édition de liens globale (S).

Coloration incrémentale (IC). Il s'agit de l'adaptation à l'invocation de méthode de la proposition de [Palacz et Vitek, 2003]. Seuls les défauts de cache et le coût des réallocations dynamiques sont sous-évalués (voir Section 5.6.1). Cette technique est compatible avec la compilation séparée et le chargement dynamique (D)

Hachage parfait de classes (PH). Cette technique est proposée avec trois fonctions de hachage : `and`, `mod` et `and+shift` (voir Section 5.3). Il n'a cependant pas été implémenté de manière optimale en espace mais la différence n'est pas significative sur une hiérarchie de cette taille. Cette technique est compatible avec (D).

Arbres binaires de sélection (BTD$_i$). Les arbres binaires de sélection (voir Section 5.8) sont proposés dans ces expériences dans une version bornée et couplée à la coloration (BTD$_i$) et dans une version non-bornée (BTD$_\infty$). Dans le schéma de compilation globale (G) , ils sont directement mis en ligne dans le code, tandis qu'en (O) ils sont partagés aux travers de *thunks*. La coloration de méthodes (MC) est utilisée quand le degré de polymorphisme d'un site dépasse la valeur i fixée.

Sous-objets (SO) Cette implémentation sans invariant de référence (voir Section 5.1) est couplée au hachage parfait de classes pour les tests de sous-typage (voir Section 5.3.4). Elle est compatible avec (D).

Cache dans les tables de méthodes (CA_x) Le cache dans les tables de méthodes est testé dans une version multiple ($x = 4$) et séparée (voir Section 5.7). Les autres politiques ont aussi été testées mais les résultats ne sont pas inclus dans les expériences suivantes. Les raisons de ce choix seront plus amplement discutées plus tard (voir Section 8.5.2).

8.1.3 Accès aux attributs

Pour toutes les implémentations, à l'exception des sous-objets, nous avons testé deux variantes d'accès aux attributs, avec la coloration d'attributs (AC) ou avec la simulation des accesseurs (AS). La coloration d'attributs (AC), couplée à n'importe quelle technique compatible avec la compilation séparée et édition de liens globale (S), permet d'extrapoler les résultats à une implémentation en sous-typage multiple et avec chargement dynamique (D). Cependant, comme il n'y a pas de distinction fondamentale entre classes et interfaces, l'expérimentation correspond à un programme dans lequel les interfaces seraient beaucoup utilisées.

Un test est représenté par un triplet composé de : la technique d'implémentation utilisée pour l'invocation de méthode et le test de sous-typage, la technique utilisé pour l'accès aux attributs, et enfin du schéma de compilation. La référence prise pour toutes les évaluations à venir est la coloration de méthodes et d'attributs en compilation séparée (MC-AC-S). Ce choix est motivé par le fait que cette configuration a la même efficacité que l'implémentation du sous-typage simple, donc sans le surcoût lié à l'héritage multiple.

8.2 Évaluations abstraites

Les techniques d'implémentation utilisées pour cette expérience peuvent être analysées a priori en se basant sur le modèle de Driesen [2001] (voir Section 4.2.1) pour le temps (Table 5.1).

L'espace est considéré suivant trois aspects (Table 8.1) : le code, les tables statiques et la mémoire dynamique (instances). Le *code* correspond au nombre d'instructions utilisées pour implémenter les mécanismes dans le pseudo-assembleur de Driesen, ces séquences étant produites par la *génération de code* (voir Section 6.3.4). Les *tables statiques* correspondent à l'espace occupé pour implémenter les structures servant de support aux mécanismes objet (principalement les tables des méthodes). Ces différentes structures sont produites par la *génération de l'implémentation* (voir Section 6.3.4). La *mémoire dynamique* représente l'espace utilisé par les instances durant l'exécution du programme.

TABLE 8.1 : Efficacité attendue

Schéma	Technique	Espace			Temps
		Code	Statique	Dyn.	Exécution
D	SO	−	−−	−−	−
	IC	−	+	+++	−
	PH-and	−	−	+++	−
	PH-mod	−	+	+++	−−
	PH-and +CA	−−	−−	+++	−−
	PH-mod +CA	−−	−	+++	−
S	MC	++	++	+++	++
G ou O	$BTD_{i<2}$	+++	+++	+++	+++
	$BTD_{i>4}$	−−−	+++	+++	−−−
	AC	+++	+++	+	++
	AS	+	+	+++	−

+++ : optimal, ++ : très bon, + : bon,
− : mauvais, −− : très mauvais, −−− : déraisonnable

Le temps peut être considéré à l'exécution, à la compilation et au chargement. Le temps d'exécution représente le temps nécessaire pour réaliser les invocations de mécanismes. Celui de la compilation correspond le temps consommé par la *génération de code*. Enfin, le temps de chargement correspond, dans le schéma de compilation séparée avec chargement dynamique (D), le temps pris par le chargement d'une nouvelle unité de code. Pour les techniques ne supportant pas le chargement dynamique, le temps correspond à celui pris par la phase globale. Cependant comme le temps à la compilation et au chargement sont trop dépendants du schéma de compilation sous-jacent, nous n'en présentons pas d'estimation.

La notation va de « −−− » à « +++ », « ++ » représentant l'efficacité de la technique de référence en héritage simple (voir Chapitre 4). Un des objectifs de cette thèse est de vérifier empiriquement ces évaluations théoriques.

8.3 Protocole d'expérimentation

Le dispositif d'expérimentation (Figure 8.1) est basé sur la méta-compilation, c'est-à-dire la compilation du compilateur par lui même. La méta-compilation a déjà été utilisée, par SMART EIFFEL par exemple [Zendra, 2000]. Le test est constitué d'un programme de test P, qui est compilé par une variété de compilateurs $C_i, i \in [1..k]$ (ou par le même compilateur avec une variété d'options). Le temps d'exécution de chacun des exécutables P_i obtenus est ensuite mesuré sur une donnée commune D. Comme le compilateur est

TABLE 8.2 : Caractéristiques des processeurs testés

processeur	nom	fréquence	cache de L2	année	mémoire	
S-1	UltraSPARC III	1.2 GHz	8192 Ko	2001	4096 Mo	distante
I-2	Xeon Prestonia	1.8 GHz	512 Ko	2001	1024 Mo	distante
P-3	PowerPC G5	1.8 GHz	512 Ko	2003	2048 Mo	distante
I-4	Xeon Irwindale	2.8 GHz	2048 Ko	2006	3546 Mo	distante
I-5	Core T2400	2.8 GHz	2048 Ko	2006	2048 Mo	locale
A-6	Athlon 64	2.2 GHz	1024 Ko	2003	2048 Mo	distante
A-7	Opteron Venus	2.4 GHz	1024 Ko	2005		distante
I-8	Core2 T7200	2.0 GHz	4096 Ko	2006	2048 Mo	locale
I-9	Core2 E8500	3.16 GHz	6144 Ko	2008	8192 Mo	locale
I-10	Core2 Q9650	3.0 GHz	6144 K	2008	3862 Mo	locale

le seul programme significatif disponible en PRM, le programme de test *P* est lui aussi le compilateur, tout comme la donnée *D*.

8.3.1 Processeurs testés

Les tests qui vont être présentés ont été réalisés sur plusieurs processeurs afin de d'examiner aussi l'influence de ces derniers. Le tableau 8.2 résume leurs caractéristiques. La plupart d'entre eux sont des processeurs INTEL (I-2, I-4, I-5, I-8, I-9 et I-10) mais nous avons aussi testé des processeurs AMD (A-6, A-7), un processeur SPARC (S-1) de SUN ainsi qu'un POWERPC (P-3) développé conjointement par IBM, APPLE et MOTOROLA. Les INTEL et les AMD utilisent le jeu d'instructions `x86` tandis que le SPARC et le POWERPC utilisent un jeu d'instructions RISC. Tous ces processeurs ont été utilisés en 32 bits, même lorsqu'ils disposaient d'un mode 64 bits.

Tous ces ordinateurs tournent sous système UNIX, la plupart étant des LINUX de la distribution UBUNTU. Le POWERPC fonctionne sous MACOS X 10.5.3 et le SPARC sur SUNOS 5.10. Tous ces ordinateurs disposent d'un compilateur GCC (version 4.0.1 sur MACOS, 4.2.2 sur SPARC et de 4.2.4 à 4.4.3 pour les LINUX). Pour l'édition de liens, GNU `ld` est utilisé sur tous les systèmes (avec des versions légèrement différentes suivant les systèmes [1]). Cependant les versions SUNOS et MACOS X de `ld` n'utilisent pas le même système de substitution de symboles et certains tests n'ont pas été réalisés sur ces systèmes.

Lorsque les machines sont disponibles localement (I-5, I-8, I-9 et I-10) les tests ont été réalisés en *recovery mode*, c'est-à-dire que le système lancé est minimal. Pour toutes les autres, nous nous contentons de lancer les expérimentations la nuit, lorsque ces machines sont moins sollicitées — il s'agit pour la plupart de serveurs faiblement sollicités mais tout de même utilisés.

1. Ceci n'influe pas sur l'efficacité du code généré.

Enfin, à notre grand étonnement, les portables récents — mais pas uniquement — bride le processeur lorsque celui-ci est trop utilisé car il chauffe. Pour remédier à cela, nos tests incluent une pause entre deux tests d'une durée presque aussi longue que le test lui-même. Cette pause a eu un effet bénéfique puisque les résultats qui étaient antérieurement fortement bruités sont devenus très réguliers, voire les plus réguliers.

Note sur la division entière. Pour réaliser les divisions entières, les processeurs INTEL utilisent une méthode exacte, appelée SRT. Les processeurs antérieurs à I-9 utilisent la SRT Radix-4 (2 bits par itération), tandis que depuis les cœurs *penryn*, c'est maintenant la SRT Radix-16 qui est utilisée (4 bits par itération). La latence est donc divisé par deux. D'après les spécifications, I-8 ne devrait pas l'utiliser pas, cependant comme nous le verrons plus tard, l'expérimentation montre le contraire sans que nous n'ayons d'explication à ce sujet. Quand aux processeurs AMD, ils utilisent la méthode de la convergence. Cette technique est censée être plus efficace quand les opérandes sont grandes.

Enfin, la division entière est implémentée par l'unité de calcul flottant, ce qui a pour effet secondaire de rajouter des étapes de conversion et de vider le pipeline du processeur.

8.3.2 Mesures temporelles

Les mesures temporelles sont effectuées avec la fonction UNIX `times(2)` qui comptabilise le temps alloué par l'ordonnanceur système pour un processus, c'est-à-dire le temps utilisé par un processus indépendamment de tout autre processus. Il est nécessaire de prendre cette précaution car tous les systèmes d'exploitation utilisés pour ces tests sont multi-tâches. Toutefois, le changement de contexte entre chaque processus surestime le nombre de défauts de cache car chaque processus change le contenu du cache du processeur.

Comme de nos jours la majorité des processeurs sont multi-cœurs, pour limiter le biais, nous nous assurons qu'un seul cœur est utilisé simultanément — le processus du compilateur est volontairement mono-processus et mono-thread.

Remarque. Les tests présentés n'ont pas pu profiter de cette remarque.

Toujours en vue d'éviter de surestimer les défauts de cache, les processus sont lancés au travers de l'utilitaire `taskset` qui permet de choisir l'affinité (préférence) d'un processus pour un processeur ou un cœur donné. Enfin, la politique d'ordonnancement du système pour le processus est changée de `SCHED_OTHER` à `SCHED_FIFO` — uniquement pour les machines locales sous LINUX. Un processus avec la politique `SCHED_FIFO` ne sera jamais préempté et il ne rendra la main que pour attendre une synchronisation du noyau ou en cas de pause explicite (`sleep`).

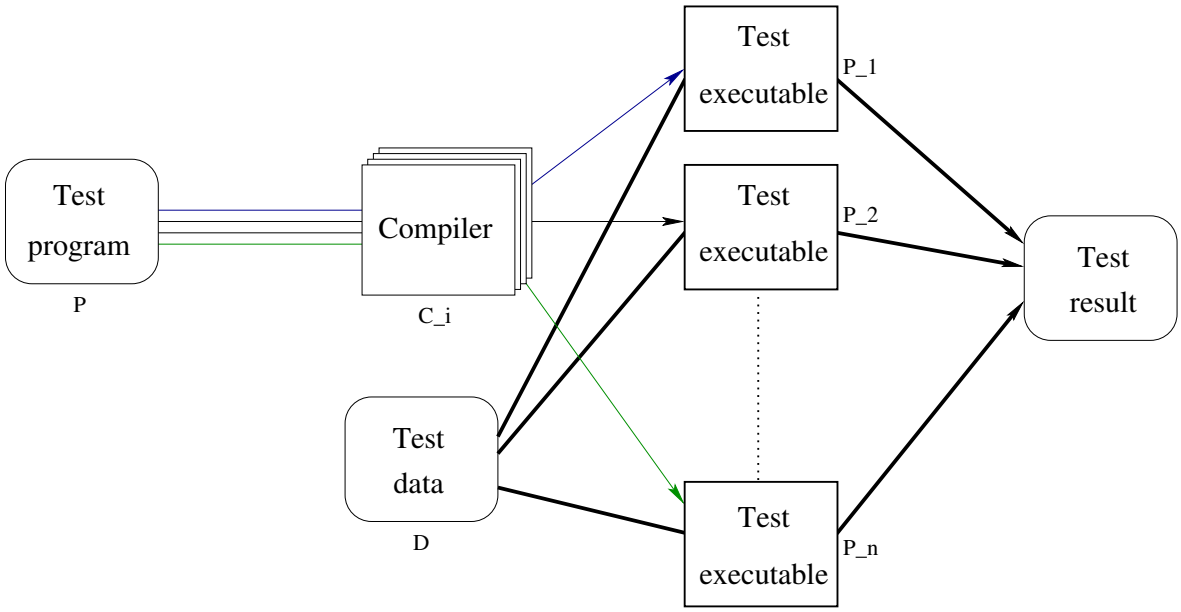


FIGURE 8.1 : Protocole d'expérimentation

8.3.3 Reproductibilité

Le compilateur PRMC est entièrement déterministe, de façon à ce que chaque test soit exactement reproductible. L'ordre de compilation des classes est déterminé par l'ordre de leur déclaration dans chaque module — modulo la relation de spécialisation —, les itérations sur les tables de hachage se font dans l'ordre d'insertion, etc. Deux compilations du compilateur avec les mêmes options produisent le même résultat final à l'octet près. Dans la figure 8.1, les flèches en gras pointent toutes sur le même résultat. Ce n'est pas une métaphore, car le résultat obtenu est strictement identique. Ceci a été vérifié à l'aide de la commande `diff` sur les fichiers C produits et sur les fichiers binaires *strippés*[2]. Malgré tous les efforts faits pour avoir des exécutions identiques, la valeur de hachage de certains objets est dépendante de la position en mémoire de ces objets — la fonction de hachage par défaut est : adresse de la référence/8. Ceci a un effet sur le chaînage des tables de hachage utilisant le *chaînage séparé* (*separate chaining*) ainsi que pour les tables utilisant une *exploration linéaire* (*linear probing*) car les collisions peuvent être différentes d'une exécution à l'autre. En effectuant les statistiques pour la table 8.3 nous avons pu constater que ces variations sont minimes, moins de 1 pour 10000. Ces variations sont donc considérées comme étant en dessous de la précision de la mesure.

