Académie de Montpellier
Université Montpellier II
— Sciences et Techniques du Languedoc —

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**

De l'expressivité à l'efficacité
une approche modulaire des langages à objets
Le langage PRM et le compilateur `prmc`

par

Jean Privat

Soutenue le vendredi 21 juillet 2006, devant le jury composé de :

Directeur de thèse
M. Roland Ducournau, professeur, LIRMM, Université Montpellier II

Rapporteurs
M. Christian Queinnec, professeur, LIP6, UPMC, Paris (président du jury)
M. Manuel Serrano, directeur de recherche, INRIA, Sophia Antipolis
M. Jan Vitek, associate professor, Purdue University, Indiana, USA (non membre du jury)

Examinateurs
M. Stéphane Ducasse, professeur, LISTIC, Université de Savoie, Annecy
M^{me} Marianne Huchard, professeur, LIRMM, Université Montpellier II
M. Olivier Zendra, chargé de recherche, INRIA, LORIA, Nancy

Table des matières

Table des matières iii

Table des figures ix

Liste des tableaux xi

Remerciements xiii

1 Introduction générale 1

I Spécification et modélisation des langages à objets 7

2 Présentation du langage de programmation PRM 9
2.1 Introduction . . . 9
2.2 Pourquoi un nouveau langage de programmation ? . . . 10
2.3 Qualité des langages de programmation . . . 11
2.3.1 Facilité d'apprentissage . . . 11
2.3.2 Expressivité . . . 12
2.3.3 Lisibilité . . . 12
2.3.4 Souplesse . . . 13
2.3.5 Assistance . . . 13
2.3.6 Agrément . . . 13
2.4 Des concepts simples pour une syntaxe claire . . . 14
2.4.1 Concepts simples et cohérents . . . 14
2.4.2 Syntaxe concise mais structurée . . . 16
2.5 Paradigme orienté objet . . . 19
2.5.1 Les quatre caractéristiques de l'approche par objet . . . 19
2.5.2 Héritage multiple . . . 20
2.5.3 Petit aperçu syntaxique en PRM . . . 21
2.6 Typage statique . . . 22
2.6.1 Sûreté . . . 23

2.6.2 Documentation et abstraction 23
2.6.3 Efficacité 24
2.6.4 Généricité 25
2.6.5 Typage covariant 25
2.7 Modules et raffinement de classes 27
2.7.1 Comparaison des différents systèmes de modules 28
2.7.2 Modules et classes : la vraie nature des modules 32
2.7.3 Raffinement de classes 33
2.8 Conclusion 37

3 Une méta-modélisation des classes et propriétés 39
3.1 Introduction 39
3.2 Classes et propriétés 41
3.2.1 Définition de hiérarchies de classes et construction de modèles . . 43
3.2.2 Formalisme 44
3.3 Spécialisation de classes 45
3.3.1 Formalisme 46
3.3.2 Variations autour de la spécialisation 50
3.4 Propriétés globales 53
3.4.1 Formalisme 54
3.4.2 Nom de propriété et envoi de message 54
3.4.3 Noms et héritage multiple 56
3.4.4 Variations autour des propriétés globales 58
3.5 Propriétés locales 59
3.5.1 Déclaration de propriétés locales 60
3.5.2 Introduction des propriétés globales 61
3.5.3 Redéfinition 62
3.5.4 Variation : la surcharge statique 63
3.6 Envoi de message et héritage des propriétés locales 63
3.6.1 Formalisme 64
3.6.2 Héritage simple 65
3.6.3 Héritage multiple 65
3.6.4 Variations autour de la résolution par défaut 68
3.7 Sélection complexe et redéfinition partielle 72
3.7.1 Propriétés locales et méta-propriétés 72
3.7.2 Sémantique additionnelle, exemple : les méthodes abstraites . . . 73
3.7.3 Redéfinition partielle, exemple : la protection 73
3.7.4 Héritage combinant, exemple : les exceptions déclarées 75
3.7.5 Redéfinition combinante, exemple : les contrats 76
3.8 Typage statique 76
3.9 Conclusion 78

4 Une méta-modélisation des modules et du raffinement de classes 79
4.1 Introduction . . . 79
4.2 Modules et classes . . . 80
4.2.1 Définition de hiérarchies de modules et construction de modèles . 83
4.2.2 Hiérarchie de modules . . . 83
4.2.3 Sémantiques et constructions . . . 85
4.3 Modules, dépendances et importations de classes . . . 86
4.3.1 Dépendance de modules . . . 86
4.3.2 Classes globales . . . 87
4.3.3 Classes locales . . . 88
4.4 Spécialisation de classes . . . 89
4.4.1 Spécialisation importée . . . 89
4.4.2 Conflit de spécialisation . . . 90
4.4.3 Spécialisation déclarée . . . 90
4.5 Héritage des propriétés . . . 91
4.5.1 Hiérarchie de classes d'une hiérarchie de modules . . . 91
4.5.2 Conflit de propriétés globales . . . 92
4.5.3 Conflit de propriétés locales . . . 94
4.5.4 Typage statique . . . 95
4.6 Autres travaux . . . 97
4.7 Conclusion . . . 98

5 De la spécification de langages aux techniques efficaces de compilation 101

II Compilation séparée et efficace des langages à objets 107

6 Compilation et langages à objets 109
6.1 Introduction . . . 109
6.1.1 Compilation . . . 109
6.1.2 Implémentation des langages à objets . . . 110
6.2 Spécialisation simple . . . 113
6.2.1 Le principe . . . 113
6.2.2 Le casting . . . 115
6.2.3 Redéfinition de types . . . 118
6.2.4 Évaluation . . . 121
6.3 Spécialisation multiple . . . 121
6.3.1 Principe d'implémentation de C++ . . . 122
6.3.2 Casting . . . 125
6.3.3 Redéfinition de types . . . 127
6.3.4 Évaluation . . . 127

6.3.5 Variations autour de l'implémentation de la spécialisation multiple 129
6.3.6 Temps et espace . . . 129
6.3.7 Spécialisation simple sans surcoût . . . 129
6.3.8 Accès aux attributs . . . 130
6.4 Techniques globales d'implémentation . . . 130
6.4.1 Recopie physique des méthodes . . . 131
6.4.2 Analyse globale des types . . . 132
6.4.3 Appel direct et mise en ligne . . . 135
6.4.4 Prédiction de type et arbre de décision . . . 136
6.4.5 Coloration . . . 137
6.4.6 SmartEiffel . . . 140
6.5 Compilation et PRM . . . 141

7 Techniques globales en compilation séparée 143
7.1 Introduction . . . 143
7.2 Principe de la compilation séparée . . . 144
7.2.1 Un peu d'histoire . . . 144
7.2.2 Schéma de compilation séparée . . . 145
7.3 Édition de liens en C++ . . . 146
7.3.1 Mutilation de nom (*name mangling*) . . . 147
7.3.2 Initialisations et destructions statiques . . . 148
7.3.3 Code non lié à un fichier source . . . 148
7.3.4 Limites de l'approche de C++ . . . 150
7.4 Compilation séparée et optimisations globales . . . 151
7.4.1 Phase locale . . . 151
7.4.2 Phase globale . . . 154
7.5 Compilation séparée et raffinement de classes . . . 155
7.6 Compilation séparée optimisante : autres approches . . . 156
7.7 Perspectives et conclusion . . . 157
7.7.1 Mise en ligne des appels monomorphes . . . 157
7.7.2 Bibliothèques partagées . . . 157
7.7.3 Conclusion . . . 158

8 prmc, le compilateur PRM 159
8.1 Présentation du compilateur prmc . . . 159
8.1.1 Programmes de test . . . 160
8.1.2 Modes de fonctionnement . . . 161
8.1.3 Langage cible : C . . . 162
8.1.4 Gestion de la mémoire . . . 164
8.1.5 Dépendance externe . . . 167
8.1.6 Les différentes options . . . 168

8.2 L'architecture de prmc . . . 171
8.2.1 Construction de l'arbre syntaxique . . . 173
8.2.2 Construction du modèle et vérification des types . . . 174
8.2.3 Génération locale . . . 175
8.2.4 Analyse globale . . . 176
8.2.5 Génération globale . . . 177
8.3 Génération de code C . . . 178
8.3.1 Représentation des objets standards . . . 178
8.3.2 Représentation des types primitifs . . . 179
8.3.3 Test d'égalité . . . 182
8.3.4 Corps des méthodes . . . 184
8.3.5 Envoi de message . . . 186
8.3.6 Création d'instance . . . 191
8.3.7 Accès aux attributs . . . 192
8.3.8 Test de types . . . 196
8.4 Comparaison de trois compilateurs . . . 197
8.4.1 Description . . . 197
8.4.2 Résutats . . . 199
8.5 Conclusion . . . 204

9 Conclusion générale 207

Annexes 215

A Spécification complète du langage PRM: PRM—The Language 215
A.1 A PRM Introduction . . . 215
A.1.1 Three Simple Examples . . . 216
A.1.2 The PRM Syntax: A First Impression . . . 217
A.1.3 Outline . . . 218
A.2 Object-Oriented Programming . . . 218
A.2.1 Class Definition . . . 218
A.2.2 A Word about Class Specialisation . . . 219
A.2.3 Properties: Attributes and Methods . . . 219
A.2.4 Object Creation . . . 230
A.2.5 Visibility . . . 232
A.2.6 Class Specialisation . . . 240
A.2.7 Genericity . . . 250
A.3 Modules . . . 252
A.3.1 Module Structure . . . 252
A.3.2 Module Dependence . . . 252

A.3.3 Class Refinement . . . 253
A.3.4 Procedural style . . . 255
A.3.5 Base Modules . . . 257
A.3.6 Base Classes . . . 259
A.4 The Base Language . . . 266
A.4.1 Source Structure . . . 266
A.4.2 Name . . . 266
A.4.3 Type . . . 267
A.4.4 Expression . . . 268
A.4.5 Statement . . . 270
A.5 A PRM Conclusion . . . 276
A.6 Index . . . 277

B Index des langages **281**
B.1 ADA . . . 281
B.2 C . . . 281
B.3 C++ . . . 282
B.4 Clos . . . 282
B.5 Eiffel . . . 283
B.6 Java . . . 283
B.7 Objective Caml . . . 284
B.8 Pascal . . . 284
B.9 Perl . . . 284
B.10 PRM . . . 284
B.11 Python . . . 285
B.12 Ruby . . . 285
B.13 Simula . . . 285
B.14 Smalltalk . . . 285

C Spécification du pseudo-langage d'assemblage **287**
C.1 Instructions et arguments . . . 287
C.1.1 Arguments . . . 287
C.1.2 Mémoire : instructions `load` et `store` . . . 287
C.1.3 Calculs : instructions `add` et `cmp` . . . 288
C.1.4 Branchements : instructions `call` et `j*` . . . 288
C.2 Processeur . . . 289

Bibliographie **291**

Index **303**

Table des figures

1.1 Programmeur, langage et ordinateur 1

2.1 Caractéristiques et critères de qualité dans les langages 11
2.2 Architecture d'un logiciel de visualisation d'images 29
2.3 Architecture d'un logiciel à base de composants 30

3.1 Méta-modèle des propriétés 41
3.2 Instance du méta-modèle de l'exemple du listing en JAVA 44
3.3 Super-classes et sous-classes 47
3.4 Super-classes, super-classes déclarées et super-classes directes 49
3.5 Instances de classes en héritage répété 52
3.6 Conflit de propriétés globales en héritage multiple 56
3.7 Conflit de propriétés locales en héritage multiple 66
3.8 Exemple pour la comparaison des mécanismes d'héritage 67
3.9 Résultat de la comparaison des mécanismes d'héritage 71
3.10 Exemple de protection en EIFFEL 74

4.1 Méta-modèle des modules et des classes 81
4.2 Méta-modèle des modules, des classes et des propriétés 82
4.3 Exemple de représentation graphique d'une définition de hiérarchie de modules 84
4.4 Conflit de classes globales 87
4.5 Définition implicite de classes locales 88
4.6 Spécialisation et classes locales 90
4.7 Hiérarchie de classes d'une hiérarchie de modules 92
4.8 Conflits de propriétés globales par raffinement et spécialisation 93
4.9 Conflits de propriétés locales par raffinement et spécialisation 94
4.10 Raffinement et typage 96

6.1 Structure des objets et des tables de méthodes en spécialisation simple . . . 114
6.2 Structure des objets et des tables de méthodes en spécialisation multiple . . 123
6.3 Tables des méthodes pour l'exemple de la figure 6.2, suivant les types statiques et dynamiques 124

6.4 Les types en jeu sur le receveur, dans un appel de méthode 125
6.5 Implémentation des classes et des objets via la coloration 138

7.1 La compilation séparée 145
7.2 Phase locale : compilation de la classe A 152
7.3 Phase globale : compilation des classes A et B 154

8.1 Hiérarchie des modules de `prmc` 172
8.2 Architecture du compilateur `prmc` : les étapes fondamentales. 173
8.3 Les étapes de la construction de l'arbre syntaxique. 173
8.4 Les étapes de la construction du modèle et de typage. 174
8.5 Les étapes de la génération locale du code destination. 175
8.6 Les étapes de l'analyse globale. 176
8.7 Les étapes de la génération globale de l'exécutable. 177
8.8 Hiérarchie de classes utilisée dans les programmes de benchmark 198
8.9 Taille de l'exécutable *strippé* 200
8.10 150 000 000 envois de messages 201
8.11 3 200 000 000 accès d'attributs 202
8.12 Casting descendant 203

Liste des tableaux

2.1 Comparaison de quelques systèmes de modules 28

8.1 Impact de la compilation en modes séparé et global 162
8.2 Impact des politiques de gestion de la mémoire (option `--gc`) 166
8.3 Impact des options `--nocheck` et `--boost` 169
8.4 Impact de l'option `--class_identifier` 179
8.5 Impact de l'option `--notag` 185
8.6 Impact temporel de l'implémentation native des méthodes primitives. 187
8.7 Impact de l'implémentation des envois de message polymorphes (option `--select`) 189
8.8 Implémentation de l'accès aux attributs (option `--attribute`) 196

A.1 The Basic Types 259
A.2 Char and String Escape Sequencesprmtli]Escape sequence 261
A.3 Reserved Names 267

Remerciements

Tout au long de mon travail, de nombreuses personne m'ont aidé et soutenu. Ce sont ces personne que je voudrais tout particulièrement remercier.

Roland Ducournau, mon directeur pour m'avoir permis d'entre-apercevoir la vraie nature des objets et de la spécialisation de classes. Je le remercie également pour nos échanges fréquents dont la portée ne m'apparaissait parfois que bien plus tard.

Christian Queinnec, Jan Vitek et Manuel Serrano, qui m'ont fait l'honneur d'être rapporteurs de ma thèse, ainsi que Stéphane Ducasse, Mariane Huchard et Oliver Zendra d'avoir accepté de faire parti de jury de thèse. Je leur suis infiniment reconnaissant.

Clémentine Nébut, pour sa relecture profonde de ce présent manuscrit et l'extermination de nombreuses fautes, coquilles et mauvaises tournures de phrases.

Ehoud Ahronovitz, Yolande Ahronovitz, Thérèse Libourel et tous les enseignants avec lesquels j'ai eu le plaisir de travailler et qui m'on appris à apprendre.

Toute l'équipe D'OC pour m'avoir guidé à travers les arcanes des objets, des langages et du génie logiciel.

Mes camarades stagiaires, doctorant et docteurs qui ont contribué à l'ambiance chaleureuse du LIRMM, qui était toujours présent quand j'avais besoin d'eux et réciproquement : Gilles pour m'avoir mis en garde, Luc pour avoir su composer lors de nos nombreuses discussions, Sylvain pour son témoignage et ses qualités naturelles, Fabien pour sa vivacité d'esprit et sa relecture pertinente, Toufik pour sa disponibilité et sa cosmo-attitude, Jérôme pour ses questionnements existentiels, Damien pour sa bosse, Benoît, Binh et Christophe pour venir boire notre café, Mehdi parce qu'il est libre, Alexandre pour son organisation, Simon pour Nacridan, Robin pour nos discussions scientifiques gratifiantes, Floréal, pour s'être porté volontaire et son dogmatisme pro-PRM, ainsi que tous les autres que je ne peux tous citer...

Mes parents qui m'ont soutenu et m'ont déchargé de nombreux soucis.

Chiêm, ma femme, pour avoir supporté mon caractère pendant mes périodes de doute ou de travail intense.

Chapitre 1

Introduction générale

Les langages de programmation

Les langages de programmation sont des outils indispensables à la chaîne de production des logiciels informatiques. Il n'en existe pas de définition universellement reconnue, toutefois, en prenant la définition la plus large possible — un langage de programmation est un système de codage permettant d'écrire un programme — on dénombrerait plus de 8000 langages[1].

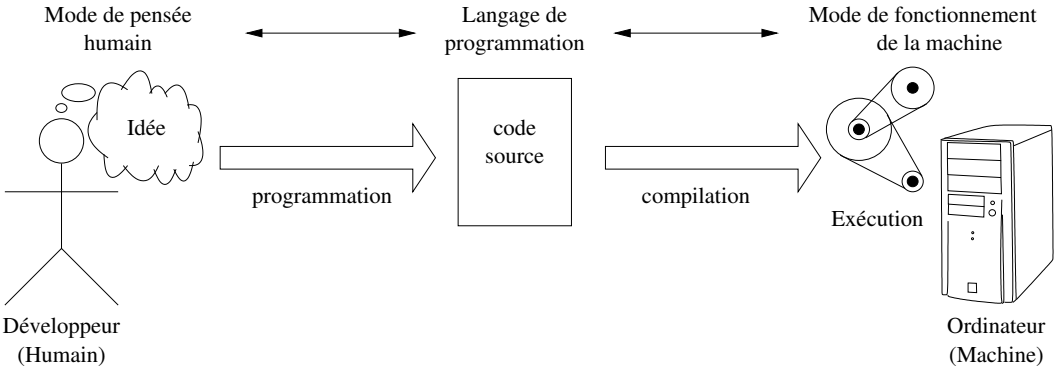


FIG. 1.1: Programmeur, langage et ordinateur

Comme l'illustre la figure 1.1, un langage de programmation est avant tout l'interface entre un développeur (l'humain) et un ordinateur (la machine).

[1]Selon l'encyclopédie des langages de programmation `http://hopl.murdoch.edu.au/`

De l'expressivité à l'efficacité

La principale thèse que défend ce mémoire peut se résumer par le principe suivant :

Principe 1 *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.*

Or, le mode de pensée humain est éloigné du mode de fonctionnement de la machine. Ceci implique que traduire efficacement un bon langage de programmation vers le mode de fonctionnement de la machine est une chose difficile... mais pas impossible.

Ainsi, les travaux que nous avons menés durant ces années de thèse, et qui font l'objet de ce mémoire, abordent tant les problématiques amont (côté humain) que les problématiques aval (côté machine) des langages de programmation. Du côté des humains, nous nous intéressons à l'expressivité des langages de programmation et étudions l'intuitivité des mécanismes objets et la nécessité des modules. Du côté des machines, nous nous intéressons à la compilation efficace des programmes à objets et à modules.

Expressivité : objets, spécialisation et modules

Langages à objets

Les langages à objets (ou langages orientés objet) sont apparus il y a plus de 30 ans — il est généralement admis que Simula [Birtwistle *et al.*, 1973] fut en 1967 le premier langage à objets, et que Smalltalk [Goldberg et Robson, 1983] fut en 1973 le premier langage où le paradigme objet est conceptualisé. Les langages à objets se sont depuis développés et la plupart des langages de programmation utilisés dans les milieux académiques et industriels appartiennent à cette grande famille. Ces langages possèdent quatre caractéristiques :

- l'*objet*, qui encapsule données et traitements ;
- l'*envoi de message*, qui permet à l'objet receveur de décider de son comportement ;
- la *classe*[2], qui regroupe les objets similaires et factorise leurs propriétés communes ;
- les notions duales d'*héritage* et de *spécialisation*, qui permettent à une classe d'hériter les propriétés des classes qu'elle spécialise.

Si le paradigme objet est devenu si populaire, c'est sans doute grâce à deux catégories de raisons : d'une part, grâce aux qualités d'ingénierie logicielle apportées par les quatre caractéristiques que nous venons de citer [Meyer, 1988; Gamma *et al.*, 1995] ; et d'autre part, grâce aux qualités intuitives apportées par l'adéquation entre le modèle à objets et ce que nous appelons l'*approche naturelle*, c'est-à-dire la façon dont l'humain se

[2]Dans cette thèse, nous ne nous intéresserons pas à la famille des langages à prototypes, c'est-à-dire à objets sans classe. Ainsi, dans la suite, quand nous mentionnerons « à objets », il faudra comprendre « à objets et à classes ».

représente le monde. Dans la culture occidentale, on peut au moins faire remonter cette approche naturelle à Aristote et l'illustrer par le syllogisme rituel :

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

Où ici, « Socrate » est un objet tandis que « homme » et « mortel » sont des classes.

Toutefois, les différents langages à objets n'ont pas toujours le même comportement vis-à-vis de la spécialisation de classes, en particulier quand celle-ci est multiple. Dans la suite des travaux de [Ducournau *et al.*, 1995], nous proposons une méta-modélisation générale de la spécialisation qui est entièrement guidée par l'approche naturelle — et donc augmente l'expressivité si l'on adhère au principe 1.

Modularité

L'approche naturelle, pour appréhender un système complexe, consiste à le diviser en plusieurs systèmes simples : c'est le principe de diviser pour régner[3]. Or les programmes informatiques sont des systèmes complexes [Fred P. Brooks, 1995] qui regroupent quantité de problématiques diverses. C'est pourquoi, idéalement, les programmes informatiques sont divisés en *modules*[4] simples, chacun d'eux ne s'occupant que de quelques problématiques [Parnas, 1972]. C'est ce que l'on appelle la *séparation des préoccupations* selon la terminologie introduite par [Dijkstra, 1976].

Toutefois, la question de la nature de ces modules se pose depuis leur introduction et des réponses extraordinairement variées ont été proposées. Dans le contexte des langages à objets, plusieurs points de vue ont été proposés :

- un module est une classe et une classe est un module ;
- un module est un ensemble de classes. On a alors parlé d'espaces de nom, de *packages* et de classes imbriquées ;
- le module est quelque chose d'orthogonal à la classe. On parle alors de nos jours d'aspect[5], d'extension de classes[6], etc.

Dans cette dernière lignée, nous proposons le mécanisme du *raffinement de classes*, une extension du modèle à objets qui permet d'ajouter *a posteriori* de nouvelles propriétés à des classes existantes. Ce mécanisme est couplé avec une notion de *hiérarchie de modules* : *un module peut raffiner les classes importées des modules dont il dépend*. Il est basé sur une analogie structurelle — et seulement structurelle — avec l'héritage de

[3] « Divide ut regnes » — Machiavel, 1532.

[4] En informatique, le terme « module » est utilisé dans de nombreuses acceptions. Ici nous l'utilisons dans sa forme la plus générale : un module est un sous-ensemble d'un programme destiné à remplir des tâches bien définies.

[5] Un aspect est un élément d'un programme ou une fonctionnalité associé à un point de vue.

[6] L'extension de classe consiste à définir une classe de façon incrémentale.

classes et la redéfinition des propriétés : *une classe peut redéfinir les propriétés héritées des classes qu'elle spécialise.*

L'originalité de ce mécanisme est de prendre en compte les spécificités de l'héritage multiple et de les intégrer à deux niveaux : celui des modules, soumis aux dépendances multiples, et celui des classes, soumises aux spécialisations multiples, aux raffinements multiples et aux combinaisons des deux.

Efficacité : compilation séparée et techniques globales

La compilation d'un langage à objets et à modules pose deux contraintes : respecter l'idée de modularité et être efficace malgré les abstractions de haut niveau que propose le langage : héritage multiple et raffinement de classes.

Pour répondre à la première contrainte, le génie logiciel préconise l'utilisation de la compilation séparée, où chaque module est compilé indépendamment de son utilisation finale. Pour répondre à la seconde contrainte, de nombreux travaux ont proposé des techniques d'implémentation globales qui, à partir de la connaissance d'un programme dans son ensemble, permettent de produire du code machine plus efficace.

Afin de répondre à ces deux contraintes contradictoires, nous présentons un schéma de compilation pour les langages à objets qui, de façon paradoxale, est à la fois séparé et intègre des techniques d'implémentation globales. Pour ce faire, le schéma de compilation proposé est divisé en deux phases. La première, locale, compile un module de façon autonome et produit du code exécutable incomplet. La seconde phase, globale, regroupe les modules compilés et produit un exécutable final optimisé par l'application de techniques globales : analyse de types, suppression du code mort, coloration et arbres binaires de sélection.

Ces techniques permettent d'annuler le coût lié à l'héritage multiple et au raffinement de classes et de réduire fortement le coût du polymorphisme inhérent à tout langage à objets. Toute l'originalité d'une telle approche réside dans le fait que l'application des techniques globales a lieu après que les modules ont été compilés.

Le langage PRM et son compilateur `prmc`

Dans le but de donner consistance à nos propositions sur la spécification et la modélisation des langages à objets, nous avons développé un nouveau langage de programmation, PRM[7], qui valide l'approche naturelle de l'héritage multiple et intègre le mécanisme du raffinement de classes.

[7]Acronymme possile : « Programmation Raffinement Modules ».

Afin de valider le schéma de compilation que nous proposons, nous avons également développé `prmc`, un compilateur pour le langage PRM. Bien que pleinement fonctionnel, `prmc` est avant tout un prototype destiné à l'expérimentation et la comparaison de différentes techniques de compilation.

Le langage PRM et le compilateur `prmc` ont déjà montré leur utilité et leur efficacité à trois reprises :

- développement de la bibliothèque de base du langage (17 modules, une centaine de classes et un peu moins de 7 000 lignes de code) ;
- utilisation en tant que langage de support à l'apprentissage de la programmation et de l'algorithmique à l'IUT de Béziers ;
- développement d'un compilateur *autogène*, c'est-à-dire écrit dans le langage qu'il compile, en l'occurrence PRM — ce dernier travail n'est toutefois pas encore arrivé à son terme.

PRM et `prmc` sont tous les deux des projets libres actuellement hébergés par Gna! à l'adresse `https://gna.org/projects/prm/`.

Le plan de la thèse

La première partie concerne notre travail sur la spécification des langages :

- le chapitre 2 est dédié à la présentation du langage PRM, nous y présentons les caractéristiques du langage et leur rapport avec différents critères de qualité ainsi que des comparaisons avec d'autres langages. Ce chapitre constitue ainsi un état de l'art sur la programmation par objets ;
- le chapitre 3 présente la modélisation — ou précisément la méta-modélisation — de la spécialisation de classes et de l'héritage des propriétés que nous proposons. Ce chapitre insiste d'une part sur la sémantique naturelle de la spécialisation et d'autre part sur la problématique de l'héritage multiple et sur la résolution de ses conflits ;
- le chapitre 4 présente notre proposition de raffinement de classes et de hiérarchie de modules. Nous réutilisons et étendons la méta-modélisation du chapitre précédent ;
- le chapitre 5 conclut cette partie et fait la transition avec la partie suivante.

La seconde partie concerne notre travail sur la compilation des langages :

- le chapitre 6 présente un état de l'art des techniques d'implémentation des langages à objets. Nous insistons sur la difficulté de compilation des langages en héritage multiple par rapport à ceux en héritage simple et présentons les diverses solutions existantes ;
- le chapitre 7 présente le schéma de compilation séparé que nous proposons et qui intègre certaines techniques présentées au chapitre précédent. Il est utilisable pour tout langage à objets statiquement typé (même sans véritable notion de module) ;
- le chapitre 8 est dédié à `prmc`, le prototype de compilateur pour le langage PRM.

Nous présentons dans ce chapitre l'architecture du compilateur, la mise en œuvre du schéma et des techniques ainsi que les résultats de différents bancs d'essais.

Enfin, le chapitre 9 conclut ce mémoire et ouvre les perspectives de notre travail.

Les annexes, comme le reste du mémoire, parlent encore de langages :

– l'annexe A, rédigée en anglais, est consacrée à la spécification quasi-complète du langage PRM. Nous y détaillons sa syntaxe et sa sémantique et nous faisons quelques comparaisons avec les mécanismes des autres langages ;
– l'annexe B présente les quelques langages de programmation auxquels nous faisons souvent référence ;
– l'annexe C présente le pseudo-langage d'assemblage utilisé dans les chapitres 6 et 7.

Première partie

Spécification et modélisation des langages à objets

Chapitre 2

Présentation du langage de programmation PRM

Préambule

Ce chapitre présente le langage PRM, et discute de ses caractéristiques et de ses qualités sans toutefois entrer dans les détails syntaxiques ou sémantiques, ceux-ci étant traités aux chapitres suivants et dans l'annexe A.

2.1 Introduction

PRM est un langage de programmation orienté objet et statiquement typé. Il se place dans la même famille que de nombreux autres langages parmi lesquels on peut citer C++, Java et Eiffel.

PRM est avant tout un langage universitaire, résultat de nos recherches en spécification des langages à objets et cible de nos recherches en compilation de ces mêmes langages. Toutefois, PRM n'en est pas moins un langage de programmation complet, utilisable en tant que langage de support pour l'enseignement mais aussi pour réaliser de « vraies » applications.

Avant d'entrer dans le vif du sujet et de présenter une à une les différentes caractéristiques du langage, la section suivante est consacrée à un historique du langage, puis la section 2.3 présente différents critères de qualité des langages de programmation qui nous permettent de discuter des caractéristiques qu'ils offrent. À partir des sections suivantes, nous décrivons les caractéristiques du langage PRM, qui seront plus amplement détaillées dans les chapitres suivants :

– section 2.4, les concepts manipulés sont simples et la syntaxe concise, mais celle-ci reste claire et structurée — l'annexe A entre dans le détail de la syntaxe et des concepts ;

- section 2.5, le langage est tout objet : toute valeur manipulée est l'instance d'une classe, toute routine est la méthode d'une classe — le chapitre 3 traite de la spécification des mécanismes objets sur laquelle est basée le langage PRM ;
- section 2.6, le langage est en typage statique ; il offre également la généricité et une politique de typage covariant — annexe A pour les détails ;
- section 2.7, le langage offre une notion simple de module et permet le raffinement de classes : les classes sont définies dans des modules, les modules peuvent importer les classes d'autres modules, une classe importée peut être raffinée — le chapitre 4 est dédié à la formalisation des modules et du raffinement.

2.2 Pourquoi un nouveau langage de programmation ?

Comme nous l'avons déjà dit, le travail que nous avons effectué au cours de cette thèse porte sur deux problématiques : la spécification des langages à objets et leur compilation efficace. Dans les deux cas, nous avions besoin de disposer d'un vrai langage de programmation afin de valider la faisabilité de nos idées et de vérifier expérimentalement l'intérêt et la qualité de nos propositions.

Que cela soit pour la spécification ou la compilation, nous avons posé quatre contraintes qui ont servi de base à la construction du langage PRM :

- typage statique ;
- héritage multiple ;
- pas de méta-niveau (du moins pas dans le noyau du langage) ;
- langage simple et entièrement objet.

Les trois premières contraintes correspondent au cadre de recherche que nous nous sommes fixés — mais que nous justifions tout de même dans les sections suivantes. La dernière contrainte correspond à une limitation plus pragmatique : puisque notre temps était limité, la réalisation d'un compilateur fonctionnel devait se focaliser sur nos besoins de compilation des mécanismes objets et nécessiter de disperser le moins possible nos efforts sur des problématiques de compilation qui n'entraient pas dans le cadre de notre travail.

La question de l'utilisation d'un langage existant s'est bien sûr posée dès le début de notre travail. En particulier, nous avions commencé à explorer la piste du langage Eiffel puisque ce langage bénéficie d'un compilateur libre SmartEiffel développé au LORIA à Nancy. Malheureusement, comme SmartEiffel est basé sur un principe de compilation globale (cf. section 6.4.6 page 140), le développement était rendu difficile par la nécessité de comprendre et de modifier en profondeur le code source. Ainsi, pour avancer dans nos recherches sur le raffinement et sur la compilation, nous avons commencé à développer en parallèle un prototype de compilateur pour un prototype de

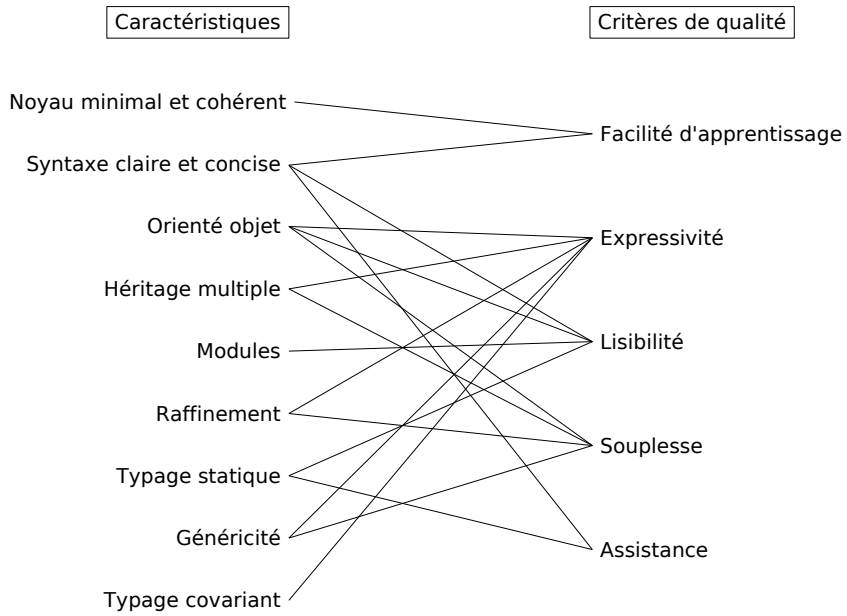


FIG. 2.1: Caractéristiques et critères de qualité dans les langages

langage qui est devenu par la suite le langage PRM.

2.3 Qualité des langages de programmation

En tant qu'outils de la chaîne de production de logiciels, les langages de programmations jouent un rôle important dans la qualité des programmes produits. Afin d'identifier précisément leur influence dans la qualité du code source des logiciels qu'ils servent à écrire, nous avons isolé différents critères de qualité propres aux langages de programmation : la facilité d'apprentissage, l'expressivité, la lisibilité, la souplesse et l'assistance.

La figure 2.1 montre les liens entre les critères de qualité et les caractéristiques des langages de programmation dont nous discutons à partir de la section 2.4.

2.3.1 Facilité d'apprentissage

Ce critère de qualité correspond à la facilité d'un programmeur à maîtriser le langage, sa syntaxe et sa sémantique.

La *facilité d'apprentissage* ne recouvre pas exactement la même chose suivant que le programmeur est déjà familier avec d'autres langages (programmeur expérimenté) ou qu'il s'agit d'un programmeur novice (étudiant débutant).

L'apprentissage de PRM est facile pour les deux catégories de programmeurs. Les retours d'expérience dont nous disposons montrent qu'un programmeur expérimenté n'a aucun mal à maîtriser rapidement le langage (étudiants en master, collègues enseignants). L'utilisation de PRM par des étudiants débutants a montré que des programmeurs novices sont capables rapidement de programmer en PRM (même s'ils ne maîtrisent pas l'ensemble des caractéristiques du langage). La meilleure preuve en est sans doute le succès de son utilisation à l'IUT de Béziers en tant que langage de support pour l'apprentissage de la programmation et de l'algorithmique.

2.3.2 Expressivité

Tous les langages généraux, dont PRM fait partie, sont des langages Turing-complets, c'est-à-dire qu'ils sont capables d'exprimer tous les algorithmes calculables. Toutefois, bien qu'ils soient capables d'exprimer tous les calculs possibles, les langages généraux ne sont pas tous équivalents en terme d'*expressivité*, c'est-à-dire dans leur capacité à permettre au programmeur de s'exprimer succinctement et directement.

Au début de l'histoire de l'informatique et des langages de programmation, l'expressivité se limitait généralement à la capacité d'exprimer simplement des algorithmes — on est alors passé des langages d'assemblage à la programmation structurée avec l'introduction de notions comme les variables ou les structures de contrôle comme `if` ou `while`. Comme les programmes deviennent de plus en plus complexes, les langages modernes doivent également se préoccuper de proposer des abstractions capables de modéliser et de faire vivre des systèmes de plus en plus complexes — ce qui correspond plus ou moins à la percée de la programmation par objets.

2.3.3 Lisibilité

Le concept de *lisibilité* du code source d'un programme informatique renvoie à la difficulté et à la vitesse de lecture de celui-ci par un programmeur — qui peut en être l'auteur mais pas forcément. Bien que ce soit la façon dont le programmeur a écrit le code source qui détermine si celui-ci est lisible ou non (noms de symboles explicites, commentaires, architecture claire, etc.), le rôle des langages des programmation n'est pas si anodin, et certains langages de programmation comme C ou Perl sont réputés pour leur capacité à permettre l'écriture de programmes illisibles[1]. Le critère de qualité des langages que nous appelons lisibilité correspond à l'impact qu'a le langage sur la lisibilité du code source.

La lisibilité dépend de la syntaxe du langage et du rapport entre la syntaxe et la sémantique. Par exemple, l'existence d'écritures syntaxiquement proches mais

[1]The International Obfuscated C Code Contest (`http://www.ioccc.org/`) est un concours annuel qui se tient sur Internet depuis 1984. Les gagnants sont ceux qui produisent les programmes en C les plus illisibles, créatifs et bizarres mais fonctionnels.

sémantiquement distantes réduit la lisibilité des programmes. C'est le cas en C avec le = et le ==.

De même que pour la facilité d'apprentissage, la lisibilité est différente en fonction des programmeurs. Si le programmeur est un expert du langage, la lisibilité se détermine principalement par la facilité et la rapidité qu'a le programmeur à comprendre ce que fait un programme. Si le programmeur est expérimenté mais n'est pas un spécialiste du langage, la lisibilité se détermine également par l'exotisme de la syntaxe du langage.

2.3.4 Souplesse

La *souplesse* regroupe les problématiques d'*évolutivité*, de *maintenabilité*, de *réutilisabilité*, d'*interopérabilité* et de *portabilité*. Elle correspond à la capacité des programmes à supporter des évolutions plus ou moins profondes (correction de bugs, ajout de fonctionnalités, réorganisation architecturale, réécriture de fonctionnalités, etc.) mais aussi à pouvoir s'adapter facilement à des changements de contexte et de plate-forme.

De même que pour la lisibilité, la souplesse dépend de la façon dont le code source a été écrit et de l'architecture des programmes. Mais, également, chaque langage de programmation peut plus ou moins faciliter l'écriture de programmes souples.

En règle générale, les langages souples sont ceux qui permettent d'écrire de façon générique et ceux qui sont indépendants des machines et des architectures.

2.3.5 Assistance

L'*assistance* est la capacité d'un langage de programmation à éviter au programmeur d'écrire des erreurs. Celle-ci peut se manifester à de nombreux niveaux : dès l'écriture du code, lors de la compilation, lors de la phase de test, etc. L'assistance a pour objectif d'aider le programmeur à produire plus rapidement des programmes conformes aux spécifications (que celles-ci soient un cahier des charges rigoureux ou non) et plus fiables.

Un langage de programmation peut avoir une influence directe sur l'assistance, par exemple en ayant une syntaxe rigoureuse qui évite les écritures erronées.

L'assistance peut également être mise en œuvre par l'intermédiaire de divers programmes : éditeurs de textes, compilateurs, outils de tests, etc. L'influence des langages de programmation est alors indirecte, puisque chaque langage se prête plus ou moins facilement et efficacement à ces outils.

2.3.6 Agrément

Ce dernier critère de qualité signifie simplement que le langage de programmation doit être agréable à utiliser. Ce critère de qualité est sans doute le moins objectif de tous puisque chaque programmeur est différent et, par définition, son appréhension de chaque langage de programmation lui est personnelle. Toutefois, en identifiant dans un

langage de programmation les causes de frustration du programmeur, il est possible de présumer si celui-ci est agréable à utiliser ou non.

Parmi les causes de frustration facilement identifiables, on retrouve en première position les autres qualités des langages de programmation. En effet, un langage est frustrant si celui-ci est difficile à apprendre, limite l'expressivité, permet trop facilement l'écriture de code illisible (en particulier pour la personne chargée d'en faire la maintenance), ne permet pas l'écriture de code souple (puisqu'il oblige le programmeur à faire de la réécriture) ou assiste peu le programmeur (qui perd du temps à chercher et corriger des erreurs triviales).

2.4 Des concepts simples pour une syntaxe claire

Comme le langage PRM est un langage neuf parti de rien, nous avons pu respecter deux points importants selon nous :

- minimalité des concepts : PRM ne doit contenir qu'un petit nombre de concepts significatifs ;
- syntaxe simple et concise : PRM doit être concis et rigoureux sans être verbeux.

2.4.1 Concepts simples et cohérents

PRM est bâti sur un langage minimal à objets (définition de classe, définition d'attribut et de méthode, envoi de message et construction d'instance), agrémenté du minimum de la programmation structurée (variables locales, boucles et structures conditionnelles). Autour de ce langage minimal se sont greffées quelques extensions : classes paramétrées, visibilité (protection), modules, raffinement, etc.

Outre le fait de faciliter le développement d'un compilateur, un langage simple et cohérent facilite l'apprentissage puisque d'une part, il y a peu de concepts à apprendre et d'autre part, il y a peu de cas irréguliers à connaître par cœur.

Le dernier point à ne pas négliger en faveur de la spécification de concepts simples est celui de la perfection :

Principe 2 (Perfection [Saint-Exupéry, 1939]) *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.*

Remarque : En cas de conflit entre le principe 1 (page 2) et le principe 2, c'est le premier qui doit l'emporter. En effet, non seulement le contraire risquerait de signifier que le langage de programmation vide est potentiellement le meilleur langage de programmation mais en plus, cela encouragerait les langages qui persistent à dire que `1+2*3 = 9` !

La compilation ne doit pas guider la spécification

Lorsque le compilateur est développé en même temps que la spécification du langage, la tentation est grande de déroger au principe suivant :

Principe 3 *Les problématiques d'implémentation ne doivent pas corrompre la spécification d'un langage de programmation.*

Ce principe n'est qu'un corollaire du principe 1 de la page 2.

Nous pensons que de nombreux défauts des langages de programmation existant viennent d'infractions à ce principe. Celles-ci peuvent se manifester de plusieurs façons :

- la spécification n'introduit pas une caractéristique parce qu'elle est difficile à implémenter. Une telle façon de faire réduit de façon évidente l'expressivité, la souplesse ou l'assistance ;
- la spécification introduit une caractéristique dégradée à la place d'une plus difficile à implémenter. L'impact sur l'expressivité, la souplesse ou l'assistance existe toujours mais celui-ci est minimisé. En contrepartie, le langage devient moins cohérent et plus difficile à apprendre puisqu'une caractéristique dégradée s'accompagne généralement de cas irréguliers. On peut sans doute illustrer ce cas par Java, l'héritage simple de classes et l'héritage multiple d'interfaces ;
- la spécification introduit à la fois la caractéristique difficile à implémenter et celle dégradée plus facile à implémenter. L'impact sur l'expressivité, la souplesse et l'assistance est alors potentiellement nul (puisque la caractéristique non dégradée existe dans le langage). Malheureusement, la cohabitation de deux caractéristiques proches ne simplifie pas l'apprentissage du langage surtout si celles-ci ont des points de conflits. On peut aisément illustrer ce cas avec le mot clé `virtual` de C++ qui est utilisé pour différencier d'une part les méthodes qui peuvent être soumises à une liaison tardive et celles qui ne peuvent pas l'être et d'autre part deux mécanismes d'héritage qui, en fonction des combinaisons, peuvent produire un héritage répété ou non — nous revenons sur ce point dans la section 3.3.2 page 50 ;
- la spécification introduit une caractéristique uniquement parce qu'elle est facile à implémenter. Or l'ajout inconsidéré de fonctionnalités va à l'encontre du principe 2, rend le langage moins simple et augmente le risque d'incohérence puisque les combinaisons des différentes caractéristiques deviennent plus nombreuses.

Des propriétés retardées

Un autre moyen que nous avons mis en œuvre pour conserver un noyau cohérent a été de ne pas inclure de caractéristiques dans le langage tant que nous n'avions pas trouvé de façons élégantes de les spécifier.

Ainsi, certaines caractéristiques parfois largement répandues dans les autres langages de programmation n'ont toujours pas été intégrées à PRM (c'est le cas des énumérations

et des constantes) ou sont intégrées avec une spécification temporaire car celle-ci n'est pas satisfaisante (c'est le cas des variables statiques).

2.4.2 Syntaxe concise mais structurée

Syntaxiquement, PRM est issu de deux familles de langages : celles des langages de scripts et celles des langages structurés (à la PASCAL). Tout particulièrement, PRM s'inspire fortement de deux langages orientés objets, RUBY (pour la première famille) et de EIFFEL (pour la seconde), qui malgré leurs divergences profondes, partagent une même idée : un langage à objets, puissant, simple, cohérent et facile à apprendre et à utiliser.

Concision des langages de scripts

Les langages de la première famille (PERL, PYTHON, RUBY, PHP, etc.) sont généralement réputés pour leur concision et leur souplesse, ainsi que pour leur facilité d'apprentissage et de développement rapide qui laisse la part belle au prototypage et à l'*extreme programming*.

RUBY [Thomas et Hunt, 2000] est un langage de script développé dès 1993 par Yukihiro Matsumoto avec un objectif premier de cohérence et d'intelligibilité. Comme de nombreux langages de script, il permet entre autres de manipuler facilement les fichiers textes (les expressions rationnelles sont natives), de faciliter la programmation web (gestion avancée des CGI, Web2.0 avec RUBY On Rails[2]) et d'aider à effectuer des tâches d'administration système. Contrairement à la majorité des autres langages de script, RUBY est un langage pleinement orienté objet.

Ainsi, RUBY peut passer pour un mélange de SMALLTALK et de PERL puisqu'il possède à la fois une syntaxe dont la concision est inspirée de celle de PERL et la rigueur du modèle à objets de SMALLTALK. Parmi ces fonctionnalités, on peut citer : toute valeur est un objet, ramasse-miettes, exceptions, définition incrémentale, fermetures et méta-niveau.

Toutefois, comme SMALLTALK, RUBY ne gère pas l'héritage multiple mais ce manque est compensé par un principe de *mixin* (cf. section 3.3.2 page 51).

Rigueur des langages structurés

Les langages de la seconde famille (PASCAL, EIFFEL, ADA, MODULA, etc.) sont caractérisés par une syntaxe claire et rigoureuse qui facilite la structuration des programmes.

[2] `http://www.rubyonrails.org/`

Pascal [Wirth, 2002], le patriarche de la famille, fut créé par Niklaus Wirth en 1970 à l'université de Zurich. Il a été conçu pour servir à l'enseignement de la programmation de manière rigoureuse mais simple.

Plus proche de nous, le langage de programmation Eiffel [Meyer, 1997] fut créé en 1985 par Bertrand Meyer et développé par son entreprise, Interactive Software Engineering (ISE) (Goleta, Californie, USA).

Le nom Eiffel est un hommage à Gustave Eiffel, ingénieur qui conçut la tour éponyme en respectant les délais et le coût prévu, et dont la structure est constituée de nombreux éléments modulaires. Pour Bertrand Meyer, aucun des langages existants n'était totalement satisfaisant du point de vue du génie logiciel. Ainsi Eiffel n'est pas une surcouche ou une variation d'un langage existant : d'une part, il propose une approche complète d'ingénierie objet [Meyer, 1988] ; d'autre part, il intègre un modèle de contrats [Meyer, 1991] — *Design by Contract* — qui vise à documenter et garantir les spécifications par la définition explicite dans le code source de préconditions, postconditions et invariants.

Depuis sa création et sa présentation, le langage n'a cessé d'évoluer. La définition du langage est dans le domaine public où elle est contrôlée par NICE, le *Non-profit International Consortium for Eiffel* fondé en 1991.

Concilier les deux mondes

Au final, la syntaxe de PRM est résolument contemporaine et réutilise les mots clés et structures syntaxiques répandus dans les autres langages — à l'exception de ceux du C et de langages dérivés comme C++, Java, C# parfois incompatibles avec une approche à la Pascal. Toutefois, il n'a pas été simple d'arriver à concilier les qualités des langages souples, celle des langages rigoureux et la minimalité des concepts de PRM.

La première solution consiste à relâcher les contraintes de structure du code lorsque celles-ci sont superflues. Ainsi, le programme le plus simple que l'on puisse écrire en PRM est le programme qui ne fait rien et qui n'est constitué que d'un fichier vide. De la même manière, le fameux *Hello World* s'écrit en PRM par un programme d'une seule ligne :

```
println("Hello World")
```

Dans les deux exemples (le fichier vide et le *Hello World*), bien que le langage soit pleinement orienté objet, la définition explicite de classes et de méthodes n'est pas nécessaire.

La seconde solution que nous avons utilisée consiste à faire largement appel à du sucre syntaxique, c'est-à-dire à la définition d'écritures syntaxiques strictement équivalentes à des écritures plus complexes. L'avantage du sucre syntaxique est qu'il ne touche pas au noyau du langage, celui-ci restant minimal, tout en apportant une légèreté d'écriture

pour de nombreux cas parmi lesquels on peut citer les accesseurs automatiques, les boucles `for in`, les paramètres par défaut, etc.

Que cela soit par le relâchement des contraintes de structure ou par l'utilisation de sucre syntaxique, l'avantage donné au programmeur est de pouvoir se libérer de certaines caractéristiques avancées du langage. Ceci est particulièrement utile pour l'enseignement.

Petit aperçu syntaxique en PRM

Concrètement, un petit programme en PRM qui ne définit explicitement aucune classe ressemble à :

```
# fichier sum.prm

def sum_array(array: Array[Int]): Int
# La somme des elements du tableau 'array'
do
        let sum := 0    # resultat
        for a in array do
                # on somme
                sum := sum + a
        end
        return sum
end

let an_array := [5, 10, 4, 3] # Un tableau litteral d'entiers
print("La somme des elements est ", sum_array(an_array))
```

Ce programme, qui définit et utilise une fonction qui calcule la somme des éléments d'un tableau d'entiers, permet de remarquer quelques caractéristiques syntaxiques du langage, ses similitudes avec les langages Ruby et Eiffel et les nombreuses différences syntaxiques avec C++ et Java :

- comme de nombreux langages de script, PRM utilise le *symbole numérique* (appelé improprement « dièse ») comme marqueur des commentaires — actuellement, il n'existe pas de commentaire multi-lignes ;
- comme dans la majorité des langages modernes, les instructions n'ont pas à être terminées par un point-virgule — de même, lors de l'invocation des méthodes d'arité nulle, l'utilisation de () est superflue ;
- à l'inverse des langages provenant du C, PRM préfère une syntaxe pascalienne : les définitions des variables et des paramètres sont de la forme `identifier: Type` au lieu de `Type identifier` et les blocs sont délimités par des mots clés (`do end`) au lieu d'accolades ;

- l'opérateur d'affectation est le := et est distinct de ceux de comparaison — qui sont = pour la comparaison de valeur (`equals` en JAVA et `is_equal` en EIFFEL) et == pour la comparaison d'identité d'objets ;
- la syntaxe de PRM est régulière dans le sens ou toute déclaration commence par un mot clé (**def** pour les attributs et méthodes, **let** pour les variables locales, etc.) ;
- les variables locales peuvent être déclarées n'importe où dans un bloc d'instructions ; de plus le type statique n'est pas explicitement requis si la variable possède une valeur initiale — le type statique de la variable est alors le même que celui de la valeur initiale ;
- il n'y a pas de syntaxe spéciale pour les types tableaux : un tableau d'éléments `T` est déclaré par `Array[T]` où `Array` est une classe paramétrée (cf. section 2.6.4) ;
- les chaînes littérales sont écrites entre guillemets, les tableaux littéraux sont écrits entre crochets ;
- PRM hérite la plupart des structures de contrôle (`if`, `while`) des langages procéduraux ou à objets traditionnels, à l'exception du **for** qui correspond au `foreach` de PERL plutôt qu'au `for` du BASIC ou à celui du C.

2.5 Paradigme orienté objet

Créés il y a plus de vingt ans, les *langages à objets* (ou *langages orientés objet*) sont devenus la norme dans les processus de développement des logiciels. La raison d'un tel succès est principalement liée à quatre caractéristiques qui apportent expressivité, lisibilité et souplesse : l'objet, la classe, l'héritage et l'envoi de message.

2.5.1 Les quatre caractéristiques de l'approche par objet

Le concept d'*objet* permet au programmeur de désigner et de manipuler de façon directe et naturelle des abstractions d'entités du monde réel. Contrairement aux langages de programmation traditionnels qui distinguent données et traitements, l'approche par objets met en avant une seule notion, l'objet, qui encapsule les données (appelées généralement attributs ou variables d'instances) et les traitements (appelés généralement méthodes). Une autre caractéristique des objets est la notion d'*identité* par opposition à la notion de *valeur* : un objet ne se réduit pas à une suite de bits.

Le concept de *classe* permet au programmeur de définir des catégories d'objets. Les classes apportent aux langages à objets la structuration nécessaire dès que l'on a de nombreux objets. Toutefois le concept de classe n'est pas strictement nécessaire à la programmation par objets : il existe des langages à objets sans classes, c'est la famille des langages dits de *prototypes* ou de *frames* tels que SELF [Ungar et Smith, 1987], ECMASCRIPT [ECMA, 1999] (c'est-à-dire JAVASCRIPT) ou plus récemment LISAAC [Sonntag et Boutet, 2004]. La structuration apportée par les classes permet d'améliorer les qualités

de lisibilité et se caractérise par le regroupement des objets de même nature — les objets d'une classe sont appelés ses *instances* — et factorise les définitions des attributs et des méthodes de ses instances.

Le mécanisme d'*héritage* permet au programmeur de définir (c'est-à-dire d'exprimer) de nouvelles classes par rapport à des classes déjà définies. L'intérêt est double :

- d'une part structurer les classes entre elles par la construction d'une hiérarchie de classes qui reflète une notion de généralisation/spécialisation entre les classes ;
- d'autre part factoriser les propriétés communes à plusieurs classes : les classes héritent les attributs et les méthodes de leurs super-classes — c'est-à-dire des classes qu'elles spécialisent.

La métaphore de l'*envoi de message* (appelée aussi *liaison tardive*) permet d'exprimer de façon générale des manipulations sur les objets sans se soucier des classes dont ils sont instances. Il remplace l'appel de procédure et de fonction des langages procéduraux. Lors de l'exécution, chaque envoi de message est résolu en fonction de l'objet receveur et du message qui lui est envoyé. Dans les langages à classes, c'est le type dynamique, c'est-à-dire la classe de l'objet receveur, qui détermine la réponse à donner pour un message particulier. Avec la notion d'objet, la notion d'envoi de message est primordiale dans l'approche à objets.

Comme nous l'avons déjà évoqué, PRM est un pur langage à objets ce qui signifie que :

- toute donnée manipulée est un objet ;
- tout objet est l'instance d'une classe ;
- tout sous-programme est une méthode d'une classe ;
- toute invocation de méthode se fait par un envoi de message.

2.5.2 Héritage multiple

L'héritage multiple est une caractéristique de certains langages à objets qui se caractérise par le fait de permettre au programmeur d'exprimer plus d'une super-classe lors de la définition d'une nouvelle classe. De fait, l'héritage multiple augmente l'expressivité du langage :

- il permet une meilleure structuration. Par exemple, la classe `Poulet` spécialise les classes `Oiseau` et `Animal de basse-cour` ;
- il améliore la factorisation — une classe peut vouloir hériter des propriétés de classes indépendantes — ce qui limite la redondance nuisible à la maintenance.

Toutefois, l'héritage multiple a provoqué et provoque toujours une réaction timide : on trouve cela « dangereux », « risqué », voire « preuve d'une mauvaise modélisation ». Le fait n'est pas nouveau ; en 1995, Bertrand Meyer faisait le constat suivant : « one still encounters people who have been told that multiple inheritance is tricky or dangerous » [Meyer, 1995].

Les langages récents n'intègrent même plus la spécialisation multiple de façon pleine et entière et se contentent au mieux d'une forme plus ou moins dégradée comme l'héritage d'interfaces ou l'héritage de *mixins* (cf. section 3.3.2 page 51). Certains langages semblent même abandonner l'idée d'une spécification simple de la spécialisation, c'est le cas d'Eiffel qui intègre désormais une notion d'héritage non-conforme (cf. section 3.3.2 page 50). Pourquoi en est-on arrivé là ?

Il nous semble, mais c'est une hypothèse que nous avançons avec prudence, que cela vient d'une complexification inutile de la notion générale de spécialisation : pour la plupart des programmeurs (que cela soit ceux qui utilisent ou ceux qui développent un langage de programmation), la spécialisation relève à la fois d'une astuce d'implémentation et d'un concept de modélisation. Ce qui est plutôt antinomique. En héritage multiple, cette complexité amène généralement la plupart des langages à mal poser la question des conflits d'héritage multiple et donc à y apporter des réponses peu satisfaisantes.

Le chapitre 3 est entièrement dédié aux problématiques d'héritage et tout particulièrement d'héritage multiple. Dans la suite des travaux de [Ducournau *et al.*, 1995], nous y présentons l'approche naturelle de la spécialisation sur laquelle est basée la spécification de PRM. L'héritage multiple en devient plus simple (à comprendre et à utiliser) et les divers conflits d'héritage peuvent alors être gérés de façon plus naturelle.

Une autre hypothèse sur le peu d'engouement que génère l'héritage multiple est qu'il vient sans doute de problèmes d'implémentation : l'héritage multiple est plus difficile à implémenter efficacement que l'héritage simple (cf. section 6.3 page 121). Pour pallier cette difficulté, certains langages spécifient alors plusieurs sortes d'héritages. C'est le cas de C++ qui distingue un héritage virtuel et un héritage non virtuel, ce qui ne simplifie pas la compréhension des mécanismes. Or, par respect du principe 3, ce n'est pas la voie que nous avons choisie pour PRM. Toutefois, comme nous le verrons dans la seconde partie de ce mémoire, même si la spécification de PRM ne fait pas de concession à l'implémentation et aux problématiques de compilation, le surcoût habituellement lié à l'héritage multiple est entièrement annulé par le compilateur `prmc`.

2.5.3 Petit aperçu syntaxique en PRM

La syntaxe de PRM associée aux mécanismes objets est résolument traditionnelle.

Le mot clé `class` permet de définir des classes :

```
class Voiture
end
```

Le mot clé `inherit` permet de déclarer des super-classes :

```
class Ambulance
        inherit Voiture
end
class VoitureToutTerrain
```

```
        inherit Voiture
end
class AmbulanceToutTerrain
        inherit Ambulance
        inherit VoitureToutTerrain
end
```

Remarque : Implicitement, toute classe spécialise la classe `Any`, la racine de la hiérarchie de classe.

Le mot clé `def` permet de définir des attributs et des méthodes :

```
class Personne
        def @nom: String

        def presente_toi
        do
                print("Bonjour, je m'appelle ", @nom, ".\n")
        end
end
```

Les attributs sont différenciés par un arobase « `@` » en début de nom que l'on peut prononcer « at » comme « at-tribut », ils possèdent également un type statique. Les méthodes possèdent une signature (éventuellement vide dans le cas des procédures sans paramètre) et un corps.

L'instanciation se fait avec le mot clé `new`, l'envoi de message utilise la notation pointée :

```
let s := new String
print(s.length) # Affiche 0
```

Remarque : Les problèmes de constructeurs et le cas de la méthode `print` sont examinés en détail dans l'annexe A.

2.6 Typage statique

Le typage statique est une technique utilisée dans certains langages de programmation pour annoter le code source en attribuant un type statique à chaque entité déclarée — une variable, un paramètre de fonction, un attribut, etc. L'ensemble des valeurs que peut prendre une entité est restreint par le type statique associé.

Vu que le typage statique a pour rôle de contraindre le champ des valeurs possibles associées aux entités, il restreint l'expressivité du langage. Toutefois, il offre généralement en contrepartie quatre avantages : sûreté, abstraction, documentation et efficacité. De plus, la généricité et le typage covariant permettent de rendre le typage statique moins restrictif au niveau de l'expressivité.

Comme le typage statique offre à la fois des avantages et des inconvénients, le débat entre les partisans des langages à typage dynamique et ceux à typage statique risque de durer encore longtemps. Toutefois, nous pensons que les avantages du typage statique sont plus nombreux que ses inconvénients, surtout dans le cadre des langages à objets :

Principe 4 (Typage statique [Meyer, 1995]) *Serious use of object technology requires static typing.*

2.6.1 Sûreté

Les outils liés à un langage de programmation, et tout particulièrement les compilateurs, peuvent utiliser les annotations de types du code source pour détecter de nombreuses erreurs bien avant l'exécution des programmes.

Définition 1 (Typage sûr) *Le typage d'un langage de programmation est sûr lorsque toutes les erreurs de types des programmes peuvent être détectées avant leur exécution.*

Remarque : Cette définition diffère d'une notion plus ancienne du typage sûr qui correspondait plutôt au fait qu'à l'exécution aucune opération ne sera appliquée à une donnée d'un mauvais type. La différence peut sembler subtile, mais est fondamentale. En effet, selon cette ancienne définition, le typage est sûr si pour toute opération invalide tentée à l'exécution, le programme s'arrête proprement ou signale une exception. Le typage est non sûr lorsque à l'application d'une opération invalide, le programme se termine de façon abrupte ou continue comme si de rien n'était. Contrairement à notre définition, cette ancienne notion peut s'appliquer indépendamment aux langages à typage statique et dynamique.

Pour les programmeurs distraits ou peu rigoureux, la plupart des erreurs de types sont des erreurs d'inattention et vont des mauvais arguments passés en paramètres de fonctions aux affectations de la mauvaise valeur à la mauvaise variable en passant par la tentative de diviser une chaîne de caractères par 3 — du moins dans les langages qui considèrent que la division d'une chaîne de caractères par un entier naturel n'est pas une opération pertinente.

Du point de vue des langages à objets, la sûreté due au typage statique s'applique principalement à l'envoi de message. Ainsi, soit un envoi de message `x.toto(args)` où `x` est une expression, `toto` le nom d'une propriété et `args` les arguments. Le typage statique permet de s'assurer que d'une part il existe une propriété `toto` connue de tout objet possible de `x`, et que d'autre part `args` sont des arguments acceptables pour cette propriété.

2.6.2 Documentation et abstraction

Par le principe même de l'annotation de types, le typage statique impose une documentation automatique du code source puisque celui-ci illustre l'intention du program-

meur. Dans le cas des méthodes, le typage statique (la signature) précise également des conditions nécessaires (mais pas forcément suffisantes) aux utilisations correctes de celles-ci. Dans le cas des variables et des attributs, leurs types statiques documentent de quelles manières ils doivent être utilisés.

Le typage statique force également le programmeur à expliciter les abstractions manipulées. C'est particulièrement le cas dans les langages à objets dans lesquels la notion de type se confond généralement avec celle de classe. Certaines classes sont alors créées pour servir de type, c'est-à-dire d'abstraction — c'est le cas de nombreuses classes abstraites ou des interfaces au sens JAVA. Dans ces langages à objets, le besoin d'abstraction nécessite l'héritage multiple — même si celui-ci n'est disponible que sous une forme dégradée.

Le typage statique permet également de différencier explicitement des propriétés homonymes mais qui correspondent à des abstractions distinctes[3]. Par exemple la propriété `hauteur` de la classe `Personne` et la propriété locale `hauteur` de la classe `Triangle` correspondent à des abstractions différentes : c'est pas la « même » hauteur. Un envoi de message, `x.hauteur` correspond à la première (resp. à la seconde) si le type statique de `x` est sous-type de `Personne` (resp. de `Triangle`).

2.6.3 Efficacité

Bien que cela ne respecte pas les principes 1 (page 2) et 3 (page 15), l'efficacité est parfois considérée comme la première qualité associée au typage statique. En effet, en restreignant le champ des valeurs possibles, le typage statique fournit des informations utiles au compilateur. Celui-ci peut alors utiliser ces informations pour produire du code plus efficace (temporellement mais aussi spatialement). En particulier, la sûreté apportée par un système de types permet de se passer des vérifications de types lors de l'exécution.

Toutefois, il ne faut pas en déduire que seuls les langages statiquement typés sont efficacement compilés. En effet, le typage n'est pour le compilateur qu'une source comme une autre d'information de types. Comme nous le verrons dans la seconde partie de ce mémoire, de nombreuses techniques permettent d'obtenir des exécutables efficaces sans avoir à se baser sur le typage statique — parce que le langage est en typage dynamique voire parce que le typage statique ne fournit pas une information suffisamment précise.

Remarque : Le fait que l'efficacité d'un programme puisse être liée aux types statiques utilisés dans son code source peut avoir des effets pervers : un programmeur peut utiliser un type statique particulier là où un autre type statique aurait été plus pertinent (meilleure abstraction et meilleure documentation) uniquement parce que le compilateur utilisé par ce programmeur produit un exécutable plus performant.

[3]Toutefois, nous voyons dans le chapitre 3 que la plupart des langages à objets en typage statique ne profitent pas complètement de cette particularité apportée par le typage statique.

2.6.4 Généricité

La *généricité*, ou *polymorphisme paramétrique*, consiste en la possibilité d'utiliser des types pour paramétrer d'autres types. Ainsi, si `List[T: Any]` est le type générique des listes, où `T` représente un type formel borné par la classe `Any`, `List[Integer]` et `List[Canard]` correspondent aux types des listes d'entiers et des listes de canards. Du point de vue de la qualité des langages de programmation, la généricité — surtout quand elle est bornée — est un facteur d'expressivité, de lisibilité et de sûreté.

La généricité fut introduite dans ML (1973) et popularisée dans les milieux industriels par ADA (1983). Certains langages à objets ont introduit très tôt la généricité : EIFFEL, BETA, C++. D'autres langages n'ont introduit la généricité que très récemment : JAVA 5.0, VB.NET 2005 ou C# 2.0.

La généricité telle qu'elle est spécifiée en PRM reprend, dans les grandes lignes, l'approche de EIFFEL.

2.6.5 Typage covariant

Le typage sûr impose des règles strictes de *substituabilité*, c'est-à-dire de sous-typage. Dans les langages à objets, ces règles se traduisent par la *contravariance* des types des paramètres des méthodes, la *covariance* du type du résultat des fonctions et l'*invariance* des types des attributs mutables. On dit qu'un langage qui impose ces règles au programmeur suit une politique de typage *contravariant*.

Ainsi, soit le listing suivant :

```
class A
        def m(t: T) ...
...
end
class B
        inherit A
        def m(t: U) ...
...
end
```

où une classe `A` munie d'une méthode `m` prenant un paramètre de type `T` et `B` une sous-classe de `A` redéfinissant `m` par une méthode prenant un paramètre de type `U`. La règle de contravariance impose que `T` soit un sous-type de `U`. En effet, le code suivant est statiquement correct au niveau des types et ne provoquera aucune erreur de type :

```
let x: A
let y: T
x := new B
x.m(y)
```

Toutefois, le typage sûr est en contradiction avec l'expressivité et la sémantique naturelle de la spécialisation [Ducournau, 2002a]. Ainsi, le programmeur définissant une classe `Animal` munie d'une méthode `mange` prenant un paramètre de type `Nourriture` et créant une sous-classe `Vache` voudra sans doute redéfinir `mange` par une méthode prenant un paramètre de type `Herbe` :

```
class Animal
        def mange(n: Nourriture) ...
...
end
class Vache
        inherit Animal
        def mange(n: Herbe) ...
...
end
```

Ce qui est incompatible avec une politique de typage contravariant. Une *politique de typage covariant* permet une meilleure expressivité mais le typage n'est alors plus sûr puisqu'il serait possible d'écrire :

```
let v: Animal
let s: Nourriture
v := new Vache
s := new Saucisse
v.mange(s)
```

La plupart des langages à objets statiquement typés ont opté pour une politique de typage contravariant[4] — avec un bémol pour JAVA dont les tableaux sont covariants et qui dans les versions précédentes à 5.0 n'autorisait pas la redéfinition covariante du type de retour des fonctions (ce qui était un comportement plus strict que ne l'imposait la théorie). EIFFEL est l'un des rares langages à objets qui suit une politique de typage covariant.

La contradiction entre l'expressivité (typage covariant) et le typage sûr (typage contravariant) est insoluble. Toutefois, dans les deux cas, le risque d'erreur est inévitable [Ducournau, 2001b; Ducournau, 2002a] : dans le premier cas, l'erreur de type est envisageable ; dans le second cas, il existe toujours le risque de nourrir les vaches avec des saucisses, ce qui, d'une certaine façon est également une erreur de type qui devra être détectée à l'exécution avec un test de sous-typage — sans doute sous la forme, au début de la méthode `mange` de la classe `Vache`, d'une coercition vers le type `Herbe` du paramètre de la méthode.

[4]La raison de ce choix est sans doute liée au mythe du typage sûr. Toutefois, il est également possible que les difficultés d'implémentation des politiques de typage non sûr y soient également pour quelque chose (cf. sections 6.2.3 page 118 et 6.3.3 page 127).

Comme les deux politiques de typage n'empêchent pas les erreurs de types, nous considérons que la meilleure politique de typage est celle qui limite le moins possible l'expressivité. C'est pourquoi PRM intègre une politique de typage covariant.

2.7 Modules et raffinement de classes

Le terme de *modules* dans les langages a fait et fait toujours couler beaucoup d'encre. Dans les langages de programmation et dans les divers travaux de recherche, de nombreuses entités jouent plus ou moins un rôle de modules. Ainsi outre le terme *module*, on parle également de *package*, d'*espace de nom*, d'*aspect*, de *composant* voire de *classe*. Ainsi, bien que le concept de module corresponde à des acceptions diverses, on retrouve systématiquement deux notions clés :

- un programme est décomposé en modules, chaque module correspondant à une préoccupation ;
- un module est une capsule contenant des définitions d'*éléments* associés à sa préoccupation — les éléments étant, en fonction du langage considéré, des variables, des procédures, des types, des classes, des méthodes, voire d'autres modules — on parle alors de modules imbriqués.

En toute généralité, quels sont les avantages apportés par les modules en terme d'ingénierie des logiciels ?

- un programme bien structuré en morceaux simples à appréhender séparément facilite la spécification, le développement et la maintenance des logiciels ;
- en se concentrant sur des préoccupations unitaires, les modules sont de bons candidats en tant qu'unité de réutilisation. De façon idéale, la création d'un nouveau logiciel devrait pouvoir être faite par la réutilisation de modules existants et la création d'un nouveau module[5], de préférence le plus simple possible ;
- la mise en place de *lignes de produits* à partir d'un logiciel correspond simplement à une variation sur ses modules (ajout, suppression et remplacement de modules) ;
- de façon idéale, l'ajout d'une nouvelle fonctionnalité à un programme existant ne devrait nécessiter que l'ajout d'un nouveau module sans avoir à modifier les modules existants.

Il ressort de ces avantages que les modules sont avant tout une façon de structurer le code source d'un programme. Toutefois, au niveau de la spécification d'un langage, cette structuration peut n'être associée à aucune sémantique particulière, c'est par exemple le cas en SMALLTALK avec les *catégories de classes*, et ne jouer alors qu'un rôle similaire à celui des commentaires. Toutefois, la notion de capsule apportée par les modules peut amener à s'interroger sur la frontière entre ce qui est dans le module et ce qui est hors du module, c'est-à-dire dans les autres modules : on parle alors de *visibilité*, d'*espace de nom*, d'*exportation* et de *protection*.

[5]Il s'agit du module dédié à la préoccupation de combiner les préoccupations des autres modules.

	Dépendance hiérarchique	Espace de nom	Protection	À plat	Unité de code
Packages JAVA	non	oui	oui	non	non
Classes JAVA	non	oui	oui	non	oui
Namespace C++	non	oui	non	non	non[a]
Classes C++	non	oui	oui	non	non[b]
Classes EIFFEL	non	non	oui	oui	oui
Clusters EIFFEL	non	oui[c]	oui	oui	oui[d]
Packages ADA	oui	oui	oui	oui	oui
Modules MODULA 3	oui	oui	oui	oui	oui
Catégories SMALLTALK [e]	non	non	non	oui	non
PRM	oui	oui	non	oui	oui

[a] Toutefois, il est conseillé en C++ d'associer un espace de nom à un fichier ou à un répertoire.

[b] Toutefois, il est conseillé en C++ d'associer une classe à un fichier.

[c] Bien qu'il n'y ait pas de notion de désignation explicite, les clusters d'EIFFEL sont considérés comme des espaces de noms de classe, le langage ACE permettant de renommer les éventuelles classes homonymes afin d'éviter les conflits.

[d] Toutefois, l'unité n'est pas le fichier mais le répertoire.

[e] En SMALLTALK, les catégories de classes n'ont qu'un rôle superficiel de rangement et ne possèdent aucune sémantique particulière au niveau du langage.

TAB. 2.1: Comparaison de quelques systèmes de modules

Dans les langages à objets, une idée souvent proposée consiste à dire qu'une classe est une entité prédestinée à jouer un rôle de module. Nous expliquerons pourquoi nous n'adhérons pas à cette idée dans la section 2.7.2.

2.7.1 Comparaison des différents systèmes de modules

Notre objectif n'est pas ici de comparer dans le détail chacune des propositions ni de proposer un formalisme unifiant les différentes approches et sémantiques des systèmes de modules — de tels comparaisons et formalismes sont disponibles dans de nombreux travaux [Bracha et Lindstrom, 1992; Ancona et Zucca, 1999; Bono *et al.*, 1999; Leroy, 2000; Bergel *et al.*, 2005].

Nous nous contenterons du tableau 2.1 qui résume les différences entre les notions de modules proposés par quelques langages en fonction des grandes caractéristiques suivantes : *dépendance hiérarchique*, *espace de noms*, *protection*, *à plat* et *unité de code*.

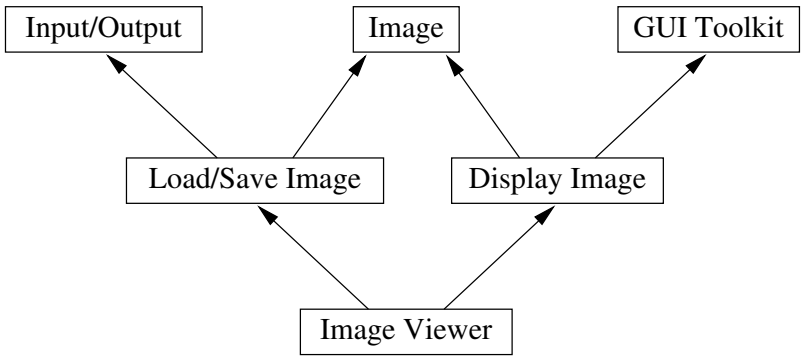


FIG. 2.2: Architecture d'un logiciel de visualisation d'images

Dépendance hiérarchique

Lorsqu'un module `B` utilise un élément d'un module `A`, on dit que `B` *dépend de* `A`. On parle de *dépendance hiérarchique* lorsque la relation de dépendance ne possède pas de circuit (c'est-à-dire qu'un module ne dépend ni directement, ni indirectement de lui-même). On parle de *dépendance spaghetti*[6] lorsque la dépendance n'est pas hiérarchique.

Il est bien évident que l'approche hiérarchique des modules est la plus structurée, donc la plus lisible : comprendre l'architecture d'un logiciel est bien plus simple lorsque la relation de dépendance est hiérarchique. Par exemple, dans la figure 2.2 qui illustre l'architecture d'un logiciel de visualisation d'images, les boîtes correspondent aux modules et les arcs aux liens de dépendance entre modules :

`Input/Output` est le module correspondant aux lectures et écritures de fichiers sur disque ;

`Image` est le module contenant les définitions des éléments (classes, méthodes, etc.) associées aux images — le *modèle* dans la terminologie *Model View Controler*) ;

`GUI Toolkit` est le module dédié aux interfaces utilisateur graphiques ;

`Load/Save Image` est le module préoccupé du chargement et de la sauvegarde des images ;

`Display Image` se préoccupe de l'affichage des images ;

`Image Viewer` est le module principal du logiciel.

Dans l'approche hiérarchique de modules, les dépendances sont uniquement des dépendances au niveau du code (les modules servent à structurer le code source) : un module `B` dépend d'un module `A` si `B` a besoin des données ou des traitements définis dans `A`. Il ne s'agit pas de dépendance au sens du fonctionnement d'un logiciel : les liens de dépendances entre modules ne correspondent pas à « ... se base sur le travail

[6] Le terme « spaghetti » est ici repris de [Szyperski, 1992].

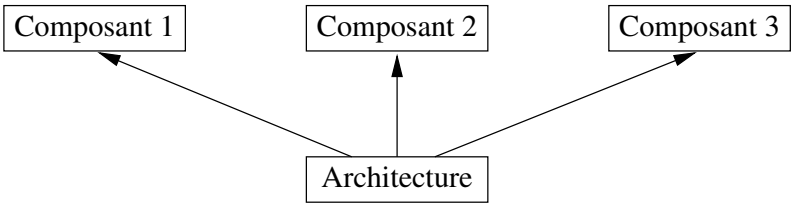


FIG. 2.3: Architecture d'un logiciel à base de composants

de ... » ni à « ... a besoin que le travail de ... ait été effectué ». Ainsi, dans l'exemple de la figure 2.2, le module `Display Image` ne dépend pas du module `Load/Save Image` même s'il parait évident que dans le cadre de notre logiciel, une image doit d'abord être chargée avant de pouvoir être affichée à l'écran.

Le cas extrême de cette idée de dépendance hiérarchique des modules est l'approche par composants [Szyperski, 2002] dans laquelle :

- chaque composant est développé indépendamment de tout autre ;
- une application n'est qu'un assemblage particulier de composants.

Une telle architecture à base de composants est illustrée dans la figure 2.3.

Espace de nom

Les modules sont des espaces de rangement dont l'un des buts est d'éviter les conflits de nom. En général, l'accès à un élément à partir de l'extérieur du module se faisant alors par une désignation explicite de la forme `module::element`. Toutefois, des facilités existent pour réduire la surcharge syntaxique : d'une part un élément défini dans un module n'a pas besoin de désigner explicitement les autres éléments de ce module ; d'autre part, un module peut importer des noms d'un autre module afin de ne plus avoir à préciser le module source (via les mots clé `use` en ADA , `import` en JAVA , `using` en C++ , `FROM IMPORT` en MODULA 3, etc.).

Protection

Tous les éléments d'un module ne sont pas utilisables de la même façon si l'on est à l'intérieur ou à l'extérieur du module. En général, c'est au programmeur que revient la tâche de spécifier ce qui est utilisable en dehors du module et ce qui ne l'est pas. Dans l'approche la plus simple, cette protection consiste à distinguer dans chaque module une partie publique et une partie privée (comme c'est le cas en ADA par exemple). Cette protection peut distinguer les éléments accessibles de l'extérieur de ceux qui ne le sont pas mais elle peut également être plus fine en précisant par exemple que telle variable statique est accessible de l'extérieur seulement en lecture.

Un oui dans le tableau 2.1 signifie que le programmeur a la capacité d'exprimer une protection des éléments d'un module qui est plus restrictive à l'extérieur qu'à l'intérieur. L'existence de protection entre modules est bien sûr indépendante du fait que le langage permette d'exprimer d'autres formes de protection — entre classes par exemple [Ardourel, 2002].

À plat

Nous disons qu'un système de modules est *imbriqué* lorsqu'un module peut contenir la définition d'autres modules. Dans le cas contraire, nous disons que le système de modules est *à plat*.

Le problème de l'imbrication de modules est que sa sémantique est extrêmement variable en fonction des langages. Au mieux ce n'est qu'une façon d'imbriquer des espaces de noms (comme avec les *namespaces* de C++, ou les *packages* de JAVA) au pire, il s'agit d'imbriquer la protection ce qui aboutit inévitablement à des mécanismes complexes et difficiles à appréhender [Ardourel, 2002].

L'avantage pour un système de modules d'être à plat est qu'il permet de considérer que les modules d'un programme forment une décomposition simple et une partition de celui-ci.

Unité de code

Un module est une unité de code source lorsque dans un langage de programmation, un module correspond à un fichier source ou à un couple de fichiers sources, l'un contenant l'interface du module et l'autre son corps. Par exemple, en C et en C++ avec les fichiers `.h` et `.c` ou en ADA avec les fichiers `.ads` et `.adb`. Lorsque les modules sont des unités de code, les programmes ont une bien meilleure lisibilité.

Lorsqu'un module correspond à un seul fichier source, il n'est pas possible d'ajouter des éléments à ce module sans modifier ce fichier source. On dit alors que le système de modules est *fermé*. Lorsque un module peut être disséminé à travers plusieurs fichiers sources, on peut ajouter des éléments à un module simplement en créant un nouveau fichier source. On dit alors que le système de module est *ouvert*. C'est par exemple le cas des packages JAVA dans lesquels n'importe qui peut ajouter une nouvelle classe.

L'intérêt des modules ouverts est qu'ils offrent une plus grande souplesse puisque des éléments peuvent être ajoutés *a posteriori*. Toutefois, comme nous le verrons par la suite, construire un module qui dépend du premier et utiliser ce nouveau module à la place du premier permet d'arriver à autant de souplesse.

Remarque : Dans le cadre d'un langage compilable séparément, on peut poser la question autrement : un module est-il une unité de compilation ? Si la réponse est « oui », c'est vraisemblablement qu'un module est une unité de code.

2.7.2 Modules et classes : la vraie nature des modules

Modules et classes ont de nombreux points communs : ce sont des entités de structuration et ce sont des capsules — voir section 2.5. De plus, comme nous le montrons dans le chapitre 4, les classes et les modules peuvent être structurellement très proches. Toutefois, comme Szyperski [Szyperski, 1992], nous pensons que les classes et les modules sont fondamentalement des abstractions distinctes : les modules structurent les programmes et correspondent à des préoccupations ; les classes décrivent leurs instances.

Deux notions distinctes en PRM

Un langage de programmation à objets devrait proposer deux notions distinctes, l'une de classe et l'autre de module, qui doivent rester dans leurs rôles respectifs : il est hors de question que la notion de module empiète sur celle de classe ou inversement. Ainsi, en PRM, les deux notions sont entièrement différenciées :

- contrairement à la classe, le module est l'unité de code (un fichier = un module), l'unité préférentielle de réutilisation et l'unité préférentielle de compilation — du moins dans un cadre de compilation séparée, comme c'est le cas avec le compilateur `prmc` ;
- un module contient uniquement des définitions de classes (donc pas de modules imbriqué) alors qu'une classe contient les définitions des propriétés de ses instances : attributs et méthodes (donc pas de classe imbriquée ni de module défini dans des classes) ;
- un module explicite l'ensemble des modules dont il dépend, la dépendance est hiérarchique ;
- un module importe les classes des modules dont il dépend. Il n'y a pas de notion de protection au niveau de la frontière entre modules, nous n'en avons pas besoin pour l'instant : lorsqu'un module `B` dépend d'un module `A`, le module `B` importe toutes les classes définies dans `A` mais également toutes les classes importées par `A`. En d'autres termes, la relation de dépendance n'est pas seulement exempte de circuit mais elle est également transitive.

Le chapitre 4 traite de façon plus approfondie les rapports entre classes et modules ainsi que les éventuels conflits liés aux dépendances multiples.

Au niveau de l'architecture des logiciels, un module définit donc les classes correspondantes à une préoccupation. Par exemple, le module `GUI Toolkit` de la figure 2.2 définit des classes comme `Widget` ou `Button`. Le module `Display Image` importe les classes des modules dont il dépend (dont les classes `Widget` et `Button`) et peut les utiliser : typage, instanciation et définition de sous-classes.

Petit aperçu syntaxique en PRM

Syntaxiquement, un module PRM correspond à un fichier, le nom du module étant déduit du nom du fichier — sans l'extension `".prm"`. La relation de dépendance entre modules s'exprime par le mot clé `import` :

```
# Fichier m1.prm
class Toto
        ...
end
```

```
# Fichier m2.prm
import m1

class Tata
        inherit Toto
        ...
end
```

Au final, le système de modules de PRM est extrêmement simple[7] comme l'illustre parfaitement son seul mot clé `import`.