2. *Stripper* est un anglicisme mais aucune des traductions françaises ne convient. *Stripper* correspond au fait d'enlever toutes les informations superflues d'un programme — table des symboles, informations de débogage, etc. Sous LINUX ceci s'obtient grâce à la commande « `strip -a` »

Pour les résultats, nous ne considérons que le meilleur temps pris sur plusieurs dizaines d'exécutions (ce nombre étant voisin de cent pour les machines rapides). Un test complet représente donc plusieurs heures de calcul. On peut s'interroger sur le choix de la meilleure mesure, au lieu, par exemple, d'une moyenne. Si i est une technique d'implémentation et j un test, la mesure t_i^j est de la forme $t_i^j = T + \epsilon_i^j$ où T est la grandeur que l'on cherche à mesurer et ϵ_i^j un bruit toujours positif. Lorsque le nombre de tests j augmente, $\min_j(\epsilon_i^j)$ tend asymptotiquement vers 0 (ou vers un bruit irréductible qui peut vraisemblablement être intégré à T sans biais particulier). D'un autre côté, la difficulté avec les moyennes est d'éliminer les cas aberrants par excès, et de savoir où s'arrêter. Conserver uniquement le meilleur temps revient à éliminer tous les autres temps, les aberrations y compris. De plus, les machines testées présentent des bruits très différents, et la prise en compte des moyennes reviendrait à « pénaliser » les plus bruitées (la comparaison du processeur n'est cependant pas l'objectif de ces tests). Finalement, cela revient à considérer que les processeurs sont déterministes. Ce choix a aussi été retenu par Alpern *et al.* [2001b] pour valider leur test de sous-typage.

8.3.4 Comparabilité

Le résultat principal de cette thèse est l'obtention d'une comparaison objective des différentes techniques d'implémentation et des différents schémas de compilation. La reproductibilité est donc un point essentiel mais ce n'est pas le seul. Il est aussi primordial que les résultats soient directement comparables. La différence entre deux tests se limite à l'objet du test *ceteris paribus* — toutes choses égales par ailleurs. En pratique, pour deux versions utilisant des techniques d'implémentation différentes, seules les définitions des macros réalisant les mécanismes primitifs et l'implémentation des tables de méthodes diffèrent. Le code généré en utilisant une technique c_1 s'obtient en substituant dans le code généré pour une technique c_2, les séquences de c_2 par les séquences de c_1. Pour les différents schémas de compilation, seules les conditions de mise en ligne et d'élimination des tests inutiles diffèrent, le code réalisant les implémentations étant identique. Grâce à cela, deux résultats dans les tables 8.5, 8.6 et 8.9 sur une même ligne ou une même colonne, sont directement comparables, toutes choses étant égales par ailleurs. En particulier, la différence entre deux techniques ne bouleverse pas la localisation du code grâce à la parfaite reproductibilité que les tests garantissent.

De plus, la signification des résultats dépend étroitement de la signification du temps de référence, qui pourrait être grossièrement surévalué, réduisant ainsi artificiellement les différences observées. Il faut donc évaluer le niveau général d'efficacité de PRMC, ce qui réclame une référence externe. Nous avons pris SMART EIFFEL : les deux langages ont des fonctionnalités équivalentes et SMART EIFFEL est considéré comme très efficace (BTD_∞-G). Sur une même machine, le temps d'une compilation de SMART EIFFEL vers C est du

même ordre de grandeur que celui de PRM vers C — une comparaison plus fine ne serait pas significative.

Néanmoins, le ramasse-miettes représente un biais quand deux versions diffèrent sur l'occupation mémoire dynamique — ceci est particulièrement vrai pour l'implémentation par sous-objets de C++. Le compilateur PRMC utilise par défaut le ramasse-miettes conservatif de Boehm [1993], qui n'a pas été optimisé spécifiquement pour chaque implémentation et n'est pas aussi efficace que le serait un ramasse-miettes dédié au modèle objet très simple de PRM dans le cadre des implémentations avec invariant de référence (Invariant 4.1). Aussi présentons-nous les résultats obtenus sans utilisation de ramasse-miettes, seulement sur quelques machines disposant d'assez de mémoire pour éviter de faire appel à la mémoire virtuelle. Ces résultats réduisent légèrement l'écart relatif entre deux tests — car le temps pris par le ramasse-miettes ne dépend pas des techniques (sauf pour les sous-objets) et $T_i = T_i^{no-gc} + T^{gc}$.

8.4 Statistiques du programme de test

A défaut de disposer d'un grand nombre de programmes pour réaliser nos tests, nous les avons effectués sur le seul programme conséquent écrit en PRM, à savoir le compilateur PRMC (voir Section 8.3).

La table 8.3 donne des statistiques sur ce programme de test (P), d'un point de vue statique (nombre d'éléments du code source du programme) et dynamique (nombre d'accès à l'exécution, lorsqu'il est appliqué à la donnée D). Ces statistiques sont calculées sur toutes les variantes, afin de vérifier qu'elles sont identiques. Les statistiques statiques sont comptées par le compilateur lors de la génération du code, tandis que les dynamiques sont mesurées par des versions instrumentalisées du programme de test. Les mesures du temps et les statistiques dynamiques sont effectuées dans des passes différentes afin d'éviter leur influence mutuelle. Pour les appels de méthodes, la table 8.3 les séparent en fonction du degré de polymorphisme du site d'appel. Les lignes BTD_i correspondent à un site d'appel pouvant s'implémenter par un arbre binaire de sélection de profondeur i (mais pas $i-1$) d'après l'analyse de types CHA (voir Section 6.3.3). Donc un site appartient à la catégorie i si et seulement si le nombre de ses branches est compris entre $(2^{i-1}+1)$ et 2^i. Les appels monomorphes sont donc représentés par la colonne BTD_0. Le fort taux d'appels monomorphes (79% des sites d'appels ou 64% du nombre effectif d'appels) de PRMC est cohérent avec les taux rapportés dans la littérature : « la majorité d'un programme objet n'est pas polymorphe ». On peut par exemple se référer aux statistiques statiques qui montrent que beaucoup de méthodes n'ont qu'une seule implémentation [Ducournau, 1997]. Les statistiques montrent que la différence entre les BTD_i et BTD_{i+1} est certainement en dessous de la précision de la mesure. Aussi n'avons-nous inclus que BTD_2 et BTD_∞ dans les résultats présentés.

Les tests de sous-typage étaient sous-estimés dans les premières versions du compi-

TABLE 8.3 : Statistiques sur le programme de test *P*, d'après [Ducournau *et al.*, 2009]

nombres de		statique	dynamique
classes	introductions	532	—
	instanciations	6651	35 M
	tests de sous-typage	194	87 M
méthodes	introductions	2599	—
	définitions	4446	—
	appels	15873	1707 M
BTD	0	124875	1134 M
	1	848	61 M
	2	600	180 M
	3	704	26 M
	4..7	1044	306 M
	8	32	228 K
attributs	introductions	614	—
	accès	4438	2560 M

La colonne "statique" représente le nombre d'éléments du code source du programme (classes, méthodes, attributs) ainsi que le nombre de sites d'appel pour chaque mécanisme. La colonne "dynamique" rapporte le nombre d'invocations de ces sites durant l'exécution (en milliers ou millions). Les appels de méthodes sont comptés séparément en fonction de leur degré de polymorphisme.

lateur, seuls les tests explicites étaient effectués. Ce point a été corrigé dans [Morandat et Ducournau, 2010] mais le chiffre rapporté est celui de la version testée.

Enfin, les taux de réussite du cache dans les tables des méthodes (voir Section 5.7.1), présentés dans la table 8.4, sont mesurés avec un cache couplé au hachage parfait de classes, bien que ces taux ne dépendent pas de la technique d'implémentation sous-jacente. Alors que le monomorphisme d'un site d'appel est une propriété statique (déterminée à la compilation), le taux de réussite du cache est dépendant de la localité temporelle. C'est donc une propriété dynamique dépendant de l'enchaînement des accès à la table des méthodes. En réalité, il est conditionné par le groupe d'éléments accédés au travers de la table de méthodes, qui est déterminé par le type statique de l'introduction d'une propriété et par le genre de propriété accédée (méthode et types virtuels d'un côté et attribut d'un autre, les tests de sous-typage utilisent un cache supplémentaire). Avec la coloration d'attributs (voir Section 5.2), ce taux varie de 60 à 80% en fonction du nombre de caches utilisés et du type de cache considéré (commun ou séparé). Comme nous le supposions, l'augmentation du nombre de caches induit une augmentation du taux de réussite du cache. Cependant, c'est principalement avec la simulation des accesseurs (voir Section 5.5) que la différence est la plus marquée. En effet, il passe de 40% à plus de 60% simplement en les séparant. Les taux rapportés par nos expérimentations sont proches des taux mentionnés dans [Palacz et Vitek, 2003; Alpern *et al.*, 2001b].

TABLE 8.4 : Taux de réussite du cache dans les tables des méthodes

cache	with AC		with AS	
number	separate	common	separate	common
1	68	63	62	39
2	71		64	
4	79		67	

Les taux de réussite du cache sont présentés en pourcentage. Ils sont considérés avec la coloration d'attributs ou la simulation des accesseurs, avec un cache soit commun à tous les mécanismes, soit dédié à chacun d'eux. Chaque ligne représente le nombre de caches utilisés par table de méthodes.

8.5 Résultats et discussions

Les résultats présentés dans les sections qui suivent (Sections 8.5.2 et 8.5.3) sont réalisés à partir de deux jeux de données distincts provenant respectivement de [Ducournau *et al.*, 2009] et [Morandat et Ducournau, 2010]. Le premier concerne uniquement les implémentations avec invariant de référence. Le second compare les sous-objets avec les principales implémentations avec invariant de référence. Enfin, les résultats présentés dans [Morandat *et al.*, 2009] ne sont pas rappelés ici puisqu'ils sont pour l'essentiel un sous-ensemble de ceux présentés dans la section 8.5.2.

Ces tables présentent pour chaque variante et chaque processeur le temps d'exécution par rapport à l'implémentation en coloration pure (M-AC-S). Cette implémentation a été choisie comme référence car elle présente la même efficacité que l'implémentation du sous-typage simple dans ce même schéma (S).

Attention. Ce chapitre contient les résultats d'expérimentations présentées dans plusieurs articles différents [Morandat *et al.*, 2009; Ducournau *et al.*, 2009; Morandat et Ducournau, 2010; Ducournau et Morandat, 2011]. Les résultats de ceux-ci ne sont pas directement comparables car l'intégration de nouveaux mécanismes — les sous-objets par exemple — a souvent nécessité un peu de *reverse engineering* ce qui a impacté les implémentations précédentes. Dans les tableaux de mesure qui font l'essentiel de ce chapitre, chaque tableau présente des mesures comparables qui concernent une unique version du programme de test. En revanche, deux tableaux peuvent concerner des versions différentes du programme de test, qui ne sont donc pas directement comparables.

8.5.1 Ordre de grandeur des différences

Avant de discuter des résultats en eux-mêmes, il est important d'interpréter correctement les ordres de grandeur considérés.

Dans l'ensemble de ces mesures, une différence en-dessous de 1% est insignifiante. D'une part, parce qu'elle est de l'ordre de la précision de la mesure, d'autre part parce

qu'un autre programme de test aurait certainement pu présenter la différence opposée. Les décimales doivent donc être considérées prudemment. En revanche, une différence de 5% est significative : elle est reproductible et peut être raisonnablement extrapolée à d'autres programmes. Une différence de 10% peut être considérée comme grande, les différences se rapportant seulement au surcoût de l'implémentation objet et non au code métier lui même. Enfin, une différence de 50% est dramatique.

Impact du ramasse-miettes. Sauf précision contraire, les mesures incluent le temps consommé par le ramasse-miettes.

Dans la première expérience (Section 8.5.2), ce temps T_{gc} est globalement équivalent pour toutes les versions testées. Comme le ramasse-miettes ne fait appel à aucun mécanisme objet, les rapports demeurent significatifs et ce rapport est augmenté en conséquence. La dernière colonne « I-8 sans GC » permet de donner une estimation du temps consommé par le ramasse-miettes — en faisant la différence des colonnes « I-8 » et « I-8 sans GC ». Le temps consommé par le ramasse-miettes est de 14.5 à 15 secondes pour toutes les variantes à l'exception de MC-BTD-O, soit quasiment la moitié du temps de référence. Les surcoûts sont donc presque doublés par rapport à une version sans ramasse-miettes. Dans la version MC-BTD-O, le surcoût du ramasse-miettes n'est que de 13.4 secondes, cependant nous sommes incapables d'expliquer cette différence.