2.7.3 Raffinement de classes

Besoin d'un peu de raffinement

Un logiciel architecturé à base de modules et de classes, tel que nous l'avons décrit dans la section précédente, peut répondre aux problématiques d'évolution et de réutilisation. Toutefois, de nombreux travaux ont montré qu'une classe ne correspond pas systématiquement à une seule préoccupation [Andersen et Reenskaug, 1992; Kiczales *et al.*, 1997; Ossher et Tarr, 2001; Apel *et al.*, 2005]. En particulier, un programme doit pouvoir être étendu non seulement en définissant de nouvelles classes mais également en ajoutant de nouvelles propriétés aux classes existantes — ce que [Findler et Flatt, 1999; Torgersen, 2004; Zenger et Odersky, 2004] appellent l'*expression problem*.

De nombreuses approches ont été proposées pour répondre à ce problème d'expression. Par exemple, dans la programmation par aspects [Kiczales *et al.*, 1997] et les langages tels que Hyper/J [Ossher et Tarr, 2001] ou AspectJ [Kiczales *et al.*, 2001], les préoccupations correspondent à des entités nommées *aspects* et sont *tissées* à des classes existantes.

Dans le cas d'architectures à base de classes et de modules, qui correspondent à celles que nous venons de présenter, l'approche la plus simple pour répondre au problème

[7] Et c'est un bon facteur de qualité si l'on en croit le principe 2 page 14.

consiste à permettre aux modules de *raffiner*[8] — c'est-à-dire modifier — les classes qu'ils importent des modules dont ils dépendent. Cette idée se retrouve dans de nombreux travaux, anciens et récents : *Virtual Classes* [Madsen et Møller-Pedersen, 1989], *Open Classes* [Clifton *et al.*, 2000], *Difference-Based Modules* [Ichisugi et Tanaka, 2002], *Classboxes* [Bergel *et al.*, 2003], *Higher-Order Hierarchies* [Ernst, 2003], *Nested Inheritance* [Nystrom *et al.*, 2004; Nystrom *et al.*, 2005].

Approche intuitive du raffinement de classes

Intuitivement, le raffinement d'une classe c par une classe c' se présente comme une définition incrémentale de classes dans laquelle les attributs et les méthodes définis dans la classe c' s'ajoutent ou prennent la place de ceux de c. Contrairement à la spécialisation de classes, une fois que c a été raffinée par c', cette dernière prend la place de c dans toutes ses occurrences dans la totalité du programme.

Un tel mécanisme est commun dans de nombreux langages en typage dynamique. Dans de nombreuses extensions objet de Lisp, comme Flavors et Clos, une telle définition incrémentale est réservée aux méthodes puisqu'elles sont définies en dehors des classes. En Yafool [Ducournau, 1991] ou en Ruby, le raffinement de classes est d'usage standard. En toute généralité, c'est possible dans tous les langages dotés d'un *protocole à méta-objet* [Kiczales *et al.*, 1991; Pavillet, 2000] comme Clos, même si des expériences montrent que ces protocoles ne sont pas toujours très adaptés à la modification des classes [Pavillet et Ducournau, 1999].

Le caractère intuitif du raffinement se perd un peu dès que l'on rencontre un raffinement multiple d'une même classe ou que l'on combine raffinement et spécialisation : l'ordre des raffinements et de la spécialisation sous-jacente n'est plus innocent. Si les langages dynamiques peuvent se reposer sur l'ordre chronologique, cela n'est plus possible dans le cadre d'un langage à typage statique comme PRM.

Les mécanismes de raffinement

Le mécanisme de raffinement que nous proposons et que nous avons spécifié[9] dans le langage PRM permet quatre raffinements atomiques que nous illustrons par le module `m0` ci-dessous et par les modules `m1`, `m2`, `m3`, `m3b` et `m4`.

```
# Fichier m0.prm
class A
        ...
```

[8] Il n'existe pas de vocabulaire commun pour désigner ce mécanisme. Le terme le plus proche de l'idée serait sans doute « extension de classe » mais a le défaut de prêter à confusion avec le mot clé `extends` de Java et la notion d'« éléments » utilisée dès que l'on parle de concepts ou d'ensembles — cf. note 10 (page 46).

[9] De façon incroyable, cette spécification ne provoque aucun surcoût syntaxique !

```
end
class B
        inherit A
        def toto ...
end
```

Ajout d'une propriété. C'est-à-dire l'ajout d'une méthode ou d'un attribut dans une classe importée :

```
# Fichier m1.prm
import m0
class B
        def tata ...
end
```

L'ajout d'une méthode est le minimum que l'on peut demander à un mécanisme de raffinement. L'ajout d'attributs n'est pas systématique dans les différentes propositions car il peut poser des problèmes comme nous le verrons dans la section 7.5 page 155.

Redéfinition d'une propriété. La nouvelle propriété prend la place de l'ancienne :

```
# Fichier m2.prm
import m0
class B
        def toto ...
end
```

Remarque : La redéfinition est plus commune pour les méthodes que pour les attributs.

Ajout d'une super-classe. La classe raffinée profite alors du sous-typage et de l'héritage :

```
# Fichier m3.prm
import m0
class A2
        def titi ...
end
class B
        inherit A2
end
...
let a2: A2
let b: B
```

```
a2 := b # B sous-type valide de A2
b.titi  # B herite 'titi' de A2
```

Toutefois, bien que nous nous intéressons principalement aux langages en héritage multiple, l'ajout de super-classe n'est pas incompatible avec les langages en héritage simple :

```
# Fichier m3b.prm
import m0
class A2b
        inherit A
        ...
end
class B
        inherit A2b
        # B n'est plus sous-classe directe
        # de A mais de A2b !
end
```

L'ajout de super-classes est sans doute le mécanisme le moins présent dans les propositions existantes que nous avons pu étudier.

Généralisation d'une propriété. C'est-à-dire la définition d'une propriété dans une super-classe de la classe qui introduit cette propriété :

```
# Fichier m4.prm
import m0
class A
        def toto ...
end
```

Ce mécanisme peut sembler n'être qu'une variation du premier (ajout d'une propriété) mais ce n'est pas forcément le cas comme nous le verrons dans la section 4.5.2 page 92.

Notre proposition. Comme nous l'avons dit, le principe du raffinement de classes n'est pas complètement nouveau en soit, toutefois les travaux existants souffrent de quelques limitations :

- la redéfinition de propriété et plus souvent l'ajout de super-classe ne sont pas toujours possibles ;
- ils unifient les notions de classes et de modules ;
- ils sont basés sur des langages en héritage simple (ou sur une forme dégradée d'héritage), il en résulte que le raffinement multiple et la combinaison entre le

raffinement et l'héritage ne sont pas toujours gérés de façon correcte — c'est-à-dire, selon le principe 1 (page 2), d'une façon proche du mode de pensée humain.

Afin de prendre particulièrement en compte ce dernier point, notre proposition, décrite au chapitre 4, se base sur une méta-modélisation des modules et des classes isomorphe à la méta-modélisation des classes et des propriétés (que nous présentons dans le prochain chapitre).

2.8 Conclusion

Nos travaux sur la spécification des langages à objets et ceux sur la compilation efficace nous ont amené à spécifier PRM, un nouveau langage de programmation qui est à la fois le résultat de nos recherches en spécification et la cible de nos recherches en compilation. Nous pensons que PRM est un bon langage de programmation — du moins selon les principes 1 et 2 que nous avons énoncés dans l'introduction de ce mémoire — ce qui se traduit par :

- des concepts simples, cohérents et intuitifs ;
- un syntaxe claire, rigoureuse et concise.

Toutefois, PRM est un langage encore jeune et son équipe de développement est restreinte. Ainsi, de nombreuses fonctionnalités ne sont pas encore disponibles mais sont prévues à plus ou moins long terme :

Contrats. Les contrats à la Eiffel permettent d'annoter les méthodes par des pré- et post-assertions. D'une part ils offrent une documentation au programmeur et d'autre part ils permettent une vérification dynamique des programmes.

Exceptions. Le mécanisme d'exception permet de traiter d'une manière curative (et non préventive) les cas d'erreurs qui se produisent lors de l'exécution des programmes. La majorité des langages de programmation orientés objet intègrent un mécanisme d'exception.

Réflexivité. La réflexivité permet d'agir sur un programme au cours de son exécution. Toutefois, seul un mécanisme d'introspection est prévu — à la Java et son package `reflect`.

Sélection multiple. La sélection multiple offre une liaison tardive dont la résolution est déterminée par le type dynamique de tous les arguments et non plus par celui du seul receveur. Dans le cadre du langage PRM, nous envisageons une spécification de la sélection multiple à l'aide de *multi-méthodes* [Mugridge *et al.*, 1991; Castagna, 1997].

Chapitre 3

Une méta-modélisation des classes et propriétés

Préambule

Ce chapitre propose une formalisation des relations entre les classes et les propriétés (attributs, méthodes, etc.) vis-à-vis de la spécialisation de classes ainsi que de l'héritage et de la redéfinition des propriétés; en particulier quand la spécialisation et l'héritage sont multiples. Nous posons comme principe de base un mécanisme d'héritage basé sur une approche naturelle de la relation de spécialisation entre les classes. La formalisation ainsi construite a les deux avantages suivants :

- *être intuitive et simple afin de pouvoir servir de base à la spécification du langage de programmation PRM ;*
- *être universelle et souple afin d'être suffisamment compatible avec l'utilisation habituelle de la programmation par objets.*

3.1 Introduction

Dans le contexte de l'approche objet, la méta-modélisation est un outil puissant qui consiste à construire des ontologies[1] qui permettent de spécifier et de comprendre les concepts et les mécanismes mis en jeu dans le paradigme objet. Une méta-modélisation peut également servir de base à la réalisation de logiciels dédiés aux traitements de programmes informatiques : (méta-)évaluateurs, compilateurs[2], outils d'analyse, etc. Méta-modéliser un sous-ensemble d'un langage de programmation à objets consiste à

[1] Une *ontologie* est la définition de concepts utilisés dans un langage donné et la description des relations logiques qu'ils entretiennent entre eux.

[2] D'ailleurs le compilateur `prmc` (cf. chapitre 8) se base sur le méta-modèle que nous présentons dans ce chapitre.

définir un modèle à objets — c'est-à-dire des entités comme des classes, des associations, des attributs, des méthodes, etc. — afin de modéliser les concepts considérés du langage. Toutefois, afin de respecter le principe 2 (page 14), toute méta-modélisation devrait respecter le principe de minimalité du rasoir d'Occam : les entités ne devraient pas être multipliées sans nécessité[3]. De plus, nous avons la volonté d'ajouter une contrainte supplémentaire qui exprime une forme de complétude :

Principe 5 (Complétude de la méta-modélisation) *Dans un programme, toute occurrence d'un identifiant dénotant une entité méta-modélisée doit désigner de façon non ambiguë une unique instance d'une classe du méta-modèle.*

Les rapports entre les classes et leurs instances ont fait l'objet de nombreuses recherches comme par exemple ObjVlisp [Cointe, 1987] qui est aujourd'hui considéré comme le méta-modèle de référence pour qui veut comprendre les implications entre instances, classes et méta-classes. Toutefois, dans les langages de programmation, le rapport entre les classes et leurs propriétés (méthodes et attributs)[4] n'a jamais vraiment été formalisé dans toute sa généralité ; peut-être parce que chaque langage de programmation propose des entités, des relations et des mécanismes qui lui sont propres.

Ce chapitre est consacré à la proposition motivée d'une méta-modélisation universelle des classes et des propriétés. L'intérêt de ce méta-modèle est multiple :

- d'une part, il permet de donner une définition conceptuelle et relationnelle des classes et de leurs propriétés dans les langages à objets ;
- d'autre part, sa construction est guidée par l'approche naturelle de la spécialisation. Il offre donc des entités intuitives et des mécanismes cohérents ;
- ensuite, il permet de mettre en évidence et de proposer des solutions pour résoudre des problèmes de modélisation qui apparaissent en héritage multiple ;
- enfin, ce méta-modèle offre une vision commune et unifiée de l'utilisation habituelle de la programmation par objet[5].

Les langages de programmation que nous considérons sont l'ensemble des langages à objets et à classes. Dans le reste du chapitre, nous évoquerons les langages C++, Clos, Eiffel, Java, Objective-Caml, Python et Smalltalk (cf. annexe B). Cependant, nous nous intéresserons plus particulièrement aux langages à typage statique et en héritage multiple pour lesquels la méta-modélisation que nous proposons est la plus adaptée.

---

[3] « Pluralitas non est ponenda sine neccesitate » — William d'Occam, XIV^e siècle.

[4] Dans toute cette thèse, le terme « propriété » désigne de façon systématique les propriétés au sens de la programmation par objet et non les propriétés au sens de la mathématique.

[5] L'universalité que nous avons prétendu dans le préambule du chapitre, la méta-modélisation que nous proposons ne cible pas les langages de programmation dans leur totalité mais bien l'utilisation de l'approche par objets dans le cadre du développement de logiciels.

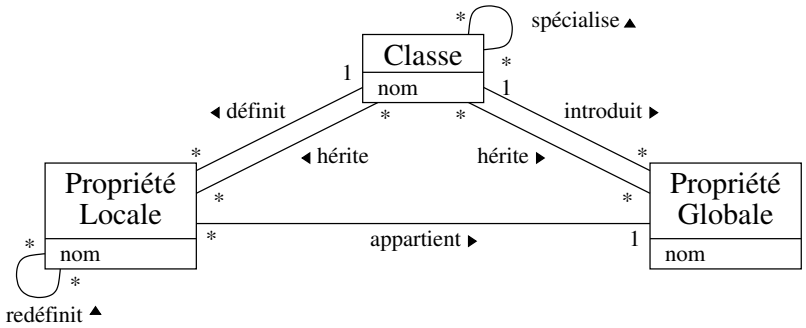


FIG. 3.1: Méta-modèle des propriétés

La section suivante présente le méta-modèle et décrit ses entités et relations de façon informelle. Ensuite chacune des quatre sections suivantes est consacrée en profondeur à une partie du méta-modèle : les classes et la relation de spécialisation pour la section 3.3, les propriétés globales et leur héritage pour la section 3.4, les propriétés locales, leur définition et l'introduction des propriétés globales pour la section 3.5 et enfin, la résolution des envois de message et l'héritage des propriétés locales pour la section 3.6. La section 3.7 est dédiée aux héritages complexes et aux cas particuliers : méthodes abstraites, protection, contrats, etc.

3.2 Classes et propriétés

Le méta-modèle que nous proposons qui modélise les *classes* et les *propriétés* est présenté figure 3.1 sous forme d'un diagramme de classes UML. Dans la suite, lorsque nous parlerons « du » méta-modèle, il s'agira de celui-ci.

Le terme *classe* désigne des entités qui :

– sont hiérarchisées par une relation de *spécialisation* ;
– possèdent des propriétés, que celles-ci soient, selon la terminologie considérée, *définies*, *introduites* ou *héritées* ;
– possèdent des instances, que celles-ci soient directes (propres) ou indirectes[6].

Ainsi nous incluons sous cette terminologie de *classe*, les classes telles qu'elles sont définies dans les langages de programmation mais aussi les *interfaces* au sens de JAVA.

Le terme *propriété* désigne de façon générale ce qui correspond aux données et aux procédures et qui s'appelle dans les différents langages : *attribut*, *champ* ou *variable d'instance* pour les données et *méthode*, *routine* voire *fonction générique* pour les procédures. Dans la suite, nous utiliserons les termes *attribut* et *méthode* lorsque la distinction sera nécessaire.

[6]Les *instances directes* d'une classe sont les objets que cette classe a instanciés elle-même. Les *instances indirectes* d'une classe sont les instances directes de cette classe et de toutes ses sous-classes.

Toutefois, pour méta-modéliser les propriétés, l'*envoi de message* (appelé également *liaison tardive* ou *late binding*) impose la définition de deux catégories d'entités afin de respecter le principe 5. La première entité correspond aux propriétés telles qu'elles sont définies dans les classes. La seconde entité correspond aux propriétés telles qu'elles sont utilisées lors des envois de message. Le méta-modèle est donc composé de trois entités principales : les *classes*, les *propriétés locales* et les *propriétés globales* (figure 3.1).

Les *propriétés locales* correspondent aux attributs et aux méthodes que le programmeur définit, c'est-à-dire décrit lors de la définition d'une *classe*, indépendamment de toute éventuelle autre définition de la « même propriété » dans les super-classes et les sous-classes. Par exemple, la définition d'une méthode dans une classe et sa redéfinition dans une sous-classe correspondent à deux propriétés locales distinctes.

Les *propriétés globales*[7] servent à modéliser cette idée de « même propriété » à travers plusieurs classes en relation de spécialisation. Les propriétés globales correspondent aux messages auxquels les instances d'une classe peuvent répondre : introduire une propriété globale dans une classe c'est donner à cette classe et à ses sous-classes la capacité à répondre à un nouveau message. Par exemple, dans le cas des méthodes, cette réponse correspond à l'invocation de la propriété locale du type dynamique du receveur qui appartient à la propriété globale.

Une propriété locale :
- est *définie* dans une seule classe — la classe dans laquelle le programmeur écrit la définition de la propriété locale ;
- *appartient* à une seule propriété globale — c'est une réponse à un envoi de message particulier ;
- peut *redéfinir*[8] une propriété locale définie dans une super-classe et qui appartient à la même propriété globale ;
- est *héritée* par les sous-classes de sa classe de définition — sauf si elle est redéfinie.

Une propriété globale :
- est *introduite* dans une seule classe ;
- regroupe des propriétés locales : celle qui sert à l'introduire et ses redéfinitions ;
- est *héritée* par toutes les sous-classes de la classe qui l'introduit.

Remarque : Le fait que les propriétés globales soient introduites dans une seule classe rend le méta-modèle en partie inadapté aux langages à typage dynamique (cf. section 3.4.2).

[7] Dans des travaux antérieurs, le terme utilisé était *propriété générique*, par analogie avec les *fonctions génériques* de CLOS .

[8] Certaines propriétés locales peuvent ne pas être redéfinissables (cf. section 3.5.3).

3.2.1 Définition de hiérarchies de classes et construction de modèles

Dans les langages de programmation, la *définition d'une hiérarchie de classes* correspond à l'ensemble des définitions de chacune des classes de cette hiérarchie.

La *définition d'une classe* est un triplet $\langle \texttt{classname}, \texttt{supernames}, \texttt{localdef} \rangle$ constitué du nom de la classe (`classname`), d'un ensemble de noms de super-classes (`supernames`) et d'un ensemble de définitions de propriétés locales (`localdef`).

La *définition d'une propriété locale* est constituée du nom de la propriété et d'autres données qui ne nous intéressent pas pour l'instant.

Exemple, à partir du listing JAVA suivant :

```
class A {
        public void toto() { ... }
}
class B extends A {
        public void toto() { ... }
        public void tata() { ... }
}
```

on associe la *définition de la hiérarchie de classes* suivante :

$$\{\langle \texttt{A}, \emptyset, \{\texttt{toto}\}\rangle, \langle \texttt{B}, \{\texttt{A}\}, \{\texttt{toto}, \texttt{tata}\}\rangle\} \ .$$

A partir de la définition d'une hiérarchie de classes, on doit pouvoir construire une instance[9] du méta-modèle (c'est-à-dire un modèle) correspondant à cette définition.

Dans l'exemple précédent, on dénombre sept entités que l'on a représentées dans la figure 3.2 sous la forme d'un diagramme d'instances UML :

- deux classes : celle nommée `A` et celle nommée `B` ;
- trois propriétés locales : celle nommée `toto` définie dans la classe `A`, celle nommée `toto` définie dans la classe `B` et celle nommée `tata` définie dans la classe `B` ;
- et deux propriétés globales : celle introduite par `toto` dans `A` et celle introduite par `tata` dans `B`. Le listing suivant montre trois sites d'envoi de message, la propriété globale associée à chacun est indiquée en commentaire :

```
A a; B b;
...
a.toto(); // toto introduite dans A
b.toto(); // toto introduite dans A
b.tata(); // tata introduite dans B
```

[9] En ingénierie des modèles, un modèle n'est pas lié à un méta-modèle par la relation « est instance de » mais par la relation « est conforme à ». C'est par abus que nous utilisons la terminologie « instance de » puisque nous nous plaçons dans le monde des objets.

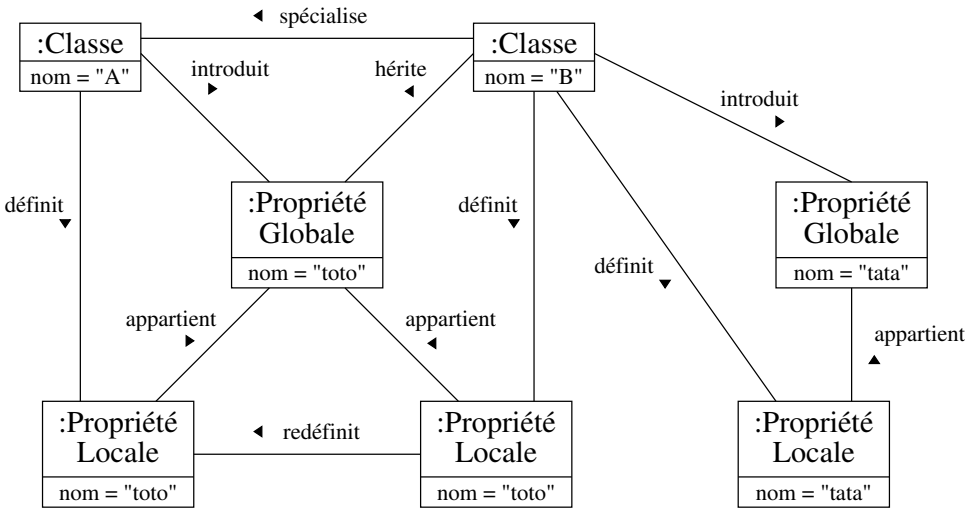


FIG. 3.2: Instance du méta-modèle de l'exemple du listing en JAVA

3.2.2 Formalisme

Notations. Soit E et F deux ensembles et $f : E \to F$ une fonction :
- $|E|$ désigne le cardinal de E, c'est-à-dire le nombre d'éléments de E ;
- $E \uplus F$ désigne l'union de E et F en insistant sur le fait qu'elle est disjointe (intersection vide) ;
- $\mathcal{P}(E)$ désigne l'ensemble des parties de E :

$$\mathcal{P}(E) \stackrel{\text{def}}{=} \{E' \mid E' \subseteq E\} \ ;$$

- la définition de f est étendue en une fonction $\mathcal{P}(E) \to \mathcal{P}(F)$ de façon habituelle :

$$\forall E' \subseteq E, f(E') = \{f(x) \mid x \in E'\} \ ;$$

- f^{-1} désigne la fonction $F \to \mathcal{P}(E)$ définie ainsi :

$$\forall x \in F, f^{-1}(x) \stackrel{\text{def}}{=} \{x' \in E \mid f(x') = x\} \ .$$

Définition 2 *[Modèle de hiérarchie] Un* modèle de hiérarchie*, c'est-à-dire une instance du méta-modèle, est un n-uplet* $\mathcal{H} = \langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, \text{nom}_{\mathcal{H}}, \text{glob}_{\mathcal{H}}, \text{intro}_{\mathcal{H}}, \text{def}_{\mathcal{H}} \rangle$ *dans lequel :*
- $X^{\mathcal{H}}$ *est l'ensemble des* classes*.*
- $\prec^{\mathcal{H}}$ *est la relation de spécialisation de classes.*
- $G^{\mathcal{H}}$ *est l'ensemble des propriétés globales.*
- $L^{\mathcal{H}}$ *est l'ensemble des propriétés locales.*

- $N^{\mathcal{H}}$ *est l'ensemble des* identifiants *(noms) de* classes *et de* propriétés.
- $\text{nom}_{\mathcal{H}} : X^{\mathcal{H}} \uplus G^{\mathcal{H}} \uplus L^{\mathcal{H}} \to N^{\mathcal{H}}$ *est la fonction de désignation.*
- $\text{glob}_{\mathcal{H}} : L^{\mathcal{H}} \to G^{\mathcal{H}}$ *associe chaque propriété locale à la propriété globale à laquelle elle appartient.*
- $\text{intro}_{\mathcal{H}} : G^{\mathcal{H}} \to X^{\mathcal{H}}$ *associe chaque propriété globale à la classe qui l'introduit ;*
- $\text{def}_{\mathcal{H}} : L^{\mathcal{H}} \to X^{\mathcal{H}}$ *associe chaque propriété locale à la classe qui la définit.*

Le méta-modèle est générique puisque toutes ses composantes sont paramétrées par $\mathcal{H}$. Toutefois, pour des raisons de lisibilité, ce paramètre $\mathcal{H}$ sera implicite dans le reste du chapitre. Dans le chapitre suivant, le paramètre sera à nouveau explicite puisque celui-ci fait intervenir plusieurs modèles de hiérarchie.

Tous les modèles de hiérarchie ne sont pas valides, les trois prochaines sections vont respectivement poser des contraintes qui concerne la validité de la spécialisation ($\prec$), celle des propriétés globales (G) et celles des propriétés locales (L). Toutefois, pour commencer, la première contrainte concerne le nom des classes. En effet, nous considérons que dans une hiérarchie de classes, chaque classe doit avoir un nom unique : la restriction de la fonction nom sur l'ensemble X doit être injective.

3.3 Spécialisation de classes

La *relation de spécialisation* et le *mécanisme d'héritage* constituent certainement la caractéristique la plus originale de l'approche objet, malheureusement, c'est aussi la cause des principales difficultés.

Dans toute cette thèse, et plus particulièrement dans ce chapitre, nous distinguons la *relation de spécialisation*, qui établit les notions de *super-classe* et de *sous-classe*, du *mécanisme d'héritage* qui s'appuie sur la relation de spécialisation et qui consiste à identifier les propriétés globales et locales d'une classe c, celles-ci étant entièrement déterminées par la définition de la classe c et celles de toutes ses super-classes — c'est le versant déductif de l'héritage, par opposition à la classification qui en est le versant inductif [Napoli, 1992].

Selon nous, la seule bonne façon d'appréhender la relation de spécialisation consiste à respecter l'approche intuitive, que nous appelons *approche naturelle de la spécialisation* — ou simplement *approche naturelle* pour faire plus court. Celle-ci remonte au moins à la tradition aristotélicienne et on peut l'illustrer avec le système de propositions que nous avons déjà évoqué dans l'introduction de ce mémoire :

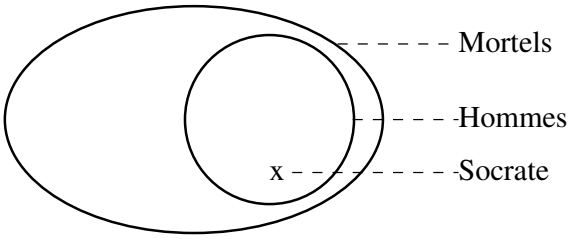
> Socrate est un homme.
> Or les hommes sont mortels.
> Donc Socrate est mortel.

Où ici « Socrate » est une instance tandis que « homme » et « mortel » sont des classes.

Vis-à-vis de la relation entre classes et instances, cela revient à poser le principe suivant :

Principe 6 (Covariance des extensions[10]) *Les instances d'une classe sont aussi les instances de ses super-classes et la relation de spécialisation entre les classes est le reflet de l'inclusion de leurs extensions.*

La figure suivante illustre la même chose de façon ensembliste :



3.3.1 Formalisme

Rappel : Nous nous plaçons dans une hiérarchie de classes $\mathcal{H} = \langle X^{\mathcal{H}}, \prec^{\mathcal{H}}, G^{\mathcal{H}}, L^{\mathcal{H}}, N^{\mathcal{H}}, \text{nom}_{\mathcal{H}}, \text{glob}_{\mathcal{H}}, \text{intro}_{\mathcal{H}}, \text{def}_{\mathcal{H}} \rangle$ telle que définie dans la section 3.2.2 et pour des raisons de lisibilité, nous omettons le paramètre $\mathcal{H}$.

La relation de spécialisation $\prec$ est un ordre partiel strict (relation transitive, anti-symétrique et anti-réflexive). Nous notons $\preceq$ la fermeture réflexive de la relation de spécialisation.

Si, pour deux classes c et $c' \in X$, $c' \prec c$, nous disons que :
- c' *spécialise* c ;
- c' est plus *spécifique* que c ;
- c' est une *sous-classe* de c ;
- c *généralise* c' ;
- c est plus *générale* que c' ;
- c est une *super-classe* de c'.

Remarque : La notation $c' \prec c$ peut paraître un peu déroutante car d'une part la « flèche » va dans le sens opposé au sens UML classique et d'autre part la sous-classe (c') est notée avant la super-classe (c) ce qui est contraire à la réalité du développement dans lequel on définit c avant c'. On peut toutefois remarquer l'analogie de notation entre la variance des classes et celle des extensions — qui varient dans le même sens, cf. principe 6 :

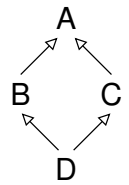
$$ext(c') \subseteq ext(c) \Leftrightarrow c' \preceq c \ ,$$

où *ext* est la fonction qui associe une classe à son extension.

[10] L'*extension* d'une classe est l'ensemble de ses instances (propres ou non).

```
class A end
class B inherit A end
class C inherit A end
class D inherit B inherit C end
```

(a) Définitions des classes



(b) Représentation graphique

Classe	Super-classes		Sous-classes	
	toutes ($\prec$)	directes ($\prec_{\mathrm{d}}$)	toutes ($\prec$)	directes($\prec_{\mathrm{d}}$)
A	$\emptyset$	$\emptyset$	{ B, C, D }	{ B, C }
B	{ A }	{ A }	{ D }	{ D }
C	{ A }	{ A }	{ D }	{ D }
D	{ A, B, C }	{ B, C }	$\emptyset$	$\emptyset$

(c) Super-classes et sous-classes

FIG. 3.3: Super-classes et sous-classes

Nous notons $X_{\uparrow c}$ (respectivement $X_{\downarrow c}$) l'ensemble des super-classes (respectivement sous-classes) de la classe $c \in X$ au sens strict (c'est-à-dire c exclue), soit :

$$X_{\uparrow c} = \{c' \in X \mid c \prec c'\} \ ;$$
$$X_{\downarrow c} = \{c' \in X \mid c' \prec c\} \ .$$

Ainsi, dans l'exemple de la figure 3.3, la classe `B` est une super-classe de la classe `D` et une sous-classe de la classe `A`.

Spécialisation directe

La *relation de spécialisation directe*, notée $\prec_{\mathrm{d}}$, est la réduction transitive de la relation de spécialisation ($\prec$).

Les *super-classes directes* d'une classe (respectivement *sous-classes directes*) sont ses super-classes les plus spécifiques (respectivement ses sous-classes les plus générales). On peut formellement désigner les super-classes directes d'une classe $c \in X$ par $\min_{\prec}(X_{\uparrow c})$ et ses sous-classes directes par $\max_{\prec}(X_{\downarrow c})$.

Lors de la représentation graphique d'une hiérarchie de classes, nous ne représentons que la relation de spécialisation directe, celle-ci étant symbolisée par des flèches — les pointes des flèches dirigées vers les super-classes. Ainsi, dans l'exemple de la figure 3.3, la classe `A` est une super-classe directe de la classe `B` (on dessine une flèche de `B` vers `A`) mais une super-classe indirecte de la classe `D` (donc pas de flèche).

Déclaration de spécialisations

Nous notons $X_{\uparrow_{\text{decl}}c}$ l'ensemble des *super-classes déclarées* d'une classe c. Elles correspondent aux classes `supernames` que le programmeur a déclaré dans le code source lors de la définition de cette classe c :

$$X_{\uparrow_{\text{decl}}c} \stackrel{\text{def}}{=} \text{nom}^{-1}(\texttt{supernames}) \ . \tag{3.1}$$

Nous notons $\prec_{\text{decl}}$ la *relation de spécialisation déclarée* définie par :

$$c' \prec_{\text{decl}} c \stackrel{\text{def}}{\Longleftrightarrow} c' \in X_{\uparrow_{\text{decl}}c} \ . \tag{3.2}$$

La relation de spécialisation ($\prec$) est construite à partir de la relation de spécialisation déclarée ($\prec_{\text{decl}}$). Puisqu'il faut garantir que la relation de spécialisation est un ordre partiel, déclarer une classe $c' \in X$ sous-classe d'une classe $c \in X$ revient à dire que c' est sous-classe de c et de toutes les super-classes de c. Au final, la relation de spécialisation ($\prec$) est définie par la fermeture transitive de la relation de spécialisation déclarée ($\prec_{\text{decl}}$) ; et la relation de spécialisation directe ($\prec_{\text{d}}$) est définie par la réduction transitive de $\prec$.

Lors de la construction d'un modèle de hiérarchie à partir d'une définition de hiérarchie, deux sortes d'erreurs peuvent se produire au niveau de la relation de spécialisation. Dans les deux cas, ces erreurs sont insolubles. Le programmeur n'a que la solution de modifier le code :

Super-classe inconnue. Le nom d'une superclasse déclarée (`supernames`) dans la définition d'une classe ne correspond à aucun nom (`classname`) d'une définition de classe :

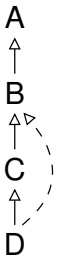
$$\text{nom}^{-1}(\texttt{supernames}) \not\subseteq X \ .$$

Circuit de spécialisation. La relation de spécialisation déclarée contient un circuit. On ne peut pas construire une relation de spécialisation qui soit un ordre partiel.

Il est important de noter que le rôle des super-classes déclarées et de la relation de spécialisation associée n'est pas sémantique, c'est juste le moyen mis à la disposition du programmeur pour définir les super-classes d'une classe. Comme les *arcs de transitivité* (c'est-à-dire les super-classes déclarées mais qui ne sont pas des super-classes directes) n'ont pas d'incidence sur les relations $\prec$ et $\prec_{\text{d}}$, la bonne façon de programmer consiste alors à ne déclarer que les super-classes nécessaires (c'est-à-dire les super-classes directes). Ainsi en pratique, si le système détecte un arc de transitivité, celui-ci doit

```
class A end
class B inherit A end
class C inherit B
class D inherit B inherit C end
```

(a) Définitions des classes



(b) Représentation graphique

Classe	Super-classes		
	toutes ($\prec$)	déclarées ($\prec_{\text{decl}}$)	directes ($\prec_{\text{d}}$)
A	$\emptyset$	$\emptyset$	$\emptyset$
B	A	A	A
C	A, B	B	B
D	A, B, C	B, C	C

(c) Super-classes

FIG. 3.4: Super-classes, super-classes déclarées et super-classes directes

l'ignorer et éventuellement le signaler. Malgré tout, la section 3.3.2 montrera des cas d'utilisation sémantiques (souvent abusifs) des super-classes déclarées.

Graphiquement, nous représentons les arcs de transitivité comme des arcs de spécialisation directe mais en pointillés. Ainsi, dans l'exemple de la figure 3.4, les classes `B` et `C` sont des super-classes déclarées de `D`. Toutefois, la déclaration de `B` comme super-classe de `D` est superflue : il s'agit d'un arc de transitivité.

Spécialisation simple et spécialisation multiple

Définition 3 (Spécialisation simple et multiple) *Nous disons qu'un langage est en* spécialisation multiple *(abusivement en* héritage multiple*) lorsqu'une classe peut posséder plusieurs super-classes directes. Nous disons qu'un langage est en* spécialisation simple *(abusivement en* héritage simple*) dans le cas contraire.*

Cette définition diffère très légèrement de la plupart de celles que l'on peut trouver

ailleurs. En effet, la définition naïve « en héritage multiple, une classe peut avoir plusieurs super-classes » n'est pas satisfaisante puisque le mot « direct » est implicite. La seconde définition « en héritage multiple, une classe peut déclarer plusieurs super-classes » est bien meilleure et en pratique elle est équivalente à celle que nous proposons. En effet, il n'y aucun intérêt à ce qu'un langage permette de déclarer plusieurs super-classes mais ne permette pas aux classes d'avoir plusieurs super-classes directes, ce qui de fait interdirait au programmeur de déclarer autre chose que des arcs de transitivité.

Remarque : Nous rappelons que le méta-modèle ne fait pas de distinction entre les classes et les interfaces au sens JAVA. C'est la raison pour laquelle il ne faut pas être surpris lorsque nous traitons JAVA comme un langage en spécialisation multiple.

3.3.2 Variations autour de la spécialisation

L'approche de l'héritage présentée dans ce chapitre n'est pas forcément la même que celles que l'on peut trouver dans la littérature et en particulier dans la spécification des langages. On peut citer quelques variations particulières : l'héritage d'implémentation, l'héritage de mixins, la transitivité explicite et les chemins d'héritages.

Héritage d'implémentation

On parle d'*héritage d'implémentation* pour désigner un mécanisme *ad hoc* qui nie l'approche naturelle de la spécialisation : on fera par exemple hériter la classe `Personne` de la classe `String` sous prétexte qu'une personne possède un nom alors que la bonne modélisation consisterait à définir un attribut `nom` de type `String` dans la classe `Personne`.

De façon moins extrême, mais tout aussi condamnable, on pourrait faire hériter la classe `Pile` de la classe `Liste` sous prétexte qu'une pile peut s'implémenter à l'aide d'une liste chaînée et que les méthodes `push`, `pop` et `is_empty` peuvent être héritées telles quelles. De la même manière, la bonne modélisation consisterait ici à utiliser un attribut de type `Liste`. Éventuellement, le langage considéré pourrait simplifier l'écriture d'une telle classe `Pile` en proposant une facilité syntaxique de délégation pour ces trois méthodes.

Il est important de bien comprendre que l'héritage d'implémentation n'est pas en tant que tel une caractéristique d'un langage de programmation. En effet, il ne s'agit que d'une utilisation de l'héritage et non d'une spécification particulière de celui-ci. Toutefois, certains langages facilitent l'utilisation de l'héritage d'implémentation. C'est le cas de C++ qui propose un mécanisme d'*héritage privé* qui n'a d'utilité que pour l'héritage d'implémentation. C'est également le cas d'EIFFEL qui inclut dans sa dernière version une notion d'*héritage non-conforme* qui correspond exactement à cette idée d'héritage d'implémentation.

Malgré tout, l'héritage d'implémentation n'est en aucun cas compatible avec l'approche naturelle de la spécialisation qui est selon nous la seule approche correcte de la

spécialisation.

Héritage de mixins

La notion de *mixin* est plutôt large [Stefik et Bobrow, 1986; Bracha et Cook, 1990; Bracha, 1992; Boyen *et al.*, 1994; Myers, 1995; Limberghen et Mens, 1996; Findler et Flatt, 1998; Bono *et al.*, 1999; Ancona *et al.*, 2000], et l'usage du terme varie de définitions formelles à des définitions beaucoup moins formelles.

Dans la plupart des approches, un mixin peut-être vue comme une *classe abstraite* mais toutefois moins abstraite que les interfaces de JAVA (un mixin, comme une interface JAVA n'a pas d'instance directe mais il peut définir des méthodes concrètes, voire des attributs). Les mixins sont généralement présentés comme un moyen de résoudre les problèmes de l'héritage multiple (ou de restreindre son usage à des cas non conflictuels). Un mixin a également pour rôle d'encapsuler des fonctionnalités présumées orthogonales aux classes et qui peuvent être combinées (mixées) avec elles sans risque de conflit.

En RUBY, la notion de *module* (qui ne correspond en rien avec celles de la section 2.7 page 27 et du chapitre 4) est en réalité un mixin. De plus, dans le méta-niveau de RUBY, la classe `Module` des modules subsume la méta-classe `Class`.

Les *traits* [Ducasse *et al.*, 2006] peuvent également être considérés comme des sortes de mixins (toutefois un trait n'a pas d'attribut).

En règle générale, les mixins ne sont qu'un moyen de factoriser et de réutiliser du code à travers plusieurs classes que cela soit via le mécanisme d'héritage ou via un autre mécanisme, mais dans tous les cas, comme pour l'héritage d'implémentation, l'approche naturelle de la spécialisation semble totalement laissée de côté.

Transitivité explicite

La *transitivité explicite* consiste à apporter une sémantique différente suivant le fait que les arcs de transitivité soient explicites ou non dans la définition d'une classe.

Ainsi, dans l'exemple suivant :

```
class A ... end
class B inherit A  ... end

class C1 inherit B end
class C2 inherit A inherit B end
```

cela correspond à traiter de façon différente l'héritage de la classe C1 et de la classe C2. Nous verrons des exemples concrets dans la section 3.6.4.

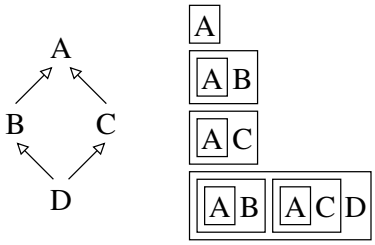


FIG. 3.5: Instances de classes en héritage répété

Chemins d'héritage

De façon plus générale, les *chemins d'héritage* consistent non pas à considérer la relation de spécialisation ($\prec$) comme la fermeture transitive de la relation de spécialisation déclarée ($\prec_{\text{decl}}$) mais comme l'ensemble des chemins qui permettent de relier deux classes par la spécialisation déclarée. La différence entre les deux points de vue apparaît de façon symptomatique dans les cas d'héritage multiple par l'existence de l'héritage dit « répété » : un héritage répété apparaît dès qu'il existe plusieurs chemins de spécialisation déclarée entre deux classes.

Or cette notion d'héritage répété n'a pas de sens avec l'approche naturelle de la spécialisation. Par exemple, dans le système suivant :

> Les philosophes sont des hommes.
> Les grecs sont des hommes.
> Les disciples de Socrate sont des philosophes et aussi des grecs.

prétendre que les disciples de Socrate sont deux fois plus des hommes que les autres philosophes n'a pas de sens.

L'approche des chemins d'héritage apparaît plus particulièrement lorsque la spécification du langage est dirigée par son implémentation. C'est de façon évidente le cas de C++ dans laquelle l'héritage est implicitement[11] considéré comme une inclusion syntaxique : une instance d'une classe *contient* une instance de chacune de ses super-classes déclarées. Dans l'exemple de la figure 3.5, la classe `B` est sous-classe déclarée de la classe `A` donc une instance de la classe `B` contient une instance de la classe `A`. De même, la classe `D` est une sous-classe déclarée des classes `B` et `C`, donc une instance de la classe `D` contient une instance des classe `B` et `C` et donc deux fois une instance de la classe `A`.

Conclusion sur les variations

L'existence de ces variations autour de la notion de spécialisation pose de nombreux problèmes. En premier lieu, elles sont incompatibles avec l'approche naturelle — qui est

[11] C'est-à-dire sans l'utilisation du mot clé `virtual`.

selon nous le principal avantage de l'approche objet. En second lieu, elles compliquent inutilement le mécanisme d'héritage en particulier lorsque la spécification du langage considéré mélange plusieurs variations de la spécialisation, comme c'est le cas en C++ où l'héritage dit « répété » peut être combiné avec celui dit « virtuel » qui correspond à l'approche naturelle.