Dans la seconde expérience (Section 8.5.3), le temps consommé par le ramasse-miettes est nettement plus variable car les techniques considérées n'utilisent pas la même quantité de mémoire. De ce fait, toutes les mesures ont été doublées pour les considérer avec ou sans ramasse-miettes.

Comme le ramasse-miettes dépend du programme considéré, de sa consommation de mémoire et de la mémoire disponible, les tests sans ramasse-miettes ont été réalisés sur des machines ayant au moins 2 Go de mémoire vive, ce qui est suffisant pour que le système et une instance du compilateur fonctionnent sans faire appel à la mémoire virtuelle.

Malgré le coût du ramasse-miettes de Boehm [1993] qui semble prohibitif, son utilité n'a pas à être remise en question. Tout d'abord car le compilateur utilise énormément de petits objets et aucune attention particulière n'a été portée à l'économie de mémoire. Ensuite, ce ramasse-miettes *conservatif* n'a pas — ou quasiment pas (voir Section 7.3.2) — été optimisé pour le modèle mémoire des objets PRM. Une version semi-conservative de ce ramasse-miettes— c'est-à-dire en rajoutant pour chaque classe la description de ses instances (positions des références, etc.) — devrait grandement améliorer ses performances [Jones et Lins, 1996].

8.5.2 Comparaisons des techniques avec invariant de référence

La table 8.5 rappelle les résultats de [Ducournau *et al.*, 2009]. Ils confirment les influences des schémas de compilation sur l'efficacité du code généré et le coût non négli-

geable de l'héritage multiple. Nous présenterons d'abord l'influence des schémas de compilation, puis nous discuterons de l'efficacité des techniques proposées.

Schémas de compilation

Conformément à nos attentes, la compilation globale (G) est une nette amélioration par rapport à la compilation séparée (S). Le fort taux d'appels monomorphes explique principalement ce résultat. La principale différence entre MC-S et MC-BTD_2-G porte sur les appels monomorphes et MC-BTD_2-G améliorent encore l'efficacité — de 10% à 20% au total. La différence entre BTD_0 et BTD_2 étant faible, nous n'avons présenté que ces derniers.

D'un autre côté, la compilation séparée avec édition de liens optimisée (O) ne représente qu'une faible amélioration. L'amélioration apportée par les appels monomorphes est contre-balancée par le coût des *thunks* des appels polymorphes. Ce résultat nous a plutôt surpris, puisqu'il nous semblait que le pipeline du processeur annulerait en grande partie le coût des *thunks*— aux défauts de cache près.

Le chargement dynamique (D) quant à lui, rajoute un surcoût non négligeable par rapport au schéma séparé. Ce surcoût représente le prix à payer pour l'utilisation de l'héritage multiple — ou pour le sous-typage multiple — par rapport à l'implémentation du sous-typage simple dans un contexte de chargement dynamique. Cependant vis-à-vis du sous-typage multiple (sans AS), à part pour les processeurs AMD, cette différence est tout de même plus faible que la différence entre la compilation séparée et la compilation globale.

Enfin, si on cumule tous ces surcoûts, l'écart entre la compilation globale (G) et la compilation séparée avec chargement dynamique (D) est impressionnant. Les écarts minimaux (par exemple pour I-8 et I-9) sont de l'ordre de 15% pour le sous-typage multiple (AC) et 40% dans un cadre d'héritage multiple pur (AS).

Les conclusions sur les différents niveaux d'optimisations globales — pour les schémas (G) et (O) — sont nettement plus modérées. Les statistiques de la table 8.3 montrent que la principale amélioration à espérer dépend des sites d'appel monomorphes (64%). Les BTD_1 et BTD_2 ne représentent que 20% du nombre de sites d'appel monomorphes et à cause des erreurs de prédictions l'amélioration attendue est forcément non proportionnelle à ce nombre. De plus, les BTD_i avec $i > 2$ sont en nombre trop faibles pour permettre de conclure quant à l'amélioration qu'ils apportent sur la coloration. C'est la raison qui nous a poussé à ne présenter que les résultats pour les BTD_2 et BTD_∞.

Des conclusions similaires sont à tirer pour les analyses de type. Malgré sa simplicité, CHA donne de très bon résultats et une approximation plus précise du degré de polymorphisme des sites d'appel ne change pas fondamentalement les conclusions. En compilation globale, un compromis entre temps de compilation et précision de l'analyse de type doit être considéré — par exemple comme nous le proposons au travers d'une option du compilateur. CHA étant très rapide, elle représente une solution intéressante pendant le développement puisqu'elle réduit les temps de recompilation. Pour une version finale du

TABLE 8.5 : Temps d'exécution en fonction des techniques d'implémentations avec invariant de référence et des processeurs, d'après [Ducournau *et al.*, 2009]

identifiant			S-1			I-2			P-3		
temps de référence			123.2s			87.4s			62.3s		
technique	schéma		AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
BTD_∞	RTA	G	-22.6	-11.5	14.4	-10.9	-2.2	9.7	***	***	***
MC-BTD_2	RTA	G	-22.2	-10.9	14.6	-11.7	-3.8	10.6	-28.4	-13.0	21.5
BTD_∞	CHA	O	***	***	***	-2.9	1.4	4.5	***	***	***
MC-BTD_2	CHA	O	***	***	***	-5.4	-2.8	2.8	***	***	***
MC		S	0	9.8	9.8	0	5.7	5.7	0	18.2	18.2
IC		D	13.7	34.1	17.9	5.3	14.5	8.7	13.4	27.2	12.1
PH-and		D	13.4	35.4	19.5	2.5	14.5	11.6	8.1	24.9	15.6
PH-and+shift		D	14.7	38.0	20.3	10.6	25.5	13.6	13.4	35.5	18.9
PH-mod		D	81.1	226.0	80.0	28.6	104.3	58.8	49.1	146.1	65.1
PH-mod+CA_4		D	33.1	121.0	66.1	21.1	81.2	49.6	24.7	87.5	50.3

identifiant			I-4			I-5			A-6		
temps de référence			43.3s			34.8s			34.0s		
technique	schéma		AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
BTD_∞	RTA	G	-13.3	-1.0	14.1	-8.9	2.9	13.0	-12.5	2.9	17.6
MC-BTD_2	RTA	G	-13.7	-1.3	14.4	-10.2	-0.8	10.5	-12.5	1.1	15.5
BTD_∞	CHA	O	-3.0	4.9	8.2	2.7	19.4	16.2	5.8	27.8	20.8
MC-BTD_2	CHA	O	-5.9	2.3	8.7	-2.7	14.2	17.3	3.4	27.3	23.1
MC		S	0	7.9	7.9	0	11.1	11.1	0	15.8	15.8
IC		D	4.3	16.6	11.8	7.7	28.5	19.2	14.9	40.1	21.9
PH-and		D	4.2	19.0	14.2	6.2	31.4	23.8	15.2	52.4	32.2
PH-and+shift		D	6.9	28.7	20.4	10.3	45.3	31.8	19.5	64.4	37.6
PH-mod		D	55.2	172.0	75.2	24.4	106.3	65.8	80.1	235.4	86.2
PH-mod+CA_4		D	28.1	98.2	54.7	21.4	82.7	50.5	40.4	141.2	71.8

identifiant			A-7			I-8			I-9			I-8 sans GC		
temps de référence			32.8s			30.4s			18.5s			15.7s		
technique	schéma		AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
BTD_∞	RTA	G	-13.7	9.9	27.3	-9.4	7.5	18.6	-10.3	0.6	12.1	-19.2	12.7	39.5
MC-BTD_2	RTA	G	-13.1	10.7	27.3	-9.7	7.0	18.5	-9.7	0.6	11.5	-18.5	12.9	38.6
BTD_∞	CHA	O	4.0	23.3	18.6	1.1	12.1	10.8	1.8	15.0	12.9	10.2	31.0	18.9
MC-BTD_2	CHA	O	2.0	22.8	20.4	-1.8	10.1	12.2	-2.5	13.0	16.0	5.4	27.9	21.3
MC		S	0	15.2	15.2	0	11.3	11.3	0	10.3	10.3	0	21.6	21.6
IC		D	12.6	38.9	23.3	7.7	29.7	20.5	8.0	29.7	20.1	14.8	56.5	36.4
PH-and		D	16.8	57.3	34.7	5.1	30.0	23.7	6.2	30.1	22.6	9.6	55.9	42.2
PH-and+shift		D	18.8	61.5	36.0	7.8	43.6	33.2	8.3	43.3	32.3	14.1	81.9	59.4
PH-mod		D	73.4	221.5	85.4	19.5	108.7	74.7	17.6	85.7	57.9	35.2	209.1	128.6
PH-mod+CA_4		D	38.3	129.3	65.8	19.6	81.3	51.6	20.8	74.6	44.5	37.4	155.7	86.1

Chaque sous-table présente les résultats pour un processeur, avec son temps de référence. Tous les autres nombres sont exprimés en pourcentage. Chaque ligne décrit une technique d'invocation de méthode et un test de sous-typage. Pour toutes les techniques, les deux premières colonnes représentent le surcoût vis-à-vis de la coloration (MC-AC-S) respectivement avec la coloration d'attributs (AC) et la simulation des accesseurs (AS). La troisième colonne est le surcoût entre la simulation des accesseurs et la coloration d'attributs. La dernière sous-table présente les mesures du processeur I-8 sans ramasse-miettes. Sur P-3, les BTD_2 sont remplacés par les BTD_0. Les tests marqués *** n'ont pas pu être réalisés.

programme, RTA ou CFA peuvent être considérées comme un élément supplémentaire d'efficacité. Dans le schéma (O), la meilleure solution est certainement de n'utiliser que CHA puisqu'elle ne nécessite pas de modèle interne, ce qui simplifie l'architecture générale du compilateur (Figure 6.5 et 6.6).

Techniques d'implémentation

Nous considérons dans un premier temps les arbres binaires de sélection dans un cadre de compilation globale (G ou O). Toutes les autres techniques serons considérées en chargement dynamique (D).

Arbres binaires de sélection (BTD). Notre banc d'essai ne nous permet pas d'estimer avec précision le taux de mauvaises prédictions de branchement pour les arbres binaires de sélection et il est peu probable qu'aucun banc d'essai ne puisse jamais conclure avec précision sur le réglage précis de la valeur i optimale. Ce réglage est bien trop dépendant du flot de contrôle du programme et du processeur en question. Néanmoins, il est tout de même possible de tirer une conclusion depuis un évaluation abstraite. Les BTD_1 constituent très certainement une amélioration sur la coloration, puisqu'une mauvaise prédiction de branchement a un coût identique à celui du saut indirect de la coloration : on économise donc un `load`. Avec les BTD_2, il suffit d'une seule prédiction correcte pour se ramener au cas précédent. Le point critique devrait certainement se situer entre les BTD_3 et les BTD_4. Sur la moitié des processeurs, les BTD_∞ sont légèrement meilleurs que les BTD_2 en compilation globale (G), mais ce n'est jamais le cas en (O). Cette observation est cohérente avec nos attentes, puisque dans le cas de (G) les arbres binaires de sélection sont *mis en ligne* tandis qu'ils sont partagés dans des *thunks* en (O). Dans le premier cas, les prédictions de branchement sont propres à chaque site d'appel, elles sont donc plus susceptibles d'être correctes. En revanche, lorsque les sites d'appels sont partagés, les erreurs de prédiction ont tendance à augmenter. Cependant, avec les BTD_∞ les *thunks* réduisent énormément la taille du programme final (Table 8.6). Finalement, cette discussion doit être reconsidérée pour tous les processeurs sans prédiction de branchement (non étudiés ici), par exemple ceux des petits systèmes embarqués, où seuls les BTD_1 peuvent être éventuellement considérés.

Hachage parfait de classes (PH). Quand le hachage parfait de classes (`and`) est utilisé uniquement pour l'invocation de méthode, il rajoute un surcoût faible si ce n'est négligeable (de 3 à 8% sur les processeurs INTEL). Ceci confirme nos études a priori et peut être expliqué par le fait qu'il rajoute des accès mémoire mais à des zones mémoire déjà utilisées par la technique de référence — donc sans risque de défaut de cache supplémentaire — ainsi que quelques instructions à 1 cycle. Ces instructions supplémentaires représentent le réel surcoût de la technique qui reste faible comparé au coût total de l'invocation de méthode (passage de paramètres).

La ou les instructions rajoutées par PH-`and+shift` entraînent un surcoût supplémentaire par rapport à PH-`and` (supérieur à 2% pour quasiment tous les processeurs). Ce résultat est un peu surprenant car la plupart du calcul des séquences supplémentaires peut être fait en parallèle. De plus, l'augmentation de la taille des séquences de code de PH-`and+shift` ne contre-balance pas l'économie réalisée sur la taille des tables de méthodes [Ducournau et Morandat, 2011]. Si l'on y rajoute le surcoût temporel (les plus de 2%), cette fonction de hachage peut être définitivement éliminée des alternatives à considérer.

Sur tous les processeurs, la division entière utilisée par PH-`mod` entraîne un surcoût exorbitant, même par rapport à nos prévisions initiales qui étaient déjà élevées. Cette expérience confirme donc que PH-`mod` doit être exclusivement réservé à des processeurs avec une division entière efficace — la plupart des processeurs testés utilisent l'unité de calcul flottant pour réaliser cette division.

En conclusion, PH-`and` surpasse toutes nos espérances quand il est utilisé uniquement pour le test de sous-typage et l'invocation de méthode. Il est plus rapide que toutes les autres alternatives testées dans le schéma (D) et d'un point de vue spatial, son surcoût est faible (Table 8.6). C'est donc un très bon candidat à l'implémentation des interfaces de JAVA ou C#. Cependant, quand il est utilisé pour l'accès aux attributs, le surcoût est nettement moins raisonnable.

Coloration incrémentale (IC). L'évaluation abstraite de la coloration incrémentale est comparable à celle de PH-`and`, ce qui se vérifie en pratique. Elle est même légèrement plus efficace que PH-`and` sur les processeurs anciens mais elle tend à l'être moins sur les processeurs modernes. Cette similarité s'explique aisément car la zone mémoire utilisée pour stocker les couleurs peut entraîner un défaut de cache supplémentaire, qui a une plus grande influence sur les processeurs récents qui sont plus rapides. Quand elle est utilisée aussi pour l'accès aux attributs, la différence entre la coloration incrémentale et PH-`and` est généralement en dessous du seuil de nos mesures. D'un point de vue spatial, la coloration incrémentale est sans doute un peu meilleure que PH-`and` mais cette différence n'est pas assez marquée pour compenser le coût du recalcul des couleurs en cas de chargement de nouvelles classes et les allocations dynamiques qui y sont associées.

Simulation des accesseurs (AS). Quelle que soit la technique d'implémentation considérée, la simulation des accesseurs rajoute un surcoût non négligeable à la coloration des attributs. Ce coût peut être considéré comme le surcoût de l'utilisation de l'héritage multiple sur le sous-typage multiple. Au vu du grand nombre d'accès aux attributs fait par PRMC (Table 8.3), ce surcoût était attendu puisque la simulation des accesseurs remplace le `load` unique de la coloration d'attributs par une séquence de code équivalente à une invocation de méthode — au branchement près. De plus, la simulation des accesseurs rajoute au moins une indirection par la table des méthodes, ce qui peut rajouter un défaut de cache

supplémentaire et l'unique `load` de coloration d'attribut est plus facilement parallélisable que la séquence de code associée à la simulation des accesseurs.

Pour toutes les techniques utilisées dans le schéma de compilation séparée avec chargement dynamique (D), la simulation des accesseurs provoque un surcoût apparemment non additif. Pour une technique, par exemple PH-`and`, la différence PH-`and`-AS et MC-AC est bien plus grande que la somme des différences — PH-`and`-AC, MC-AC et MC-AC et MC-AS. Ceci s'explique car la première différence ne concerne que l'invocation de méthode tandis que la seconde concerne les deux mécanismes. L'extrapolation doit donc être faite en la multipliant, d'après les statistiques de la table 8.3, par le rapport $\frac{2560+1707}{1707} \approx 2.5$. Le rapport PH-`and`-AS/MC-AC doit être comparé à 2.5 × PH-`and`-AC/MC-AC+MC-AS/MC-AC. Cette observation justifie les surcoûts de la simulation des accesseurs couplée à IC, PH-`and` et même PH-`mod`, malgré le surcoût dramatique de ce dernier.

Enfin, ces conclusions sur la simulation des accesseurs ne doivent pas être considérées comme définitives car le programme utilise de nombreux « vrais » accesseurs. Ces accesseurs ajoutent donc un appel de méthode à la simulation des accesseurs, et des optimisations sont envisageables.

Caches dans les tables de méthodes (CA). Les statistiques sur le taux de réussite du cache (Table 8.4), montrent qu'il est grandement variable en fonction de la configuration de caches testée et les résultats temporels le confirment. Ces différentes configurations ont été testées avec PH-`and` et PH-`mod`, cependant quelle que soit la configuration retenue, elle dégrade PH-`and` sur tous les processeurs. De plus, seuls les caches séparés et multiples améliorent PH-`mod`, c'est pourquoi nous ne présentons que les résultats de PH-`mod`+CA_4. Avec la simulation des accesseurs, les caches améliorent systématiquement PH-`mod`. Enfin, avec la coloration des attributs, ils n'améliorent que les machines où la division entière est particulièrement coûteuse — il s'agit de tous les processeurs n'utilisant pas la SRT Radix-16 — et ce au prix d'une grosse augmentation de la taille du code et des tables des méthodes.

Finalement, le cache ne doit être considéré comme une solution que si la technique d'implémentation sous-jacente est particulièrement inefficace ou si le nombre d'entités à *cacher* est très faible, ce qui entraîne un taux de réussite du cache disproportionné comme dans le cas du test de sous-typage pour les interfaces de JAVA où ce taux était voisin de 100% dans SPECJVM 98 [Click et Rose, 2002].

Taille des exécutables

La taille des exécutables a été étudiée de manière similaire au temps, la table 8.6 montre ces résultats. Ils sont à prendre avec prudence car le compilateur n'est pas particulièrement optimisé pour l'occupation mémoire. Il est cependant possible d'en tirer certaines conclusions.

Les schémas (G) et (O) réduisent fortement la taille de l'exécutable. Bien que la compilation globale profite de l'élimination du code mort, l'architecture modulaire du compila-

TABLE 8.6 : Taille des exécutables strippés sur le processeur I-5

			taille de la ref. 914 Ko		
technique	schéma		AC	AS	AS/AC
BTD_∞	RTA	G	22.0	30.9	7.3
MC-BTD_2	RTA	G	-26.9	-17.0	13.5
BTD_∞	CHA	O	-5.1	-0.1	5.2
MC-BTD_2	CHA	O	-14.9	-10.4	5.3
MC		S	0	11.7	11.7
IC		D	19.0	33.3	11.9
PH-`and`		D	24.5	45.5	16.9
PH-`and+shift`		D	37.0	61.6	18.0
PH-`mod`		D	25.9	41.6	12.5
PH-`mod`+CA_4		D	75.6	104.7	16.7

Tailles en Ko et en pourcentage. Les conventions sont les mêmes que pour les mesures de temps.

teur en limite l'effet. Les BTD_∞ augmentent considérablement la taille du code quand ils sont spécifiques à chaque site dans le schéma (G). Cependant ils ne peuvent être vraiment efficaces que s'ils sont dédiés à un site d'appel. La taille du programme constitue donc un argument supplémentaire à l'utilisation combinée des arbres binaires de sélection et de la coloration.

La coloration incrémentale permet d'obtenir des exécutables légèrement plus petits que le hachage parfait de classes mais elle impose des allocations dynamiques que notre expérimentation ne prend pas en compte. De plus, le hachage parfait de classes n'a pas été implémenté de manière optimale en espace. La numérotation parfaite de classes donnerait, en effet, de bien meilleurs résultats [Ducournau et Morandat, 2011]. Malgré cette limitation évidente, plusieurs conclusions se dégagent sur l'utilisation du hachage parfait de classes. Tout d'abord, PH-`and+shift` augmente considérablement la taille du code et d'après les statistiques de Ducournau et Morandat [2011], le gain à espérer sur la taille des tables est négligeable. Comme il est moins efficace en temps et n'améliore pas l'espace, il peut être définitivement éliminé des alternatives à considérer. Ensuite, la différence entre PH-`mod` et PH-`and` n'est pas vraiment marqué et le compromis espace/temps de PH-`mod` sur PH-`and` précédemment considéré par [Ducournau, 2008] n'est pas flagrant — d'autant qu'avec les processeurs testés le temps de PH-`mod` est prohibitif. Cependant, cette différence avait été considérée sur des hiérarchies de classes nettement plus grandes que celle de PRMC (voir Tables 5.3, 5.4).

Enfin, les caches dans les tables de méthodes, lorsqu'ils sont mis en ligne, ne sont définitivement pas compétitifs puisqu'ils augmentent excessivement la taille de l'exécutable pour une amélioration temporelle minime — voire une dégradation dans certains cas. Ils doivent donc être réservés à des techniques non mises en ligne et couplés à des méca-

TABLE 8.7 : Comparaison du temps d'exécution en fonction de la technique d'implémentation et des processeurs, d'après [Ducournau et Morandat, 2011]

cluster	1		2		3	
technique	AC	AS	AC	AS	AC	AS
compile-time MC+BTD$_0$	-1.0	-0.3	-1.0	0.5	-1.0	0.2
link-time MC	0.0	0.5	0.0	1.2	0.0	1.1
IC	0.5	1.2	1.1	3.1	0.7	2.9
PH-`and`	0.4	1.3	1.3	4.3	0.5	2.9
PH-`and+shift`	0.6	1.8	1.5	4.9	0.8	4.2
PH-`mod`	3.0	9.2	6.0	17.9	1.9	9.3
unité	19.0%		12.8%		10.7%	
nb. processeurs	4		2		4	

Chaque sous-table présente les résultats pour un *cluster* de processeurs et chaque nombre correspond à la moyenne des mesures de tous les processeurs du *cluster*. Chaque ligne décrit une technique d'implémentation pour l'invocation de méthode et le test de sous-typage. Les deux colonnes représentent le surcoût par rapport à la coloration pure (link-time MC-AC), respectivement avec la coloration d'attributs (AC) et la simulation des accesseurs (AS).

nismes très lents — par exemple avec des interpréteurs de *bytecode*. Leur utilisation au travers de *thunks* partagés pourrait constituer une alternative.

Influence des processeurs

Les différents processeurs testés influent sur les résultats de cette expérience, bien qu'ils n'en changent pas les conclusions. La plupart d'entre eux ont un comportements similaire, il y a cependant quelques exceptions. Par exemple, A-7 est le seul processeur où la coloration incrémentale est bien meilleure que PH-`and`. Sur tous les processeurs qui ne sont pas des INTEL, les surcoûts sont globalement doublés par rapport aux INTEL.

PH-`mod` est bien plus efficace sur les INTEL récents — I-8 et I-9— ce qui est partiellement expliqué par la division SRT Radix-16 [3]. Sur S-1, I-5, A-6 et A-7 l'efficacité de PH-`mod` est ridicule.

Nous avons classé les processeurs dans les résultats de la table 8.5, de gauche à droite et de haut en bas, en fonction du temps de référence. On retrouve une certaine corrélation entre cet ordre et leur date de sortie.

Dans [Ducournau et Morandat, 2011], nous avons utilisé un algorithme de *clustering* pour présenter ces mêmes résultats (Table 8.7). Dans cette représentation, l'unité de référence correspond à la différence normalisée entre MC-AC-S et MC-BTD$_0$-G. En quelque sorte, elle représente le coût de l'hypothèse du monde ouvert pour le sous-typage multiple par rapport à l'hypothèse du monde clos. Le premier *cluster* est composé de S-1, I-2, P-3

3. Bien que d'après les listes de processeurs trouvés sur internet seul I-9 aurait dû en profiter.

et I-4, le second des processeurs AMD A-6 et A-7. Enfin, le dernier *cluster* est composé des processeurs INTEL récents I-5, I-8, I-9 et I-10. Les *clusters* obtenus mettent effectivement en emphase des corrélations entre les dates de sortie et les comportement des processeurs, les anciens et les nouveaux processeurs INTEL se trouvant séparés dans des *clusters* différents, tandis que les processeurs AMD se retrouvant isolés dans leur propre *cluster*.

Malgré ces comportement visibles, l'échantillon de processeurs testés est trop réduit pour que l'on puisse tirer une conclusion de ces chiffres sur l'influence d'une famille de processeurs ou sur un constructeur particulier.

8.5.3 Comparaisons des techniques avec et sans invariant de référence

Dans cette seconde expérience [Morandat et Ducournau, 2010] l'objectif est concentré sur la comparaison d'une technique avec invariant de référence — le hachage parfait de classes (`and`) couplé à la simulation des accesseurs — à une technique sans cet invariant — les sous-objets — dans le cadre du chargement dynamique (D) et de l'héritage multiple. C'est pour cela que le hachage parfait de classes n'est considéré que couplé à la simulation des accesseurs. L'implémentation par sous-objets utilise PH-`and` pour le test de sous-typage et les *downcasts* (voir Section 5.3.4).

Cette seconde expérience utilise la même plate-forme de test, bien que les statistiques sur le programme de test soient légèrement différentes. La table 8.8 présente les statistiques pour cette version, les colonnes sont les mêmes que dans la table 8.3 et la ligne *ajustements de pointeurs* a été rajoutée. Les ajustements de pointeurs ne concernent que l'implémentation par sous-objets. Pour les attributs, ils sont maintenant séparés entre ceux dont le type statique du receveur est identique au type d'introduction (rst=aic) — donc ne demandant pas d'ajustement de pointeur dans l'implémentation par sous-objets — et les autres.

Comme expliqué dans la section 8.5.1, les sous-objets et le hachage parfait de classes ne consomment pas la même quantité de mémoire à l'exécution, ce qui influe directement sur le temps consommé par le ramasse-miettes qui sera déclenché plus souvent dans le premier cas. De plus, dans l'implémentation par sous-objets, il peut y avoir plusieurs points d'entrée sur une instance, ce qui complique encore la tâche du ramasse-miettes.

Comme dans l'expérience précédente (Section 8.5.2), le chargement dynamique couplé à l'héritage multiple ajoute un surcoût non négligeable. Il n'y a rien de surprenant à cela, car les séquences de code pour chaque mécanisme sont plus longues et les structures plus complexes.

Du point de vue spatial, l'implémentation par sous-objets est désastreuse aussi bien statiquement que dynamiquement. La quantité de mémoire utilisée par cette implémentation est plus que doublée. Conformément à nos attentes et aux évaluations abstraites, la taille des tables dans cette implémentation augmente sérieusement la taille de l'exécutable. Néanmoins les ajustements de pointeurs ne sont pas en reste et représentent 12% de

TABLE 8.8 : Statistiques sur le programme de test *P* pour la comparaison avec et sans invariant de référence, d'après [Morandat et Ducournau, 2010]

nombres de		statique	dynamique
classes	introductions	532	—
	instanciations	6649	35 M
	tests de sous-typage	194	484 M
méthodes	introductions	2598	—
	définitions	4445	—
	appels	15672	1712 M
attributs	introductions	614	—
	accès	4437	2547 M
	`rst` ≠ `aic`	*1442*	*653 M*
ajustements	*upcasts*	*9439*	*785 M*
de pointeurs	*downcasts*	*3254*	*1393 M*

La légende de cette table est identique à celle de la table 8.3. Les lignes en italique ne concernent que l'implémentation par sous-objets. Les accès aux attributs sont comptés séparément quand le type statique (rst) du receveur est différent de la classe d'introduction (aic). Les *upcasts* correspondent aux affectations ou au passage de paramètres polymorphes, quand le type de la source est un sous-type strict de la cible. Enfin, les *downcasts* représentent les types de retour polymorphes.