Il existe une dernière forme de variation autour de la spécialisation qui consiste à définir pour chaque classe un ordre total de préférence de ses super-classes. Elle est généralement syntaxiquement mise en œuvre par l'ordre dans laquelle les super-classes sont déclarées lors de la définition d'une sous-classe. Toutefois cette variation n'est pas incompatible avec l'approche naturelle de la spécialisation : elle se contente d'ajouter une *sémantique additionnelle* que l'on peut naïvement associer à un point de vue préférentiel : les disciples de Socrate sont-ils avant tout des philosophes ou des grecs ? Comme nous le voyons dans la section 3.6.4 (page 68), cette sémantique additionnelle est utilisée dans les langages comme CLOS pour ses techniques de linéarisation mais aussi de façon abusive par PYTHON ou OBJECTIVE-CAML.

3.4 Propriétés globales

Afin de modéliser le rapport entre classes et propriétés globales et plus particulièrement l'héritage de celles-ci, notre point de départ est à nouveau l'approche naturelle de la relation de spécialisation que nous illustrons par le système suivant :

> Les hommes sont mortels.
> Or les grecs sont des hommes.
> Donc les grecs sont mortels.

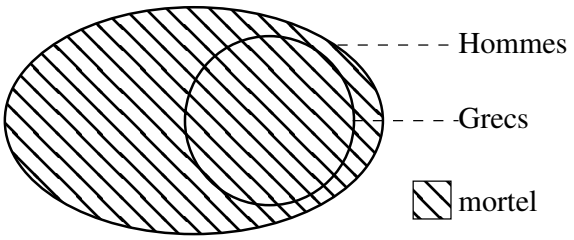
où « homme » et « grec » correspondent à des classes et où « mortel » correspond à une propriété globale[12].

Vis-à-vis de la relation entre classes et propriétés globales, cela vient à poser la proposition suivante :

Principe 7 (Contravariance des intensions[13]) *Si une classe peut répondre à un message donné, toutes ses sous-classes le peuvent également : une classe hérite les propriétés globales de ses super-classes.*

La figure suivante illustre l'exemple précédent de façon ensembliste :

[12] Si l'on considère « mortel » comme un substantif, la notion « mortel » est une classe.
[13] L'*intension* d'une classe est l'ensemble de ses propriétés.



3.4.1 Formalisme

Etant donnée une classe $c \in X$, nous notons $G_c \subseteq G$ l'ensemble des propriétés globales de la classe c. L'héritage des propriétés globales est une simple inclusion des ensembles de propriétés :

$$\forall c, c' \in X \quad c' \prec c \Rightarrow G_{c'} \supseteq G_c \ . \tag{3.3}$$

On peut alors séparer les propriétés globales d'une classe $c \in X$ en deux ensembles disjoints : celui que nous notons $G_{\uparrow c}$ qui correspond aux propriétés globales *héritées* d'une super-classe de c et celui que nous notons G_{+c} qui correspond aux propriétés globales *introduites* par la classe c. Nous avons donc le système d'équation suivant :

$$G_{+c} = \text{intro}^{-1}(c) \ , \tag{3.4}$$

$$G_c = G_{\uparrow c} \uplus G_{+c} = \biguplus_{c \preceq c'} G_{+c'} \ , \tag{3.5}$$

$$G_{\uparrow c} = \bigcup_{c \prec_{\text{d}} c'} G_{c'} = \biguplus_{c \prec c'} G_{+c'} \ , \tag{3.6}$$

$$G = \bigcup_{c \in X} G_c = \biguplus_{c \in X} G_{+c} \ . \tag{3.7}$$

Rappel : Tous les G_{+c} sont disjoints deux à deux puisque dans le méta-modèle les propriétés globales ne sont introduites que dans une seule classe.

3.4.2 Nom de propriété et envoi de message

Nous considérons les envois de message dans leur généralité : ils regroupent bien évidemment les invocations de méthodes (procédures et fonctions) mais aussi les accès aux attributs.

Les envois de message sont localisés dans le code source par des *sites d'appel* qui peuvent prendre plusieurs formes syntaxiques en fonction des langages et des utilisations. Sans perte de généralité, nous représentons un site d'appel par la syntaxe pointée habituelle `x.p` où `x` représente le *receveur* et `p` le *sélecteur*. Le receveur peut être une

expression explicite ou implicite, dans ce dernier cas, le receveur est le receveur courant (suivant les langages celui-ci s'appelle `self`, `this`, `Current`, etc.). Par exemple, en SMALLTALK, l'accès aux attributs se fait systématiquement avec le receveur implicite tandis que l'invocation de méthodes se fait avec un receveur explicite.

Nous excluons de notre champ d'étude les appels statiques, c'est-à-dire les envois de message dont la résolution est statiquement déterminée et donc indépendante du type dynamique du receveur. En C++ par exemple, une invocation de méthode est statique si l'une des conditions suivantes est vérifiée :

- le receveur n'est ni un pointeur ni une référence sur une instance d'une classe ;
- la méthode est définie sans le mot clé `virtual` ;
- le receveur est le receveur courant (nommé `this` en C++) et le site d'appel est situé dans un constructeur.

Le sélecteur doit permettre de distinguer de façon non ambiguë une propriété globale. Dans le cas contraire, le système doit pouvoir détecter l'ambiguïté et, soit la lever, soit déclencher une erreur. Malheureusement, la notion de sélecteur varie profondément suivant les langages et peut prendre en compte : la forme syntaxique, un nom de propriété (un symbole, éventuellement sensible ou non à la casse), le nombre voire le type des arguments des méthodes (c'est le cas en C++ et JAVA), etc. De plus, des règles plus ou moins complexes de surcharge statique (cf. section 3.5.4 page 63), de masquage, d'espace de nom ou de protection (ou *export*) [Ardourel, 2002] peuvent exister. C'est pourquoi, dans un souci de simplification, nous nommons « nom » le sélecteur d'une propriété globale et nous considérons que celui-ci est indépendant des classes. La fonction de nommage nom : $G \rightarrow N$ permet d'associer une propriété globale au nom qui lui a été donné.

Le besoin de distinguer les propriétés globales de leurs noms est dû au fait que les propriétés globales correspondent à une notion de « même idée ». Par exemple, la propriété locale `hauteur` de la classe `Personne` et la propriété locale `hauteur` de la classe `Triangle` ne peuvent pas appartenir à la même propriété globale puisque que la hauteur d'une personne et celle d'un triangle ne correspondent pas à une même idée.

C'est pourquoi dans un langage à typage statique, nous considérons qu'une propriété globale n'est introduite que dans une seule classe : c'est la seule façon de garantir le respect de l'intention du programmeur. Ainsi, au niveau d'un site d'appel de la forme `x.p` la propriété globale est déterminée à la fois par le type statique du receveur `x` et par le sélecteur `p`.

En typage dynamique, le receveur n'intervient statiquement en rien pour déterminer la propriété globale associée à un site d'appel : une propriété globale est entièrement déterminée par son nom. Il n'est donc pas possible de distinguer deux propriétés globales homonymes[14]. La fonction de nommage des propriétés globales n'a donc d'intérêt que

[14]C'est une des supériorité du typage statique sur le typage dynamique dans le cadre des langages à objets.

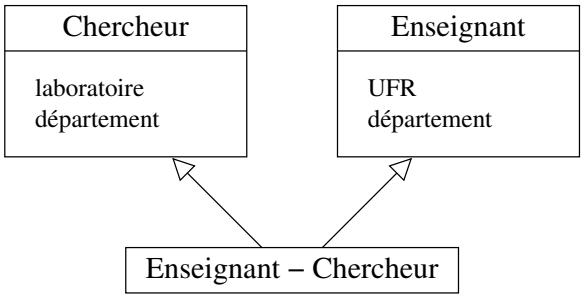


FIG. 3.6: Conflit de propriétés globales en héritage multiple

dans le cadre des langages à typage statique.

3.4.3 Noms et héritage multiple

Les choses se compliquent en héritage multiple puisque des conflits peuvent apparaître lors de l'héritage des propriétés globales.

Définition 4 (Conflit de propriétés globales[15]) *Un* conflit de propriétés globales *survient lorsqu'une classe $c \in X$ spécialise deux super-classes directes c_1 et $c_2 \in X$ distinctes qui possèdent des propriétés globales $g_1 \in G_{c_1}$ et $g_2 \in G_{c_2}$ également distinctes mais homonymes.*

L'exemple de la figure 3.6 montre deux classes (`Chercheur` et `Enseignant`) possédant toutes deux une propriété globale nommée `departement`, l'une désignant un département dans un laboratoire de recherche, l'autre un département d'enseignement dans une université. Il est alors attendu, selon le principe 7, que la sous-classe commune `Enseignant-Chercheur` hérite les propriétés globales de ses super-classes, or le nom `departement` serait ambigu dans le contexte de la sous-classe commune.

Dans tous les cas, un conflit de propriétés globales n'est qu'un problème de vocabulaire et un renommage systématique garantirait l'absence de conflits de noms. Ce renommage peut être libre ou spécifié, local ou global en fonction des caractéristiques qu'offre le langage considéré.

Aucun renommage (c'est-à-dire erreur) : la spécification du langage ne fournit pas de moyen de résoudre les conflits de noms.

Cette situation force le programmeur à renommer au moins l'une des deux propriétés globales dans tous les programmes qui l'utilisent ce qui peut impliquer la modification d'un grand nombre de classes (avec les erreurs potentielles inhérentes) voire s'avérer impossible si ces classes ne sont pas modifiables (code source indisponible par exemple).

[15] Appelé *conflit de nom* dans [Ducournau *et al.*, 1995; Privat et Ducournau, 2005b].

Désignation explicite : c'est-à-dire une écriture syntaxique étendue alternative qui juxtapose au nom de la propriété le nom d'une classe dans laquelle le nom de la propriété n'est pas ambigu. Cette classe peut être par exemple la classe qui introduit la propriété globale.

Dans la spécification de la classe `Enseignant-Chercheur` de l'exemple, `Enseignant::departement` désignerait la propriété globale connue sous le nom `departement` dans la classe `Enseignant` et `Chercheur::departement` désignerait la propriété globale connue sous le nom `departement` dans la classe `Chercheur`. Cette solution est utilisée dans C++ pour les attributs[16].

Cette solution possède au moins deux inconvénients au niveau de la syntaxe du langage. Le premier est une écriture plus lourde qui est nécessairement utilisée dans la classe en conflit mais également dans toutes ses sous-classes. Le second est le non respect du principe d'anonymat qui veut que les références explicites aux super-classes apparaissent le moins possible afin d'améliorer la souplesse du code.

Renommage local : Afin de garantir que, dans une classe, un nom ne désigne qu'une seule propriété globale, le renommage local permet de changer la désignation des propriétés dans une classe et dans ses sous-classes.

Dans la classe problématique de l'exemple, on peut renommer `departement` hérité de `Enseignant` en `dept-ens` et `departement` hérité de `Chercheur` en `dept-rech`. Ainsi `departement` dans `Chercheur` et `dept-rech` dans `Enseignant-Chercheur` désignent la même propriété globale et comme attendu, dans la classe `Enseignant-Chercheur`, `dept-rech` et `dept-ens` désignent deux propriétés globales distinctes.

Cette solution est utilisée dans EIFFEL mais possède l'inconvénient majeur de modifier le vocabulaire en fonction des classes.

Unification : Les langages dynamiques (CLOS, PYTHON, etc.), JAVA via ses interfaces et C++ pour les méthodes considèrent que si deux propriétés globales sont homonymes alors elles ne sont pas distinctes. Il n'y a donc pas de conflit de propriétés globales possible et les ambiguïtés de l'héritage multiple sont reportées sur l'héritage de propriétés locales (voir section 3.6.3).

Selon nous, cette solution est la pire de toutes puisqu'elle ne permet pas de respecter l'intention du programmeur d'avoir deux propriétés globales distinctes — dans l'exemple de la figure 3.6, les deux « départements » représentent bien deux concepts distincts. Si l'intention du programmeur était un seul concept, celui-ci aurait défini une super-classe commune qui introduit la propriété globale correspondante à ce concept.

Le fait est que, malgré tout, cette solution est celle appliquée par quasiment tous les langages à objets. L'explication en est triviale pour les langages à typage dynamique dans lesquels les propriétés globales sont entièrement déterminées par

[16] Pour les méthodes, l'opérateur :: correspond à un appel statique : c'est donc une propriété locale qui est désignée et non la propriété globale.

leur nom. Par contre, pour les langages à typage statique (C++, Eiffel, Java, Objective-Caml) il n'existe pas de réelle justification si ce n'est peut-être une volonté « de faire comme les autres langages ».

De base, aucune solution au conflit de propriétés globales n'est prise en compte par le méta-modèle. Toutefois, les modifications à y apporter ne seraient pas trop difficiles à spécifier. Dans le cas de la désignation explicite, il faut considérer la gestion des noms étendus. Dans le cas du renommage, la fonction de nommage doit prendre en compte la classe qui sert de contexte, la signature de la fonction nom changerait en $X \times G \rightarrow N$. Dans le cas de l'unification, la multiplicité de l'association *introduit* changerait puisqu'une propriété globale pourrait être introduite par plusieurs classes.

Quant à la question de « se mouiller » et de choisir entre la désignation explicite et le renommage, nous avons choisi pour PRM le renommage car il semble présenter moins d'inconvénients, surtout si le langage considéré inclut également le raffinement de classes — cf. chapitre suivant. Toutefois, l'utilisation du renommage se doit d'être strictement encadré afin d'éviter les abus, d'une part en limitant l'utilisation du renommage au cas de conflits de propriété globale et d'autre part en spécifiant le renommage de façon claire : c'est la propriété globale que l'on renomme.

3.4.4 Variations autour des propriétés globales

La contribution principale du méta-modèle que nous proposons est de rendre non ambiguë la spécification des langages lorsque des problèmes de noms surviennent, c'est-à-dire lorsqu'un même nom ne permet pas de distinguer dans un même contexte plusieurs propriétés. Ces conflits se produisent en héritage multiple (conflits de propriétés globales) mais également avec la surcharge statique [Meyer, 2001]. En l'absence de ces problèmes, lors de la spécification d'un langage ou de la réalisation d'un compilateur, il n'y a pas de nécessité (hors celle conceptuelle, qui d'une certaine façon nous guide depuis le début de ce mémoire) de distinguer les propriétés globales des noms des propriétés. Dans cette section, nous passons en revue le statut de ce qui correspondrait aux propriétés globales dans quelques langages à objets.

Dans la terminologie Smalltalk , les *méthodes* et les *sélecteurs* correspondent respectivement aux propriétés locales et globales. Toutefois, les sélecteurs ne sont réifiés que sous la forme de symboles : les sélecteurs ne possèdent pas en tant que tels d'informations telles que les classes avec lesquelles ils sont associés. Il n'y a pas non plus d'équivalent pour les attributs — *variable d'instance* selon la terminologie Smalltalk.

En Clos , les *méthodes* et les *fonctions génériques* correspondent aux propriétés locales et globales. Elles sont réifiées (elles ont un statut de classe) mais à cause de la *sélection multiple*, Clos n'entre pas tout à fait dans le cadre de notre méta-modèle : les propriétés n'appartiennent pas à une classe particulière et ne sont pas héritées de la façon habituelle. Pour les attributs, *slots* dans la terminologie Clos, ils sont réifiés

par deux sortes de *description de slot* qui peuvent être *directes* (*direct slot description*) ou *effectives* (*effective slot description*). Toutefois, aucune des deux sortes de slots ne correspond aux propriétés globales.

Malgré le cas des attributs, CLOS et SMALLTALK semblent être les seuls langages qui possèdent une terminologie qui permet partiellement de distinguer les deux notions clés que nous appelons propriété globale et propriété locale. Dans les autres langages, un seul mot (*méthode*, *propriété*, *feature*, etc.) correspond aux deux notions.

En JAVA , la notion de propriété globale n'apparaît pas dans les spécifications du langage. De plus, il n'y a pas de réification des propriétés globales, ni dans les capacités d'introspection (package `java.lang.reflect`), ni dans les extensions réflexives du langage comme OPENJAVA ou JAVASSIST [Chiba, 1998].

En EIFFEL , la notion de *propriété* (*feature*) regroupe attribut et méthode mais sans distinguer les propriétés locales et globales. Toutefois, une notion de *feature seed* peut se rapprocher des propriétés globales : vis-à-vis de notre modélisation, elle correspond à la propriété locale qui introduit une propriété globale (voir la section 3.5.2).

Il est impossible de parler de méta modélisation en objet sans parler de UML. Vis-à-vis des propriétés, le méta-modèle UML ne distingue pas les propriétés globales : aucune entité UML n'y correspond. Le *features diagram* de UML 2.0 [OMG, 2004, page 27] ne décrit qu'une sorte d'entité nommé *feature*. Une raison possible est l'absence de spécification rigoureuse de l'envoi de message et de l'héritage multiple.

Les travaux de [Huchard, 2003] se basent sur une notion appelée *propriété générique*. Celle-ci représente un ensemble de propriétés locales sémantiquement liées et munie d'un ordre partiel de spécialisation sur cet ensemble. D'une certaine façon, ces propriétés génériques et nos propriétés globales ont en commun l'objectif de regrouper les propriétés locales qui correspondent à une « même idée ». La différence est que nos propriétés globales correspondent exactement à ce que le programmeur a exprimé sans chercher à deviner quelle était son intention alors que ces propriétés génériques sont extraites des hiérarchies de classes par diverses heuristiques, l'objectif étant de « deviner » la réelle intention du (ou des) programmeurs pour refactoriser cette hiérarchie en une meilleure hiérarchie. Toutefois, les propriétés génériques et les propriétés globales se confondent une fois qu'une hiérarchie a été refactorisée.

3.5 Propriétés locales

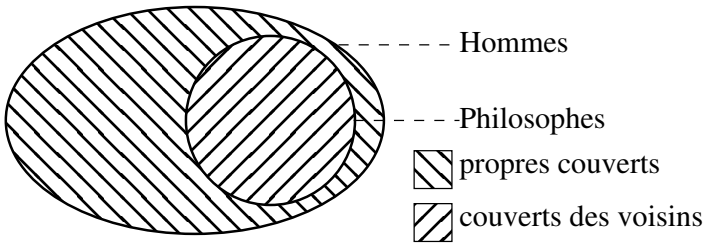
L'approche naturelle de la définition et de la redéfinition des propriétés locales est plus difficile à identifier que celles de la spécialisation de classes ou de l'héritage de propriétés globales. Malgré tout, on peut tenter d'illustrer cette approche avec le système suivant :

> Les philosophes sont des hommes.
> Les hommes mangent avec leurs propres couverts.

Les philosophes mangent avec les couverts de leurs voisins[17].

Dans ce système, une interprétation universelle stricte fait long feu : les hommes mangent avec leurs propres couverts or les philosophes sont des hommes donc les philosophes mangent avec leurs propres couverts ; or les philosophes mangent avec les couverts de leurs voisins donc contradiction ; le système est incohérent.

Le cerveau humain n'aime pas les incohérences et tente généralement de les lever. La solution ici pour lever la contradiction consiste à considérer que la seconde proposition doit être interprétée en tant que « les hommes qui ne sont pas philosophes mangent avec leurs propres couverts ». On peut facilement illustrer cette idée intuitive du « sauf » par la figure suivante :



À partir de cette approche naturelle, on peut poser le principe suivant :

Principe 8 (Propriétés locales) *Définir une propriété locale dans une classe consiste à définir la réponse pour une propriété globale dans cette classe et dans les sous-classes qui ne définissent pas une propriété locale plus spécifique.*

3.5.1 Déclaration de propriétés locales

Dans les langages de programmation, une propriété locale est définie dans une classe, appelée *classe de définition*. Cette classe peut être explicite, la définition d'une propriété est alors généralement syntaxiquement placée à l'intérieur de la définition d'une classe ; par exemple en JAVA :

```
class Toto {
    public int tata(void) { ... }
}
```

Cette classe peut également être implicite. Par exemple en RUBY, les propriétés définies en dehors d'une définition de classe sont implicitement des propriétés définies dans `Object`, la classe racine de la hiérarchie[18]. Toutefois, nous considérons que la définition d'une classe $\langle$`classname`, `supernames`, `localdef`$\rangle$ nous permet d'identifier les propriétés locales définies dans cette classe (`localdef`).

[17] Selon Edsger Dijkstra [Silberschatz et Peterson, 1988].

[18] C'est également le cas en PRM (cf. annexe A).

Formellement, soit une classe $c \in X$, nous désignons par $L_c \in L$ les propriétés locales définies dans c. De façon inverse, la fonction def : $L \rightarrow X$ associe à une propriété locale sa classe de définition.

Remarque : En toute généralité, nous ne faisons aucune hypothèse sur la nature des propriétés locales. Celles-ci peuvent être des méthodes, des attributs, etc. De plus, dans les langages à typage statique, certaines méthodes peuvent être déclarées *abstraites* c'est-à-dire ne pas posséder d'implémentation. On trouve aussi les termes de *virtuelles pures* en C++ ou retardées (`deferred`) en EIFFEL. En particulier, c'est le cas en JAVA pour toutes les méthodes déclarées dans des interfaces. Celles-ci sont néanmoins considérées comme des propriétés locales à part entière : elles possèdent une classe de définition, une signature, elles sont héritables, etc.

3.5.2 Introduction des propriétés globales

Vis-à-vis des propriétés globales, la définition dans une classe $c \in X$ d'une propriété locale $l \in L_c$ peut correspondre à deux mécanismes:

- l'*introduction* d'une nouvelle propriété globale, c'est-à-dire la capacité donnée à c de répondre à un message que n'avaient pas ses super-classes, l étant définie comme la réponse à ce message ;
- la *redéfinition* d'une propriété globale héritée $g \in G_{\uparrow c}$, c'est-à-dire l'assignation d'une réponse particulière l lors de la réception d'un message associé à la propriété globale g.

Par exemple, soit le listing PRM suivant :

```
class A
        def toto: Int
        do
                return 1
        end
end
class B
        inherit A
        def toto: Int
        do
                return 2
        end
end
let o: A
...
o.toto
```

La propriété locale `toto` définie dans la classe `A` introduit une nouvelle propriété globale : le site d'appel `o.toto` y est associé et la réponse à un envoi de message sera 1 si `o` est

une instance directe de `A`. La propriété locale `toto` définie dans la classe `B` redéfinit cette propriété globale héritée de `A` : vis-à-vis du site d'appel `o.toto`, la réponse à un envoi de message sera 2 si `o` est une instance directe de `B`.

Ainsi, pour une classe $c \in X$, nous notons $L_{\uparrow c}$ l'ensemble des propriétés locales qui redéfinissent une propriété globale héritée et L_{+c} l'ensemble des propriétés locales qui introduisent une nouvelle propriété :

$$L_c = L_{\uparrow c} \uplus L_{+c} \ . \tag{3.8}$$

De façon simplifiée, la distinction entre les deux mécanismes se fait en fonction du nom de la propriété locale et du nom des propriétés globales héritées :

Définition 5 (Introduction ou redéfinition ?) *Dans une classe $c \in X$ la définition d'une propriété locale $l \in L_c$ introduit une propriété globale si et seulement si:*

$$l \in L_{+c} \quad \overset{\text{def}}{\Longleftrightarrow} \quad \forall g \in G_{\uparrow_c}, \mathrm{nom}(g) \neq \mathrm{nom}(l) \ . \tag{3.9}$$

Dans le cas contraire, l'absence de conflits de propriétés globales ou leurs résolutions doit nous permettre d'identifier une unique propriété globale $g \in G_{\uparrow_c}$ homonyme à l. C'est cette propriété globale qui est redéfinie par l.

Dans les deux cas, la propriété locale l est associée à la propriété globale g introduite ou redéfinie :

$$\mathrm{glob}(l) = g \ .$$

À la fin, on a :

$$\mathrm{glob}(L_{\uparrow c}) \subseteq G_{\uparrow c} \ , \tag{3.10}$$

$$\mathrm{glob}(L_{+c}) = G_{+c} \ . \tag{3.11}$$

3.5.3 Redéfinition

Définition 6 (Redéfinition) *Nous notons $\ll$ la relation de redéfinition entre propriétés locales définie par :*

$$l' \ll l \quad \overset{\text{def}}{\Longleftrightarrow} \quad \mathrm{glob}(l') = \mathrm{glob}(l) \quad \wedge \quad \mathrm{def}(l') \prec \mathrm{def}(l) \ . \tag{3.12}$$

La redéfinition entre propriétés locales est un ordre partiel strict. De plus, nous notons $\ll_{\mathrm{d}}$ la relation de redéfinition directe, celle-ci étant définie par la réduction transitive de la relation $\ll$.

Si pour deux propriétés locales l et $l' \in L$, $l' \ll l$, nous disons que l' est *plus spécifique* que l, et que l est *plus générale* que l'.

En fonction du langage de programmation considéré, le champ des possibles redéfinitions est extrêmement variable. Par exemple, en C++, en JAVA et en SMALLTALK, les attributs ne peuvent pas être redéfinis contrairement à EIFFEL. De plus, certaines propriétés locales habituellement redéfinissables peuvent être spécifiquement déclarées non redéfinissables — via l'utilisation du mot clé `final` en JAVA ou `frozen` en EIFFEL (cf. chapitre 5).

3.5.4 Variation : la surcharge statique

Dans certains langages, des écritures syntaxiques, appelées « surcharge statique » dans la littérature, ressemblent à de la redéfinition mais n'en sont pas.

Par exemple, en C++, d'une part les attributs ne peuvent pas être redéfinis et d'autre part, le type statique des paramètres des méthodes sont pris en compte dans le nom des propriétés. Ainsi, dans le listing suivant :

```
class A {
public:
        int x;
        virtual int toto(int);
};
class B: public virtual A {
public:
        int x;
        virtual int toto(double);
};
```

La classe `B` ne redéfinit ni l'attribut `x` ni la méthode `toto` définis dans la classe `A` mais introduit deux nouvelles propriétés globales. À la fin, la classe `B` possède quatre propriétés globales.

3.6 Envoi de message et héritage des propriétés locales

Le mécanisme de l'envoi de message impose une résolution non ambiguë des liaisons tardives qui passe par la sélection des propriétés locales :

Définition 7 (Sélection) *La sélection d'une propriété locale consiste à déterminer une unique propriété locale pour chaque propriété globale d'une classe.*

Dans toute cette section, nous considérons d'une part que toutes les propriétés locales sont traitées de la même manière vis-à-vis de la sélection et d'autre part que le mécanisme de sélection ne « crée » pas de propriétés locales : seules les propriétés locales définies

peuvent être sélectionnées. Nous appelons *résolution par défaut* un tel mécanisme de sélection.

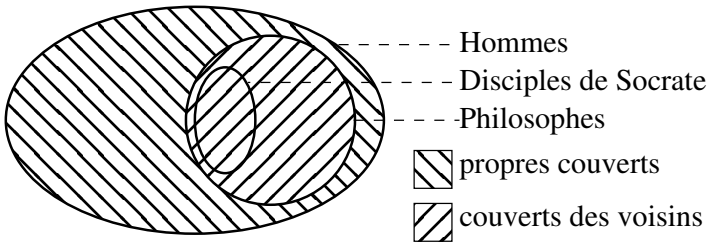
Lorsque la classe définit (ou redéfinit) une propriété locale associée à la propriété globale, c'est celle-ci qui doit être sélectionnée. Autrement, une propriété locale doit être héritée. Cet héritage impose deux contraintes triviales :

– la propriété locale héritée doit appartenir à la propriété globale ;
– la propriété locale héritée doit être définie dans une des super-classes de la classe.

La troisième contrainte imposée à l'héritage des propriétés locales est moins triviale. Celle-ci relève de l'approche naturelle et nous l'illustrons par le système suivant :

> Les philosophes sont des hommes.
> Les disciples de Socrate sont des philosophes.
> Les disciples de Socrate sont des hommes (déduite des deux précédentes propositions).
> Les hommes dînent avec leurs propres couverts.
> Les philosophes dînent avec les couverts de leurs voisins.

La question que l'on se pose concerne l'identité des propriétaires des couverts dont se servent les disciples de Socrate pour dîner. Comme nous l'avons présenté au début de la section 3.5, l'interprétation sans incohérence du sous-système constitué des propositions 1, 4 et 5, nécessite que la proposition 5 soit interprétée par « Les hommes qui ne sont pas philosophes dînent avec leurs propres couverts ». Une fois cela fait, il est facile de déduire que les disciples de Socrate dînent avec les couverts de leurs voisins. Ce qui correspond bien à l'intuition naturelle que la figure suivante illustre :



Ainsi, nous pouvons déduire de l'approche naturelle le principe suivant :

Principe 9 (Monotonie) *Pour un message donné, la réponse d'une classe ne peut pas être moins spécifique que la réponse d'une de ses super-classes.*

3.6.1 Formalisme

Nous notons sel : $X \times G \rightarrow L \cup \{\perp\}$ la fonction qui sélectionne dans une classe la propriété locale associée à une propriété globale ou qui retourne $\perp$ en cas d'erreur. Toutefois, nous considérons que la sélection ne peut pas échouer pour une classe et une propriété globale de cette classe :

$$\forall c \in X, g \in G, \quad g \in G_c, \Leftrightarrow \text{sel}(x, g) \neq \perp \ . \tag{3.13}$$

Les deux contraintes triviales sur la sélection imposent :

$$\forall c \in X, g \in G_c, \quad c \preceq \mathrm{def}(\mathrm{sel}(x, g)) \quad \wedge \quad \mathrm{glob}(\mathrm{sel}(x, g)) = g \ . \tag{3.14}$$

Le principe de monotonie impose :

$$\forall c, c' \in X, g \in G_c, c' \prec c \Rightarrow \mathrm{sel}(c, g) \not\ll \mathrm{sel}(c', g) \ . \tag{3.15}$$

Au final, nous notons spec : $X \times G \to \mathcal{P}(L)$ la fonction qui associe à une classe et à une propriété globale les propriétés locales qui respectent les contraintes de la sélection, c'est-à-dire « les plus spécifiques » :

$$\mathrm{spec}(c, g) = \min_{\ll} (\{l \in L \mid (\mathrm{def}(l) \preceq c) \wedge (\mathrm{glob}(l) = g)\}) \ . \tag{3.16}$$

3.6.2 Héritage simple

Dans les langages en spécialisation simple, le mécanisme d'héritage mis en œuvre fait l'unanimité bien qu'il puisse s'exprimer de quatre façons différentes (mais équivalentes) :

- la propriété locale héritée est celle sélectionnée dans sa super-classe directe ;
- la propriété locale héritée est celle sélectionnée dans sa super-classe déclarée ;
- la propriété locale héritée est la plus spécifique parmi celles définies dans ses super-classes ;
- la propriété locale héritée est la plus spécifique parmi celles sélectionnées dans ses super-classes.

Les deux premières sont trivialement équivalentes puisqu'en spécialisation simple, la super-classe directe et la super-classe déclarée désignent la même classe. Les deux dernières sont également trivialement équivalentes puisqu'une propriété locale sélectionnée dans une classe est alors forcément définie dans cette classe ou dans une de ses super-classes, et réciproquement.

3.6.3 Héritage multiple

En héritage multiple, les choses se compliquent un peu. Néanmoins l'héritage multiple des propriétés locales n'est pas incompatible avec l'approche naturelle de la spécialisation. Ainsi, dans le système suivant :

Les philosophes sont des hommes.
Les grecs sont des hommes.
Les disciples de Socrate sont des philosophes.
Les disciples de Socrate sont des grecs.
Les hommes dînent avec leurs propres couverts.
Les philosophes dînent avec les couverts de leurs voisins.

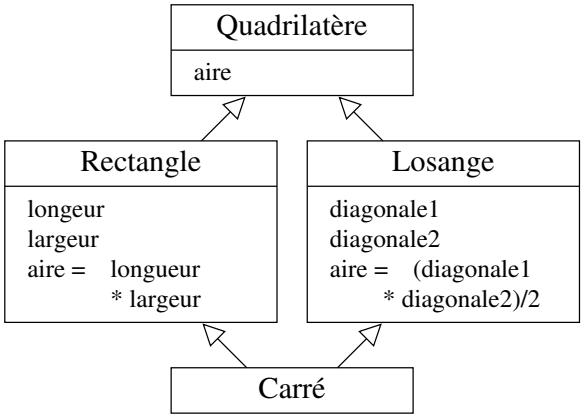
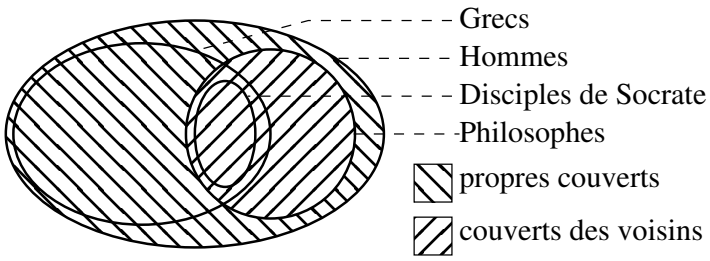


FIG. 3.7: Conflit de propriétés locales en héritage multiple

l'approche naturelle permet de déterminer que les grecs dînent avec leurs propres couverts et que les disciples de Socrate dînent avec les couverts de leurs voisins. Comme on peut l'illustrer par la figure suivante, par rapport au système présenté au début de la section 3.6, le fait que les disciples de Socrate soient des grecs ne change rien à leur manie d'utiliser des couverts qui ne sont pas à eux :



Toutefois, contrairement à l'héritage simple, la résolution par défaut ne permet pas de résoudre tous les cas d'héritage multiple.

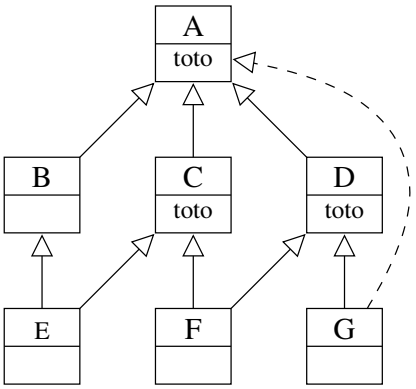
Définition 8 (Conflit de propriétés locales) *Un* conflit de propriétés locales *survient lorsque pour une classe $c \in X$ et une propriété globale $g \in G_c$, $|\text{spec}(c, g)| > 1$.*

L'exemple de la figure 3.7 illustre un cas de conflit[19] de propriétés locales. Il montre deux classes (`Rectangle` et `Losange`), toutes deux redéfinissant la méthode `aire` dont la propriété globale a été introduite dans la classe `Quadrilatere`. Dans la sous-classe commune `Carre`, laquelle doit être héritée ?

[19]Toutefois, d'un point de vue purement géométrique, il ne s'agit pas d'un vrai conflit puisque le résultat « devrait » être le même dans les deux cas.

Classe	Super-classes déclarées	Propriété `toto`
A	∅	introduite
B	A	héritée
C	A	redéfinie
D	A	redéfinie
E	B, C	héritée
F	C, D	héritée
G	A, D	héritée

(a) Définitions



(b) Représentation graphique

FIG. 3.8: Exemple pour la comparaison des mécanismes d'héritage

Contrairement au conflit de propriétés globales, il n'y a pas de solution intrinsèque à ce problème (comme un renommage massif).

Le langage peut apporter une sémantique additionnelle pour s'affranchir de ce problème — voir sections 3.6.4 et 3.7. Dans le cas contraire (ou dans le cas où la sémantique additionnelle n'a pas non plus suffit à résoudre le conflit), une erreur doit être signalée dans le but de forcer le programmeur à modifier le programme et résoudre lui-même le conflit. Généralement, ceci est fait en redéfinissant la propriété globale par une nouvelle propriété locale dans la classe où le conflit apparaît. Toutefois, d'autres moyens de résolution peuvent également exister, par exemple dans le cas des méthodes en EIFFEL, le programmeur peut sélectionner la propriété locale à hériter parmi celles en conflits par un jeu complexe de clauses d'héritage `undefine` (cf. section 3.7.2).

3.6.4 Variations autour de la résolution par défaut

Afin de comparer les différents mécanismes d'héritage, nous définissons sept classes (cf. figure 3.8) dans les langages C++, Eiffel, Clos, Python et Objective-Caml.

C++

Par défaut, C++ considère un héritage répété et des méthodes non soumises à l'envoi de message (mais héritées). Nous spécifions systématiquement le mot clé `virtual` dans ses deux acceptions afin de retrouver l'approche naturelle de la spécialisation.

```
class A { public: virtual void toto(); };
class B: virtual public A { };
class C: virtual public A { public: virtual void toto(); };
class D: virtual public A { public: virtual void toto(); };
class E: virtual public B, virtual public C { };
class F: virtual public C, virtual public D { };
class G: virtual public A, virtual public D { };
```

Le mécanisme d'héritage de C++ correspond à la résolution par défaut sans sémantique additionnelle : `B`, `E` et `G` héritent la propriété locale la plus spécifique (respectivement celles définies dans `A`, `C` et dans `D`) tandis que pour la classe `F`, un conflit est signalé entre les propriétés redéfinies dans `C` et dans `D`.

Eiffel

```
class A feature toto is do ... end end
class B inherit A end
class C inherit A redefine toto end feature toto is do...end end
class D inherit A redefine toto end feature toto is do...end end
class E inherit B C end
class F inherit C D end
class G inherit A D end
```

Le mécanisme d'héritage d'Eiffel ne correspond pas à la résolution par défaut. En effet, la règle d'Eiffel est d'hériter la propriété locale qui fait l'unanimité parmi celles connues de ses super-classes déclarées. Cette règle peut être considérée comme une généralisation abusive de la règle de l'héritage simple : « la propriété locale héritée est celle connue de sa super-classe déclarée ».

Ainsi dans l'exemple, la classe `B` hérite la propriété locale `toto` connue de `A` et un conflit se produit dans la classe `F` entre les propriétés locales `toto` connues dans `C` et `D`. Toutefois, cette règle possède deux problèmes importants, chacun d'eux produisant des conflits dans les classes `E` et `G` que la résolution par défaut aurait pu éviter.

Le premier problème est caractérisé par le conflit dans la classe `G` entre les propriétés locales `toto` connues des classes `A` et `D`. Ce problème est lié à la *transitivité explicite*[20] puisque la règle parle des super-classes déclarées. Une façon de s'affranchir du problème consiste à substituer dans la règle « super-classes déclarées » par « super-classes directes ». Dans le cas de la classe `G`, la propriété locale héritée serait alors celle connue de `D` puisque `D` est la seule super-classe directe de `G`.

Toutefois, cette nouvelle formulation ne résout pas le second problème, caractérisé par le conflit dans la classe `E` entre les propriétés locales `toto` connues des classes `B` et `C`. Ce conflit n'est pas nécessaire puisque la propriété locale `toto` connue de `B` est moins spécifique que celle connue de `C`. Imaginer alors que la propriété connue dans `B` puisse être héritée dans `E` contredirait le principe de monotonie.

Clos

De façon stricte, Clos n'est pas compatible avec le méta-modèle puisque les fonctions génériques et les méthodes ne sont pas associées à une classe en particulier mais à un n-uplet de classes. Toutefois, en se restreignant à des méthodes d'arité 1, on se ramène à des n-uplets d'une seule classe ce qui nous permet de comparer Clos au regard du méta-modèle.

```
(defclass a () ())
(defclass b (a) ())
(defclass b (a) ())
(defclass d (a) ())
(defclass e (b c) ())
(defclass f (c d) ())
(defclass g (d a) ())
(defgeneric toto (x))
(defmethod toto ((x a)) ...)
(defmethod toto ((x c)) ...)
(defmethod toto ((x d)) ...)
```

Clos respecte l'approche naturelle de la spécialisation puisque la classe `B` hérite de la propriété locale `toto` définie dans `A`, la classe `E` celle définie dans `C` et la classe `G` celle définie dans `D`.

Toutefois, Clos intègre une technique de linéarisation [Ducournau *et al.*, 1994; Barrett *et al.*, 1996] : en utilisant l'ordre dans lequel les super-classes sont déclarées, on détermine pour chaque classe un ordre total sur ses super-classes qui inclut la relation de spécialisation (l'ordre partiel $\prec$). L'intérêt de déterminer un ordre total est que celui-ci permet de résoudre l'héritage sans jamais causer de conflit de propriétés

[20] Cf. section 3.3.2.

locales[21]. Dans tous les cas, CLOS respecte l'approche naturelle puisque la propriété locale héritée est toujours choisie dans spec(c, g).

Dans l'exemple, comme la classe `F` déclare la super-classe `C` avant la super-classe `D`, le conflit est résolu par l'héritage de la propriété locale `toto` définie dans `C`.

Python

```
class A(): def: toto: ...
class B(A): pass
class C(A): def: toto: ...
class D(A): def: toto: ...
class E(B,C): pass
class F(C,D): pass
class G(A,D): pass
```

PYTHON utilise un mécanisme d'héritage différent de la résolution par défaut. En effet, la propriété locale héritée est celle de la première super-classe déclarée qui en fournit une. Fondamentalement, cette règle correspond à déterminer un ordre total sur les super-classes d'une classe et permet donc de résoudre les conflits de propriété locale. Par exemple, ici la classe `F` hérite la propriété locale `toto` connue dans `C`.

Malheureusement, l'ordre total construit par PYTHON ne respecte pas la relation de spécialisation et n'en est donc pas une extension linéaire. L'approche intuitive de la spécialisation se trouve complètement niée puisque le principe de monotonie n'est pas respecté : ici, les classes `E` et `G` héritent la propriété connue dans `A` (et non celle respectivement connue dans `C` et dans `D`).

Objective-Caml

```
class a = object method toto = ... end;;
class b = object inherit a end;;
class c = object inherit a method toto = ... end;;
class d = object inherit a method toto = ... end;;
class e = object inherit b inherit c end;;
class f = object inherit c inherit d end;;
class g = object inherit a inherit d end;;
```

OBJECTIVE-CAML utilise un mécanisme proche de ceux de PYTHON et d'EIFFEL. Proche de PYTHON car la propriété héritée est celle de la dernière super-classe déclarée qui en fournit une — en PYTHON c'est la première mais l'idée est la même. Proche d'EIFFEL car un warning est généré lorsqu'il y a un « conflit », il s'agit bien sûr d'un conflit au sens d'EIFFEL, c'est-à-dire dans les classes `E`, `F` et `G` de l'exemple.

[21] Modulo les erreurs signalées lorsque la linéarisation est impossible.

	B	E	F	G
C++	A	C	Erreur	D
Eiffel	A	Erreur	Erreur	Erreur
Clos	A	C	C	D
Python	A	A	C	A
Objective-Caml	A	C (+ warning)	D (+ warning)	D (+ warning)
Approche naturelle (et PRM)	A	C	Erreur	D + warning de transitivité

Fig. 3.9: Résultat de la comparaison des mécanismes d'héritage

PRM

```
class A def toto do ... end end
class B inherit A end
class C inherit A def toto do ... end end
class D inherit A def toto do ... end end
class E inherit B C end
class F inherit C D end
class G inherit A D end
```

Sans surprise, la spécification de PRM respecte strictement la sémantique naturelle de la spécialisation : `E` hérite la propriété définie dans `C`, `G` hérite celle définie dans `D` et conflit pour `F`.

Conclusion sur les variations

La figure 3.9 résume, pour chaque langage et pour chaque classe de l'exemple (figure 3.8) héritant une méthode `toto`, la classe dans laquelle est définie la méthode héritée.

De notre point de vue, un langage de programmation orienté objet idéal doit au minimum respecter l'approche naturelle de la spécialisation. Comme nous l'avons vu, c'est le cas de tous les langages en héritage simple. Toutefois, pour les langages en héritage multiple, ce n'est malheureusement pas systématique. De façon surprenante, parmi les langages étudiées ici, seuls C++ et Clos (et PRM) respectent l'approche naturelle de la spécialisation, toutefois avec deux bémols : pour C++, l'usage systématique du mot clé `virtual` est requis ; pour Clos l'analyse est partielle puisque notre méta-modèle ne permet pas d'intégrer la sélection multiple.

3.7 Sélection complexe et redéfinition partielle

Dans la section précédente, nous avons considéré trois axiomes :
- l'homogénéité des propriétés : les propriétés locales ont la même nature et le même comportement vis-à-vis de l'héritage ;
- redéfinition totale : une propriété locale est soit totalement redéfinie, soit totalement héritée.
- héritage atomique : les propriétés locales héritées sont celles définies dans les super-classes ;

Dans cette section, nous dépasserons ces considérations. En effet la nature des propriétés locales est éminemment complexe. Et en pratique, les mécanismes de redéfinition et d'héritage des langages prennent en compte cette nature complexe :
- l'héritage peut intégrer une sémantique additionnelle en fonction de la nature des propriétés ;
- la *redéfinition partielle* consiste à redéfinir une partie d'une propriété et d'hériter l'autre partie ;
- l'*héritage combinant* permet d'hériter une propriété locale construite par la combinaison des propriétés locales des super-classes ;
- la *redéfinition combinante* est un mélange des deux précédents puisque la propriété locale d'une classe est construite par la combinaison de la redéfinition partielle éventuelle et des propriétés locales héritées des super-classes.

3.7.1 Propriétés locales et méta-propriétés

Comme nous l'avons signalé, la nature des propriétés locales est complexe et varie suivant les langages. Parmi les grandes familles de propriétés locales, on peut citer les méthodes (procédurales ou fonctionnelles), les attributs (stockés dans les instances ou stockés dans la classe à la CLOS), mais également les types (en C++) ou les invariants de classes (en EIFFEL).

Les propriétés locales étant des entités complexes, elles possèdent des propriétés propres que nous pouvons appeler *méta-propriétés* :
- un nom — que nous avions déjà modélisé ;
- une implémentation pour les méthodes ;
- en typage statique, un type pour les attributs et une signature pour les méthodes ;
- une protection (visibilité, droits d'accès) ;
- des exceptions déclarées ;
- des contrats (pré- et post-assertions) ;
- méta-information : numéro de version, auteur, obsolescence éventuelle (*deprecated*)... ;
- etc.

3.7.2 Sémantique additionnelle, exemple : les méthodes abstraites

Pour rappel, une *méthode abstraite* (appelée *méthode virtuelle pure* en C++ et *méthode retardée* en EIFFEL) est une méthode qui ne fournit pas d'implémentation.

En héritage multiple, les conflits de propriétés locales dans lesquels les propriétés impliquées sont toutes des méthodes abstraites sauf une peuvent être levés facilement : c'est la méthode concrète qui est héritée.

Dans le langage EIFFEL, nous avons vu que les conflits de propriétés locales entre méthodes peuvent être résolus par le programmeur en choisissant la propriété à hériter. En fait, la mise en œuvre de ce mécanisme consiste à « rendre » abstraite (via le mot clé `undefine`) les méthodes en conflit sauf une, la méthode héritée étant alors celle qui n'a pas été rendue abstraite.

3.7.3 Redéfinition partielle, exemple : la protection

La *protection* [Ardourel, 2002] (appelée aussi visibilité ou *export*) permet à chaque propriété locale de préciser quelles classes sont autorisées à l'invoquer.

Certains langages de programmation, comme EIFFEL, permettent une redéfinition partielle de la protection, c'est-à-dire l'héritage d'une propriété locale mais en redéfinissant la protection de celle-ci. L'inverse est également possible (redéfinir une propriété locale mais hériter la protection).

En EIFFEL , les propriétés sont définies dans des blocs `feature` (figure 3.10). Lors de la définition d'une propriété locale, sa protection est celle déclarée par le bloc `feature` ou, à défaut, `ANY` (la racine de la hiérarchie de classes). Ainsi dans la classe `A` de l'exemple, `toto` est visible pour `ANY` et ses sous-classes (c'est-à-dire visible dans toutes les classes) tandis que `tata` et `titi` ne sont visibles que pour `SOME` et ses sous-classes.

Lors de la redéfinition d'une propriété locale, EIFFEL permet de redéfinir la protection et/ou l'implémentation. Dans le premier cas, il est nécessaire d'utiliser la clause d'héritage `export` (voir `titi` dans l'exemple), dans le second cas, la propriété doit être redéfinie dans un bloc `feature` qui ne déclare pas de protection (voir `tata` dans l'exemple) et dans le dernier cas, la propriété doit être redéfinie dans un bloc `feature` qui déclare une protection (voir `toto` dans l'exemple). Ainsi, dans la classe `B` de l'exemple, `toto` et `tata` sont visibles pour `SOME` et ses sous-classes tandis que `titi` est visible dans toutes les classes.

En conclusion, EIFFEL permet une redéfinition fine permettant de préciser si la protection des propriétés locales doit être redéfinie ou héritée. Toutefois, la syntaxe est hétérogène et inutilement complexe : la syntaxe utilisée pour ne redéfinir que la protection n'a syntaxiquement rien à voir avec celle utilisée pour ne redéfinir que l'implémentation. De plus, la spécification du langage est ambiguë sur le mélange des clauses `export` et des redéfinitions dans des blocs `feature`.

```
class A
feature
   toto ...
   -- toto est visible pour ANY

feature {SOME}
   tata ...
   -- tata est visible pour SOME

   titi ...
   -- titi est visible pour SOME
end -- A

class B
inherit
   A
      export
         {OTHER} titi
          -- titi est visible pour OTHER
      redefine toto, tata
      end

feature
   tata ...
   -- tata est visible pour SOME

feature {SOME}
   toto ...
   -- toto est visible pour SOME
end -- B
```

FIG. 3.10: Exemple de protection en EIFFEL

C++ possède également un mécanisme de protection. Les propriétés peuvent être soit `public`, soit `private` soit `protected`. Lors de la redéfinition des propriétés locales, il est obligatoire de préciser la protection. Toutefois, il est possible de redéfinir globalement la protection des propriétés héritées en utilisant lors de la déclaration des super-classes :

- le mot clé `public` : il n'y a pas de redéfinition ;
- le mot clé `protected` : les propriétés `public` héritées sont partiellement redéfinies en propriétés `protected` ;
- le mot clé `private` : toutes les propriétés héritées sont partiellement redéfinies en propriétés `private`.

3.7.4 Héritage combinant, exemple : les exceptions déclarées

La signature des méthodes JAVA doit préciser l'ensemble des exceptions susceptibles d'être signalées. Cet ensemble fait partie des méta-propriétés des propriétés locales et peut varier lors des redéfinitions des méthodes.

Vis-à-vis des exceptions déclarées, l'héritage multiple des interfaces JAVA est combinant. Les exceptions déclarées d'une propriété locale héritée sont déterminées par l'intersection des exceptions déclarées des propriétés locales des super-classes.

Par exemple, dans le listing JAVA suivant, l'interface `D` hérite une propriété locale `toto` susceptible de signaler des exceptions de type `ExceptionT` mais en aucun cas des exceptions de types `ExceptionU` ou `ExceptionV`.

```
interface A {
        public void toto() throws Exception;
}
interface B extends A {
        public void toto() throws ExceptionT, ExceptionU;
}
interface C extends A {
        public void toto() throws ExceptionT, ExceptionV;
}
interface D extends B, C  { }
```

Remarque : Les exceptions déclarées de JAVA varient de façon covariante, ce qui est conforme aux règles du typage sûr. Toutefois, la nature même des exceptions[22] devrait rendre caduque l'idée de toute règle au sujet de la variation des exceptions. Ainsi, les langages ne devraient ne pas imposer la déclaration des exceptions ou alors uniquement dans une optique de documentation — c'est-à-dire avec la même absence de sémantique que les commentaires.

[22]Dans le cas général, une exception, n'a pas à être prévue entre l'endroit où elle est signalée et l'endroit où elle est rattrapée.

3.7.5 Redéfinition combinante, exemple : les contrats

Dans le langage EIFFEL , les méthodes d'une classe contiennent des contrats [Meyer, 1991] matérialisés par des pré-conditions et des post-conditions.

Que cela soit lors de l'héritage ou lors de la redéfinition d'une propriété, le contrat de la propriété locale est construit par combinaison des contrats des propriétés locales des super-classes : disjonction pour les pré-conditions et conjonction pour les post-conditions.

Par exemple, dans le listing EIFFEL suivant, le contrat de la méthode `toto` de la classe `B` est constitué de la pré-condition `ra` $\vee$ `rb` et de la post-condition `ea` $\wedge$ `eb` :

```
class A
feature
        toto is
                require
                        ra: ...
                do
                        ...
                ensure
                        ea: ...
                end
end -- A

class B
inherit A redefine toto end
feature
        toto is
                require
                        rb: ...
                do
                        ...
                ensure
                        eb: ...
                end
end -- B
```

3.8 Typage statique

Nous partons du principe largement répandu dans les différents langages que les types correspondent aux classes et que la relation de sous-typage ($<:$) n'est rien d'autre que la relation de spécialisation ($\preceq$).

Rappel : à chaque site d'appel de la forme `x.toto(args)` est associé la propriété globale nommée `toto` dans le type statique de `x`. Ainsi, en typage statique, un tel site d'appel est invalide si cette propriété globale n'existe pas. Au niveau de l'exécution, les receveurs potentiels désignés par `x` sont sous-type du type statique de `x`.

L'un des rôles du typage sûr est d'assurer lors de l'exécution que :
- le message est valide — pas de `doesNotUnderstand:` pour parler à la SMALLTALK ;
- les arguments sont valides — pas d'erreur de type au niveau des arguments ;
- le résultat est valide — pas d'erreur de type au niveau de la valeur de retour.

Le premier point est garanti puisque qu'une classe hérite les propriétés globales de ses sous-classes. Afin de garantir les deux autres points, nous avons besoin d'introduire $<\in L^2$, la *relation de masquage* entre propriétés locales qui se définit ainsi :

$$\forall c, c' \in X, g \in G_c, \quad c' \prec c \quad \Rightarrow \quad \mathrm{sel}(c', g) < \mathrm{sel}(c, g) \ . \tag{3.17}$$

En français : les propriétés locales redéfinies ou héritées dans une classe masquent les propriétés locales des mêmes propriétés globales dans les super-classes.

Lorsque $l' < l$, cela signifie que si un envoi de message peut se résoudre par la propriété locale l alors cet envoi de message pourrait également se résoudre par la propriété locale l' ; donc, de fait, l' est substituable à l. Selon Liskov [Liskov et Wing, 1993], la notion de sous-typage est basée sur la notion de substituabilité.

Le typage sûr impose alors la *règle de contravariance* suivante :

Définition 9 (Règle contravariante de typage) *Pour deux propriétés l et $l' \in L$ de signatures respectives $t \to u$ et $t' \to u'$,*

$$l' < l \quad \Rightarrow \quad (t <: t') \quad \wedge (u' <: u) \ .$$

Au niveau du typage, les attributs peuvent être vus comme des propriétés de signature $t \to t$ où t est le type statique de l'attribut ; il en résulte que la règle de contravariance impose l'invariance des attributs.

Une politique de typage covariante est caractérisée par la règle suivante :

Définition 10 (Règle covariante de typage) *Pour deux propriétés l et $l' \in L$ de signatures respectives $t \to u$ et $t' \to u'$,*

$$l' < l \quad \Rightarrow \quad (t' <: t) \quad \wedge (u' <: u) \ .$$

Elle a notre préférence puisqu'elle correspond à l'approche naturelle de la spécialisation [Ducournau, 2001b; Ducournau, 2002a].

Dans tous les cas, les règles de contravariance ou de covariance peuvent être statiquement vérifiées — par exemple par un compilateur.

Lorsque l'héritage des propriétés locales est conforme à l'approche naturelle de la spécialisation et que la spécification du langage n'introduit aucune sémantique additionnelle (c'est-à-dire les conflits de propriétés locales étant alors résolus par la définition

d'une nouvelle propriété locale), nous avons :

$$l' \ll l \quad \Leftrightarrow \quad l' < l \ .$$

Le système peut se contenter de vérifier si les redéfinitions de propriétés locales sont conformes.

Lorsque l'approche naturelle de la spécialisation n'est pas respectée ou que la spécification du langage introduit une sémantique additionnelle, il peut être nécessaire de vérifier également que les propriétés locales héritées sont conformes aux propriétés locales qu'elles masquent.

3.9 Conclusion

Dans ce chapitre, nous avons présenté une méta-modélisation des mécanismes d'héritage des langages à objets basée sur une approche naturelle de la notion de spécialisation de classes — et qui correspond généralement à la façon dont sont utilisés les langages de programmation à objets.

Cette méta-modélisation a aussi l'avantage de proposer une gestion simple et naturelle de l'héritage multiple nécessaire à tout usage de l'approche par objets dans un contexte de typage statique. Simple parce que le rôle de classe n'est assuré que par une seule entité, le rôle de la spécialisation n'est assuré que par une seule relation et qu'au niveau des propriétés nous n'avons pas besoin de distinguer les attributs des méthodes ; bref, nous n'avons pas multiplié les entités sans nécessité — cf. principe 2 page 14. Naturelle car que l'héritage soit simple ou multiple, la ligne directrice est la même : le respect du mode de représentation de l'humain — cf. principe 1 page 2.

Ce méta-modèle n'est pas seulement destiné à poser un cadre formel, il a également une vocation pratique : c'est celui qui est implémenté dans le compilateur `prmc` que nous présentons dans le chapitre 8.

Dans le chapitre suivant nous allons montrer que ce méta-modèle des classes et des propriétés est réutilisable pour d'une part modéliser le rapport entre les modules et les classes et d'autre part le rapport entre les classes et leurs raffinements.

Chapitre 4

Une méta-modélisation des modules et du raffinement de classes

Préambule

Ce chapitre propose une formalisation des relations entre les modules (tels que nous les avons présentés dans la section 2.7 page 27) et les classes vis-à-vis de la dépendance entre modules et de l'importation et du raffinement de leur classes. En particulier, nous nous intéressons aux cas dans lesquels :

- *la dépendance entre modules est multiple ;*
- *l'importation des classes est multiple ;*
- *le raffinement et la spécialisation de classes sont combinés.*

Nous posons comme principe de base une méta-modélisation des modules et des classes isomorphe à la méta-modélisation des classes et des propriétés que nous avons présentée dans le chapitre précédent.

4.1 Introduction

Comme nous l'avons déjà montré dans la section 2.7, les modules et les classes correspondent à deux notions en opposition:

- les classes n'ont de sens que par leurs instances (directes et indirectes) alors que les modules n'en ont aucune notion ;
- un module correspond à une préoccupation alors qu'une classe regroupe généralement plusieurs préoccupations et qu'une préoccupation est généralement dispersée à travers plusieurs classes.

Les modules sont définis par quatre caractéristiques de base :

- un module dépend d'autres modules ;
- le graphe de dépendance entre modules est sans circuit ;

– les modules sont composés de définitions de classes ;
– les modules importent les classes des modules dont ils dépendent ;
et si l'on rajoute le raffinement de classes — cf. section 2.7.3 page 33 :
– les modules peuvent raffiner les classes qu'ils importent.

Nous pouvons illustrer cette idée de modules, de classes et de raffinement de classes par le programme PRM suivant :

```
# Module text
class String
        def length: Int
        do
                ...
        end

        ...
end
```

```
# Module crypt
import text
class String
        def encode(key: String): String
        do
                ...
        end

        def decode(key: String): String
        do
                ...
        end
end
```

Ici, le module `crypt` dépend du module `text` et importe ses classes, dont la classe `String` qu'il raffine en y ajoutant deux méthodes, l'une de chiffrement et l'autre de déchiffrement.

4.2 Modules et classes

L'amorce de notre travail est basée sur une analogie structurelle entre les modules et les classes. En effet, si l'on y regarde attentivement, on peut se rendre compte que d'une certaine façon, les classes sont aux modules ce que les propriétés sont aux classes :
– les classes spécialisent d'autres classes ;
– le graphe de spécialisation entre classes est sans circuit ;

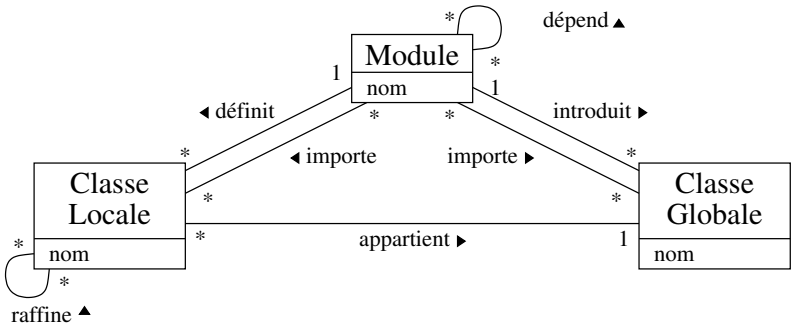


FIG. 4.1: Méta-modèle des modules et des classes

– les classes sont composées de définitions de propriétés ;
– les classes héritent les propriétés des classes qu'elles spécialisent ;
– les classes peuvent redéfinir les propriétés qu'elles héritent.

Ainsi, le méta-modèle que nous proposons qui modélise les *modules* et les *classes* est présenté figure 4.1 sous forme d'un diagramme de classes UML. Celui-ci est de façon évidente une adaptation du méta-modèle des classes et des propriétés de la figure 3.1 (page 41).

Les modules (analogues aux classes) sont des entités qui :
– sont hiérarchisées par une relation de *dépendance* ;
– possèdent des classes locales: celles-ci sont *définies* ou *importées* ;
– possèdent des classes globales: celles-ci sont *introduites* ou *importées*.

Les *classes locales* (analogues aux propriétés locales) correspondent aux classes que le programmeur définit, c'est-à-dire écrit lors de la définition d'un *module*, indépendamment de toute éventuelle autre définition de la « même classe » dans les super-modules et les sous-modules. Par exemple, la définition d'une classe dans un module et son raffinement dans un sous-module correspondent à deux classes locales distinctes.

Les *classes globales*[1] servent à modéliser cette idée de « même classe » à travers plusieurs modules en relation de dépendance. Dans le chapitre précédent, la nécessité des propriétés globales était due à l'envoi de message puisqu'à chaque site d'appel `x.foo` correspond une propriété globale précise. Avec les classes globales, on peut retrouver cette analogie non pas avec l'envoi de message mais avec l'instanciation puisqu'à chaque *site d'instanciation* `new Foo`, correspond une unique classe globale. Avec le raffinement de classe, la sémantique d'instanciation d'un module dépendent alors des programmes qui incluent ce module.

Par exemple, dans le programme précédent, la classe locale `String` du module `text` (appelons-la l) est distincte de la classe locale `String` du module `crypt` (appelons-là

[1]Pour des raisons évidentes, nous n'avons pu utiliser la terminologie de *classe générique* et ainsi profiter de l'analogie avec les *fonctions génériques* de CLOS.

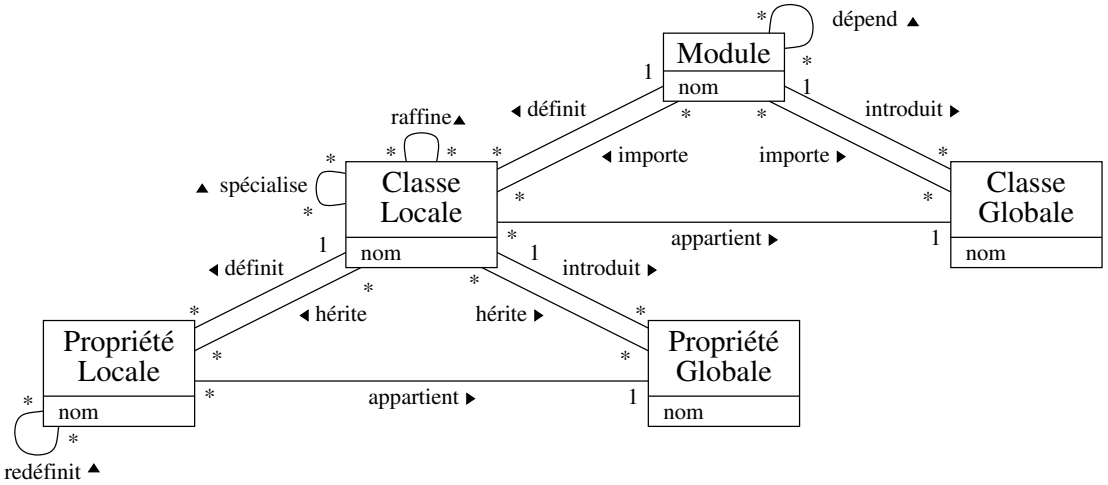


FIG. 4.2: Méta-modèle des modules, des classes et des propriétés

l') — en effet, l n'a pas les méthodes `encode` et `decode` que l' possède. À l'exécution, les objets qui seront créés à partir d'un site d'instanciation de la forme `new String` dépendent des programmes. Si le programme n'est constitué que des modules `text` et `crypt`, la classe directe de ces instances sera l'. Mais si le programme n'est constitué que du module `text`, la classe directe de ces instances sera l.

Bref, dans la méta-modélisation que nous proposons, une classe locale :

- est *définie* dans un seul module — le module dans lequel le programmeur écrit la définition de la classe locale ;
- *appartient* à une seule classe globale ;
- peut *raffiner* une classe locale définie dans un super-module et qui appartient à la même classe globale ;
- est *importée* par les sous-modules de son module de définition et éventuellement redéfinie.

Tandis qu'une classe globale :

- est *introduite* dans un seul module ;
- regroupe des classes locales : celle qui sert à l'introduire et ses raffinements ;
- est *importée* par les sous-modules du module qui l'introduit.

Si l'on ajoute les propriétés au méta-modèle de la figure 4.1, c'est-à-dire si on le combine avec le méta-modèle de la figure 3.1 du chapitre précédent, on obtient le méta-modèle regroupant modules, classes et propriétés présenté figure 4.2 sous forme d'un diagramme de classes UML où les classes locales correspondent aux classes telles qu'elles sont définies dans le méta-modèle du chapitre précédent.

Remarque : Dans le méta-modèle de la figure 4.2, il n'y a aucune relation directe entre les propriétés globales et les classes globales, contrairement à ce que pourrait laisser

supposer leur noms.

4.2.1 Définition de hiérarchies de modules et construction de modèles

La *définition d'une hiérarchie de modules* correspond à l'ensemble des définitions de chacun des modules de cette hiérarchie.

La *définition d'un module* est un triplet $\langle$`modname`, `supermodnames`, `localclassdef`$\rangle$ constitué du nom du module, (`modname`), d'un ensemble de noms de super-modules (`supermodnames`) et d'un ensemble de définitions de classes locales (`localclassdef`).

La *définition d'une classe locale* correspond à la *définition d'une classe* telle que définie dans la section 3.2.1.

Par exemple, à partir du petit programme précédent composé des modules `text` et `crypt`, on associe la définition de la hiérarchie de modules suivante :

$$\{\langle \texttt{text}, \emptyset, \{\langle \texttt{String}, \emptyset, \{\text{length}\}\rangle\}\rangle, \langle \texttt{crypt}, \emptyset, \{\langle \texttt{String}, \emptyset, \{\texttt{code}, \texttt{decode}\}\rangle\}\rangle\}$$

Nous posons également des conventions pour la représentation graphique d'une définition de hiérarchie de modules :

- les définitions de modules correspondent à des boîtes nommées (ou numérotées) ;
- la relation de dépendance déclarée entre modules est représentée par des flèches fines noires — elles relient chaque module aux modules dont elle déclare dépendre ;
- la boîte d'un module contient des boîtes plus petites correspondant aux classes que le module définit ;
- les boîtes des classes sont nommées (avec la première lettre en majuscule) ;
- la relation de spécialisation déclarée est représentée par des flèches larges blanches — elles relient chaque classe aux super-classes qu'elle déclare spécialiser ;
- la définition des propriétés locales correspond aux petits noms inscrits sous le nom des classes (les noms de propriétés locales sont en minuscules).

Ainsi, à partir du même petit programme à deux modules, on associe la représentation graphique de la figure 4.3.

Nous posons également la notation suivante : `m::c` désigne la classe locale de nom `c` définie dans le module de nom `m`.

4.2.2 Hiérarchie de modules

Sans surprise, nous allons réutiliser la formalisation des hiérarchies que nous avons présentée dans la section 3.2.2 page 44 du chapitre précédent. Toutefois, plusieurs hiérarchies coexistent : un *module* est une hiérarchie de classes et un programme une *hiérarchie* de modules (donc une hiérarchie de hiérarchies). Dans le reste du chapitre, nous expliciterons systématiquement la hiérarchie à laquelle nous faisons référence.

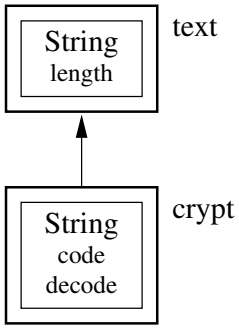


FIG. 4.3: Exemple de représentation graphique d'une définition de hiérarchie de modules

Définition 11 *[Modèle de hiérarchie de modules] Un* modèle de hiérarchie de modules*, c'est-à-dire une instance du méta-modèle de la figure 4.1, est un n-uplet* $\mathcal{P} = \langle X^{\mathcal{P}}, \prec^{\mathcal{P}}, G^{\mathcal{P}}, L^{\mathcal{P}}, N^{\mathcal{P}}, \mathrm{nom}_{\mathcal{P}}, \mathrm{glob}_{\mathcal{P}}, \mathrm{intro}_{\mathcal{P}}, \mathrm{def}_{\mathcal{P}} \rangle$ *dans lequel :*

- $X^{\mathcal{P}}$ *est l'ensemble des* modules *;*
- $\prec^{\mathcal{P}}$ *est la relation de dépendance entre modules ;*
- $G^{\mathcal{P}}$ *est l'ensemble des classes globales ;*
- $L^{\mathcal{P}}$ *est l'ensemble des classes locales ;*
- $N^{\mathcal{P}}$ *est l'ensemble des* identifiants *(noms) de* modules*, de* classes *et de* propriétés*.*
- $\mathrm{nom}_{\mathcal{P}} : X^{\mathcal{P}} \uplus G^{\mathcal{P}} \uplus L^{\mathcal{P}} \to N^{\mathcal{P}}$ *est la fonction de désignation ;*
- $\mathrm{glob}_{\mathcal{P}} : L^{\mathcal{P}} \to G^{\mathcal{P}}$ *associe chaque classe locale à la classe globale à laquelle elle appartient ;*
- $\mathrm{intro}_{\mathcal{P}} : G^{\mathcal{P}} \to X^{\mathcal{P}}$ *associe chaque classe globale au module qui l'introduit ;*
- $\mathrm{def}_{\mathcal{P}} : L^{\mathcal{P}} \to X^{\mathcal{P}}$ *associe chaque classe locale au module qui la définit.*

Définition 12 (Modèle de hiérarchie de classes d'un module) *Chaque module* $\mathcal{M} \in X^{\mathcal{P}}$ *est lui-même une hiérarchie de classes* $\mathcal{M} = \langle X^{\mathcal{M}}, \prec^{\mathcal{M}}, G^{\mathcal{M}}, L^{\mathcal{M}}, N^{\mathcal{M}}, \mathrm{nom}_{\mathcal{M}}, \mathrm{glob}_{\mathcal{M}}, \mathrm{intro}_{\mathcal{M}}, \mathrm{def}_{\mathcal{M}} \rangle$*, appelée* modèle de hiérarchie de classes d'un module*, dans lequel :*

- *les noms utilisés sont les mêmes :*

$$N^{\mathcal{P}} = N^{\mathcal{M}} \ ;$$

- *les classes du module* $\mathcal{M}$ *correspondent aux classes locales du module* $\mathcal{M}$ *dans le programme* $\mathcal{P}$ *:*

$$X^{\mathcal{M}} = L^{\mathcal{P}}_{\mathcal{M}} \ .$$

4.2.3 Sémantiques et constructions

Sémantique et construction d'un module

La *sémantique d'un module* est le modèle de la hiérarchie de classe de ce module. Celle-ci dépend de la définition de ce module mais aussi de la définition de ses super-modules. En effet, même si un modèle de hiérarchie de classes d'un module $\mathcal{M}$ possède les mêmes entités qu'une hiérarchie de classe $\mathcal{H}$ telle que nous l'avons présentée dans le chapitre précédent, l'existence du raffinement de classe induit de nouveaux mécanismes et donc une formalisation différence des relations entre ces entités.

Intuitivement, le mécanisme du raffinement peut se concevoir comme la *définition incrémentale* des classes dans lequel le contenu des classes locales d'un module est construit à partir des définitions successives de chacune des classes de ce module et de ses super-modules. Nous modélisons cette définition incrémentale des classes grâce à la construction d'une immense hiérarchie de classes locales du module et de ses super-modules de telle façon qu'une classe locale hérite les propriétés des classes locales qu'elle raffine.

Remarque : Une approche différente consiste à modéliser directement cette notion de définition incrémentale [Privat et Ducournau, 2006]. Toutefois, en passant par une hiérarchie de classes intermédiaire et en assimilant le raffinement de classes à une sorte de relation de spécialisation, nous retrouvons exactement le cadre que nous avons développé dans le chapitre précédent.

Dans le cadre du développement modulaire de logiciels, chaque module devrait être développé, testé et compilé indépendamment des programmes auxquels il appartient — et plus précisément, indépendamment de tout module dont il ne dépend pas[2]. Ce mode de développement est bien sûr compatible avec notre modélisation puisque la hiérarchie de classes d'un module ne dépend que de ce module et de ses super-modules.

Sémantique et construction d'un programme

Un programme est une hiérarchie de modules qui possède un *module principal*, c'est-à-dire un module qui dépend de tous les autres modules de la hiérarchie[3]. Toutefois, pour toute hiérarchie de modules, on peut considérer l'existence d'un module principal implicite qui ne contient aucune définition de classe et se contente de dépendre des autres modules de la hiérarchie.

Remarque : Le module principal d'un programme n'est pas forcément le module qui contient la définition du point d'entrée du programme (la méthode `main` de la classe

[2]Cela peut passer pour une lapalissade, mais nous pensons qu'il est utile de le préciser.

[3]Sur les figures, le module principal est donc en bas : il serait mal avisé de confondre le module principal d'une hiérarchie de module avec le module *racine*, c'est-à-dire celui dont dépend tous les autres modules (appelé `kernel` en PRM, cf. annexe A).

`Sys` en PRM). En effet, rien n'interdit de raffiner un tel module sans toucher au point d'entrée.

La sémantique d'un programme correspond à la sémantique du module principal de ce programme, c'est-à-dire à la hiérarchie de classes du module principal.

Remarque : Dans le cadre d'un développement efficace, il serait souhaitable que :
- le module principal soit le plus petit possible : de préférence vide, voire entièrement implicite[4] ;
- le risque d'avoir un conflit dans le module principal soit le plus faible possible — en partant du principe que tous les autres modules du programme sont exempts de conflit.

Toutefois, si conflits il y a, ils peuvent être résolus pour la plupart dans le module principal comme nous le détaillons dans les sections suivantes.

4.3 Modules, dépendances et importations de classes

Cette section se concentre sur l'analogie entre modules et classes — cf. chapitre précédent.

4.3.1 Dépendance de modules

La relation de dépendance entre modules est analogue à la relation de spécialisation entre classes. Si, pour deux modules m et $m' \in X^{\mathcal{P}}$, $m' \prec^{\mathcal{P}} m$, nous disons-que :
- m' *dépend* de m ;
- m' est un *sous-module* de m;
- m est un *super-module* de m'.

Remarque : Un sous-module n'est pas un module imbriqué (aucun de nos modules n'est imbriqué de toute façon). Un super-module n'est pas un module principal (au contraire, le module principal est plutôt le plus petit sous-module).

La relation de dépendance entre modules ($\prec^{\mathcal{P}}$) est construite par la fermeture transitive de la relation de dépendance déclarée ($\prec^{\mathcal{P}}_{\text{decl}}$) qui est elle-même construite à partir la définition de la hiérarchie de modules (`supermodnames`). De même que pour les classes et la relation de spécialisation, lors de la construction d'un modèle de hiérarchie de modules à partir d'une définition de modules, deux sortes d'erreurs peuvent se produire : *super-module inconnu* et *circuit de dépendance* — cf.section 3.3.1 page 48.

[4] Ainsi, le module principal du programme compilé par la commande `prmc module1 module2` n'est que le module implicite qui dépend des modules `module1` et `module2` (cf. chapitre 8).

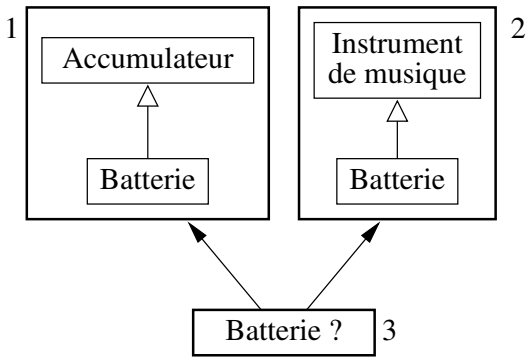


FIG. 4.4: Conflit de classes globales

4.3.2 Classes globales

Les modules importent les classes globales de leurs super-modules — cf. les équations 3.3 à 3.7 page 54 :

$$\forall m \in X^{\mathcal{P}}, m' \in X^{\mathcal{P}} \quad m' \prec^{\mathcal{P}} m \Rightarrow G^{\mathcal{P}}_{m'} \supseteq G^{\mathcal{P}}_{m} \ , \tag{4.1}$$

$$G^{\mathcal{P}}_{+m} = \mathrm{intro}^{-1}_{\mathcal{P}}(m) \ , \tag{4.2}$$

$$G^{\mathcal{P}}_{m} = G^{\mathcal{P}}_{\uparrow m} \uplus G^{\mathcal{P}}_{+m} = \biguplus_{m \preceq^{\mathcal{P}} m'} G^{\mathcal{P}}_{+m'} \ , \tag{4.3}$$

$$G^{\mathcal{P}}_{\uparrow m} = \bigcup_{m \prec^{\mathcal{P}}_{d} m'} G^{\mathcal{P}}_{m'} = \biguplus_{m \prec^{\mathcal{P}} m'} G^{\mathcal{P}}_{+m'} \ , \tag{4.4}$$

$$G^{\mathcal{P}} = \bigcup_{m \in X^{\mathcal{P}}} G^{\mathcal{P}}_{m} = \biguplus_{m \in X^{\mathcal{P}}} G^{\mathcal{P}}_{+m} \ . \tag{4.5}$$

où $G^{\mathcal{P}}_{m}$ désigne les classes globales du module m, $G^{\mathcal{P}}_{+m}$ désigne celles introduites par m et $G^{\mathcal{P}}_{\uparrow m}$ désignent celles héritées par m.

Les conflits de classes globales sont analogues à ceux des propriétés globales — voir la section 3.4.3 page 56. Ils surviennent lorsque un module importe deux classes globales distinctes mais homonymes comme l'illustre la figure 4.4.

Les résolutions de tels conflits sont également analogues : lorsque les modules sont considérés comme des espaces de noms, la désignation explicite est ici la solution la plus naturelle ; toutefois, un mécanisme de renommage des classes à la EIFFEL et son langage de configuration LACE pourrait également résoudre le problème. Dans tous les cas, la cohérence de la spécification d'un langage est un bon point de départ : si les conflits de propriétés globales sont solvables d'une façon, les conflits de classes globales devraient l'être d'une façon analogue. En PRM, ces conflits peuvent donc être résolus par renommage.

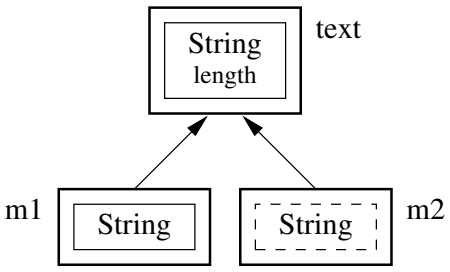


FIG. 4.5: Définition implicite de classes locales

4.3.3 Classes locales

L'ensemble des classes locales définies dans un module $m \in X^{\mathcal{P}}$ est noté $L_m^{\mathcal{P}}$; celui-ci est partitionné en deux sous-ensembles, celui des classes locales qui introduisent une nouvelle classe globale (noté $L_{+m}^{\mathcal{P}}$) et des classes locales qui raffinent une classe globale héritée (noté $L_{\uparrow m}^{\mathcal{P}}$). Comme pour les propriétés, la correspondance entre classes locales et globales est faite sur leurs noms — cf. équation 3.9.

La relation de raffinement entre classes locales ($\ll^{\mathcal{P}}$) est analogue à la relation de redéfinition entre propriétés locales ($\ll^{\mathcal{H}}$) — cf. l'équation 3.12 page 62:

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

Les modules importent les classes locales de leurs super-modules, toutefois les classes locales sont des entités complexes et si l'on utilise la terminologie de la section 3.7 leur importation est *combinante*. En effet, le « contenu » d'une classe est construit par combinaison des différents raffinements d'une classe globale. En particulier, raffiner dans un module une classe par une classe vide est équivalent à ne pas l'avoir du tout raffinée. Ainsi, les deux modules suivants sont strictement équivalents :

```
# Module m1
import text
class String
end
```

```
# Module m2
import text
```

Ainsi, pour des raisons de simplicité, nous considérons au niveau de la formalisation que pour chaque classe globale importée, il existe une classe locale définie : soit par le programmeur, soit implicitement définie vide. Graphiquement, nous représentons ces définitions de classes implicites par des boîtes en pointillés. Par exemple, la représentation graphique des modules `text`, `m1` et `m2` est illustrée dans la figure 4.5.

Toutefois, cette différentiation graphique n'a aucun impact sur les mécanismes d'importation ou de raffinement.

Cette définition implicite apporte une petite variation à l'analogie des équations 3.10 et 3.11 :

$$\begin{aligned} \text{glob}(L^{\mathcal{P}}_{\uparrow c}) &= G^{\mathcal{P}}_{\uparrow c} \ , \\ \text{glob}(L^{\mathcal{P}}_{+c}) &= G^{\mathcal{P}}_{+c} \ , \\ \text{glob}(L^{\mathcal{P}}_{m}) &= G^{\mathcal{P}}_{c} \ . \end{aligned}$$

4.4 Spécialisation de classes

4.4.1 Spécialisation importée

En important les classes, les modules importent également la relation de spécialisation. Par exemple, soient les deux modules suivants :

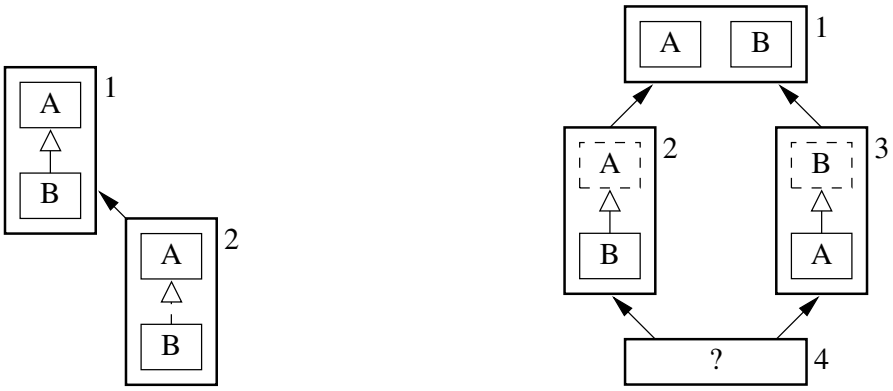
```
# module 1
class A
   ...
end
class B
inherit A
   ...
end
```

```
# module 2
import 1
class A
   ...
end
class B
   ...
end
```

Dans le module `2`, `2::B` est sous-classe de `2::A` puisque dans le module `1` dont dépend le module `2`, `1::B` est sous classe de `1::A`.

Plus formellement, dans un module $\mathcal{M} \in X^{\mathcal{P}}$, nous notons $\prec^{\mathcal{M}}_{\leftarrow}$ la *relation de spécialisation importée*, définie par la plus petite relation qui vérifie :

$$\begin{aligned} &\forall c, c' \in X^{\mathcal{M}}, \forall \mathcal{M}' \in X^{\mathcal{P}}, \mathcal{M} \prec^{\mathcal{P}} \mathcal{M}', \forall (c'', c''') \in X^{\mathcal{M}'} \\ &\quad (c'' \prec^{\mathcal{M}'} c''') \wedge (c \ll^{\mathcal{P}} c'') \wedge (c' \ll^{\mathcal{P}} c''') \Rightarrow (c \prec^{\mathcal{M}}_{\leftarrow} c') \end{aligned} \tag{4.6}$$



(a) Importation de la spécialisation (b) Conflit de spécialisation

FIG. 4.6: Spécialisation et classes locales

Dans les représentations graphiques, nous utilisons des flèches en pointillés pour représenter la relation de spécialisation importée — cf. figure 4.6(a). Toutefois, pour des raisons de lisibilité, nous ne représentons que sa réduction transitive.