TABLE 8.9 : Résultats temporels et spatiaux de PH-`and` par rapport aux sous-objets, d'après [Morandat et Ducournau, 2010]

	M-AC-S	PH-`and`-AS	SO
I-9 avec ramasse-miettes	17.3s	29%	46%
sans ramasse-miettes	10.0s	46%	39%
I-10 avec ramasse-miettes	17.9s	30%	45%
sans ramasse-miettes	10.0s	48%	40%
I-4 avec ramasse-miettes	43.7s	19%	32%
sans ramasse-miettes	25.4s	40%	43%
Executable Size	952.3K	45%	149%
Dynamic memory usage	699M	0%	119%

La première colonne représente les temps d'exécution de la coloration pure (attributs et méthodes) en compilation séparée (S). Les colonnes suivantes représentent respectivement le surcoût en pourcentage du hachage parfait de classes avec la fonction `and` et la simulation des accesseurs (PH-`and`-AS) et des sous-objets (SO). Enfin, les deux dernières lignes présentent la taille de l'exécutable (strippé) et la taille de la mémoire allouée durant l'exécution.

la taille totale de l'exécutable. Du point de vue du passage à l'échelle, ce premier constat donne encore l'avantage au hachage parfait de classes sur les sous-objets.

Si l'on considère le temps, les résultats sont moins tranchés. Sans ramasse-miettes, les sous-objets sont plus rapides que PH-and mais lorsqu'on l'utilise, le surcoût monstrueux des sous-objets, inverse la tendance. Dans tout les cas, l'utilisation de l'héritage multiple sans restriction rajoute un surcoût non négligeable (entre 30 et 40% sur les machines récentes). Avec un ramasse-miettes, l'espace supplémentaire requis par les sous-objets entraîne que le hachage parfait de classes est autour de 15% plus efficace — 15% représente une très grande différence — tandis que sans ramasse-miettes, cette différence n'est que de 7% dans l'autre sens. Cette différence est significative mais pas démesurée et comparée au surcoût mémoire, le hachage parfait de classes doit être préféré aux sous-objets. Cette conclusion est renforcée par la simplicité de l'implémentation du hachage parfait de classes puisqu'elle maintient l'invariant de référence et ne nécessite donc pas d'ajustement de pointeurs

Limite de la comparaison

Ces résultats ne permettent pourtant pas de conclure définitivement sur l'implémentation par sous-objet car plusieurs optimisations n'ont pas été intégrées au compilateur par manque de temps.

Optimisation des sous-objets vides. L'implémentation par sous-objet utilisée dans cette expérience n'utilise pas l'optimisation des sous-objets vides (voir Section 5.1.4). Les sous-objets vides touchent un grand nombre de classes et permettraient de réduire le surcoût en mémoire — statique et dynamique — de cette implémentation. Cependant les modules et le raffinement de classe utilisés par PRM ne permettent pas d'éliminer les ajustements de pointeurs, et seuls les gains en mémoire dynamique seraient effectifs.

Utilisation de *thunks*. L'implémentation par sous-objets peut être réalisée en utilisant des *thunks* pour faire ces *décalages* (voir Section 5.1.2). L'utilisation de *thunks* permet, lors de l'invocation de méthodes, de ne pas faire les *décalages* lorsqu'ils sont nuls. L'effet de cette optimisation est encore accru lorsqu'elle est couplée à celle des sous-objets vides. Mais au prix d'une augmentation de l'espace statique utilisé par le programme.

Implémentation hétérogène des génériques. Les *templates* de C++— et leur implémentation hétérogène — sont une des caractéristiques primordiales de l'implémentation par sous-objets (voir Section 5.10.3). Elles contre-balancent le surcoût temporel de cette implémentation — principalement lorsque les *templates* sont instanciées par des types primitifs — au prix d'une duplication du code. Actuellement, la plate-forme PRM ne permet cependant pas d'utiliser l'implémentation hétérogène des génériques.

En fin de compte, cette expérience ne prétend pas conclure sur l'efficacité du langage C++ mais seulement d'en évaluer la technique d'implémentation par sous-objets. Surtout qu'en C++ l'héritage virtuel est sous-utilisé — en pratique la plupart des programmes se contentent d'un héritage arborescent — et les langages autorisent la définition de méthodes non virtuelles qui ne sont pas régies par le polymorphisme. Toutes ces optimisations et restrictions sur l'usage du polymorphisme rendent C++ très efficace mais ce au détriment de la réutilisabilité.

8.5.4 Comparaison avec les tests effectués sur des programmes artificiels

Lors de sa thèse Privat [2006] avait proposé une évaluation du schéma de compilation séparée avec édition de liens optimisée (O) couplée aux BTD_2. Cette comparaison a été réalisée par un *benchmark* composé de programmes synthétiques, générés dans trois langages différents supportant l'héritage multiple — PRM, EIFFEL et C++. A l'inverse de nos comparaisons « toutes choses égales par ailleurs », c'est donc une comparaison hétérogène, réalisée avec trois compilateurs différents respectivement PRMC (version RUBY), SMART EIFFEL (compilation globale et BTD_∞) et G++ (compilation séparée et sous-objets).

Pour que la comparaison soit la plus objective possible, le *benchmark* n'utilise aucune spécificité des langages sous-jacents.

Chaque jeu de test est une hiérarchie de classes composée :

- d'une racine (R).
- de 15 sous-classes directes de la racine, fournissant chacune un attribut (A_X).
- d'une sous-classe commune aux fournisseurs d'attributs et servant de type statique manipulé (S).
- d'un ensemble de classes cousines à S et héritant d'une des classes A_X (P_X).
- d'un arbre binaire de sous-classes de S et de certains P_X — environ 20% de chance d'hériter d'un A_X au travers d'un P_X (D_X). Le nombre de classes dans l'arbre est dépendant du paramétrage du script et permet d'augmenter le degré de polymorphisme des sites d'appels.

Chacune des classes de cette hiérarchie (re)définit des méthodes et introduit un nouvel attribut.

Le programme associé à cette hiérarchie initialise un tableau [4] typé par S de n éléments pris au hasard parmi les classes D_X. Le tirage au sort a lieu à la génération, de telle sorte que ce soient exactement les mêmes tableaux pour les trois langages. Enfin, une double boucle effectue une série d'actions sur les éléments de ce programme. En fonction du test effectué, l'action est soit un test de sous-typage, soit une invocation de méthode, soit un accès à un attribut.

4. Tous ces langages autorisent un accès primitif aux tableaux et les itérations sont faites manuellement pour éviter le biais qui pourrait être introduit par des bibliothèques de plus haut niveau (STL, ...).

Dans l'ensemble, les conclusions obtenues sont similaires à celles proposées dans nos expériences. Néanmoins, il y a deux grandes différences.

Tout d'abord le test de sous-typage utilisé par G++ le met complètement hors-jeu. Certes G++ implémente son test par un appel de fonction mais ceci n'explique pas tout. Ce surcoût ne se retrouve pas dans nos expérimentations puisqu'il est implémenté par le hachage parfait de classes, ce qui le rend comparable — voire compétitif. Le hachage parfait de classes représente donc une implémentation intéressante pour le test de sous-typage dans une implémentation par sous-objets, donc pour C++.

La seconde différence notable est liée au surcoût des *thunks* pour PRM, car le surcoût annoncé par cette expérience était nettement moins important que celui que nous avons pu constater. Il peut y avoir plusieurs explications à cela. Tout d'abord, dans cette expérimentation tous les types de paramètres sont invariants, ce qui n'est pas le cas dans le compilateur PRM, puisqu'il a une politique covariante. Le travail à faire dans nos *thunks* est donc plus important que dans l'expérience sus-mentionnée et certains *thunks* ne sont pas terminaux (à cause des boites). De plus, dans notre expérience, seule l'implémentation du schéma (O) utilise des *thunks*, leur nombre pourrait être sur-évalué et nous ne sommes pas complètement à l'abri d'une erreur dans leur implémentation.

8.6 Conclusions sur ces expérimentations

Plusieurs conclusions peuvent être dégagées de ces deux expérimentations. La première est d'ordre méthodologique. Les évaluations abstraites et les simulations permettent de se faire une idée assez précise de l'efficacité d'une technique d'implémentation sans la tester. Le modèle de processeur que nous utilisons (P95 de Driesen [2001]) a été choisi de façon un peu arbitraire et un modèle différent aurait pu être utilisé. Cependant, l'utilisation d'un autre modèle ne changerait pas radicalement les conclusions. Malgré toutes les caractéristiques des processeurs modernes — parallélisme, prédiction de branchement, etc. — l'évaluation abstraite donne des résultats assez fiables et permet à moindre coût de prévoir l'efficacité des techniques d'implémentation sans avoir à les implémenter. Il en va de même pour l'efficacité spatiale, qui se simule facilement sur des schémas très simplifiés de hiérarchies de classes, ce qui permet de tester leur passage à l'échelle ce qui est impossible avec des programmes réels [Ducournau, 2008, 2011; Ducournau et Morandat, 2011] et pour paraphraser une des conclusions de [Ducournau et Morandat, 2010] : « Les compilateurs sont des bombes atomiques, il vaut mieux les simuler avant de les tester ».

La seconde conclusion concerne le hachage parfait de classes. Ainsi, malgré l'implémentation rudimentaire utilisée lors des tests — hachage parfait de classes au lieu de numérotation parfaite de classes — les résultats spatiaux sont comparables aux autres techniques et leur efficacité temporelle est très bonne, particulièrement lorsque le hachage parfait de classes est couplé à la coloration. PH-`and` représente donc une implémentation très intéressante dans le cadre du sous-typage multiple pour le test de sous-typage et de

l'invocation de méthode — donc pour JAVA ou C#. Sur les processeurs classiques, le coût prohibitif de la division entière, même avec la SRT Radix-16, met hors-jeu PH-`mod`. Cette alternative peut être mise de côté tant que les processeurs ne proposeront pas une division entière véritablement efficace.

Les arbres binaires de sélection sont réellement efficaces sur les processeurs modernes. Leur coût en espace peut cependant être élevé ce qui justifie pleinement leur utilisation bornée conjointe à la coloration.

Les techniques d'implémentation reposant sur des caches dans les tables de méthodes ne doivent définitivement plus être considérées, en tout cas pour dans les langages en typage statique et des programmes compilés. On leur préférera systématiquement une technique en temps constant.

Enfin, ces tests ne permettent pas de dégager une technique d'implémentation efficace compatible avec l'héritage multiple et le chargement dynamique dans le cadre que nous avons étudié, c'est-à-dire sans recompilation. Les seules techniques envisageables dans ce cadre sont l'implémentation par sous-objets et le hachage parfait de classes couplé à la simulation des accesseurs.

L'implémentation par sous-objets entraîne un réel surcoût en terme de mémoire tant statique que dynamique. De plus, elle impose un travail supplémentaire non négligeable au compilateur. S'imposer, dans ce langage, un style de programmation purement objet en héritage multiple conduit à une perte d'efficacité notable. Comme l'héritage virtuel est sous-utilisé en C++, ce surcoût est moindre en pratique. En comparant le surcoût prohibitif du test de sous-typage de G++ (voir Section 8.5.4) à celui obtenu dans la section 8.5.3, le hachage parfait de classes représente une implémentation particulièrement intéressante pour le test de sous-typage dans une implémentation par sous-objets. Mais il faudrait l'adapter à la présence d'héritage non virtuel.

Le hachage parfait de classes avec la simulation des accesseurs permet une implémentation aussi efficace que les sous-objets avec un surcoût mémoire dynamique nul et un surcoût statique nettement inférieur. Cette implémentation est aussi beaucoup plus simple à mettre en œuvre puisqu'elle maintiens l'invariant de référence.

En tout état de cause, le hachage parfait de classes ouvre des perspectives intéressantes pour l'héritage multiple et le chargement dynamique dans un cadre de compilation adaptative. En effet, la majorité des attributs ont une position dans l'objet invariante par spécialisation. En utilisant un protocole de double compilation inspiré de celui de Myers [1995], il serait possible d'utiliser un code efficace (celui de l'implémentation du sous-typage simple) quand la position est invariante et d'utiliser le hachage parfait de classes dans les autres cas. Cette alternative a été proposée en même temps qu'un protocole de recompilation dans [Ducournau et Morandat, 2010].

Les schémas de compilation ont un réel impact sur l'efficacité du code généré et il serait intéressant quel que soit le langage, de disposer d'un compilateur à géométrie variable qui offrirait différents mode de compilation, du plus séparé, pour le développement, au plus global, pour la distribution. Quel que soit le langage considéré, le choix du schéma doit être

laissé au programmeur afin de lui permettre de proposer des programmes efficaces dans son contexte d'utilisation. La spécification d'un langage ne devrait donc jamais imposer de schéma de compilation associé, tout du moins si le langage se prétend universel — le chargement dynamique ne devant être qu'une fonctionnalité supplémentaire proposée au programmeur.

CHAPITRE

9 Conclusion

L'objectif de cette thèse était de comparer de façon empirique l'efficacité de diverses techniques d'implémentation et schémas de compilation des langages objet. Cet objectif a été atteint et les résultats ont été décrits et discutés dans le chapitre 8.

Pour l'atteindre, nous avons dû développer un compilateur et un banc de tests. C'est bien sûr le compilateur qui a occupé la plus grande partie de ce travail. L'intégration a posteriori des sous-objets a, en particulier, soulevé des difficultés extraordinaires et imprévues. Rétrospectivement, il aurait mieux valu commencer par cette technique pour éviter que le compilateur se construise autour de l'invariance des références.