4.4.2 Conflit de spécialisation

Définition 13 (Conflit de spécialisation) *Un* conflit de spécialisation *survient lorsque la relation de spécialisation importée n'est pas un ordre partiel.*

La figure 4.6(b) illustre un tel conflit. Il n'existe pas de résolution intrinsèque sauf par l'*unification de classes globales*, c'est-à-dire considérer que les classes locales en relation de spécialisation cyclique appartiennent à la même classe globale.

4.4.3 Spécialisation déclarée

À cause des modules, la construction de la relation de spécialisation entre classes ($\prec^{\mathcal{M}}$) diffère de ($\prec^{\mathcal{H}}$) que nous avons vue dans la section 3.3.1 page 48. Dans un module $\mathcal{M} \in X^{\mathcal{P}}$, la relation de spécialisation ($\prec^{\mathcal{M}}$) est la fermeture transitive de la relation de l'union de spécialisation déclarée ($\prec^{\mathcal{M}}_{\text{decl}}$) et de spécialisation héritée ($\prec^{\mathcal{M}}_{\text{decl}}$).

Autre différence : les déclarations de spécialisation superflues ne se limitent plus aux arcs de transitivité mais concernent également les déclarations de spécialisation redondantes avec la relation de spécialisation importée. Par exemple, dans le listing PRM suivant, le lien de spécialisation déclaré dans le module `m2` est superflu :

```
# module m1
class A
end
class B
```

```
inherit A
end
```

```
# module m2
import m1
class B
inherit A # Avertissement ! Information superflue
end
```

À noter également que c'est bien évidemment une erreur d'introduire des circuits de spécialisation :

```
# module m3
import m1
class A
inherit B # Erreur ! Circuit
end
```

4.5 Héritage des propriétés

Le raffinement de classe exacerbe le besoin d'héritage des propriétés entre classes :
- les classes locales héritent les propriétés des classes locales qu'elles spécialisent ;
- les classes locales héritent les propriétés des classes locales qu'elles raffinent.

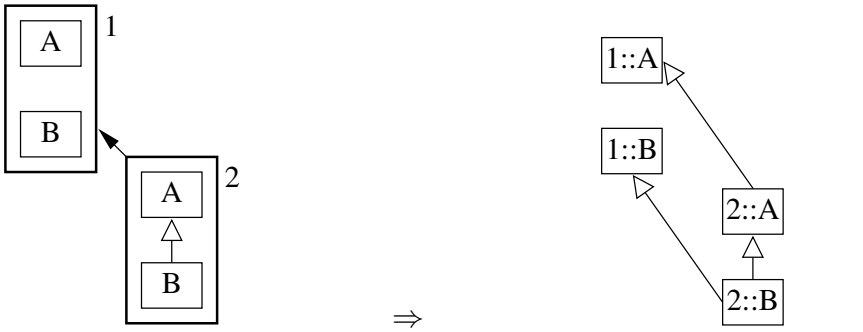
En outre, il n'est plus possible de faire l'économie des problèmes que posent l'héritage multiple. En effet, comme le montre la figure 4.6(a), `2::B` hérite de façon multiple les propriétés de `2::A` et de `1::B` malgré le fait que le raffinement de modules ainsi que la spécialisation de classes soient simples.

Dans le chapitre précédent, nous avons proposé une formalisation des classes et des propriétés qui gère le plus naturellement possible l'héritage, en particulier multiple. Or pour ce faire, nous avons besoin d'une simple hiérarchie de classes qui correspond à la hiérarchie $\mathcal{H}$ du chapitre précédent. C'est pourquoi nous introduisons une notion de *hiérarchie de classes d'une hiérarchie de modules* qui sera cette hiérarchie-là.

4.5.1 Hiérarchie de classes d'une hiérarchie de modules

Définition 14 (Hiérarchie de classes d'une hiérarchie de modules) *La* hiérarchie de classes d'une hiérarchie de modules $\mathcal{P}$ *est un n-uplet* $\mathcal{Q} = \langle X^{\mathcal{Q}}, \prec^{\mathcal{Q}}, G^{\mathcal{Q}}, L^{\mathcal{Q}}, N^{\mathcal{Q}}, \mathrm{nom}_{\mathcal{Q}}, \mathrm{glob}_{\mathcal{Q}}, \mathrm{intro}_{\mathcal{Q}}, \mathrm{def}_{\mathcal{Q}} \rangle$ *dans lequel :*

- $X^{\mathcal{Q}} = L^{\mathcal{P}} = \biguplus_{\mathcal{M} \in \mathcal{P}} X^{\mathcal{M}}$ *est l'ensemble des classes locales de la hiérarchie de modules ;*



(a) $\mathcal{P}$, une hiérarchie de modules (c'est à dire une hiérarchie de hiérarchies de classes)

(b) $\mathcal{Q}$, la hiérarchie classes de la hiérarchie de modules $\mathcal{P}$

FIG. 4.7: Hiérarchie de classes d'une hiérarchie de modules

- $\prec^{\mathcal{Q}}$ *est la fermeture transitive de l'union de la relation de raffinement* ($\ll^{\mathcal{P}}$) *et de la relation de spécialisation de chaque module* $\mathcal{M} \in X^{\mathcal{P}}$ ($\prec^{\mathcal{M}}$).
- $G^{\mathcal{Q}} = \bigcup_{\mathcal{M} \in X^{\mathcal{P}}} G^{\mathcal{M}}$ *est l'ensemble des propriétés globales de la hiérarchie de modules ;*
- $L^{\mathcal{Q}} = \bigcup_{\mathcal{M} \in X^{\mathcal{P}}} L^{\mathcal{M}}$ *est l'ensemble des propriétés locales de la hiérarchie de modules ;*
- $N^{\mathcal{Q}} = N^{\mathcal{P}}$ *est l'ensemble des noms ;*
- $\mathrm{nom}_{\mathcal{Q}}$, $\mathrm{glob}_{\mathcal{Q}}$, $\mathrm{intro}_{\mathcal{Q}}$ *et* $\mathrm{def}_{\mathcal{Q}}$ *sont les unions des fonctions équivalentes pour chaque module* $\mathcal{M} \in X^{\mathcal{P}}$ *;*

La figure 4.7 illustre l'exemple d'une hiérarchie de modules et de la hiérarchie de classe associée. La notion de module a bien disparue, seules restent des classes et une relation de spécialisation : la hiérarchie $\mathcal{Q}$ peut alors être utilisée de la même façon qu'une hiérarchie de classe $\mathcal{H}$ pour déterminer les propriétés locales et globales héritées et détecter les éventuels conflits associés (conflits de propriétés globales et conflits de propriétés locales).

Toutefois, comme nous le voyons dans les sous-sections suivantes, certains de ces conflits peuvent être résolus si l'on prend en compte la distinction que l'on peut faire entre le raffinement et la spécialisation.

4.5.2 Conflit de propriétés globales

Outre le conflit de propriétés globales lié à la seule spécialisation — cf. figure 3.6 page 56 — on dénombre de base trois autres configurations de conflits liés au raffinement multiple (figure 4.8(a)), à l'ajout de super-classes (figure 4.8(b)) et à la combinaison d'héritage et de raffinement (figure 4.8(c)).

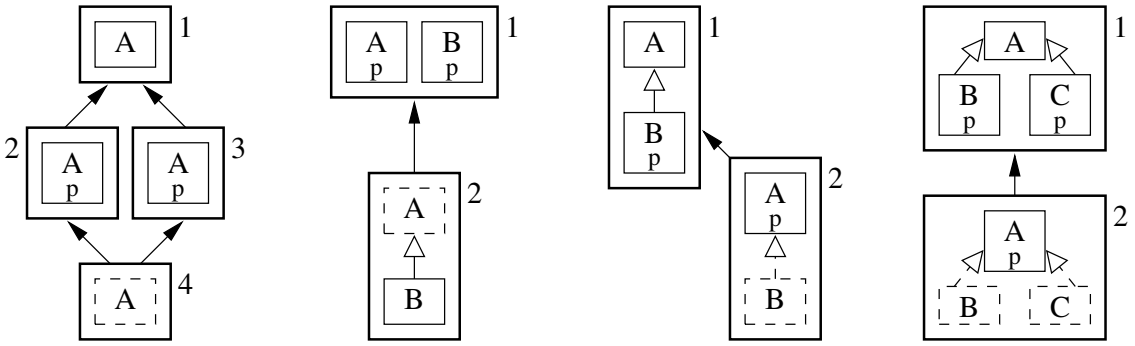


FIG. 4.8: Conflits de propriétés globales par raffinement et spécialisation

Généralisation de propriété globale

Le cas de la figure 4.8(c) est intéressant. En effet, il nous semble légitime de poser le principe suivant :

Principe 10 (Programmeur éclairé) *Un programmeur connaît les super-modules qu'il utilise pour développer un nouveau module.*

Ainsi, l'intention du programmeur qui a défini dans la classe `2::A` une propriété `p` n'était sans doute pas celle de provoquer un conflit de propriétés globales mais plutôt de généraliser à la classe `A` une propriété globale seulement connue de `B` : c'est ce que l'on appelle la *généralisation de propriété globale*. En effet, du point de vue du raffinement, l'ajout d'une propriété locale à une propriété globale n'a pas de raison de se limiter à la redéfinition dans des sous-classes.

Formalisme

La modification du méta-modèle pour qu'il prenne en compte la généralisation des propriétés globales consiste, pour chaque classe $c \in X^\Omega$, à partitionner l'ensemble des propriétés globales (G_c^Ω) en trois :

- les propriétés globales héritées : $G_{\uparrow c}^\Omega$;
- les propriétés globales introduites : G_{+c}^Ω ;
- les propriétés globales généralisées : $G_{\leftarrow c}^\Omega$;

L'ensemble des propriétés globales généralisées d'une classe locale $c \in X^\Omega$ est inclus dans un ensemble plus grand, celui des propriétés globales généralisables, noté $G_{\leftarrow ?c}^\Omega$, qui sont les propriétés globales des classes raffinées par les classes qui spécialisent c (et qui ne sont pas déjà héritées par c) :

$$G_{\leftarrow ?c}^\Omega = \bigcup_{c' \prec^{\mathcal{M}} c} \bigcup_{c' \ll^{\mathcal{P}} c''} G_{c''}^\Omega \setminus G_{\uparrow c}^\Omega \quad \text{avec } \mathcal{M} = \text{def}_{\mathcal{P}}(\text{c}) \ . \tag{4.7}$$

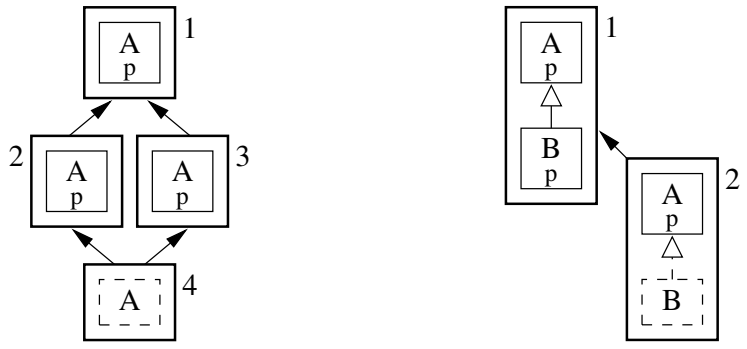


FIG. 4.9: Conflits de propriétés locales par raffinement et spécialisation

Dans l'exemple de la figure 4.8(c), les classes qui spécialisent `2::A` sont $\{$`2::B`$\}$; les classes raffinées par $\{$`2::B`$\}$ sont $\{$`1::B`$\}$; les propriétés globales de $\{$`1::B`$\}$ sont la propriété globale introduite par `p` dans `1::B` ; or la classe `2::A` n'hérite pas de cette propriété globale ; donc cette propriété globale est généralisable par cette classe.

Dans une classe $c \in X^{\mathrm{Q}}$, la définition d'une propriété locale $l \in L_c^{\mathrm{Q}}$ peut alors correspondre à trois mécanismes (et non plus deux) :

- la redéfinition d'une propriété globale, si le nom de la propriété locale correspond à celui d'une propriété globale héritée : $\mathrm{nom}_{\mathrm{Q}}(l) \in \mathrm{nom}_{\mathrm{Q}}^{-1}(G^{\mathrm{Q}}_{\uparrow c})$;
- la généralisation d'une propriété globale, si le nom de la propriété locale correspond à celui d'une propriété globale généralisable : $\mathrm{nom}_{\mathrm{Q}}(l) \in \mathrm{nom}_{\mathrm{Q}}^{-1}(G^{\mathrm{Q}}_{\leftarrow c})$;
- l'introduction d'une nouvelle propriété globale, sinon.

Conflits restants

Cette variation du modèle ne libère pas toutefois des conflits de propriétés globales liés à la combinaison du raffinement et de l'héritage. En effet, pour une classe donnée, deux propriétés globales généralisables qui sont homonymes ne peuvent pas être généralisées. La figure 4.8(d) montre un conflit de propriétés globales lié à cette impossibilité de généralisation.

Dans tous les cas, les conflits de propriétés globales peut être levés par les techniques que nous avons présentées dans la section 3.4.3 page 56.

4.5.3 Conflit de propriétés locales

Outre le conflit de propriétés locales lié à la seule spécialisation — cf. figure 3.7 page 66 — on dénombre de base deux autres configurations de conflits liées au raffinement multiple (figure 4.9(a)) et à la combinaison d'héritage et de raffinement (figure 4.9(b)).

Définition incrémentale

Pour l'exemple 4.9(b), si l'on se réfère à nouveau au principe 10, l'intention du programmeur du module `2` n'est pas de provoquer un conflit de propriétés locales dans la classe `2::B` mais d'hériter la propriété locale `p` définie dans `1::B`. En effet, c'est le comportement naturel si l'on interprète le raffinement comme une définition incrémentale des classes.

Il est facile de prendre en compte ce comportement : il suffit de spécifier une *sémantique additionnelle*. Elle consiste à hériter les propriétés dans $\text{spec}'_\Omega : X^\Omega \times G^\Omega \to \mathcal{P}(L^\Omega)$ définie par :

$$\text{spec}'_\Omega = \min_{\lhd^\Omega}(\text{spec}_\Omega) \ .$$

où l'ordre partiel $\lhd^\Omega$ entre propriétés locales L^Ω est défini par :

$$\begin{aligned} l' \lhd^\Omega l \quad \overset{\text{def}}{\Longleftrightarrow} \quad & \exists c \in X^\mathcal{M}, \wedge\, (\text{glob}_\Omega(l) = \text{glob}_\Omega(l')) \\ & \wedge\, (\text{def}_\Omega(l') \preceq^\mathcal{M} c) \\ & (\text{glob}_\mathcal{P}(\text{def}_\Omega(l)) = \text{glob}_\mathcal{P}(c)) \quad \text{avec } \mathcal{M} = \text{def}_\mathcal{P}(\text{def}_\Omega(l')) \ . \end{aligned}$$

Remarque : Il est bien évident que cette sémantique additionnelle est conforme à la résolution par défaut : une propriété locale $l \in L^\Omega$ héritée par une classe $c \in X^\Omega$ appartient bien à $\text{spec}_\Omega(c, \text{glob}_\Omega(l))$.

Cette sémantique additionnelle permet de résoudre le conflit de la figure 4.9(b) : puisque pour l la propriété locale `p` de la classe `2::A` et l' celle de la classe `1::B`, on a bien $l' \lhd^\Omega$'). Et dans le cas de la figure 4.9(a) le conflit n'est pas résolu : les propriétés locales `p` des modules `2` et `3` sont incomparables par la relation $\lhd^\Omega$.

4.5.4 Typage statique

Le raffinement est compatible avec toutes les politiques de typage — qu'elles soient sûres ou non. Nous nous intéressons ici seulement au sous-typage mais les autres éléments de conformité comme les contrats ou les exceptions déclarées devraient être également examinés. Bien évidemment, la vérification du typage ne se fait pas globalement au niveau du programme mais localement au niveau de chaque module.

Pour rappel, nous partons du principe que les types correspondent à des classes locales et que dans un module $\mathcal{M}$ la relation de sous-typage ($<:$) n'est autre que la relation de spécialisation ($\preceq^\mathcal{M}$). Ainsi, à chaque annotation de type, on doit pouvoir faire correspondre une classe locale du module courant.

Typage sûr

Dans un cadre de typage sûr, le raffinement et la spécialisation imposent la même politique de contravariance. Ainsi, les règles de conformité associées à la relation $\prec^\Omega$

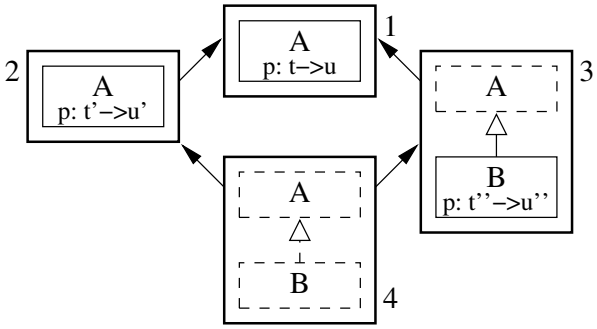


FIG. 4.10: Raffinement et typage

sont les mêmes que celles que nous avons associées à la relation $\prec^{\mathcal{H}}$ dans le chapitre précédent.

Soit la hiérarchie de modules de la figure 4.10 dans laquelle un module `1` définit une classe locale `1::A` qui définit une propriété locale l de nom `p` et de signature $t \to u$; et un sous-module `2` raffine cette classe en redéfinissant l par une propriété locale l' de signature $t' \to u'$. Le typage sûr nous impose la règle de contravariance, c'est-à-dire $t <: t'$ et $u' <: u$.

Toutefois, la même règle de contravariance s'applique à la spécialisation. Ainsi, dans un troisième module `3`, dépendant du module `1` mais indépendant du module `2`, la classe `3::A` est spécialisée par une classe `3::B` qui redéfinit la propriété locale l par une propriété locale l'' de signature $t'' \to u''$. La règle de contravariance nous impose alors que $t <: t''$ et $u'' <: u$.

Soit alors un quatrième module `4` dépendant à la fois des modules `3` et `4` mais ne définissant rien. La propriété locale héritée par `4::A` est l' et celle héritée par `4::B` est l''. La règle de contravariance impose cette fois que $t' <: t''$ et $u'' <: u'$.

Or rien ne permet de le garantir de façon modulaire sauf si $t = t'$ et $u = u'$, c'est-à-dire si l'on impose l'invariance des types lors du raffinement.

Typage non sûr

Une politique de typage covariante impose que la conformité soit basée sur la règle de covariance. Ainsi, en reprenant l'exemple du module `3` de la figure 4.10, la règle de covariance impose que $t'' <: t$ et $u'' <: u$.

Le point restant à éclaircir concerne la pertinence de la règle de covariance pour le raffinement de classes. En fait, lorsque l'on considère le raffinement d'une classe c par une classe c', c et c' correspondent à la « même classe » puisqu'elles ont les mêmes instances. C'est pourquoi, on ne peut pas prétendre que la sémantique du raffinement est celle de la spécialisation, et l'argumentation en faveur de la covariance ne peut donc pas s'appliquer.

Au contraire, le raffinement d'une classe correspond à l'intégration d'une nouvelle préoccupation. C'est-à-dire à l'extension de son comportement et peut légitimement s'accompagner d'une extension de ses méthodes afin qu'elles puissent accepter plus de valeurs en paramètre (donc contravariance du type des paramètres) et retourner plus de valeurs en résultat (donc contravariance du type de retour[5]).

Cas des constructeurs

Dans les langages statiques sans raffinement, les constructeurs d'instances ne sont pas soumis au polymorphisme : le type dynamique des instances qui seront créées est déjà déterminé par le type statique utilisé dans le constructeur. Ainsi au niveau des langages, soit les méthodes particulières qui ont un rôle de constructeur d'instances ne sont pas héritées (JAVA ou C++), soit c'est leur rôle de constructeur qui ne l'est pas (EIFFEL).

Avec le raffinement, les choses sont un peu différentes. Le type dynamique des instances qui seront créées est statiquement inconnu dans le module puisque la classe locale que le module manipule statiquement peut être raffinée dans d'éventuels sous-modules. Ainsi, lors d'un raffinement de classes, les constructeurs doivent d'une part être pleinement héritables et d'autre part, les classes doivent s'assurer que les constructeurs introduits dans les classes qu'elles raffinent restent cohérents — en les redéfinissant si nécessaire.

4.6 Autres travaux

MULTIJAVA [Clifton *et al.*, 2000] propose des unités de compilation, analogues aux modules présentés ici, qui sont munis d'une relation de dépendance via le mot-clé `import`. Il permet d'étendre des classes existantes en ajoutant des fonctions aux classes par une syntaxe *ad hoc*. Par contre la redéfinition de méthodes, l'ajout d'attributs ou la déclaration d'implémentation d'interfaces JAVA ne sont pas permis. Néanmoins, MULTIJAVA est compatible avec la compilation séparée et le chargement dynamique. Il propose également une mise en œuvre des multi-méthodes.

Les *classboxes* [Bergel *et al.*, 2003] permettent d'étendre les classes en SMALLTALK par des ajouts ou des redéfinitions de méthodes et d'attributs tout en contrôlant la visibilité des ajouts puisque ces amendements n'ont que des impacts locaux, les réponses aux envois de message étant déterminés à la fois par le receveur et par la *classbox*. Ainsi, contrairement à notre contribution, les amendements de classes apportés par une *classbox* ne sont appliqués qu'à cette *classbox* et aux *classboxes* qui l'importent,

[5]Ce qui n'est pas sûr au niveau des types mais vu le titre de la section, ce n'est sans doute qu'un point de détail.

les envois de message provenant d'autres *classboxes* ne seront donc pas affectés par la modification.

MixJuice [Ichisugi et Tanaka, 2002], basé sur le langage Java, propose des modules en relation de dépendance et des amendements de classes autorisant la définition et redéfinition de méthodes, l'ajout d'attributs et la déclaration supplémentaire d'implémentation d'interfaces Java. En cas de dépendance multiple entre modules, les conflits de propriétés globales sont résolus par désignation explicite, les conflits de propriétés locales sont résolus par une linéarisation à la Clos. L'approche est compatible avec la compilation séparée mais ne permet pas le chargement dynamique. MixJuice ne fait pas apparaître l'analogie entre les classes et les modules ni n'analyse les différentes configurations conflictuelles et leurs résolutions. C'est avant tout une implémentation fonctionnelle du mécanisme de raffinement (dans les limites imposées par Java d'invariance des types et d'héritage simple), qui permet de montrer la faisabilité d'une telle approche.

Les « hiérarchies d'ordre supérieur » de [Ernst, 2003] ressemblent beaucoup à nos modules. Les motivations sont identiques et l'isomorphisme des méta-modèles des classes (ordre 1) et des modules (ordre 2) permettrait une généralisation immédiate à un ordre quelconque. Ce n'est cependant pas notre objectif et nous ne poussons pas la métaphore des classes aussi loin : finalement, comme le dit [Szyperski, 1992], nos modules ne sont pas des classes, au moins en ce sens qu'elles n'ont pas d'instances. Un module est de la dimension d'un programme : on comprend bien qu'il puisse avoir une unique instance, correspondant aux données statiques d'une exécution. Mais si l'on poursuit la métaphore, une classe d'ordre 3 serait un système d'exploitation, et la proposition des hiérarchies d'ordre supérieur devient un peu mystérieuse. Enfin, techniquement, les deux approches diffèrent sensiblement sur le traitement de l'héritage multiple, dont nous avons vu qu'il était inévitable. [Ernst, 2003] propose une combinaison de *mixins* totalement ordonnés alors que notre proposition repose sur l'interprétation de l'héritage multiple dans le méta-modèle, la seule bonne façon d'aborder l'héritage multiple selon nous.

4.7 Conclusion

Dans ce chapitre, nous avons formalisé le principe du raffinement de classes qui permet aux langages à modules, à objets, en typage statique et en héritage multiple de définir « de façon incrémentale » une même classe à travers plusieurs modules. Cette formalisation a servi à la spécification du langage PRM et en partie à l'implémentation du compilateur `prmc` (avec toutefois, un bémol sur la généralisation de propriété globale qui est actuellement refusée par le compilateur).

Bien que les modules et les classes soient d'utilité fondamentalement différente [Szyperski, 1992], cette formalisation est basée sur une analogie structurelle stricte entre ces

deux notions puisqu'elles sont décrites par des méta-modèles semblables[6].

La limitation principale du raffinement vient de ce qu'il n'est pas possible *a posteriori* de revenir sur des choix déjà effectués comme par exemple la suppression de propriétés ou de relations de spécialisation. Toutefois, cette monotonie dans la définition des classes n'est pas nouvelle puisqu'elle est présente également dans la spécialisation de classes.

[6] « Import is inheritance, why we need both. » pour paraphraser et partiellement contredire [Szyperski, 1992].

Chapitre 5

De la spécification de langages aux techniques efficaces de compilation

Préambule

Ce chapitre sert de transition entre la spécification de langages de programmation et leur compilation efficace. Comme il est très court, nous avons pensé à compenser par l'utilisation de très longs titres de section.

De la spécification de langages de programmation à la spécification d'un langage de programmation particulier, comme PRM par exemple

Dans un premier temps, notre travail sur la spécification des langages à objets nous a conduit à nous pencher plus particulièrement sur les problématiques de spécialisation de classes et d'héritage de propriétés — surtout lorsque spécialisation et héritage sont multiples. Dans un second temps, nous nous sommes intéressés aux problématiques de modules et de raffinements de classes, là aussi dans l'optique du « multiple » : dépendance multiple de modules, raffinement multiple de classes et combinaison de raffinement et de spécialisation de classes. Dans les deux cas nous avons proposé une modélisation et formalisé les mécanismes mis en œuvre.

Afin de concrétiser ces mécanismes nous avons spécifié un nouveau langage de programmation : PRM (cf annexe A pour la spécification du langage). Deux différences majeures existent entre notre travail sur les caractéristiques des langages et notre travail sur la spécification du langage PRM :

La nécessité de faire des choix. Un langage de programmation ne peut pas intégrer

toutes les variations d'un mécanisme particulier — et même s'il le pouvait, il ne le devrait pas sous peine d'enfreindre le principe 2. Il faut donc pour chaque trait particulier du langage choisir parmi les différentes possibilités.

La spécification doit être complète. Lorsque l'on spécifie un langage, on ne peut pas se contenter de ne spécifier que des fragments : il faut spécifier le langage en entier. Et comme il faut faire des choix, cela revient pour chaque caractéristique du langage à devoir décider entre différentes spécifications. La difficulté est alors de devoir trancher entre diverses spécifications sans avoir eu matériellement le temps d'étudier pleinement pour chaque caractéristique chacun des choix possibles.

Malgré tout, pour chacune des caractéristiques du langage PRM nous avons essayé de fournir le mécanisme qui respecte les nombreux principes que nous avons énoncés jusqu'à présent, en particulier l'intuitivité (principe 1 page 2) et la simplicité (principe 2 page 14).

De la spécification des langages de programmation à leur compilation, si possible efficace

La compilation consiste à traduire le code d'un programme (écrit dans le langage de programmation considéré) en code machine (correspondant au mode de fonctionnement de la machine).

Malheureusement, tous les langages ne sont pas logés à la même enseigne du point de vue de la compilation, en effet :

Principe 11 (Compilation facile) *Un langage de programmation est d'autant plus facile à compiler qu'il est proche du mode de fonctionnement de la machine.*

Premier corollaire : les bons langages de programmation sont difficiles à compiler puisque selon le principe 1, ils sont proches du mode de pensée humain, donc éloignés du mode de fonctionnement de la machine.

Second corollaire : le compilateur le plus simple, à savoir la fonction *identité*, permet de compiler du code machine en ce même code machine — puisque le code machine est le langage qui correspond exactement au mode de fonctionnement de la machine.

Pourquoi est-ce plus facile de compiler efficacement[1] un langage proche du mode de fonctionnement de la machine ? Tout simplement parce que les prérogatives du compilateur augmentent en même temps que la distance au mode de fonctionnement de la

[1]Ici, la notion d'efficacité concerne à la fois l'efficacité du processus de compilation et l'efficacité du résultat de la compilation.

machine. Dit autrement, dans un langage de programmation de *bas niveau*[2], la charge de l'implémentation revient plus au programmeur que dans un langage haut niveau.

Par exemple, en assembleur, c'est le programmeur qui doit s'occuper de l'utilisation des registres et de la pile. En C, c'est le compilateur qui s'en occupe.

Les directives de compilation, ou quand les langages de programmation se mêlent de ce qui ne les regarde pas, en l'occurrence de la compilation

Afin de réduire la distance entre un langage de programmation et le mode de fonctionnement des machines, de nombreux langages incluent des *directives de compilation*. Elles consistent généralement en des annotations particulières destinées au compilateur et lui expliquant comment implémenter tel ou tel mécanisme. L'efficacité (ou l'inefficacité) de la compilation est alors du ressort du programmeur : le compilateur s'en lave les mains.

Ainsi, les directives de compilation permettent au programmeur de prendre le pas sur le compilateur lorsqu'il sait que ce dernier risque de ne pas compiler ses programmes de façon optimale. En contrepartie, le programmeur peut avoir à se restreindre au niveau de la sémantique.

Il est bien évident, au vu du discours que nous avons tenu jusque là, que nous réprouvons l'utilisation de telles directives : elles ne respectent ni le principe 1, puisqu'elles imposent au programmeur de se préoccuper du mode de fonctionnement de la machine, ni le principe 2, puisqu'elles ajoutent une complexité au langage considéré. De façon intégriste, nous dirons même que la présence de directives de compilation dans la spécification d'un langage de programmation est la marque de deux manques de compétence :

- le manque de compétence à spécifier un langage de qualité — par le non-respect des principes 1 et 2 ;
- le manque de compétence à proposer un compilateur efficace — par la nécessité de demander au programmeur comment un programme doit être implémenté.

Toutefois, ce point de vue intégriste n'a que peu de justification dans un monde en perpétuelle évolution où la recherche scientifique propose jour après jour des nouveaux traits de langage et techniques d'implémentation. C'est pourquoi nous classons ces directives de compilation en deux grandes familles : les *directives bénignes*, qui sont

[2]Un langage de programmation de *bas niveau* est un langage de programmation proche du mode de fonctionnement de la machine ; et de façon symétrique, un langage de *haut niveau* est un langage éloigné du mode de fonctionnement de la machine. Toutefois, en toute généralité, un langage de haut niveau n'est pas systématiquement un langage proche du mode de pensée humain.

acceptables dans un langage de programmation ; et les *directives malignes*, qui le sont beaucoup moins.

Directives de compilation bénignes

Les directives bénignes de compilation ne font pas partie de la sémantique des programmes, ou au pire se contentent d'une *restriction de la sémantique*. Dans tous les cas, les directives de compilation bénignes peuvent être ignorées. Ainsi, si un programme rempli de directives bénignes se comporte d'une certaine façon, il se comporte exactement de la même façon une fois qu'on lui a retiré toutes ses directives bénignes. Toutefois, l'inverse n'a pas à être vrai : ajouter une directive de compilation bénigne à un programme valide peut le rendre invalide, c'est que l'on appelle la *restriction de la sémantique*.

Par exemple, le mot clé `register` en C est une directive de compilation bénigne. Il permet de déclarer qu'une variable locale doit être stockée dans un registre du processeur et non dans la pile. La restriction de sémantique est qu'une telle variable ne peut plus être référencée — pas le droit à l'opérateur `&` sur cette variable.

Les mots clés `final` de JAVA et `frozen` d'EIFFEL sont aussi, d'une certaine façon, des directives de compilation bénignes : ils permettent d'invoquer des méthodes sans avoir recours à une liaison tardive. La restriction de la sémantique est qu'une telle méthode ne peut plus être redéfinie dans une sous-classe. Dans certaines documentations, le côté « efficace » de ces deux mots clés n'est pas explicitement mentionné[3]. Toutefois, ces deux mots clés ont bien les caractéristiques des directives bénignes : ils peuvent être ignorés sans changer la sémantique du programme, ils permettent potentiellement une implémentation plus efficace et ils n'ont que peu de justification en soi : en effet interdire la redéfinition va à l'encontre de l'approche par objet — [Meyer, 1997] fait la même remarque et ne justifie l'utilisation du mot clé `frozen` que pour les méthodes primitives du langage comme `standard_is_equal`.

Du point de vue du développement du compilateur, les directives de compilation bénignes peuvent être respectées ou simplement ignorées :

- un compilateur peut utiliser les directives pour produire des exécutables plus efficaces, c'est leur usage premier ;
- un compilateur peut ignorer les directives car il implémente des techniques de compilation efficaces qui les rendent superflues. Par exemple, les compilateurs C modernes ignorent[4] généralement toutes les directives de compilation.

[3] C'est toutefois le cas de [Gosling, 2000] qui vante l'efficacité comme premier avantage du mot clé `final`.

[4] Certains vont même jusqu'à générer des *warning* et autres *deprecated* pour impressionner le programmeur.

– un compilateur en développement[5] peut ignorer en toute sécurité les directives de compilation, la seule contrepartie est une éventuelle baisse des performances.

Ainsi, la seule justification d'une directive de compilation bénigne est celle de palier une faiblesse des compilateurs en attendant qu'une technique d'implémentation performante apparaisse un jour.

Directives de compilation malignes

Les directives de compilation malignes ont un impact sur la sémantique des programmes mais leur justification première est avant tout l'efficacité. Contrairement aux directives bénignes, les directives de compilation malignes ne peuvent pas être ignorées par un compilateur puisque la sémantique des programmes en dépend.

Comment reconnaître une directive de compilation maligne d'un trait légitime d'un langage ? Il n'y a pas de règle universelle mais nous pouvons proposer les critères suivants :

– son utilisation limite les qualités du langage considéré (cf. section 2.3 page 11) ;
– son absence dans la spécification ne limite pas les qualités du langage considéré ;
– son utilisation fait miroiter un gain de performances ;
– son utilisation ne peut pas être ignorée.

Si un trait donné satisfait les quatre critères[6], nous considérons qu'il s'agit d'une directive de compilation maligne.

Comme exemple de directive maligne, la candidate idéale ne peut être que le mot clé `virtual`[7] de C++ dans ces deux acceptions.

De la spécification d'un langage de programmation particulier, en l'occurrence PRM, à sa compilation, si possible efficace

Après avoir spécifié le langage PRM dans la première partie de ce mémoire, nous attaquons la seconde partie avec l'objectif de compiler ce langage efficacement.

Toutefois, malgré le fait de porter la double casquette de la spécification et de la compilation du langage PRM, nous avons choisi de pas faire interférer la spécification

[5] Ou un compilateur qui expérimente des techniques de compilation sur un nombre restreint de traits du langage considéré.

[6] Si les seuls trois premiers critères sont satisfaits, il s'agit alors d'une directive de compilation bénigne.

[7] En l'occurrence, `virtual` est plutôt une « anti-directive » de compilation maligne puisque c'est l'absence du mot clé qui permet un éventuel gain de performance.

avec la compilation : la compilation dépend de la spécification et non l'inverse[8].

La partie qui suit présente d'abord les techniques de compilation : le chapitre 6 présente l'implémentation des langages à objets, en particulier les techniques efficaces qui rendent superflues les directives de compilation `final`/`frozen` et `virtual` ; le chapitre 7 présente un schéma de compilation séparé efficace.

Ensuite, le chapitre 8 est consacré à `prmc`, le compilateur PRM que nous avons développé. Toutefois, comme pour la spécification d'un langage, la réalisation d'un compilateur particulier diffère de la proposition de techniques de compilation : le compilateur doit être le plus complet possible et ne pas se restreindre à un sous ensemble du langage. Heureusement pour nous, le langage PRM est simple et son noyau minimal.

[8]Avec du recul, nous nous disons que peut-être nous aurions pu éviter de nombreuses nuits blanches si nous n'avions pas été aussi inflexible à ce sujet.

Deuxième partie

Compilation séparée et efficace des langages à objets

Chapitre 6

Compilation et langages à objets

Préambule

Ce chapitre, principalement inspiré de [Ducournau, 2002b; Ducournau, 2006], présente les techniques de compilation utilisées dans le cadre des langages à objets. Après avoir présenté rapidement le cadre (compilateur + objets), nous présentons les techniques d'implémentation des langages de programmation statiquement typés. Nous partons des langages en spécialisation simple, montrons les difficultés d'implémentation liées à la spécialisation multiple puis présentons des techniques globales d'implémentation qui permettent de résoudre ces difficultés au prix d'une perte de modularité de la compilation.

6.1 Introduction

6.1.1 Compilation

Qu'est-ce qu'un compilateur ?

Il est d'usage dans tout travail sur la compilation de préciser ce que l'on entend par *compilateur*.

En pratique, il existe deux façons d'exécuter un programme. La première, appelée *interprétation*, consiste à utiliser un *interpréteur* qui va lire et exécuter le programme. Le seconde, appelée *compilation*, consiste à utiliser un *compilateur* qui va lire et traduire le programme sous forme d'*exécutable*, c'est-à-dire un fichier écrit en *code machine*, qui peut être exécuté directement par la machine.

L'intérêt de la compilation sur l'interprétation est qu'elle permet d'exécuter les programmes de façon plus efficace. L'explication vient du fait que certains calculs sont réalisés une fois pour toutes lors de la compilation et ne seront plus à faire lors de

l'exécution de l'exécutable.

Le terme de compilation peut être étendu à la traduction d'un programme écrit dans un langage source de haut niveau vers un langage destination de moins haut niveau.

En réalité, les choses sont moins tranchées et de nombreuses façons intermédiaires entre interprétation et compilation existent. Par exemple :

- *machines virtuelles* : un programme est compilé en un langage intermédiaire qui est lui-même interprété — c'est par exemple le cas en JAVA.
- *compilation juste-à-temps* (ou historiquement *traduction dynamique*) qui consiste lors de l'interprétation ou du chargement à traduire au fur et à mesure le programme en langage machine.

Performance d'un compilateur

La performance d'un compilateur peut se représenter de plusieurs façons. En premier lieu, il y a bien sûr la « qualité » du travail réalisé par le compilateur : c'est-à-dire de l'exécutable produit. Un compilateur est donc performant si les exécutables qu'il produits sont :

- petits — on parle ici de la taille du fichier binaire ;
- rapides — l'exécutable doit être efficace temporellement ;
- économes en mémoire — l'exécutable doit être peu gourmand en ressource mémoire.

En second lieu, il y a également la performance du compilateur lui-même :

- rapidité — le temps que met le compilateur pour traduire un programme en un exécutable doit être le plus court possible ;
- économie en mémoire — le compilateur doit être peu gourmand en ressource mémoire pour traduire un programme en un exécutable.

6.1.2 Implémentation des langages à objets

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

Mécanismes considérés

Dans le contexte particulier des langages à objets, sont concernés les deux mécanismes fondamentaux :

- d'accès aux attributs d'un objet, en lecture et en écriture ;

– d'envoi de message, c'est-à-dire de sélection et d'appel de la méthode correspondant au type dynamique du receveur — liaison tardive.

À ces deux mécanismes fondamentaux, s'ajoutent divers mécanismes secondaires qui sont tous indispensables dans n'importe quel langage à objets et dont l'implémentation n'est pas toujours aussi triviale qu'il n'y paraît :

– la vérification dynamique de type, nécessaire pour les constructions comme le casting descendant, doit être implémentée efficacement, idéalement en temps constant ;
– l'implémentation doit en particulier gérer la redéfinition des types dans les sous-classes, qu'il s'agisse du type de retour, des paramètres ou des attributs, et que cette redéfinition soit sûre du point de vue des types, ou pas ;

Toutefois, dans ce chapitre nous n'aborderons pas les problématiques suivantes :

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

Le lecteur pourra toutefois se reporter à [Ducournau, 2002b].

Efficacité spatiale des exécutables

Sur le plan spatial, on peut distinguer trois catégories d'occupation mémoire qu'il faut prendre en compte :

– une part complètement dynamique consiste en l'implémentation des objets eux-mêmes, en général comme une table de leurs attributs, pointant sur les structures de données associées à leur classe ;
– une part statique, représentée par des structures de données associées aux classes, en lecture seule ;
– la part du code proprement dit, dans laquelle la mise en œuvre d'un mécanisme peut occuper une séquence d'instructions plus ou moins longue.

Dans le même ordre d'idées, il faut aussi s'intéresser à la gestion automatique de la mémoire : l'implémentation des objets doit rester compatible avec un ramasse-miettes efficace (cf. section 8.1.4 page 164).

L'efficacité temporelle et l'efficacité spatiale varient, en général, en sens inverse : un critère unique est donc impossible et le choix résultera toujours d'un compromis qui pourra dépendre du contexte. Si une analyse de la mémoire statique a été menée dans le cadre des techniques globales qui nécessitent un compactage de tables [Ducournau, 1997], l'efficacité spatiale a été relativement peu analysée. Quant à la mémoire dynamique, elle ne semblait pas poser beaucoup de problèmes : les techniques standards (cf. section 6.2) nécessitent un surcoût faible et la coloration (cf. section 6.4.5) est la seule

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

Efficacité temporelle des exécutables

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

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

En tout état de cause, notre appréciation de l'efficacité des différentes techniques ne dépassera pas un niveau très intuitif, basé éventuellement sur un pseudo-code écrit dans un pseudo-langage d'assemblage. Pour plus de détails sur ce pseudo-code, le lecteur se reportera à l'annexe C.

Caractéristiques des langages considérés

On peut classer les langages à typage statique en trois catégories en fonction de la complexité de la relation de spécialisation entre classes (cf. section 3.3 page 45) :

- les langages en spécialisation simple, comme SIMULA ou dans une moindre mesure C++, c'est-à-dire sans le mot clé `virtual` et en héritage simple ;
- les langages avec héritage d'interfaces, comme JAVA, EIFFEL# ou C# ;

– les langages pleinement en spécialisation multiple, comme C++ [1], Eiffel et bien évidemment PRM.

Dans le cadre de ce chapitre nous ne nous intéresserons qu'aux deux extrémités, c'est-à-dire à l'implémentation standard de la spécialisation simple et l'implémentation de la spécialisation multiple.

Remarque : Dans [Ducournau, 2002b; Ducournau, 2006], les trois catégories sont respectivement désignées par *héritage et sous-typage simples*, *héritage simple et sous-typage multiple* et *héritage et sous-typage multiples*. Nous avons choisi de ne pas garder cette dénomination puisque nous considérons que Java est quand même un langage en spécialisation multiple (cf. chapitre 3).

6.2 Spécialisation simple

6.2.1 Le principe

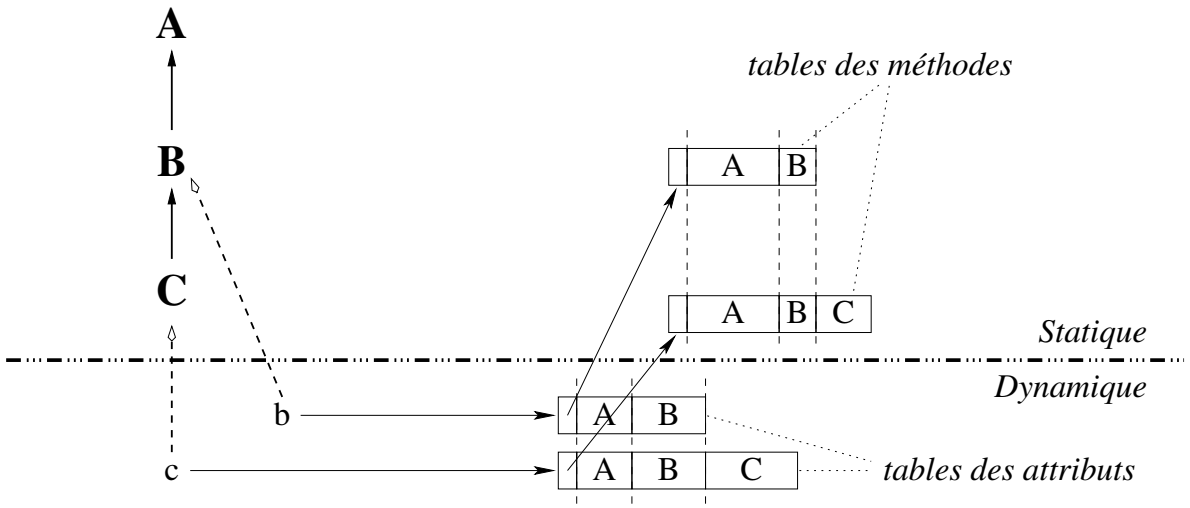
En spécialisation simple, un objet est implémenté comme une table de ses attributs, auxquels s'ajoute un pointeur sur une table de méthodes commune à toutes les instances d'une même classe (figure 6.1). La première table contient, pour chaque objet, la valeur de ses attributs : nous parlons de *table d'instance* ou de *table des attributs*. La seconde table contient les adresses des méthodes : nous parlons de *table de classe*, de *table des méthodes* ou de VFT (*Virtual Function Table*, table des fonctions virtuelles pour reprendre le jargon C++).

La relation de cette implémentation avec le méta-modèle présenté dans le chapitre 3 est la suivante : chaque propriété globale a une *position* déterminée[2] dans la table correspondante (table de classe pour les méthodes et table d'instance pour les attributs). Pour les méthodes, l'entrée correspondante dans la table des méthodes contient l'adresse de la propriété locale. Pour les attributs, l'entrée correspondante dans la table des attributs contient la valeur de l'attribut.

Remarque : Le rôle des tables de classes ne se limite pas à contenir des pointeurs vers des méthodes : elles sont également utilisées pour stocker les informations nécessaires à la vérification de sous-typage (cf. section 6.2.2). Dans le cas de la redéfinition non sûre des types des méthodes (paramètres et valeur de retour), cf. section 6.2.3, plusieurs indices peuvent être associés à une même propriété globale : un seul associé à l'adresse de la propriété locale, les autres associés à l'adresse d'une fonction chargée d'effectuer les vérifications de sous-typage et l'appel à la propriété locale. Dans le cadre de la redéfinition des types d'attributs, un indice peut être réservé dans la table des méthodes pour contenir le type des attributs et servira à la vérification de sous-typage lors de certains accès.

[1] Avec usage systématique du mot clé `virtual`.

[2] On parlera d'*indice*, de *décalage* (*offset*) voire de *couleur* (cf. section 6.4.5).



Trois classes : `A`, `B` et `C` et deux objets : `b` instance directe de `B` et `c` instance directe de `C`.

FIG. 6.1: Structure des objets et des tables de méthodes en spécialisation simple

L'implémentation des objets et de l'envoi de message se caractérise donc par les deux invariants suivants :

Invariant 1 *La valeur d'une référence — paramètre, variable, attribut ou valeur de retour d'un appel fonctionnel — sur un objet est invariante relativement à son type statique.*

Invariant 2 *Chaque propriété globale (attribut ou méthode) a un indice non ambigu et invariant par spécialisation, donc indépendant du type statique du receveur.*

La spécialisation simple se caractérise ainsi par une invariance absolue vis-à-vis des types statiques, qui n'ont aucun rôle à l'exécution. L'implémentation se conforme donc à la sémantique fondamentale de l'approche objet, pour laquelle le type dynamique exprime l'essence de l'objet, le type statique étant purement contingent. On notera par la suite s le type statique d'une entité et d son type dynamique, avec $d <: s$[3]. Le type statique est celui qui a servi à annoter une entité d'un programme, alors que le type dynamique est la classe dont l'instanciation a créé l'objet qui value l'entité considérée.

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

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

[3] $<:$ est la relation de sous-typage (cf. section 3.8 page 76).

où `#tableOffset` est une constante, généralement 0 et `#methodOffset` est l'indice de la propriété globale. L'accès à un attribut est immédiat en lecture :

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

où `#attributeOffset` est l'indice de la propriété globale. Il est tout autant immédiat en écriture :

```
store attribute → [object + #attributeOffset]                   1
```

Remarque : Le pseudo-code qui illustre les différentes techniques est globalement emprunté à [Driesen, 1999] : s'il est représentatif du nombre d'instructions nécessaires, en tant que tel il ne l'est pas du nombre de cycles de processeur. Chaque séquence est accompagnée d'une estimation du nombre de cycles nécessaires pour son exécution : L représente la latence de chargement et vaut 2 ou 3, et B la latence de branchement indirect et peut valoir de 3 à 15. Cf. Annexe C.

Le calcul des tables de méthodes (pour la classe) et d'attributs (pour les instances) est un cas particulier de l'heuristique de coloration (cf. section 6.4.5 page 137). L'algorithme est le suivant : pour chaque classe dont la super-classe a déjà été compilée, on numérote les propriétés globales attributs (respectivement méthodes) introduites dans la classe, en partant de l'indice maximum des propriétés globales attributs (respectivement méthodes) de la super-classe. L'absence d'héritage multiple garantit la correction du résultat, c'est-à-dire qu'il n'y a jamais aucun problème avec les indices des méthodes héritées et qu'il n'est jamais nécessaire de vérifier que l'indice que l'on veut affecter n'est pas déjà occupé.

C'est l'implémentation de base de la plupart des langages à objets, aussi bien celle de C++, tant qu'on reste en héritage simple et « non virtuel » (sans usage du mot-clé `virtual` pour l'héritage, cf. section 3.3.2 page 50), que celle de Java en ne se servant pas des interfaces. La figure 6.1 résume cette implémentation.

6.2.2 Le casting

La notion de casting

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

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

– la *réinterprétation d'une zone mémoire* par une structure d'un autre type que celui qui l'a allouée ou initialisée : ce style de programmation — courant dans les langages, comme C ou PL/1, qui permettent l'arithmétique de pointeurs ou dont

les pointeurs ne sont pas typés — enfreint tous les canons de la programmation évoluée et du typage orthodoxe et n'a rien à voir avec la programmation objet ;
- la *conversion* ou *coercition* entre types voisins, qui agit par copie : typiquement, la conversion entre représentations différentes des nombres (simple ou double précision, etc.) ;
- la *migration d'instances*, telle qu'elle est spécifiée en CLOS par la fonction générique `change-class`, qui agit comme un effet de bord sur l'objet concerné qui devient instance d'une autre classe, tout en conservant son « identité » ;
- la *classification d'instances* est un cas particulier du précédent, dans lequel la classe cible est contrainte à être une sous-classe de la classe source : l'objet reste donc instance de la classe source, mais indirectement ; bien que cette fonctionnalité soit étrangère au monde de la programmation, [Ducournau et Pavillet., 2001] montre qu'elle n'est pas incompatible ;
- le *casting ascendant*, souvent qualifié d'implicite parce qu'il ne nécessite aucun mécanisme syntaxique, correspond à l'affectation (ou au passage de paramètre) polymorphe, lorsque l'on affecte à une entité `x` de type statique X la valeur d'une entité `y` d'un type statique Y sous-type de X (i.e. $Y <: X$). Ce soi-disant casting ascendant n'a aucune existence conceptuelle — c'est l'essence même du polymorphisme (dit d'inclusion) — mais il peut néanmoins nécessiter une implémentation non triviale ;
- le *casting descendant* (ou *downcast*) nécessite une vérification dynamique de type — donc un signalement potentiel d'exception — et revient à faire l'hypothèse qu'une entité de type statique X est en fait d'un sous-type Y : ce casting peut être effectué au travers d'une affectation ou d'un passage de paramètre, ou par une construction comme `dynamic-cast` en C++ ou les tentatives d'affectation (*assignment attempts*) en EIFFEL ; l'usage du casting descendant se justifie souvent par le fait que les modèles naturellement covariants sont implémentés dans des langages contravariants — cf. section 2.6.5 page 25.

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

Dans ce mémoire, nous ne considérerons le terme de casting que sous ses deux variantes ascendantes et descendantes, la première avec une justification purement implémentatoire.

À ces deux directions et sémantiques différentes du casting, on peut enfin rajouter une distinction suivant que le type cible est connu statiquement ou pas. On pourra donc parler aussi de *casting statique* et de *casting dynamique*[4]. Tel qu'il est pratiqué

[4]Sans que cela ait le moindre rapport avec les mots-clés `static_cast` et `dynamic_cast` de C++, qui sont tous les deux statiques.

explicitement par un programmeur, le casting est toujours statique, mais la redéfinition de type, peut se réaliser, au choix, par des castings statiques ou dynamiques. Le casting dynamique impose que l'identifiant du type cible soit accessible à partir des objets considérés.

Implémentation en spécialisation simple

En spécialisation simple, comme les références à l'objet ne dépendent pas de leur type statique, la notion de *casting ascendant* ne se justifie pas plus du point de vue implémentatoire que du point de vue conceptuel.

Le *casting descendant*, qui se réduit à une vérification dynamique de type, peut s'implémenter en mettant dans la table des méthodes des étiquettes du nom des super-classes successives : une instance de type statique A est une instance de type dynamique B, si l'objet possède, au bon endroit — par exemple à un indice Δ_B correspondant au nombre de méthodes des super-classes de B — une étiquette `"B"`. De fait, ces Δ ne dépendent pas du type statique de départ (ici A) : ils peuvent donc être donnés relativement à l'origine de l'objet. Cette technique simple et utilisable en compilation complètement séparée est efficace en temps, mais elle l'est moins en espace. Le pseudo-code serait le suivant :

```
load [object + #tableOffset] → table                    2L + 2
load [table + #downTypeOffset] → id
cmp id = #downTypeId
jne #fail
; casting OK
```

C'est encore un cas particulier de l'heuristique de coloration (cf. section 6.4.5) et il semble, d'après [Raynaud et Thierry, 2001], qu'elle ait été décrite d'abord par [Cohen, 1991].

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

$$n_2(c) = \max_{c' \preceq c}(n_1(c')) \ .$$

Alors :

$$c' \prec c \quad \Leftrightarrow \quad n_1(c) < n_1(c') \leq n_2(c) \ . \tag{6.1}$$

Seuls deux petits entiers sont nécessaires, dont l'un (n_1) peut servir à identifier la classe. La numérotation est globale et n'est pas incrémentale ce qui rend problématique un chargement ou une édition de liens dynamiques[5]. On notera que, lors du test (6.1), $n_1(c')$ est dynamique (c' est en fait le type dynamique d), alors que $n_1(c)$ et $n_2(c)$ sont statiques et peuvent être compilés comme des constantes de l'exécutable. Pour un surcoût non négligeable (les statistiques indiquent en moyenne un faible nombre d'attributs, de 5 à 10 attributs par classe selon [Ducournau, 2006]), on pourra mettre l'identifiant de classe n_1 dans l'objet lui-même.

6.2.3 Redéfinition de types

Redéfinition covariante du type de retour

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

Redéfinition covariante du type des paramètres

La redéfinition covariante du type des paramètres se justifie dans le cadre d'une politique de typage covariante comme c'est le cas en Eiffel ou en PRM (cf. section 2.6.5). Elle nécessite juste une vérification de type.

L'exemple PRM suivant illustre un exemple de redéfinition covariante du type des paramètres. On suppose également l'existence de variables locales (`a`, `b`, `t`, etc.) qui sont statiquement typées par leur majuscule (`A`, `B`, `T`, etc.) :

```
class A
        def toto(p: T)
        ...
end
class B
        def toto(p: U) # U <: T
        ...
end
...
let a: A; let b: B; let t: T; let u: U
a.toto(t); a.toto(u); b.toto(u)
```

Cette vérification peut se faire, au début de la méthode appelée, par la technique décrite précédemment. Ainsi, dans l'exemple précédent, cela revient à tester si `p` est d'un sous-type de `U` au début de la méthode `toto` définie dans la classe `B`. L'inconvénient est

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

que cette vérification est systématique : il n'est pas possible de l'éviter puisqu'elle fait partie de la méthode appelée. Cette vérification est superflue lorsque le type statique de l'argument est un sous-type du type du paramètre dans la méthode du type dynamique du receveur. C'est par exemple le cas de `b.toto(u)`.

Dans un contexte un peu différent, [Myers, 1995] propose l'utilisation de *thunks*[6], bien connue dans l'implémentation de C++ (cf. section 6.3) mais qui s'applique aussi ici. Un thunk est une petite fonction intermédiaire qui fait un petit traitement avant d'effectuer un branchement sur la vraie méthode.

Ici, au lieu d'affecter un indice à chaque sélecteur de méthode, on l'affecte à chaque signature, un peu comme si on voulait implémenter la surcharge statique (qui est bien entendu incompatible avec la redéfinition covariante). Ainsi, pour le listing précédent, on aurait un indice i_1 pour `toto(T)`, un indice i_2 pour `toto(U)` et non plus un seul indice pour `toto`.

Chaque appel de méthode fait appel à l'indice correspondant aux types statiques des arguments de l'appel. Ainsi, on utiliserait l'indice i_1 pour le deux site d'appel `a.toto(t)`, tandis que l'on utiliserait l'indice i_2 pour le site d'appel `b.toto(u)`. Pour le site d'appel `a.toto(u)`, on utiliserait i_1 puisqu'en compilation séparée chaque classe n'a un indice que pour les signatures des méthodes définies dans la classe ou ses super-classes.

Au niveau de la table des méthodes, à chaque indice on associe :

- l'adresse de la méthode si l'indice est celui de sa signature : c'est le cas pour l'indice i_1 et la méthode de la classe `A` ; c'est également le cas pour l'indice i_2 et la méthode de la classe `B` ;
- l'adresse d'un thunk, qui fait les vérifications de types et saute à l'adresse de la méthode, si l'indice est celui d'une signature plus générale (c'est-à-dire la signature d'une méthode d'une des super-classes) : c'est le cas pour l'indice i_1 et la méthode de la classe `B`.

Pour une propriété globale méthode et un type dynamique d donné, il y a au plus autant de thunks que de types statiques s (s super-classe de d, $d \preceq s$) (re-)définissant la méthode[7].

Redéfinition contravariante du type de retour

La redéfinition contravariante du type de retour pourrait se justifier dans le cadre du raffinement et d'une politique de typage non-sûr comme c'est le cas en PRM (cf. sec-

[6]L'étymologie du mot est obscure : il ne figure dans aucun de nos dictionnaires. Selon Wikipédia (cf. `http://en.wikipedia.org/wiki/Thunk`) le terme désigne entre autre un calcul différé. [Myers, 1995] utilise le terme trampoline : on pourrait traduire par tremplin.

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

tion 4.5.4 page 96). Comme pour la redéfinition covariante du type des paramètres, la redéfinition contravariante du type de retour nécessite juste une vérification de type.

Cette vérification peut se faire, après chaque invocation de fonction. L'inconvénient est que cette vérification est systématique : il n'est pas possible de l'éviter puisqu'elle fait partie du code de l'invocation. Elle peut également être effectuée dans un thunk, malheureusement avec un inconvénient : l'appel de la méthode par le thunk n'est plus terminal.

Redéfinition du type des attributs

La redéfinition covariante du type des attributs se justifie dans le cadre d'une politique de typage covariante. L'accès en écriture nécessite une vérification de type.

La redéfinition contravariante du type des attributs se justifie dans le cadre du raffinement. L'accès en lecture nécessite une vérification de type.

Il y a deux façons de réaliser ces vérifications : soit l'on recourt systématiquement à des accesseurs (cf. section 6.3.8) et l'on traite le problème comme décrit précédemment, soit l'on fait la vérification de type dans la méthode appelante : c'est alors un casting dynamique, puisque le type dépend de d. Il faut donc stocker l'identifiant du type des attributs dans la table des méthodes.

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

Conclusion sur les redéfinitions de types

Quelle que soit la politique de typage et sa justification éventuelle, on observe deux grandes catégories de cas :

Cas sûr. Les cas sûrs ne nécessitent aucune implémentation ni aucun traitement particulier.

Cas non-sûr. Les cas non-sûrs nécessitent juste une vérification de type dynamique (c'est d'ailleurs la caractéristique du typage non-sûr). Le problème des cas non-sûrs est qu'ils induisent un surcoût même si le programme est sûr (c'est-à-dire si les tests de types n'échouent jamais). Toutefois, dans le cadre du développement d'un logiciel, il est légitime de n'activer ces vérifications de types que lors du développement même[8]. Les tests de types sont désactivés (par exemple en PRM, avec l'option `--boost`, cf. section 8.1.6) dans la version finale du logiciel que des tests approfondis sont censés avoir montré être sans erreur de type.

[8]Et encore, il est d'usage de désactiver les vérifications des parties du logiciel déjà testées.

6.2.4 Évaluation

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

- les objets occupent une place optimale : un « champ » par attribut, plus un pointeur vers la table de la classe ;
- les tables de méthodes ne dépendent que du type dynamique des objets, ce qui fait que chaque classe a une table unique ; elles occupent globalement une place égale au nombre de couples classe-méthode valides, ce qui correspond à l'optimum de compactage des tables en typage dynamique et spécialisation multiple [Ducournau, 1997] ; pour une classe $c \in X$[9] si $M_c \subseteq G_c$ désigne les méthodes connues (introduites ou héritées), la place totale occupée par la hiérarchie de classes d'un programme est en $\Sigma_{c \in X} |M_c|$;
- la vérification dynamique de type (casting descendant) se fait en temps constant et ne nécessite dans la table d'une classe qu'un champ par super-classe.

La transparence à peu près complète du casting — réduit à un simple test pour la vérification de type nécessaire au casting descendant — enlève tout surcoût à la redéfinition non-sûre.

Cette implémentation à peu près optimale de la spécialisation simple est la référence à laquelle on peut mesurer le « surcoût » de la spécialisation multiple ou le « gain » des autres techniques d'implémentation, en ce qui concerne l'efficacité temporelle (nombre d'indirections) ou spatiale (occupation mémoire statique des tables de méthodes et dynamique des objets).

6.3 Spécialisation multiple

En spécialisation multiple, le problème se complique considérablement comme le montre [Ellis et Stroustrup, 1990, chapitre 10] dans le cas de C++. La complication est d'autant plus importante que C++ offre des traits de langage — en l'occurrence le double rôle du mot-clé `virtual` — qui ne relèvent pas d'un bon usage de l'approche objet (cf. chapitres 2, 3 et 5). Nous nous placerons, dans cette section, dans un cadre orthodoxe qui reviendrait, en C++, à utiliser systématiquement `virtual` :

- toutes les fonctions sont virtuelles, au sens où elles sont toutes sélectionnées par liaison tardive,
- tous les héritages sont virtuels, au sens où chaque super-classe n'est « utilisée » qu'une seule fois (cf. section 3.3.2).

[9] Cf. chapitre 3, pour la définition des ensembles X et G d'une hiérarchie de classes.

Ces précautions liminaires sont inutiles pour un langage objet normalement constitué comme EIFFEL ou PRM.

Le problème causé par la spécialisation multiple s'énonce simplement : l'indice d'une méthode ou d'un attribut ne peut plus être invariant par spécialisation (invariant 2), en tout cas tant que ces indices sont calculés de façon séparée, en cherchant à les minimiser (cf. section 6.4). La raison en est la suivante : étant donné deux classes incomparables `B` et `C`, qui occupent les mêmes indices, il est toujours possible d'en définir une sous-classe commune `D`. On se trouve donc en situation de conflit puisque deux attributs ou deux méthodes sont en compétition pour le même indice.

Ce premier constat a une conséquence décisive : si l'on veut que l'envoi de message s'effectue par une indirection dans une table, un pointeur sur un objet n'est plus invariant suivant son type statique (invariant 1).

On verra plus loin comment, et à quel prix, cette conséquence peut être inversée : l'invariance de pointeur donne à l'envoi de message un coût non constant ou nécessite la mise en œuvre de techniques globales (cf. section 6.4).

6.3.1 Principe d'implémentation de C++

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

Invariant 3 *Chaque propriété globale attribut a un indice non ambigu et invariant dans le contexte du type statique qui l'introduit, donc indépendamment du type dynamique.*

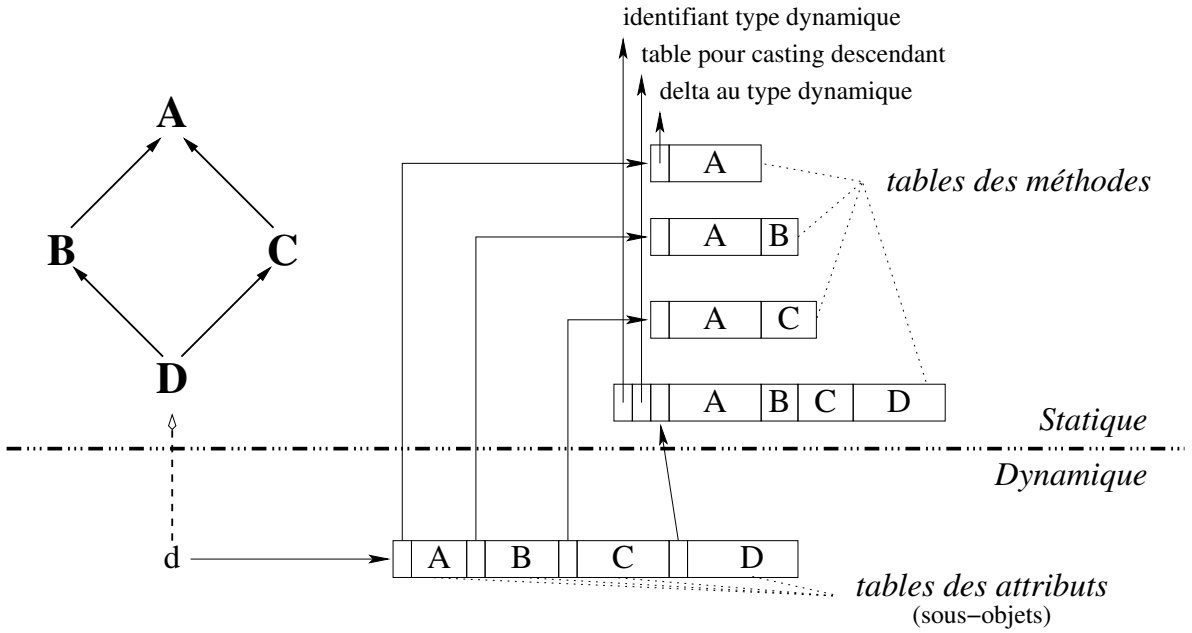
Invariant 4 *Chaque propriété globale méthode a un indice non ambigu et invariant dans le contexte d'un type statique qui l'introduit ou l'hérite, donc indépendamment du type dynamique.*

Mais ces indices déterminent des positions qui ne sont plus invariantes par spécialisation, c'est-à-dire entre deux types statiques liés par une relation de spécialisation. De plus, la valeur de `self` et de tout pointeur sur un objet dépend de son type statique. Tout se passe comme si les tables des attributs et des méthodes étaient constituées de sous-tables (ou sous-objets), une par super-classe. Et l'invariant principal est alors le suivant (figure 6.2) :

Invariant 5 *Toute entité de type statique s est liée au sous-objet correspondant à s, muni de sa propre table de méthodes.*

Cet invariant est trivialement vérifié par l'implémentation de la spécialisation simple (invariant 1). En spécialisation multiple, il est nécessaire de rajouter une propriété de non trivialité, qui est la source du surcoût de cette implémentation :

Invariant 6 *Deux sous-objets de types statiques différents sont distincts.*



Quatre classes : `A`, `B`, `C` et `D` et un seul objet : `d` instance directe de `D`.

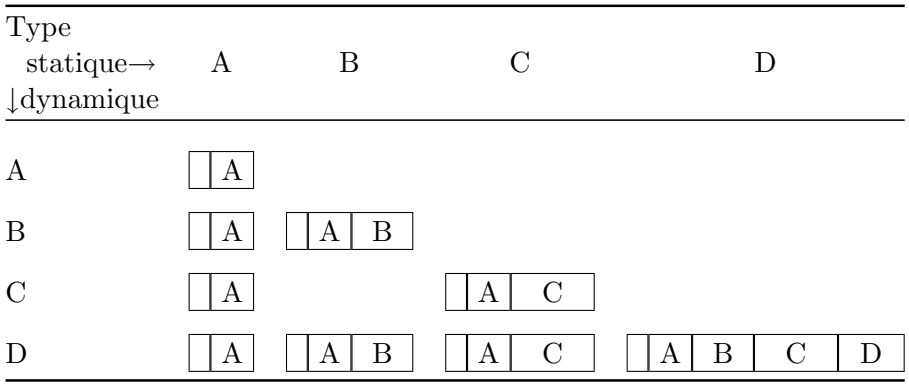
FIG. 6.2: Structure des objets et des tables de méthodes en spécialisation multiple

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

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

Chaque sous-objet ne contient que les attributs introduits par son type statique. Chaque table de méthodes d'un type statique contient toutes les méthodes (propriétés globales) connues par ce type (héritée ou introduites), mais avec des valeurs (adresses) correspondant aux méthodes (propriétés locales) effectivement héritées ou définies par le type dynamique. Ainsi, deux instances directes de classes différentes ne partagent pas les tables de méthodes de leurs super-classes communes : ces tables ont la même structure, mais pas le même contenu (figure 6.3).

Pour un type statique donné, l'ordre de ses méthodes est a priori quelconque, mais il est raisonnable de les regrouper par classe (ce qui est fait dans la figure) et d'assurer



Pour un même type statique (verticalement), les tables sont isomorphes mais diffèrent par leurs contenus (adresses des méthodes) ; en revanche, pour un même type dynamique (horizontalement), les morceaux isomorphes contiennent les mêmes adresses mais pas les mêmes décalages.

FIG. 6.3: Tables des méthodes pour l'exemple de la figure 6.2, suivant les types statiques et dynamiques

une certaine invariance par spécialisation, lorsque c'est possible (cas de spécialisation simple). Mais cette organisation n'a aucun effet sur l'efficacité de l'implémentation.

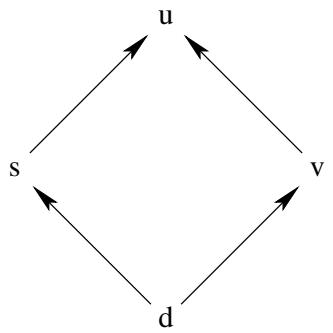
L'invariant 5 impose de recalculer la valeur de `self` à chaque envoi de message : lorsque le receveur est une entité de type statique s, et que la méthode sélectionnée a été définie dans la classe u, il faut savoir de combien il faut incrémenter ou décrémenter `self` pour obtenir, à partir du sous-objet de type s un sous-objet de type v, ce que l'on notera $\Delta_{s\to v}$[10] — figure 6.4. La table des méthodes est donc double : elle contient, pour chaque méthode, l'adresse de la méthode ainsi que la valeur de cet incrément.

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

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

La technique des *thunks* permet une alternative : un thunk s'occupe de faire le décalage puis de se brancher sur la méthode. Ces thunks peuvent être partagés par tous les appels de la même méthode avec le même décalage. En cas d'implémentation par thunk, la séquence d'instructions est la même qu'en spécialisation simple, mais elle provoque un branchement au code suivant :

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



- s est le type statique du receveur ;
- d est le type dynamique de receveur ;
- u est la classe qui introduit la méthode appelée ;
- v est la classe qui définit la méthode appelée ;

FIG. 6.4: Les types en jeu sur le receveur, dans un appel de méthode

```
add object + #delta → object
jmp #method
```

On économise ainsi une indirection dans une table, au prix d'un saut constant[11].

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

6.3.2 Casting

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

Notons d'abord deux propriétés fondamentales des décalages Δ : pour toute sous-

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

classe commune à t, u et v :

$$\Delta_{t\to v} = \Delta_{t\to u} + \Delta_{u\to v} \tag{6.2}$$
$$\Delta_{t\to t} = 0 \tag{6.3}$$

Casting ascendant

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

Invariant 7 *Chaque classe a un indice non ambigu et invariant dans le contexte statique de ses sous-classes.*

On notera $i_s(t)$ cet indice : $\Delta_s^\uparrow(t)$ est alors donné par $\Delta_s^\uparrow[i_s(t)]$.

Au lieu d'être implémenté comme une table indépendante, $\Delta^\uparrow$ peut être implémenté dans les tables de méthodes. Celle-ci contiendrait le décalage au lieu de l'adresse d'une procédure : cela fait gagner une indirection.

Accès aux attributs

Cette table $\Delta_s^\uparrow$ sert aussi à l'accès aux attributs lorsqu'ils sont introduits dans une super-classe de s. Si $\delta(p, s)$ représente la position d'un attribut p relativement au sous-objet de type s et que δ_p est l'indice de l'attribut p dans le type t_p qui l'introduit (invariant 3) (avec bien évidemment $s <: t_p$), on obtient la position de l'attribut par :

$$\delta(p, s) = \Delta_{s\to t_p} + \delta(p, t_p) = \Delta_s^\uparrow(t_p) + \delta_p \ .$$

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

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

Bien entendu, sur toute affectation `a.@x := b.@y`[12], il faut composer le casting ascendant entre les types de `y` et de `x`, avec les castings ascendants sur `a` et `b`. Ces castings se parallélisent en partie — les deux objets en parallèle, mais l'affectation en séquence —, et la séquence résultante est 5 fois plus longue qu'en spécialisation simple :

[12]Syntaxe en PRM.

```
load [object1 + #tableOffset] → table1                    5L + 3
load [object2 + #tableOffset] → table2
load [table1 + #cast1Offset] → delta1
load [table2 + #cast2Offset] → delta2
add object1 + delta1 → object1
add object2 + delta2 → object2
load [object1 + #attribute1Offset] → attribute
load [attribute + #tableOffset] → table
load [table + #castOffset] → delta
add attribute + delta → attribute
store attribute → [object2 + #attribute2Offset]
```

Casting descendant

Pour passer du type statique s à un sous-type (statique) t, il faut à la fois une vérification de type et le $\Delta_{s \to t}$. Le problème réside dans la difficulté (au moins sans connaissance globale de l'ensemble des sous-classes de s, cf. section 6.4) d'un accès direct : une recherche séquentielle permet de se débarrasser du problème, une meilleure solution consiste à utiliser une table de hachage [Ducournau, 2002b].

6.3.3 Redéfinition de types

Si, en spécialisation simple, la redéfinition de types s'implémentait de façon à peu près transparente, ce n'est plus le cas en spécialisation multiple. En effet, contrairement aux redéfinitions de types en spécialisation simple, le surcoût systématique imposé par une éventuelle redéfinition de types est exacerbé[13], que celle-ci soit sûre ou non [Ducournau, 2002b].

À tel point que bien que la norme ANSI de C++ autorise la redéfinition covariante du type de retour depuis 1998, le compilateur `g++` ne l'autorisait pas jusqu'à la sortie de la version 3.4 en 2004.

6.3.4 Évaluation

Le surcoût de la spécialisation multiple est à la fois évident et potentiellement important :

- le surcoût spatial dynamique dans chaque objet est égal au nombre de super-classes de la classe de l'objet ;
- la taille des tables de méthodes n'est plus quadratique dans le nombre de classes, mais potentiellement cubique : le nombre de tables est lui-même quadratique ; de

[13] La redéfinition de type coûte même si elle n'est pas utilisée.

plus, il n'y a pas de possibilité immédiate de partage puisque les Δ ne sont jamais invariants ; dans un contexte d'héritage massivement « non virtuel », [Driesen et Hölzle, 1995; Driesen, 1999] rapportent des facteurs supérieurs à 3, semble-t-il, sur le nombre d'entrées des tables ;
- toute affectation ou passage de paramètres qui n'est pas à type statique constant exige un casting ;
- l'efficacité de l'accès aux attributs est significativement réduite puisqu'il faut passer par un décalage stocké dans la table de méthodes ;
- l'efficacité de l'appel de méthode lui-même est plus discutable parce que le décalage, qui représente un surcoût effectif, peut s'effectuer dans les latences du processeur (pipeline) ou en parallèle (processeur super-scalaire) [Driesen, 1999] ; de même que les décalages nécessaires sur les paramètres ; mais la remarque qui suit sur les thunks laisse à penser que le surcoût est sérieux ;
- les décalages ont un effet spatial non négligeable, sur le code ou sur les tables statiques, de l'ordre du doublement ;
- le casting descendant ne se fait pas en temps constant[14].

Les expérimentations de [Driesen, 1999] semblent donner un léger avantage à la technique des thunks d'un point de vue temporel. Mais cette conclusion vaut pour des programmes C++ classiques, qui pratiquent un héritage majoritairement « non virtuel » (cf. section 4.2), dans lesquels les décalages entre types différents sont souvent nuls. Ce sont donc ces décalages nuls, qui disparaissent en pratique, qui rendent les thunks plus efficaces ; cela tendrait à prouver que dans la technique sans thunk, le décalage a un surcoût sensible. Ces expérimentations ne permettent donc pas de conclure pour l'implémentation standard. D'un point de vue spatial, les deux techniques sont à peu près à égalité : il y moins de thunks que d'entrées dans les tables, mais ils occupent deux mots au lieu d'un. De plus, la séquence de l'envoi de message a deux instructions de moins.

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

Ensuite, le surcoût de l'implémentation est très notablement augmenté — à peu près doublé — dès que l'on accepte des redéfinitions covariantes, même dans le cas parfaitement sûr du type de retour. Cela étant, tout le traitement de la redéfinition covariante des méthodes peut être intégré dans les thunks, sans en compliquer la combinatoire — un par couple (s, d) —, avec l'avantage de n'imposer le surcoût d'un casting qu'en cas de besoin, c'est-à-dire pour les redéfinitions effectives entre s et d —comme en spécialisation simple avec la technique de [Myers, 1995], mais sans avoir besoin de complexifier l'indiçage. Dans ce cas particulier, la technique des thunks semble s'imposer.

[14]Sauf au prix de très grandes tables de fermeture transitive [Ducournau, 2002b].

En revanche, pour les attributs, les thunks sont à éviter pour éviter une implémentation par de vrais accesseurs et un appel de méthode supplémentaire. En contrepartie, les castings doivent être faits systématiquement.

Enfin, cette implémentation de la spécialisation multiple augmente le coût de la gestion mémoire : un ramasse-miettes classique ne permet en général pas de pointer au milieu d'un objet.

6.3.5 Variations autour de l'implémentation de la spécialisation multiple

Cette complexité de l'implémentation de la spécialisation multiple explique à la fois certains mauvais côtés de C++, le fait que de nombreux langages, comme Java, aient partiellement renoncé à la spécialisation multiple, ainsi que la recherche de techniques alternatives, dont la plus radicale est celle du compilateur SmartEiffel qui évite l'usage de tables de méthodes, au prix d'une compilation globale (cf. section 6.4.6).

De nombreuses alternatives à l'implémentation tentent de réduire ces défauts. Elles consistent à privilégier un type d'efficacité, à chercher une implémentation qui s'applique sans surcoût à la spécialisation simple ou à réduire, sans l'annuler, la question de l'invariance des indices des attributs

6.3.6 Temps et espace

Des alternatives consistent à privilégier un type d'efficacité (temporelle, spatiale statique ou dynamique) par rapport à un autre : soit en réduisant l'espace mémoire en partageant les tables de classes au prix de diverses indirections, soit, à l'opposé, à éviter des indirections en implémentant une partie des tables de classes dans les objets, au détriment de l'occupation mémoire dynamique. Dans cette dernière catégorie, la table de décalage nécessaire au casting ascendant, $\Delta_s^{\uparrow}$, est la principale visée. Elle sert en effet en permanence pour les accès aux attributs, les passages de paramètres et les affectations polymorphes. En C++, une variation connue sous le nom VBPTR, pointeurs sur des *virtual base classes*, consiste à placer dans les sous-objets d'une instance des pointeurs vers les autres sous-objets de cette instance. Ce qui permet de réduire le coût d'accès aux attributs qui passe de $3L + 1$ à $2L$:

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

6.3.7 Spécialisation simple sans surcoût

Il existe une variation de l'implémentation de la spécialisation multiple qui s'applique sans surcoût à la spécialisation simple, au prix de la généralité ou de la sémantique de

la spécialisation : c'est l'héritage multiple non virtuel de C++ qui permet d'annuler le surcoût de la spécialisation multiple lorsqu'en pratique la spécialisation est simple (cf. section 3.3.2 page 50). Si l'on considère que l'héritage répété est une abomination et donc que la différence entre virtuel et non virtuel n'a de raison que d'implémentation, il est possible de déterminer, par une analyse statique globale, les liens de spécialisation qui doivent être virtuels et ceux pour lesquels c'est inutile [Eckel et Gil, 2000]. Mais le gain n'est possible qu'en compilation globale.

6.3.8 Accès aux attributs

L'*implémentation par accesseurs*[15] permet d'abandonner toute hypothèse d'invariance des indices des attributs, puisque ceux-ci restent encapsulés dans les accesseurs : lorsque l'indice d'un attribut change dans une sous-classe, les accesseurs correspondants sont reconstruits.

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

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

Si les attributs sont regroupés par classe d'introduction, il est encore mieux de ne mettre dans la table des méthodes que la position de chaque groupe d'attributs :

```
load [object + #tableOffset] → table                    3L + 1
load [table + #attributeGroupOffset] → offset
add object + offset → place
load [place + #attributeOffsetInGroup] → attribute
```

6.4 Techniques globales d'implémentation

Jusqu'à présent, les techniques d'implémentation considérées partaient du principe qu'à aucun moment, la totalité du programme n'était connue. Dans cette section, nous

[15] On ne confondra pas cette *implémentation par accesseurs* avec le style de programmation consistant à encapsuler tous les accès aux attributs dans des accesseurs redéfinissables dans les sous-classes, comme c'est le cas en PRM (cf. section A.2.5 page 237). La présence d'accesseurs définis par le programmeur ne devrait pas empêcher le compilateur d'en définir pour ses propres besoins.

allons présenter des techniques d'implémentation efficaces qui nécessitent d'avoir une connaissance globale du programme à compiler. En effet, la connaissance de la totalité du code d'un programme et de son point d'entrée rend possible des analyses fines sur la façon dont chaque morceau du programme est utilisé, ce qui permet de le compiler plus efficacement.

6.4.1 Recopie physique des méthodes

L'héritage a été souvent qualifié de copie virtuelle : certains auteurs proposent d'aller au bout de la métaphore en implémentant l'héritage des méthodes par une copie physique.

Le principal avantage de cette technique, proposée sous le nom de *particularisation* (*customization*) [Chambers et Ungar, 1989], est que la pseudo-variable `self` n'est plus polymorphe[16] :

- tout appel de méthode sur `self` peut donc se résoudre statiquement.
- si les attributs sont bien encapsulés de façon orthodoxe, à la SMALLTALK, cette recopie rend la variance des indices d'attributs parfaitement transparente. Pour les attributs qui ne seraient pas encapsulés, il est nécessaire d'effectuer cette encapsulation au moyen d'accesseurs, dont la génération peut être à la charge du compilateur ;
- toutes les méthodes qui ne peuvent s'appliquer qu'à `self` (par la contrainte d'un mécanisme de protection spécifique)[17], ou qui ne s'appliquent de fait qu'à `self` (par un constat lors d'une analyse de types), n'ont plus besoin de figurer dans les tables de méthodes, ou dans n'importe quelle structure d'appel de méthode polymorphe
- la recopie de méthodes donne des informations de type plus précises, dans tous les cas de redéfinitions : sur le type de retour des méthodes et sur le type des paramètres et des attributs.

Bien entendu, cette technique a plusieurs inconvénients notables :

- la compilation d'une classe nécessite le source de ses super-classes, ce qui est totalement contradictoire avec la compilation séparée ;
- l'objectif de simplification de l'implémentation n'est pas atteint pour tous les appels de méthodes : seuls les appels à `self` sont concernés et la réduction des tables obtenue n'est que partielle ;
- une amélioration non négligeable des performances temporelles est obtenue, au prix d'une dégradation très importante des ressources spatiales : la duplication du code des méthodes coûte considérablement plus cher que la petite amélioration obtenue sur les appels de méthodes.

[16] Cf. section 6.4.2.

[17] Comme en EIFFEL ou en PRM (cf. section A.2.5).

Cette technique n'est donc envisageable que dans un cadre de compilation globale, associée à d'autres améliorations permettant de réduire significativement la taille du code généré : de toute évidence, l'élimination du code non utilisé (« mort ») est une nécessité. C'est ainsi que la technique est utilisée dans le compilateur SmartEiffel— cf. section 6.4.6.

Dans le même ordre d'idées, on retrouve l'*implémentation hétérogène* [Odersky et Wadler, 1997] des classes génériques afin d'adapter le code de la classe paramétrée à chacune des instanciations de ses paramètres.

6.4.2 Analyse globale des types

L'un des objectifs d'une analyse globale de type est de déterminer, pour chaque expression du programme, l'ensemble des types dynamiques que pourront prendre les valeurs de cette expression, pour toute exécution.