Le banc de test a soulevé lui aussi quelques difficultés. Si l'idée de méta-compilation est apparue immédiatement, les questions de reproductibilité et de comparabilité nous ont durablement occupés avant que nous ne trouvions une solution satisfaisante. Curieusement, nous n'avons pas trouvé trace, dans nos lectures, de semblables préoccupations pour les mesures de performance de programmes, et bien sûr de compilateurs. Une contribution imprévue de ce travail est donc cette analyse de la reproductibilité et de la comparabilité des expériences.

PRMC est le premier compilateur permettant de générer des programmes qui utilisent certaines techniques :

- la coloration à l'édition de liens [1] ;
- le test de Cohen optimisé (sans test de bornes, ni tables de taille fixe) [2] ;
- les arbres binaires de sélection bornés, complétés par la coloration [2] ;
- le hachage parfait de classes utilisé pour le test de sous-typage seul, associé à l'envoi de message, ou comme implémentation de base pour les trois mécanismes.

1. La première version de PRMC écrite en RUBY par Privat ne la proposait que dans le schéma avec édition de liens optimisée, donc au travers de *thunks*.
2. Utilisé aussi dans PRMC RUBY.

La coloration est une technique ancienne (elle a plus de 20 ans) mais, étonnamment, elle n'avait jamais été testée dans le cadre pour lequel elle était proposée par Pugh et Weddell [1990] — compilation séparée et édition de liens globale — ni même dans un vrai compilateur. Le hachage parfait de classes de son côté est une technique très récente mais qui n'a non plus jamais été testée.

Avant PRM, elle n'avait été testée que dans le cadre d'études algorithmiques et, de façon très marginale, en YAFOOL. PRMC a permis de tester en vraie grandeur ces techniques qui pour le moment n'avaient été évaluées que de façon abstraite. Nos évaluations empiriques ont montré la viabilité de ces techniques en pratique, alors que leur passage à l'échelle avait été démontré antérieurement avec des *benchmarks* de très grande taille [Ducournau, 2008, 2010; Ducournau et Morandat, 2011]. Ces expérimentations ont aussi permis de confirmer nos attentes a priori pour ces techniques, aussi bien en temps qu'en espace.

Enfin, cette thèse présente des conclusions plus terre-à-terre et immédiates pour les concepteurs de langages ou les implémenteurs de système d'exécution — compilateur, machines virtuelles, ...

Nous commençons par faire le point sur l'héritage multiple. En toute généralité, l'héritage multiple ne peut pas être ignoré même dans les systèmes où il est relégué à l'arrière plan (JAVA par exemple). Son besoin est réel — même limité à la forme dégradée du soustypage multiple. Néanmoins l'héritage multiple a un coût qui peut être annulé par le choix d'un schéma de compilation adapté. Toutefois ce coût est faible si on se limite au soustypage multiple (légèrement supérieur à 5% sur les machines récentes). Sans restriction, ce coût est nettement moins acceptable (plus de 30% sur les machines récentes). Il reste donc encore de la place pour apporter des améliorations aux techniques existantes ou en concevoir de nouvelles.

Comme nous l'avons montré dans cette thèse, il existe un grand nombre de schémas de compilation avec une efficacité et une modularité variant en sens inverse. Les résultats montrent qu'évidemment le schéma de compilation globale est le plus efficace mais c'est au détriment de la modularité prônée par le génie logiciel. Finalement, ce schéma semble particulièrement adapté aux systèmes embarqués et aux versions finales des programmes. À l'autre extrême, il y a la compilation séparée avec chargement dynamique qui met le plus à l'épreuve l'efficacité des techniques d'implémentation. La compilation séparée avec édition de liens globale représente un moyen terme. Elle répond bien aux critères du génie logiciel et permet une implémentation efficace, avec la coloration, qui annule même le coût temporel de l'héritage multiple — au prix de quelques trous dans les instances. Cependant ce schéma pourrait être amélioré par un schéma hybride (non testé) qui conserverait le caractère modulaire tout en permettent d'améliorer l'efficacité du code produit. Cette alternative serait idéale pendant le développement, puisqu'elle conserve le temps de recompilation rapide et permet des exécutions efficaces. Comme il est impossible de désigner un schéma unique et idéal, il nous semble que les langages devraient être spécifiés indépendamment de leur schéma de compilation — comme ils devraient l'être vis-à-vis des techniques d'implémentation. Ceci permettrait de concevoir et développer des com-

pilateurs à géométrie variable pour un langage. Les utilisateurs du langage n'auraient alors qu'à choisir le schéma de compilation le plus adapté à leurs besoins et pourraient en changer si nécessaire (par exemple, pour une version finale). Le chargement dynamique vient ternir cette conclusion car il a un coût non négligeable — du moins dans le cadre que nous avons étudié de l'héritage multiple, même s'il n'est pas utilisé. Cette fonctionnalité ne devrait pas, être imposée par la spécification d'un langage et son utilisation devrait correspondre à un choix fait par les développeurs — modularité contre efficacité. Elle pourrait n'être utilisée que pour des systèmes interagissant avec l'utilisateur final — au travers des *plugins* — ou pour les systèmes demandant une forte disponibilité (*high availability*).

Maintenant, nous rappelons les principales conclusions pour les techniques d'implémentation. La coloration, bien que proposée depuis longtemps, est une technique d'avenir. Elle permet tout d'abord d'annuler le coût de l'héritage multiple et ce en compilation séparée — mais avec édition de liens globale. Son surcoût mémoire est limité, notamment par les heuristiques proposées dans [Ducournau, 2010], surcoût qui peut encore être réduit à l'aide d'un peu de profilage. Outre son utilisation en compilation séparée, cette technique, intrinsèquement globale, a toute sa place en compilation globale. Elle réduit notamment la taille des exécutables par rapport aux arbres binaires de sélection non bornés et lorsqu'elle est utilisée avec des arbres binaires de sélection bornés, elle réduit d'une part la taille des exécutables et d'autre part en améliore aussi l'efficacité. Elle semble aussi tout indiquée pour les processeurs sans prédiction de branchement, donc pour les systèmes embarqués [Sallenave et Ducournau, 2010].

Enfin, le hachage parfait de classes semble être une technique très prometteuse selon plusieurs points de vue. Premièrement, c'est la seule technique, à notre connaissance, qui soit efficace en temps comme en espace pour implémenter le test de sous-typage et qui passe gracieusement à l'échelle — espace linéaire dans la relation de spécialisation — dans le cadre de l'héritage multiple et du chargement dynamique. Elle devrait donc être utilisée, avec les sous-objets de C++, pour le test de sous-typage comme pour les *downcasts* car actuellement il n'existe aucune alternative efficace. Dans le cadre de l'héritage multiple, son surcoût est faible par rapport à l'implémentation du sous-typage simple. Par une extrapolation raisonnable au sous-typage multiple, cette implémentation est parfaitement adaptée à l'implémentation des interfaces. Elle devrait donc être sérieusement envisagée pour les implémentations de machines virtuelles, tout au moins lorsque l'espace n'est pas une priorité absolue. Dans le cas contraire, par exemple, sur de tout petits systèmes embarqués, l'utilisation d'une technique en temps non constant peut être envisagée. Toutefois nous utilisons la dénomination hachage parfait de classes qui représente la technique d'implémentation, les tables générées sont bien plus compactes avec la numérotation parfaite de classes. Elle doit donc être préféré à la place du hachage parfait de classes pour le calcul des tables.

Cette thèse permet aussi de dégager une conclusion plus générale d'ordre méthodologique. Les évaluations abstraites et les simulations permettent de se faire une idée assez précise de l'efficacité d'une technique d'implémentation sans pour autant devoir la tester

dans un vrai compilateur. Compte tenu du coût de développement d'un compilateur, ou plus simplement, d'adaptation d'une nouvelle technique dans un contexte existant, une évaluation a priori est un préalable nécessaire. De plus, le passage à l'échelle ne peut être vérifiée que par des simulations sur des *benchmarks* de taille supérieure aux programmes réels.

Pour conclure sur une touche plus personnelle, cette thèse m'aura permis d'entrevoir la *vraie* philosophie des objets, philosophie plus ou moins bien comprise par les concepteurs de langages et généralement très mal expliquée aux étudiants qui débutent.

Perspectives

Les perspectives à ce travail sont multiples, tant sur le langage PRM et son compilateur PRMC, que pour les autres langages (principalement JAVA, .NET, C++, SCALA, ...).

PRM. Le langage PRM, bien qu'utilisable, est encore incomplet. L'amélioration des spécifications de PRM n'était pas à l'ordre du jour de cette thèse, même si quelques points, comme les types virtuels et les chaînes immutables, se sont révélés nécessaires pour des questions d'implémentation. Les points cruciaux à spécifier sont la combinaison de méthodes, les constantes et les exceptions. Comme point annexe on peut au moins citer la visibilité.

En cas de conflits de propriétés locales, le programmeur est obligé de redéfinir la propriété globale conflictuelle et d'appeler manuellement les *supers*. Ceci est fastidieux et entraîne un risque de double évaluation — dans les losanges. Il faudrait disposer de linéarisations et du `call-next-method` de CLOS.

Les constantes sont des éléments d'abstraction très utiles, elles rendent le code plus lisible et clair et permettent d'éviter bien des erreurs à un prix quasiment nul. Cependant aucune des solutions actuellement proposées par les langages objets ne nous satisfont, elles ne sont tout simplement pas objet du tout. La solution en cours d'élaboration pour NIT semble une solution intéressante mais elle n'a pas encore trouvé d'implémentation. Dans cette proposition (non publiée), les constantes sont vues comme des instances de classe singleton, elle peuvent donc avoir des propriétés.

Les exceptions permettent un contrôle des erreurs bien plus souple et efficace que les tests explicites faits par le programmeur. Un mécanisme d'exception est donc à spécifier pour le langage PRM. Le système utilisé par JAVA ne présente pas que des avantages mais il est vraiment très simple, il représente certainement un bon point de départ pour une réflexion plus poussée sur le sujet.

La visibilité de PRM est actuellement très restreinte et elle devrait être enrichie. Elle devrait permettre au programmeur de choisir l'interface qu'il exporte pour un module (classes, méthodes, ...). De la même manière le programmeur devrait signaler qu'il utilise un module ou qu'il prévoit de le raffiner.

PRMC. Du point de vue du compilateur, il reste encore du travail. Tout d'abord, il ne dispose pas d'un ramasse-miettes adapté, cette lacune devrait être corrigée. Ensuite, plusieurs techniques peuvent encore être optimisées, par exemple l'optimisation des sous-objets vides pour les sous-objets, de meilleures heuristiques pour la coloration, la numérotation parfaite de classes, etc. Ces optimisations amélioreraient le passage à l'échelle des techniques sans avoir cependant d'effet notable sur nos tests.

Enfin, PRMC utilise l'implémentation homogène de la généricité. Il faudrait, dans un premier temps, l'étendre à l'implémentation hétérogène, cette implémentation améliorerait sensiblement l'implémentation par sous-objets que PRMC propose. A notre avis, une implémentation semi-hétérogène, comme en C#, serait un bon compromis. Toutefois cette implémentation repose fortement sur le compilateur à la volée (JIT) de la machine virtuelle. Nous réfléchissons plus précisément à une solution basée sur une compilation multiple mais statique, où les méthodes seraient compilées pour le cas général (`Any`) et pour une combinaison de leurs types formels instanciables par un type primitif. Pour les classes qui auraient trop de paramètres, cette solution n'est certainement pas envisageable — la combinatoire est trop élevée — mais en pratique les classes qui ont plus de deux ou trois paramètres sont rares. De plus, cette implémentation utiliserait une table de méthodes différentes en fonction des types instanciés, ce qui permettrait d'éliminer l'autre défaut de l'implémentation homogène : une instance de classe générique ne peut pas savoir, à l'exécution, quels sont ses types formels.

Techniques d'implémentation dans les systèmes existants. Nous voyons principalement deux cibles à nos travaux : le sous-typage multiple et l'implémentation de C++.

Le hachage parfait de classes est une technique efficace et simple à mettre en œuvre. Il serait particulièrement intéressant de la tester en pratique sur les machines virtuelles des langages à sous-typage multiple utilisés en production (JAVA, .NET, ...). Nous avons rapidement tenté l'expérience sur JIKES RVM mais pour des raisons techniques, nous ne sommes pas allés plus loin dans l'expérimentation — la principale raison est que JIKES RVM est écrit en JAVA et nous n'avons pas réussi à faire cohabiter les entiers avec les descripteurs de méthodes de la table des méthodes. On pourrait compléter cette implémentation, avec l'utilisation du test de Cohen optimisé.

Une seconde application du hachage parfait de classes en *vraie* grandeur est possible. Il s'agit du test de sous-typage dans l'implémentation par sous-objets de C++ ainsi que de la table des *downcasts*. Toutefois, comme C++ ne propose pas que de l'héritage virtuel, il est au préalable nécessaire d'adapter la technique au cas de l'héritage non virtuel.

Une machine virtuelle toute en héritage multiple. Cette thèse n'a pas pu dégager une implémentation de l'héritage multiple à la fois efficace et compatible avec l'hypothèse du monde ouvert, du moins en l'absence de recompilation dynamique. Le hachage parfait de classes permet une implémentation efficace en l'absence d'attributs.

Pour implémenter efficacement l'héritage multiple dans l'hypothèse du monde ouvert, les recompilations dynamiques semblent nécessaires et les schémas adaptatifs doivent être considérés. En se servant des statistiques de PRM, nous avons commencé à spécifier un protocole de recompilation dynamique inspiré de la double compilation de Myers [1995] pour une machine virtuelle toute en héritage multiple [Ducournau et Morandat, 2010]. Les accès aux attributs sont compilés d'un côté de manière efficace (implémentation du sous-typage simple) pour les cas où ils auraient une position invariante, soit en utilisant le hachage parfait de classes dans les autres cas. Au chargement d'une nouvelle classe, si elle fait varier la position d'un attribut, les méthodes utilisant cet attribut passent de l'implémentation rapide à l'implémentation par défaut. Les premières simulations sur PRMC montrent que le nombre de méthodes à recompiler n'est pas démesuré. Ceci constitue un premier pas vers une machine virtuelle tout en héritage multiple, une machine qui reste cependant à spécifier.

Bibliographie

H. S. Alavi, S. Gilbert et R. Guerraoui : Extensible encoding of type hierarchies. *In Proc. POPL'08*, pages 349–358. ACM, 2008. ISBN 978-1-59593-689-9. Cité page 71.

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), pages 108–124. ACM Press, 2001a. Cité pages 100 et 111.

B. Alpern, A. Cocchi et D. Grove : Dynamic type checking in Jalapeño. *In Proc. USENIX JVM'01*, 2001b. Cité pages 66, 108, 182 et 184.

D. Ancona, G. Lagorio et E. Zucca : True separate compilation of Java classes. *In Proc. ACM PPDP'02*. ACM Press, 2002. Cité page 112.

P. André et J.-C. Royer : Optimizing method search with lookup caches and incremental coloring. *In Proc. OOPSLA'92*, SIGPLAN Notices, 27(10), pages 110–126, Vancouver, 1992. ACM Press. Cité page 84.

M. Arnold, S. J. Fink, D. Grove, M. Hind et P. F. Sweeney : A survey of adaptive optimization in virtual machines. *In Proc. of the IEEE, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation*, 2005. Cité page 147.

D. F. Bacon : *Fast and effective optimization of statically typed object-oriented languages*. Thèse de doctorat, University of California, Berkeley, 1997. Cité page 135.

D. F. Bacon, M. Wegman et K. Zadeck : Rapid type analysis for C++. Rapport technique, IBM Thomas J. Watson Research Center, 1996. Cité page 135.

D. Bacon et P. Sweeney : Fast static analysis of C++ virtual function calls. *In Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pages 324–341. ACM Press, 1996. Cité pages 104 et 135.

A. Bergel, S. Ducasse et R. Wuyts : Classboxes : A minimal module model supporting local rebinding. *In Proc. Joint Modular Languages Conf. (JMLC'03)*, LNCS 2789, pages 122–131. Springer, 2003. Cité page 37.

G. Birtwistle, O. Dahl, B. Myhrhaug et K. Nygaard : *SIMULA Begin*. Petrocelli Charter, New York (NY), USA, 1973. Cité pages 7 et 37.

H.-J. Boehm : Space-efficient conservative garbage collection. *In Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'93)*, ACM SIGPLAN Notices, 28(6), pages 197–206, 1993. Cité pages 58, 161, 166, 183 et 186.

D. Boucher : GOld : a link-time optimizer for Scheme. *In* M. Felleisen, éditeur : *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, pages 1–12, 2000. Cité page 144.

G. Bracha et W. Cook : Mixin-based inheritance. *In Proc. OOPSLA/ECOOP'90*, SIGPLAN Notices, 25(10), pages 303–311. ACM Press, 1990. Cité pages 112 et 116.

G. Bracha : Pluggable type systems. *In Workshop on Revival of Dynamic Languages, OOSPLA'04*, 2004. Cité page 12.

G. Bracha et D. Griswold : Strongtalk : typechecking smalltalk in a production environment. *SIGPLAN Not.*, 28(10):215–230, 1993. ISSN 0362-1340. Cité page 12.

B. Calder et D. Grunwald : Reducing indirect function call overhead in C++ programs. *In Proc. ACM Symp. on Principles of Prog. Lang. (POPL'94)*, pages 397–408, 1994. Cité page 74.

L. Cardelli : Type systems. *In* A. B. Tucker, éditeur : *The Computer Science and Engineering Handbook*, chapitre 97. CRC Press, 2nd édition, 2004. Cité pages 12 et 53.

E. Chailloux : An conservative garbage collector with ambiguous roots for static typechecking languages. *In IWMM 92*, pages 218–247, St. Malo, France, 1992. Y. Bekkers and J. Cohen. Cité page 58.

C. Chambers : Predicate classes. *In* O. Nierstrasz, éditeur : *Proc. of ECOOP'93*, LNCS 707, pages 268–296. Springer-Verlag, Berlin, 1993. Cité page 143.

C. Chambers, D. Grove, G. DeFouw et J. Dean : Call graph construction in object-oriented languages. *In Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 108–124. ACM Press, 1997. Cité page 135.

C. Chambers et D. Ungar : Customization : Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. *In Proc. OOPSLA'89*, pages 146–160. ACM Press, 1989. Cité page 144.

C. Chambers, D. Ungar et E. Lee : An efficient implementation of SELF, a dynamically-typed object-oriented languages based on prototypes. *In Proc. OOPSLA'89*, pages 49–70. ACM Press, 1989. Cité page 143.

S. Chiba : A metaobject protocol for C++. *In Proceedings of OOPSLA'95, Austin (TX), USA*, special issue of ACM SIGPLAN Notices, 30(10), pages 285–299, 1995. Cité page 24.

S. Chiba : Javassist—a reflection-based programming wizard for Java. *In Proc. ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998. Cité page 24.

C. Click et J. Rose : Fast subtype checking in the Hotspot JVM. *In Proc. ACM-ISCOPE conference on Java Grande (JGI'02)*, pages 96–107, 2002. Cité pages 100, 109 et 191.

C. Clifton, G. T. Leavens, C. Chambers et T. Millstein : MultiJava : Modular open classes and symetric multiple dispatch for Java. *In Proc. OOPSLA'00*, SIGPLAN Notices, 35(10), pages 130–145. ACM Press, 2000. Cité page 37.

N. Cohen : Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4):626–629, 1991. Cité pages 65, 66, 67, 69, 70, 84, 102, 107, 109, 110, 114, 172, 201 et 205.

P. Cointe : Metaclasses are first class : the ObjVlisp model. *In Proc. OOPSLA'87*, SIGPLAN Notices, 22(12), pages 156–167. ACM Press, 1987. Cité page 24.

S. Collin, D. Colnet et O. Zendra : Type inference for late binding. the SmallEiffel compiler. *In Proc. Joint Modular Languages Conference*, LNCS 1204, pages 67–81. Springer, 1997. Cité pages 104 et 143.

D. Colnet et O. Zendra : Optimizations of Eiffel programs : SmallEiffel, the GNU Eiffel compiler. *In 29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, volume 10, pages 341–350. IEEE Computer Society, 1999. Cité page 159.

Z. J. Czech, G. Havas et B. S. Majewski : Perfect hashing. *Theor. Comput. Sci.*, 182(1-2):1–143, 1997. ISSN 0304-3975. Cité page 90.

A. de Saint-Exupéry : *Terre des hommes.* Gallimard, Paris, 1939. Cité page 49.

J. Dean, D. Grove et C. Chambers : Optimization of object-oriented programs using static class hierarchy analysis. *In* W. Olthoff, éditeur : *Proc. ECOOP'95*, LNCS 952, pages 77–101. Springer, 1995. Cité page 135.

J. Dean et C. Chambers : Optimization of object-oriented programs using static class hierarchy analysis. Rapport technique 94-12-01, University of Washington, Seattle, 1994. Cité page 135.

L. DeMichiel et R. Gabriel : The Common Lisp Object System : An overview. *In* J. Bezivin, P. Cointe, J.-M. Hullot et H. Liebermann, éditeurs : *Proc. ECOOP'87*, LNCS 276, pages 201–220. Springer, 1987. Cité page 38.

N. Desnos : Un garbage collector pour la coloration bi-directionnelle. Mémoire de D.E.A., Université Montpellier 2, 2004. Cité pages 89 et 161.

D. Detlefs et O. Agesen : Inlining of virtual methods. *In* R. Guerraoui, éditeur : *Proc. ECOOP'99*, LNCS 1628, pages 258–277. Springer, 1999. Cité page 144.

L. Deutsch et A. Schiffman : Efficient implementation of the Smalltalk-80 system. *In Proc. ACM Symp. on Principles of Prog. Lang. (POPL'84)*, pages 297–302, 1984. Cité page 103.

E. W. Dijkstra : Recursive programming. *Numer. Math.*, 2:312–318, 1960. Cité page 65.

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, 1989a. Cité page 84.

R. Dixon, T. McKee, P. Schweitzer et M. Vaughan : A fast method dispatcher for compiled languages with multiple inheritance. *In Proc. OOPSLA'89*, pages 211–214. ACM Press, 1989b. Cité page 84.

K. Driesen : *Efficient Polymorphic Calls*. Kluwer Academic Publisher, 2001. Cité pages 27, 62, 63, 99, 117, 118, 177 et 198.

S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts et A. Black : Traits : A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2005. Cité pages 37 et 112.

R. Ducournau : *Y3 : YAFOOL, le langage à objets, et YAFEN, l'interface graphique*. Sema Group, Montrouge, 1991. Cité pages 31, 38 et 84.

R. Ducournau : La compilation de l'envoi de message dans les langages dynamiques. *L'Objet*, 3(3):241–276, 1997. Cité page 183.

R. Ducournau : La coloration pour l'implémentation des langages à objets à typage statique. *In* M. Dao et M. Huchard, éditeurs : *Actes LMO'2002* in *L'Objet vol. 8*, pages 79–98. Lavoisier, 2002. Cité page 15.

R. Ducournau : Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6):1–56, 2008. Cité pages 66, 90, 97, 192, 198 et 202.

R. Ducournau : Coloring, a versatile technique for implementing object-oriented languages. *Softw. Pract. Exper.*, 40(?):1–39, 2010. (to appear). Cité pages 84, 85, 87, 89, 118, 202 et 203.

R. Ducournau : Implementing statically typed object-oriented programming languages. *ACM Comp. Surv.*, 43(4), 2011. Cité pages 80, 81, 82, 83, 94, 97, 118 et 198.

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. Cité pages 23, 32 et 34.

R. Ducournau et F. Morandat : Perfect class hashing and numbering for object-oriented implementation. *Softw. Pract. Exper.*, 41(?):1–32, 2011. (to appear). Cité pages 92, 93, 94, 105, 110, 112, 119, 121, 122, 124, 185, 190, 192, 193, 198, 202 et 221.

R. Ducournau, F. Morandat et J. Privat : Modules and class refinement : a metamodeling approach to object-oriented languages. Rapport de Recherche LIRMM-07021, Université Montpellier 2, 2007. Cité pages 37 et 42.

R. Ducournau, F. Morandat et J. Privat : Empirical assessment of object-oriented implementations with multiple inheritance and static typing. *In* G. T. Leavens, éditeur : *Proc. OOPSLA'09*, SIGPLAN Notices, 44(10), pages 41–60. ACM Press, 2009. Cité pages 97, 101, 184, 185, 186, 188 et 221.

R. Ducournau et J. Privat : Metamodeling semantics of multiple inheritance. *Science of Computer Progamming*, pages 1–40, 2011. (to appear). Cité pages 23, 26, 27, 28, 35 et 219.

R. Ducournau et F. Morandat : Towards a Full Multiple-Inheritance Virtual Machine. *In ECOOP'10 : Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'10)*, page 6, Maribor Slovénie, 06 2010. ACM. URL http://hal-lirmm.ccsd.cnrs.fr/lirmm-00486462/en/. Cité pages 147, 198, 199 et 206.

M. Ellis et B. Stroustrup : *The annotated C++ reference manual*. Addison-Wesley, Reading, MA, US, 1990. Cité pages 74 et 76.

E. Ernst : Higher-order hierarchies. *In* L. Cardelli, éditeur : *Proc. ECOOP'2003*, LNCS 2743, pages 303–329. Springer, 2003. Cité page 37.

E. Ernst : *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Thèse de doctorat, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999. Cité page 31.

E. Ernst : Safe dynamic multiple inheritance. *Nord. J. Comput*, 9(1):191–208, 2002. Cité page 112.

M. F. Fernandez : Simple and effective link-time optimization of Modula-3 programs. *In ACM Conference on Programming Language Design and Implementation*, pages 103–115, 1995. Cité page 144.

R. B. Findler et M. Flatt : Modular object-oriented programming with units and mixins. *In Proc. ICFP'98*, SIGPLAN Notices, 34(1), pages 94–104. ACM Press, 1999. Cité page 144.

I. R. Forman et S. H. Danforth : *Putting Metaclasses to Work*. Addison-Wesley, 1999. Cité page 24.

E. M. Gagnon : *A Portable Research Framework for the Execution of Java Bytecode*. Thèse de doctorat, Department of Computer Science, McGuill University, Montréal, 2002. Cité pages 89 et 111.

E. M. Gagnon et L. J. Hendren : Sablevm : A research framework for the efficient execution of java bytecode. *In Java Virtual Machine Research and Technology Symposium*, pages 27–40. USENIX, 2001. ISBN 1-880446-11-1. Cité pages 111 et 123.

E. Gamma, R. Helm, R. Johnson et J. Vlissides : *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (MA), USA, 1994. Cité pages 20 et 51.

E. Gamma, R. Helm et R. Johnson : *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. Cité page 7.

M. Garey et D. Johnson : *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco (CA), USA, 1979. Cité page 84.

J. Gil et A. Itai : The complexity of type analysis of object oriented programs. *In Proc. ECOOP'98*, LNCS 1445, pages 601–634. Springer, 1998. Cité page 135.

J. Gil et Y. Zibin : Efficient subtyping tests with PQ-encoding. *ACM Trans. Program. Lang. Syst.*, 27(5):819–856, 2005. Cité page 71.

A. Goldberg et D. Robson : *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading (MA), USA, 1983. Cité pages 7, 24 et 25.

D. Grove et C. Chambers : A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001. Cité pages 104 et 134.

D. Grove : The impact of interprocedural class inference on optimization. *In CASCON'95 Centre for Advanced Studies Conference*, pages 195–203, 1995. Cité page 134.

D. Grove, G. DeFouw, J. Dean et C. Chambers : Call graph construction in object-oriented languages. *In Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 108–124. ACM Press, 1997. Cité page 134.

U. Hölzle, C. Chambers et D. Ungar : Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. *In* P. America, éditeur : *Proc. ECOOP'91*, LNCS 512, pages 21–38. Springer, 1991. Cité pages 103 et 104.

J.-M. Hullot : CEYX version 15. Technical Report 44, 45 et 46, I.N.R.I.A., 1985. Cité page 106.

Y. Ichisugi et A. Tanaka : Difference-based modules : A class-independent module mechanism. *In* B. Magnusson, éditeur : *Proc. ECOOP'2002*, LNCS 2374, pages 62–88. Springer, 2002. Cité page 37.

R. Jones et R. Lins : *Garbage Collection.* Wiley, 1996. Cité pages 58 et 186.

S. Keene : *Object-Oriented Programming in* COMMON LISP. *A programmer's guide to* CLOS. Addison-Wesley, Reading, MA, 1989. Cité page 24.

G. Kiczales, J. des Rivières et D. Bobrow : *The Art of the Meta-Object Protocol.* MIT Press, 1991. Cité page 24.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm et W. G. Griswold : An overview of AspectJ. *In* J. L. Knudsen, éditeur : *Proc. ECOOP'2001*, LNCS 2072, pages 327–355. Springer, 2001. Cité page 24.

D. E. Knuth : *The art of computer programming, Sorting and Searching*, volume 3. Addison-Wesley, 1973. Cité page 119.

H. Lieberman et C. Hewitt : A realtime garbage collector based on the lifetimes of objects. *Communication of the ACM*, 26(6):419–429, 1983. Cité page 58.

S. Lippman : *Inside the C++ Object Model.* Addison-Wesley, New York, 1996. Cité pages 74 et 82.

B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson et A. C. Myers : THETA reference manual. Technical report, MIT, 1995. Cité page 19.

B. Meyer : *Object-Oriented Software Construction.* Prentice Hall International Series in Computer Science, C.A.R. Hoare Series Editor. Prentice Hall International, Hemel Hempstead, UK, 1988. Nouvelle édition révisée : [Meyer, 1997]. Cité pages 7 et 15.

B. Meyer : *Eiffel : The Language.* Prentice-Hall, 1992. Cité page 213.

B. Meyer : *Eiffel, le langage.* InterEditions, Paris, 1994. Traduction française de [Meyer, 1992]. Cité page 33.

B. Meyer : *Object-Oriented Software Construction.* Prentice-Hall, second édition, 1997. Cité pages 128 et 213.

B. Meyer : Static typing. *In OOPSLA '95 : Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, pages 20–29, New York (NY), USA, 1995. ACM Press. Cité page 12.

F. Morandat, R. Ducournau et J. Privat : Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique. *In* B. Carré et O. Zendra, éditeurs : *Actes LMO'2009*, pages 17–32. Cépaduès, 2009. Cité page 185.

F. Morandat et R. Ducournau : Empirical Assessment of C++-Like Implementation for Multiple Inheritance. *In ECOOP'10 : Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'10)*, page 5, Maribor Slovénie, 2010. ACM. URL http://hal-lirmm.ccsd.cnrs.fr/lirmm-00486644/en/. Cité pages 184, 185, 194, 195 et 221.

S. Muthukrishnan et M. Muller : Time and space efficient method lookup for object-oriented languages. *In Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 42–51. ACM/SIAM, 1996. Cité page 70.

A. Myers : Bidirectional object layout for separate compilation. *In Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 124–139. ACM Press, 1995. Cité pages 76, 94, 199 et 206.

N. Nystrom, S. Chong et A. C. Myers : Scalable extensibility via nested inheritance. *In* J. M. Vlissides et D. C. Schmidt, éditeurs : *Proc. OOPSLA'04*, SIGPLAN Notices, 39(10), pages 99–115. ACM Press, 2004. Cité page 37.

N. Nystrom, X. Qi et A. C. Myers : $\mathcal{JE}$: Nested intersection for scalable software composition. *In* P. L. Tarr et W. R. Cook, éditeurs : *Proc. OOPSLA'06*, SIGPLAN Notices, 41(10), pages 21–35. ACM Press, 2006. Cité page 37.

G. Ockham : *Quaestiones et decisiones in quatuor libros Sententiarum cum centilogio theologico*, volume II. (Inconnu), 1319. Cité page 49.

M. Odersky, L. Spoon et B. Venners : *Programming in Scala, A comprehensive step-by-step guide*. Artima, 2008. Cité page 112.

K. Palacz et J. Vitek : Java subtype tests in real-time. *In* L. Cardelli, éditeur : *Proc. ECOOP'2003*, LNCS 2743, pages 378–404. Springer, 2003. Cité pages 67, 97, 100, 142, 176 et 184.

B. H. C. Pfister et J. Templ : Oberon technical notes. Rapport technique 156, Eidgenossische Techniscle Hochschule Zurich–Departement Informatik, 1991. Cité page 66.

J. Privat : Analyse de types et graphe d'appels en compilation séparée. Mémoire de dea, Université Montpellier 2, 2002. Cité page 134.

J. Privat et R. Ducournau : Intégration d'optimisations globales en compilation séparée des langages à objets. *In* J. Euzenat et B. Carré, éditeurs : *Actes LMO'2004* in *L'Objet vol. 10*, pages 61–74. Lavoisier, 2004. Cité page 144.

J. Privat et R. Ducournau : Link-time static analysis for efficient separate compilation of object-oriented languages. *In ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05)*, pages 20–27, 2005. Cité page 144.

J. Privat : *De l'expressivité à l'efficacité, une approche modulaire des langages à objets — Le langage PRM et le compilateur prmc.* Thèse de doctorat, Université Montpellier II, 2006. Cité pages 3, 20, 37, 49, 151, 152, 164, 197 et 201.

J. Privat et F. Morandat : Coloring for shared object-oriented libraries. *In Workshop ICOOOLPS at ECOOP'08*, 2008. Cité page 97.

W. Pugh et G. Weddell : Two-directional record layout for multiple inheritance. *In Proc. PLDI'90*, ACM SIGPLAN Notices, 25(6), pages 85–91, 1990. Cité pages 84, 89, 172 et 202.

W. Pugh et G. Weddell : On object layout for multiple inheritance. Rapport technique CS-93-22, University of Waterloo, 1993. Cité pages 84 et 89.

F. Qian et L. J. Hendren : A study of type analysis for speculative method inlining in a jit environment. *In* R. Bodík, éditeur : *CC*, volume 3443 de *Lecture Notes in Computer Science*, pages 255–270. Springer, 2005. ISBN 3-540-25411-0. Cité page 135.

C. Queinnec : Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*, 64(6):315–321, 1998. Cité pages 66 et 106.

O. Raynaud et E. Thierry : A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. *In* J. L. Knudsen, éditeur : *Proc. ECOOP'2001*, LNCS 2072, pages 165–180. Springer, 2001. Cité page 70.

J. G. Rossie et D. P. Friedman : An algebraic semantics of subobjects. *In Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 187–199. ACM Press, 1995. Cité page 74.

B. Ryder : Constructing the call graph of a program. *IEEE Transaction on Software Engineering*, 5(3):216–225, 1979. Cité page 134.

O. Sallenave et R. Ducournau : Efficient Compilation of NET Programs for Embedded Systems. *In ECOOP'10 : Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'10)*, Maribor Slovénie, 2010. ACM. Cité pages 85, 123 et 203.

L. Schubert, M. Papalaskaris et J. Taugher : Determining type, part, color and time relationship. *Computer*, 16:53–60, 1983. Cité page 66.

M. Serrano : *Vers une compilation portable et performante des langages fonctionnels*. Thèse de doctorat, Universitè Paris 6, 1994. Cité pages 159 et 160.

D. L. Shang : Are cows animals ? URL http$:$//www.visviva.com/transframe/papers/covar.htm. 1996. Cité pages 15 et 16.

O. Shivers : Control-flow analysis in scheme. *In Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, 1988. Cité page 135.

O. Shivers : *Control-Flow Analysis of Higher-Order Languages*. Thèse de doctorat, Carnegie Mellon University, 1991. Cité page 135.

R. Sprugnoli : Perfect hashing functions : a single probe retrieving method for static sets. *Comm. ACM*, 20(11):841–850, 1977. Cité page 90.

A. Srivastava : Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems*, 1(4):355–364, 1992. Cité page 133.

G. Steele : *Common Lisp, the Language*. Digital Press, second édition, 1990. Cité pages 27 et 38.

M. Stefik et D. Bobrow : Object-oriented programming : Themes and variations. *AI Magazine*, 6(4):40–62, 1986. Cité page 112.

B. Stroustrup : *The C++ programming Language, Special ed.* Addison-Wesley, 2000. Cité page 74.

C. Szyperski : Import is not inheritance. Why we need both : Modules and classes. *In* O. L. Madsen, éditeur : *Proc. ECOOP'92*, LNCS 615, pages 19–32. Springer, 1992. Cité page 37.

S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder et P. Leroy, éditeurs. *Ada 2005 Reference Manual : Language and Standard Libraries*. LNCS 4348. Springer, 2006. Cité page 62.

P. Takhedmit : Coloration de classes et de propriétés : étude algorithmique et heuristique. Mémoire de D.E.A., Université Montpellier 2, 2003. Cité page 84.

F. Tip et J. Palsberg : Scalable propagation-based call graph construction algorithms. *In OOPSLA '00 : Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-200-X. Cité page 135.

F. Tip et P. F. Sweeney : Class hierarchy specialization. *Acta Informatica*, 36(12):927–982, 2000. Cité page 37.

M. Torgersen : Virtual types are statically safe. *In Proc. of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL 5)*, San Diego, CA, 1998. Cité page 15.

J. Vitek, R. Horspool et A. Krall : Efficient type inclusion tests. *In Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 142–157. ACM Press, 1997. Cité page 84.

T. Wang et S. Smith : Precise constraint-based type inference for Java. *In* J. L. Knudsen, éditeur : *Proc. ECOOP'2001*, LNCS 2072, pages 99–117. Springer, 2001. Cité page 134.

O. Zendra : *Traduction et optimisation globale dans les langages de classes*. Thèse de doctorat, Université Nancy 1, 2000. Cité page 178.

O. Zendra et D. Colnet : Vers un usage plus sûr de l'aliasing en Eiffel. *In* C. Dony et H. A. Sahraoui, éditeurs : *Actes LMO'2000*, pages 183–194. Hermès, 2000. Cité page 157.

O. Zendra, D. Colnet et S. Collin : Efficient dynamic dispatch without virtual function tables : The SmallEiffel compiler. *In Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 125–141. ACM Press, 1997. Cité pages 104 et 143.

O. Zendra et K. Driesen : Stress-testing control structures for dynamic dispatch in Java. *In 2nd Java Virtual Machine Research and Technology Symposium (JVM 2002), San Francisco, California, USA*, pages 105–118. Usenix — The Advanced Computing Systems Association, août 2002. Cité page 106.

Y. Zibin et J. Gil : Incremental algorithms for dispatching in dynamically typed languages. *In Proc. POPL'03*, pages 126–138. ACM, 2003. Cité page 89.

Table des figures

Liste des tableaux

Listings

Abstract

This thesis is about efficient compilation of object oriented languages with multiple inheritance. Object oriented programing is characterized by a main mechanism, *late binding* — the invoked method only depends on the dynamic type of one special parameter, the *receiver*. In order to be efficient, this mechanism needs an implementation which depends on some compilation scheme — separate compilation with dynamic loading, global compilation, etc. However, object oriented programming presents an incompatibility between three concerns: multiple inheritance, efficiency and open world assumption — especially with dynamic loading.

In this thesis, we have studied common implementation techniques compatible with multiple inheritance and a promising alternative, perfect class hashing. The context of this study is static typing, our conclusion holds for languages like C++, EIFFEL, JAVA, C#, etc. Different compilation schemes are considered, from open world assumption to closed world assumption. These techniques and schemes are implemented in the PRM bootstraped compiler. The efficiency of all these artifacts has been tested with a rigorous meta-compilation experimental protocol, and these tests have been performed on a variety of different processors. Results of these experiments are discussed and compared to an a priori evaluation of the implementation techniques. They mainly confirm perfect class hashing as an interesting implementation for multiple subtyping, a la JAVA.

Keywords: *Object Oriented Language, Static Typing, Multiple Inheritance, Compilation, Implementation, Modules, Open World Assumption, Closed World Assumption, Empirical Assessment*

Résumé

Cette thèse traite de la compilation efficace des langages à objets en héritage multiple. La programmation objet est caractérisée par un mécanisme fondamental, *la liaison tardive* — la méthode appelée dépend du type dynamique d'un paramètre distingué, le *receveur*. L'efficacité de ce mécanisme nécessite une implémentation adéquate qui est conditionnée par le schéma de compilation utilisé — compilation séparée avec chargement dynamique, compilation globale, etc. Cependant la programmation par objets présente une apparente incompatibilité entre trois termes : l'héritage multiple, l'efficacité et l'hypothèse du monde ouvert— en particulier, le chargement dynamique.

Nous avons étudié les techniques d'implémentation compatibles avec l'héritage multiple couramment utilisées ainsi qu'une alternative prometteuse, le hachage parfait de classes. Nous nous plaçons dans le cadre du typage statique, donc nos conclusions peuvent valoir pour des langages comme C++, EIFFEL, JAVA, C#, etc. Différents schémas de compilation sont considérés, de l'hypothèse du monde ouvert à l'hypothèse du monde clos. Ces techniques et ces schémas ont été mis en œuvre dans le compilateur auto-gène du langage PRM. L'influence sur l'efficacité de tous ces éléments a été testée dans un protocole expérimental rigoureux de méta-compilation et les tests ont été réalisés sur une variété de processeurs différents. Les résultats de ces expérimentations sont discutés et comparés aux évaluations a priori effectuées sur les techniques d'implémentation. Ils confirment aussi que le hachage parfait de classes est une technique d'implémentation intéressante pour le sous-typage multiple à la JAVA.

Mots clefs : *Langage objet, Typage statique, Héritage multiple, Compilation, Implémentation, Modules, Hypothèse du monde ouvert, Hypothèse du monde clos, Évaluations empiriques*

LIRMM — 161, rue Ada — 34095 Montpellier cedex 5 — France