Principe

Elle consiste à approximer trois groupes d'ensembles : l'ensemble des classes effectivement instanciées, celui des types dynamiques que pourra prendre chaque expression dans toutes les exécutions possibles (on l'appelle son *type concret*) et d'autre part l'ensemble des procédures potentiellement appelées pour chaque site d'envoi de message.

Dans sa généralité, le problème est indécidable puisqu'il pose trivialement le problème de l'arrêt d'un programme, mais des hypothèses simplificatrices le rendent polynomial [Gil et Itai, 1998]. Les trois groupes d'ensembles sont en dépendance circulaire, puisque les méthodes qui peuvent être appelées dépendent des types dynamiques du receveur, lesquels dépendent à leur tour des classes instanciées et ces dernières des méthodes effectivement appelées. Cette circularité explique la difficulté du problème et la variété des solutions.

On dit qu'une expression est :

- *monomorphe* lorsque son type concret est un singleton — cela veut dire que lors de toute exécution du programme, cette expression n'a qu'un seul type dynamique possible ;
- *polymorphe* lorqu'elle n'est pas monomorphe ;
- *oligomorphe* lorsqu'elle est polymorphe mais que son type concret contient peu de types dynamiques ;
- *megamorphe* lorsqu'elle est polymorphe mais pas oligomorphe — c'est-à-dire que son type concret contient beaucoup de types dynamiques.

On réutilise la même terminologie pour qualifier les sites d'appel (de la forme `x.toto`) : on dit qu'un site d'appel est monomorphe si, dans le programme, une seule méthode est potentiellement invocable par ce site d'appel[18].

[18]Hors particularisation, un site d'appel peut être monomorphe alors que le receveur est polymorphe,

Détection du code mort

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

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

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

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

En revanche, certaines catégories d'applications ne semblent pas pouvoir se prêter de façon satisfaisante à cette optimisation : les applications pour lesquelles l'instanciation des classes résulte, pour partie, d'un processus interactif, avec un utilisateur, un SGBD, un autre programme ou un réseau, voire l'ensemble. Dans un tel contexte, toutes les classes « applicatives » sont a priori vivantes, ainsi que tout leur code qui peut être activé par des procédés du même genre. Cette restriction apparaît en particulier dès qu'un méta-niveau entre en jeu, comme en JAVA et son package `reflect`.

Techniques d'analyse de types

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

Voici quelques-unes des techniques les plus répandues. Nous les présentons par ordre croissant de complexité (pour plus de détails sur chacune des techniques, le lecteur se référera à [Grove et Chambers, 2001; Privat, 2002]) :

---

tout simplement parce que pour tous les types du type concret du receveur, c'est la même propriété locale qui est sélectionnée. C'est typiquement, le cas lorsqu'une propriété globale ne contient qu'une seule propriété locale.

***Reachability Analysis* (RA).** RA est la première technique mise au point [Srivastava, 1992]. Son principe n'est pas très compliqué : une méthode est vivante si et seulement si son nom apparaît dans un site d'appel d'une méthode vivante.

***Unique Name* (UN).** UN, dans un même ordre d'idée, permet de détecter les sites d'appel monomorphes [Calder et Grunwald, 1994]. Son comportement est très simple également : s'il existe une seule méthode de nom `n` alors le site d'appel `e.n` est monomorphe. L'argument peut paraître trivial mais il n'est pas sans efficacité : en SMALLTALK, sur 7 623 sélecteurs il s'applique à 3 313 [Ducournau, 1997].

***Class Hierarchy Analysis* (CHA).** CHA prend en compte les informations de types pour affiner le travail de RA : il va restreindre les méthodes potentielles à celles redéfinissant la méthode du type statique du receveur [Dean et Chambers, 1994; Dean *et al.*, 1995].

***Rapide Type Analysis* (RTA).** RTA est l'une des techniques les plus utilisées. Elle est plus performante que CHA car elle prend en compte les informations d'instanciation [Bacon *et al.*, 1996; Bacon et Sweeney, 1996; Bacon, 1997]. L'idée est de déterminer, en plus des méthodes, quelles sont les classes vivantes, c'est-à-dire les classes qui possèdent des instances.

***Control-Flow Analysis* (CFA).** 0-CFA est la première technique *à flot*. L'idée de 0-CFA est de construire un réseau où un nœud (un ensemble de types) est associé à chaque entité typable du programme (expressions, paramètre, attributs de classe, etc.) et les arcs (inclusion entre les ensemble de types) sont définis en fonction des informations contenues dans le programme [Shivers, 1988; Shivers, 1991; Palsberg et Schwartzbach, 1991; Palsberg et Schwartzbach, 1994; Grove, 1995].

Les techniques *polyvariantes* consistent à analyser les méthodes avec un point de vue différent en fonction des utilisations. L'idée est donc de construire un bout du réseau par utilisation particulière des méthodes. On parle de *contour de méthode*[19].

Une première proposition 1-CFA [Palsberg et Schwartzbach, 1991; Oxhoj *et al.*, 1992] associe un contour par site d'appel. La généralisation k-CFA [Vitek *et al.*, 1992; Palsberg et Schwartzbach, 1994; Plevyak et Chien, 1994; Phillips et Shepard, 1994] associe un contour en fonction du site d'appel et des $k-1$ derniers éléments de la *chaîne d'appels* (ou *call string*) [Ryder, 1979].

Une autre extension, k-l-CFA [Vitek *et al.*, 1992; Palsberg et Schwartzbach, 1994], a été proposée pour résoudre le problème de pollution dans le type concret des attributs des classes. Dans cette proposition, les types qui « circulent » ne sont plus des classes mais des contours de classes : un contour de classe est associé en fonction du site d'instanciation et des $l-1$ derniers éléments de la chaîne d'appels.

[19] Le terme de « contour » vient sans doute du fait que dans le graphe d'analyse à flots on peut regrouper les noeuds ayant trait à l'une ou autre de ces utilisations particulières ce qui fait que graphiquement on entoure généralement chaque ensemble de noeuds pour que cela soit plus lisible.

Malgré tout, la famille des techniques CFA pose des problèmes : trop de contours sont créés et les bonnes valeurs de k et l dépendent de chaque programme et ne peuvent être devinées à l'avance.

***Cartesian Product Algorithm* (CPA).** CPA, développé par Agesen [Agesen, 1994; Agesen, 1995; Agesen, 1996], construit les contours de manière plus intelligente et se base non pas sur la chaîne d'appel mais sur les types concrets des receveurs et arguments. La technique consiste à créer des contours monomorphes de méthode, donc autant de contours qu'il y a d'élément dans le produit cartésien des types concrets du receveur et des arguments[20]. Pour éviter une explosion combinatoire du nombre de contours, une version bornée de la technique traite les nœuds mégamorphes de façon spécifique (en l'occurrence de façon polymorphe)

***Simple Class Set* (SCS).** SCS, décrit dans [Grove *et al.*, 1997], ne crée qu'un contour par méthode dans un site d'appel défini par les types concrets actuellement calculés du receveur et des arguments. Bien que le nombre de contours potentiellement crées soit prohibitif dans le pire cas ($\mathcal{O}(n^{n^a})$ où n est le nombre de méthodes et a l'arité des méthodes), [Grove et Chambers, 2001] montre qu'en pratique SCS serait plus rapide que la version bornée de CPA.

Autre utilisation

L'analyse de types n'est pas en tant que tel une technique d'implémentation. C'est un outil d'analyse statique globale qui a de nombreuses utilités, par exemple [Wang et Smith, 2001] et SmartEiffel l'utilisent pour vérifier la sûreté du casting descendant et des redéfinitions covariantes.

Les analyses de types polyvariantes permettent également de déterminer de bons candidats à de la spécialisation de code (particularisation) : un contour de méthode de classe ou de méthode peut être compilé de façon spécifique. C'est dans cette optique que [Dean *et al.*, 1995] proposent la *spécialisation de méthode* et [Tip et Sweeney, 2000] la *spécialisation de hiérarchies de classes*.

6.4.3 Appel direct et mise en ligne

Les statistiques présentées dans la littérature montrent que la plupart des envois de message sont en réalité des appels monomorphes : une analyse globale, souvent simple (CHA par exemple), permettrait de déterminer statiquement la méthode à appeler.

Lorsqu'un site d'appel monomorphe est détecté par le compilateur, celui-ci doit :

– le compiler par un *appel direct*, c'est-à-dire sans implémenter de liaison tardive ;

[20] Pour un appel de méthode `x.f(y1,y2,...,yn)`, il y a un contour par élément du produit cartésien $[x] \times [y1] \times [y2] \times \cdots \times [yn]$ où $[z]$ désigne le type concret du nœud `z`.

– compiler l'appel *en ligne* (*inlining*), c'est-à-dire remplacer l'appel par le contenu de la méthode invoquée.

Bien sûr, les mises en lignes n'ont d'intérêt que pour de petites méthodes (accesseur, méthode mandataire, résultat immédiat, etc.).

La mise en ligne peut également être utilisée dans des sites d'appel polymorphes, on parle alors de *mise en ligne gardée* [Detlefs et Agesen, 1999]. Cette technique permet, pour le prix d'un test (la garde), de gagner le coût d'un branchement (au sens large, c'est-à-dire en incluant le passage des paramètres et la sauvegarde du mot d'état) :

```
load [object + #tableOffset] → table                    2L + B + 2 ou 2L + 3
load [table + #methodOffset] → method
cmp method = #inlinedMethod
jne #methodCallLabel
; inlining code of the method #inlinedMethod
jmp #endLabel
#methodCallLabel:
call method
#endLabel:
```

Ici, `#inlinedMethod` correspond à l'adresse de la méthode mise en ligne.

6.4.4 Prédiction de type et arbre de décision

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

Dans ce cadre, deux approches sont possibles :

– la plus extrême consiste à énumérer tous les types possibles, dans une succession de tests de type qui implémentent un arbre de décision (*binary tree dispatch* ou BTD). Le typage statique rend cette technique plus envisageable qu'en typage dynamique, puisque l'ensemble des types à considérer est borné par le type statique. Mais une compilation globale est nécessaire puisque c'est la seule façon de connaître la totalité des types : la hiérarchie doit être « fermée ».
– une solution plus modérée — le cache en ligne, éventuellement polymorphe — consiste à prévoir (prédire) le cas d'un ou plusieurs types plus probables, en faisant appel au mécanisme général d'envoi de message dans les autres cas.

L'ensemble des tests peut être structuré de diverses manières — séquentiellement ou de façon arborescente — en garantissant des propriétés de la structure (par exemple, arbre équilibré) ou des propriétés d'efficacité dynamique (probabilités décroissantes).

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

Toutefois, le mécanisme n'est plus en temps constant car le nombre maximum de tests est $\lceil log_2(k) \rceil$ (où k est le cardinal du type concret du receveur). Les arbres de décisions atteignent alors leur limite dans le cas d'appels mégamorphes.

Une solution alternative, proposée par [Queinnec, 1997], consiste à utiliser un test de sous-typage à la place d'un test d'égalité de classes. Le nombre maximum de tests est alors $\lceil log_2(m) \rceil$ (où m est le nombre de propriétés locales associées au site d'appel). En contrepartie, le coût d'un test unitaire est plus élevé : [Queinnec, 1997] est en spécialisation simple et se base sur [Cohen, 1991] pour le test de sous-typage.

6.4.5 Coloration

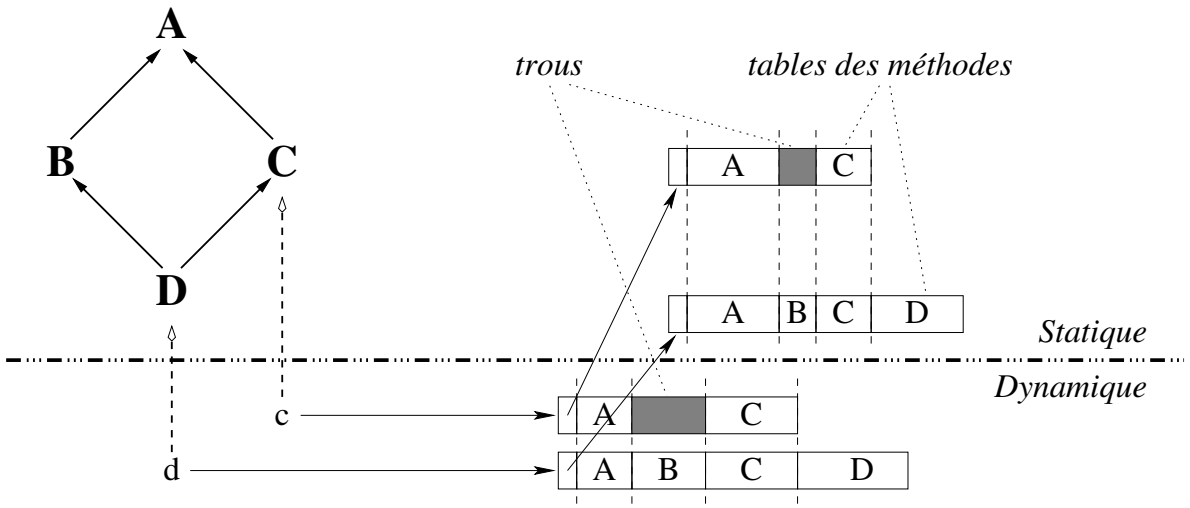
La coloration s'inscrit dans le cadre des techniques d'implémentation avec tables de méthodes. Elle s'applique aussi bien aux attributs et aux méthodes, qu'aux classes pour le test de sous-typage.

Le principe de la coloration a été décrit, à peu près indépendamment, dans [Dixon *et al.*, 1989] (pour les méthodes) et [Pugh et Weddell, 1990] (pour les attributs) et implémenté en YAFOOL [Ducournau, 1991] (pour les attributs). [Vitek *et al.*, 1997] proposent une implémentation des tests de sous-typage, appelée *pack-encoding*, qui correspond à la coloration des classes et qui généralise la technique de [Cohen, 1991]. [Dixon *et al.*, 1989] propose d'abord une coloration de sélecteurs qui est expérimentée par [André et Royer, 1992], à la suite de quoi elle a été écartée par la plupart des auteurs à cause de son apparente inefficacité. Les heuristiques proposées par [Pugh et Weddell, 1990; Ducournau, 1997; Ducournau, 2001a; Ducournau, 2003; Takhedmit, 2003; Ducournau, 2006] démontrent pourtant des performances qui justifient pleinement sa prise en considération.

Principe

Comme nous l'avons déjà remarqué, la technique que nous avons décrite pour la spécialisation simple est un cas particulier de l'heuristique de coloration. Dans le cas général, c'est-à-dire en spécialisation multiple, la coloration consiste à associer à une propriété globale méthode un indice unique et non ambigu dans toutes les classes qui possèdent la propriété globale. Plusieurs propriétés globales peuvent se partager le même indice, mais elles n'appartiennent jamais à la même classe.

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



Quatre classes : `A`, `B`, `C` et `D` et deux objets : `c` instance directe de `C` et `d` instance directe de `D`.

FIG. 6.5: Implémentation des classes et des objets via la coloration

Invariant 8 *Chaque propriété globale attribut (resp. méthode) a un indice (une couleur) invariant par spécialisation. Deux propriétés globales attributs (resp. méthodes) de même couleur n'appartiennent pas à la même classe.*

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

La technique de coloration permet donc d'implémenter la spécialisation multiple de façon aussi efficace que la spécialisation simple, au prix d'un inconvénient non négligeable : l'existence de trous dans les tables, c'est-à-dire de cases non occupées. La figure 6.5 schématise cette implémentation, les trous sont représentés en grisé.

Le respect de ces invariants est aisé, mais la minimisation de la taille des tables — c'est-à-dire du nombre de trous — est un problème proche de la coloration de graphes et prouvé NP-difficile [Pugh et Weddell, 1990; Pugh et Weddell, 1993; Takhedmit, 2003]. Heureusement, les hiérarchies de classes constituent apparemment des instances faciles du problème et de nombreuses heuristiques ont été proposées, qui donnent des résultats satisfaisants, même sur de très grosses hiérarchies [Pugh et Weddell, 1990; Ducournau, 2001a; Ducournau, 2003; Takhedmit, 2003].

La coloration peut être monodirectionelle (couleurs uniquement positives) ou bidirectionelle (couleurs positives et négatives). La coloration bidirectionelle, introduite par [Pugh et Weddell, 1990], réduit le nombre de trous.

Les trous ne sont pas trop gênants pour les tables de méthodes ou de casting, où ils seront largement compensés par la réduction du nombre de tables. En revanche, les

trous dans les instances peuvent provoquer un surcoût puisqu'ils sont multipliés par le nombre d'instances. [Pugh et Weddell, 1990] rapporte un taux de trous de 6 % pour les 563 classes et 2245 attributs de FLAVORS ; [Takhedmit, 2003] affiche une moyenne de 5 % de trous sur 3367 classes et 8963 attributs ; [Ducournau, 2006] rapporte des taux moyens de trous inférieurs à 11 % mais montre également que le taux maximum de trous dans une classe peut atteindre 700 %.

La solution pour éviter une éventuelle inflation de trous dans les instances consiste à utiliser des accesseurs simulés (cf. section 6.3.8). De tels accesseurs sont stockés dans la table des méthodes et leur indice calculés par coloration. Cette coloration peut porter sur chacun des attributs (c'est-à-dire une couleur par propriété globale), ou mieux, porter sur des groupes d'attributs (c'est-à-dire une couleur par super-classe qui introduit des attributs).

Casting descendant et coloration de classes

Comme en spécialisation simple, le casting ascendant n'a plus de raison d'être. Quant au casting descendant, il se ramène à une vérification de sous-typage puisqu'il n'y a pas de décalage à retourner, `self` étant invariant.

Cette vérification de sous-typage peut être implémentée par l'heuristique de coloration, en généralisant l'implémentation qui a été proposée pour la spécialisation simple [Cohen, 1991]. Elle consiste à colorer les classes de façon à respecter l'invariant suivant :

Invariant 9 *Chaque classe a une couleur. Deux classes de même couleur n'ont pas de sous-classe commune.*

Cet invariant renforce, globalement, l'invariant 7 qui n'est que local.

On associe alors à chaque classe c une table $\Gamma^{\uparrow}_{c}$ contenant tous les identifiants de ses super-classes de telle façon que :

$$\forall c, c' \in X, \quad c' <: c \quad \Leftrightarrow \quad \Gamma^{\uparrow}_{c'}[k(c)] = n(c) \ . \tag{6.4}$$

où $k(c)$ et $n(c)$ sont respectivement la couleur et l'identifiant de la classe c.

Comme pour les tables $\Delta^{\uparrow}$, cette table $\Gamma^{\uparrow}$ peut être intégrée dans la table des méthodes, ce qui évite une indirection supplémentaire : la seule condition est qu'un identifiant de classe $n(c)$ ne puisse pas être confondu avec une adresse de méthode (ou avec tout autre infomation contenue dans la tables des méthodes). De plus, afin d'éviter un test sur la longueur de la table, celles-ci doivent être contigues et munies de bords larges. Le pseudo-code pour une vérification de type, donc pour un casting descendant, est alors le suivant :

```
load [object + #tableOffset] → table                    2L + 2
load [table + #colorOffset] → classid
```

```
cmp classid = #id
jne #fail
; casting OK
```

La première technique proposée pour le casting descendant en sous-typage simple [Cohen, 1991] était une variante de cette technique, pas optimale dans ce cadre-là. [Queinnec, 1997] réutilise par la suite cette technique, dont [Vitek *et al.*, 1997] propose une généralisation à la spécialisation multiple et, comme [Dixon *et al.*, 1989; André et Royer, 1992], avec le nombre de couleurs comme critère de minimisation. Comme technique de codage d'ordre, la coloration est loin de l'optimal souhaité par les algorithmiciens [Raynaud et Thierry, 2001], mais son coût est relativement marginal, puisqu'elle correspond à la définition d'une méthode supplémentaire par classe. Elle ressort de l'étude de [Vitek *et al.*, 1997] comme une des plus efficaces : sur le plan temporel, il est peu probable que l'on fasse jamais beaucoup mieux. D'un point de vue spatial, la coloration de classes a autant d'entrées occupées que la taille de la fermeture transitive, soit hN, si h est le nombre moyen de super-classes indirectes d'une classe et N le nombre de classes, à comparer aux N^2 d'une table de fermeture transitive. Les statistiques de [Vitek *et al.*, 1997] donnent à h (resp. N/h), des valeurs de l'ordre de 10 à 30 (resp. 50 à 300).

6.4.6 SmartEiffel

Le compilateur GNU SmartEiffel [Collin *et al.*, 1997; Zendra *et al.*, 1997; Zendra, 2000] pour le langage Eiffel est caractéristique des approches récentes à base de techniques globales. Il s'agit sans doute de l'une de leurs premières applications à un langage à objets à typage statique.

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

1. la recopie de méthodes est systématique : `self` n'est donc plus polymorphe (cf. section 6.4.1) ;
2. une analyse de types (RTA amélioré) permet de déterminer, pour chaque appel de méthode, le ou les types dynamiques possibles du receveur : cet ensemble de types est d'autant plus réduit que `self` est monomorphe après l'étape précédente (cf. section 6.4.2) ;
3. le code mort peut alors être identifié et éliminé[21], qu'il s'agisse de classes entières, non instanciées, ou de méthodes non utilisées d'une classe « vivante » ;

[21] En réalité, le code mort n'est pas analysé du tout. En effet, le compilateur part du point d'entrée du programme et visite les classes en fonction du code qu'il découvre au fur et à mesure. Ainsi, par construction, le code vivant est le code analysé par le compilateur alors que le code mort est le code qui n'a jamais été analysé.

4. chaque appel de méthode encore polymorphe après la seconde phase est compilé par l'appel d'un thunk qui contient un arbre de décision à base de comparaisons numériques (basées sur une simple numérotation des classes) (cf. section 6.4.4) ; un traitement du même ordre est effectué pour les accès polymorphes aux attributs — lorsque l'indice de l'attribut varie suivant les différents types déterminés par la seconde étape — de même que pour la vérification de type associée au casting descendant ou à la redéfinition covariante du type des paramètres ;
5. enfin, une compilation en ligne (*inlining*) est effectuée dans divers cas (cf. section 6.4.3).

La rapidité de la recompilation est assurée par la production de code C : seule la compilation d'Eiffel à C est globale, les fichiers C non modifiés n'étant pas recompilés. Ainsi, dans les bons cas de figure, la recompilation après une modification est très rapide car seul le fichier C correspondant a été modifié. Mais dans les mauvais cas, une petite modification peut rendre vivantes ou mortes de grandes quantités de méthodes et impliquer la recompilation de nombreux fichiers.

6.5 Compilation et PRM

Au terme de ce chapitre, il apparaît ainsi clairement que la compilation séparée de la spécialisation multiple est difficile et coûteuse, avec une aggravation non négligeable en cas de redéfinition covariante et contravariante du type des paramètres et de retour. Malgré tout, des implémentations alternatives de la spécialisation multiple existent.

Il est enfin possible, à plus court terme, d'améliorer les compilateurs existant en combinant les techniques. Ainsi, SmartEiffel a prouvé qu'il était possible de faire des compilateurs efficaces de « vrais » langages à objets statiquement typés en spécialisation multiple, voire plus efficaces que les compilateurs existants, en mettant en pratique des idées simples.

C'est pourquoi, en suivant cette voie, nous proposons d'implémenter le compilateur `prmc` en intégrant les techniques les plus efficaces ou les plus prometteuses :

- implémentation de l'envoi de message en fonction de la « taille » des sites d'appel (donc analyse de types) ;
- appel direct pour les monomorphes ;
- arbres de sélection binaires pour les oligomorphes ;
- tables de méthodes basées sur la coloration pour les mégamorphes ;
- utilisation de *thunks* pour les redéfinitions covariantes et contravariantes des types des paramètres et de retour ;
- implémentation des accès aux attributs par coloration ou simulation d'accesseurs — au choix de l'utilisateur ;
- suppression du code mort.

Comme nous l'avons vu dans les chapitres précédents, le langage PRM est basé sur un principe de modularité. Afin de respecter ce principe de développement modulaire dans un compilateur, celui-ci doit fonctionner selon un schéma de compilation séparée. Toutefois, les implémentations alternatives de la spécialisation multiple nécessitent généralement une certaine dose d'analyse globale ou de connaissance du code des autres classes, et sont donc généralement mises en œuvre dans des compilateurs globaux. C'est pourquoi, dans le chapitre suivant, nous proposons un schéma de compilation séparée qui permet malgré tout la mise en œuvre de techniques d'implémentation globale parmi celles que nous avons évoquées.

Chapitre 7

Techniques globales en compilation séparée

Préambule

Nous proposons dans ce chapitre un schéma de compilation qui allie modularité et efficacité en conciliant la compilation séparée du code avec l'analyse globale de la hiérarchie de classes (et de modules).

Dans ce chapitre, nous tentons de rester le plus général possible, en effet, ce schéma est compatible avec tout langage à classes en spécialisation multiple et n'est pas limité aux langages à modules comme PRM. *La concrétisation de ce schéma est présentée au chapitre suivant, avec le compilateur* `prmc`.

7.1 Introduction

Comme nous l'avons évoqué dans l'introduction de ce mémoire et dans la section 2.7 (page 27), en ingénierie du logiciel, on cherche à produire des logiciels modulaires. Or, en ingénierie du logiciel, on cherche également à réaliser la production de logiciels d'une façon modulaire. En pratique, on peut identifier trois avantages à la production modulaire de logiciels :

- une petite modification dans le code source ne nécessite pas la recompilation de tout le programme ;
- une seule compilation d'une bibliothèque est nécessaire, même si elle est utilisée par plusieurs programmes ;
- une bibliothèque peut être distribuable sous une forme compilée.

La solution, qui existe depuis FORTRAN, est la compilation séparée : les fichiers sources d'un programme sont compilés indépendamment de leurs utilisations, puis liés pour produire un programme exécutable.

Le problème est que, comme nous l'avons vu dans le chapitre précédent, la connaissance de la totalité du programme permet une compilation plus efficace. C'est pourquoi les travaux précédents ont utilisé ces techniques dans des schémas de compilation globale, donc incompatibles avec la production modulaire de logiciels.

La section 7.2 de ce chapitre décrit le principe général de la compilation séparéer. Nous présentons, dans la section 7.3, la façon de faire de C++ et identifions ses limites. Dans la section 7.4, nous proposons un schéma de compilation séparée pour les langages à objets tandis que la section 7.5 concerne l'adaptation d'un tel schéma à un langage de programmation à modules et à raffinement de classes comme PRM. La section 7.6 discute des travaux connexes. Pour terminer, la section 7.7 ouvre les perspectives et conclut ce chapitre.

7.2 Principe de la compilation séparée

7.2.1 Un peu d'histoire

La compilation séparée est apparue dans l'histoire de l'informatique pour des raisons techniques (FORTRAN II en 1957). Les compilateurs de l'époque n'étaient alors pas capables de compiler de gros programmes en une fois : la mémoire nécessaire pour compiler un gros programme d'un coup était trop importante pour l'époque.

Toutefois, de nos jours, même si la puissance des microprocesseurs et la quantité de mémoire des machines ont augmentés, il en est de même de la taille des logiciels. Par exemple, le logiciel de bureautique OpenOffice.org[1], qui peut être considéré comme un gros logiciel puisqu'il fait plus de 10 000 000 lignes de code, met environ 8 h à compiler sur un petit PC actuel cadencé à 3 Ghz.

À ce point, il est important d'introduire la distinction entre la *compilation indépendante* et la *compilation séparée* [Clark et Horning, 1971; Horning, 1976].

Compilation indépendante

La compilation indépendante est celle effectuée par la majorité des assembleurs mais aussi par divers langages de plus haut niveau tels que FORTRAN ou PL/1.

Chaque unité est compilée de façon complètement indépendante, c'est-à-dire indépendamment :

- de ses utilisations, c'est-à-dire des programmes dont cette unité fait partie ;
- des unités dont elle dépend : une unité n'a aucun moyen d'accéder à des informations contenues dans les autres unités — comme le nom des méthodes disponibles, leur signature, le type des variables statiques, etc.

Chaque unité est donc une boîte noire pour les autres.

[1] `http://www.openoffice.org/`

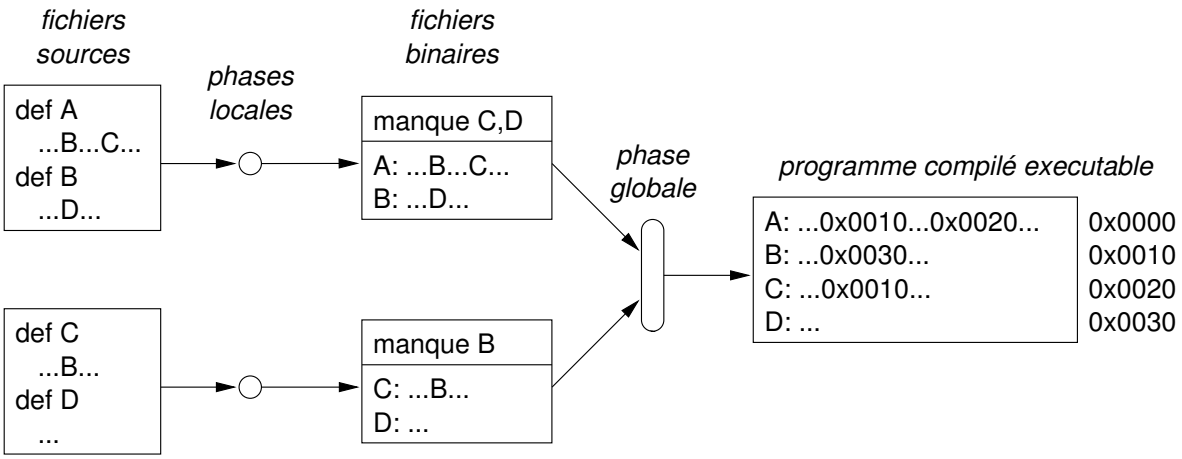


FIG. 7.1: La compilation séparée

L'inconvénient de la compilation indépendante est qu'elle ne permet pas de garantir le niveau de cohérence de la compilation globale. En effet, il n'est pas possible de vérifier lors de la compilation d'une unité la consistance des interfaces avec les unités dont elle dépend. Si incohérence il y a, elle sera au mieux détectée lorsque les unités compilées sont regroupées pour former un programme exécutable.

Compilation séparée

La compilation séparée réconcilie la sûreté de la compilation (en particulier la sûreté des types) et les raisons pragmatiques et sémantiques de la compilation par morceaux.

Les unités sont toujours compilées indépendamment de leurs utilisations (c'est-à-dire des programmes) mais cette fois-ci la compilation d'une unité nécessite la connaissance de certaines spécifications (l'*interface*) des unités dont elle dépend. Chaque unité est pour les autres une boîte noire avec un bord blanc.

Une des solutions pour mettre en œuvre cette notion d'interface consiste à diviser une unité en deux fichiers, l'un qui spécifie l'interface de l'unité (on parle aussi d'entête) et l'autre qui spécifie l'implémentation de l'unité (on parle aussi de corps). On retrouve cette méthode de compilation dans divers langages comme ADA ou dans une moindre mesure C et son petit frère C++.

Dans la suite, nous nous intéressons, bien entendu, à la compilation séparée.

7.2.2 Schéma de compilation séparée

Les schémas de compilation séparée mettent en œuvre deux phases lors de la génération de programmes (figure 7.1). La première est dite *locale* car son rôle est

de compiler chaque *fichier source* indépendamment de tout programme l'utilisant. Les résultats de cette phase sont appelés *fichiers binaires*. Ils sont écrits dans le langage cible de la compilation mais ne sont pas fonctionnels en l'état puisque les informations fournies par les autres fichiers sources sont manquantes. La seconde phase est dite *globale* car elle assemble et adapte les fichiers binaires afin de construire un *programme compilé* opérationnel et pleinement exécutable.

Contrairement à la compilation globale, la compilation séparée des programmes implique nécessairement une modularité du processus de compilation puisque la phase locale s'effectue de façon indépendante des programmes finaux. Cette modularité offre deux avantages immédiats. D'une part, un même fichier binaire peut être utilisé pour construire deux programmes distincts. D'autre part, la reconstruction d'un programme permet de réutiliser les fichiers binaires déjà compilés si leurs fichiers sources n'ont pas été modifiés.

Dans les langages procéduraux comme C, Pascal ou Ada, la phase locale, appelée simplement *phase de compilation*[2], produit des fichiers binaires appelés *fichiers objets*[3]. Ceux-ci sont écrits en langage machine mais utilisent des *noms symboliques* pour désigner les adresses des fonctions et des variables statiques qui sont inconnues et doivent être fournies par les autres unités d'un programme. La phase globale, appelée *édition de liens*, regroupe les différents fichiers binaires, calcule les adresses des fonctions et variables statiques et substitue aux noms symboliques les valeurs numériques calculées [Levine, 1999].

7.3 Édition de liens en C++

La programmation par objets telle qu'elle est introduite par C++ ajoute trois difficultés pour un éditeur de lien procédural. La première est la gestion des noms puisqu'un même nom peut désigner plusieurs propriétés distinctes. Le *name mangling* permet de résoudre ce problème. La seconde difficulté est la gestion des initialisations statiques qui correspond au code qui doit être exécuté avant que la main ne soit donnée à la fonction principale — c'est-à-dire le point d'entrée du programme. La troisième et dernière difficulté est la gestion du code exécutable non lié à un fichier source particulier comme les fonctions en ligne ou les classes paramétrées. Cela peut amener, dans une approche de compilation séparée, à générer plusieurs fois un même code exécutable.

[2]Suivant le contexte, le terme « compilation » désigne soit la phase locale, soit le processus dans son ensemble (phase locale + phase globale). Dans les langages procéduraux, le terme est utilisé pour désigner la phase locale. Pour désigner le processus dans son ensemble, on trouve souvent le terme de « construction », ou *build*. Toutefois, étymologiquement, « compiler » (probablement du latin *compilare*) désigne le fait de regrouper et combiner ensemble des éléments d'origines diverses (*compiler une encyclopédie* par exemple), ce qui correspondrait mieux à la phase globale.

[3]Le terme « objet » relève d'une polysémie remarquable dans le domaine de l'informatique.

7.3.1 Mutilation de nom (*name mangling*)

Un même nom peut désigner des éléments distincts. En C++ cette distinction est fonction de la classe de définition, du type des arguments ou de l'espace de nom considéré. Utiliser le nom d'un élément comme symbole est alors impossible puisqu'un symbole doit désigner un élément de façon non ambiguë et unique. Le *name mangling*, littéralement « mutilation de nom », permet de construire ces symboles à partir du nom des éléments.

À l'origine (vers les années 1970), le *name mangling* était utilisé pour séparer les routines écrites par le programmeur de celles inhérentes au système. L'approche prise par les systèmes Unix fut de mutiler les noms des procédures C et Fortran afin d'éviter les collisions par inadvertance. Les noms des procédures C étaient modifiés par l'ajout d'un caractère souligné, `main` devenant `_main`. Sur d'autres systèmes, d'autres politiques furent prises, comme l'utilisation de caractères interdits en C pour nommer les symboles réservés au système, comme le point (.) ou le dollar ($).

Pour C++, la mutilation de nom obéit à des règles complexes dépendantes de chaque compilateur. Toutefois, [Ellis et Stroustrup, 1990] propose un schéma de *name mangling* qui est utilisé par la majorité des compilateurs C++ à quelques variations près. Dans tous les cas, la liste des éléments prenant part à ces règles reste la même :

- l'espace de nom de la définition (que cet espace de nom soit un *namespace* ou une classe) ;
- le nom de la fonction ou de la variable. Dans le cas des fonctions, certains noms comme les opérateurs nécessitent une transformation ;
- le type statique des paramètres dans le cas des fonctions. Toutefois, les noms des types doivent être également mutilés pour intégrer les informations d'espace de nom, et d'instanciations formelles pour les classes paramétrées.

Exemple, la méthode de classe dont la signature est

```
foo::B::C<int, 4>::operator+(bar::D<int, 4, 5>)
```

est mutilée par g++-2.95 en

```
__pl__Q33foo1Bt1C2Zii4GQ23bart1D3Zii4i5
```

et par g++-3.3 et supérieur en

```
_ZN3foo1B1CIiLi4EEplEN3bar1DIiLi4ELi5EEE
```

La mutilation de nom répond bien à l'objectif qui consiste à donner un nom unique à chaque routine et variable. Le prix à payer est la manipulation de noms très longs

et complètement illisibles pour un utilisateur humain lorsqu'ils apparaissent dans des listings de code ou dans des messages d'erreur[4].

7.3.2 Initialisations et destructions statiques

En C++, les variables statiques peuvent être initialisées. Ces initialisations doivent avoir lieu avant l'exécution de la fonction principale du programme (`main`). De même, lors de la terminaison correcte de l'exécution du programme les variables statiques doivent être proprement détruites, par l'invocation de destructeurs si c'est nécessaire. La terminaison correcte d'un programme est causée par le `return` de la fonction `main` ou par l'appel de la fonction `exit`. La terminaison incorrecte d'un programme est causée par la fonction `abort` ou par le signalement d'une exception non attrapée.

La solution généralement utilisée consiste, lors de la compilation séparée, à regrouper les initialisations de chaque fichier dans un sous-programme particulier, puis, lors de l'édition de liens, à générer des appels à ces sous-programmes avant l'appel à `main`. La compilation séparée produit donc une ou plusieurs fonctions particulières, non exportées, qui regroupent les initialisations et les destructions des variables statiques. Lors de l'édition de liens, ces fonctions sont collectées et regroupées aux endroits adéquats dans l'exécutable.

Puisque ces initialisations et destructions ont lieu sans l'intervention du programmeur, il est légitime de se poser la question de leur ordre. [Stroustrup, 2000] est clair sur le sujet : dans un même fichier source, l'ordre des initialisations est celui des déclarations par contre, l'ordre des initialisations entre plusieurs fichiers sources est indéterminé — en pratique et avec `g++`, l'ordre semble être celui d'apparition des fichiers de la ligne de commande de l'édition de liens. En ce qui concerne l'ordre des destructions, celui-ci n'est pas mentionné mais les compilateurs que nous avons testés considèrent l'ordre inverse de celui des initialisations.

7.3.3 Code non lié à un fichier source

En C++, certaines parties d'un programme ne sont pas associées à un fichier source particulier. Nous appelons ces parties le *code non associé*. Ce code correspond à celui écrit ou implicitement déclaré dans les fichiers d'entête.

Pour rappel, en C++, l'unité de compilation est indépendante de la notion de classe. La déclaration de la structure des classes est répétée dans chacun des fichiers corps (*body*, .cpp, .C, .c++ ou .cc) qui utilisent cette classe. N'oublions pas que le `#include` n'est rien d'autre qu'une macro-commande interprétée par le préprocesseur. Pour le compilateur

[4]Mais certains en sont à relativiser en précisant que de toute façon, C++ a un problème intrinsèque vis-à-vis de ses espaces de noms imbriqués et de sa compilation hétérogène des classes paramétrées puisque les façons littérales de désigner de façon non ambiguë les différents objets de C++ sont parfois aussi incompréhensibles que la version mutilée.

il n'y a aucune différence entre le résultat d'un `#include` et la recopie à la main des fichiers d'entête (*headers*, `.h` ou `.hh`) et des fichiers de corps correspondant à des classes et routines génériques (*templates*) dans les fichiers de corps les incluant.

Ainsi en C++, le code correspondant aux structures suivantes n'est pas lié à un fichier source particulier :

- les fonctions en ligne. Déclarées dans les fichiers d'entêtes, elles agissent la plupart du temps comme des macros, leur implémentation étant alors recopiée à chaque utilisation. Toutefois, dans certains cas, il est nécessaire que ces fonctions se comportent comme de vraies fonctions (pour la liaison tardive par exemple) ;
- les classes paramétrées instanciées. Le code, déclaré formellement dans les fichiers d'entête, est adapté à chaque utilisation dans un fichier source et doit être compilé spécifiquement (implémentation hétérogène [Odersky et Wadler, 1997]) ;
- les constructeurs et destructeurs implicites. Bien que par définition, ils ne sont pas écrits dans le code source, ils existent à l'exécution et doivent être générés lors de la compilation du programme ;
- les tables des fonctions virtuelles (*Virtual Function Tables* ou VFT).

Lors de la compilation séparée de ces langages, se pose la question de l'endroit où doit être généré le code exécutable correspondant au code non associé. La solution compatible avec le principe de la compilation séparée consiste à compiler plusieurs fois ce même code : la compilation séparée de chaque fichier source qui a besoin de code non associé, générera sa propre version. Lors de la phase globale, l'éditeur de liens aura pour tâche de ne conserver dans l'exécutable final qu'une seule version compilée de chaque code non associé.

Ainsi, prenant l'exemple d'un simple `.h` implémentant un compteur de la façon suivante :

```
// cpt.h
class Cpt {
        int val;
public:
        virtual void raz(void) { val=0; }
        virtual int next(void) { return ++val; }
};
```

La compilation de fichiers sources incluant `cpt.h` peut nécessiter une génération multiple. Ainsi, les fichiers `unit1.cpp` et `unit2.cpp` du programme `prog`, utilisant tous les deux un compteur, se retrouvent avec du code en double.

```
$ nm -C unit1.o
00000000 W Cpt::raz()
00000000 W Cpt::next()
00000000 W Cpt::Cpt()
00000000 V typeinfo for Cpt
```

```
00000000 V typeinfo name for Cpt
00000000 V vtable for Cpt
...
$ nm -C unit2.o
00000000 W Cpt::raz()
00000000 W Cpt::next()
00000000 W Cpt::Cpt()
00000000 V typeinfo for Cpt
00000000 V typeinfo name for Cpt
00000000 V vtable for Cpt
...
```

Une fois le programme `prog` lié, l'exécutable ne contient alors qu'une version compilée.

```
$ nm -C prog
080484d8 W Cpt::raz()
080484e8 W Cpt::next()
08048502 W Cpt::Cpt()
08048680 V typeinfo for Cpt
08048688 V typeinfo name for Cpt
08048670 V vtable for Cpt
...
```

Toutefois, il est possible de réduire les générations multiples en associant de manière arbitraire le code à un fichier particulier. Ainsi, on peut associer les constructeurs implicites et les tables des fonctions virtuelles au fichier qui définit la première méthode déclarée de la classe considérée.

Une autre solution consiste à générer le code de façon incrémentale : lors de la première compilation, le code non associé n'est pas compilé, puis lors de l'édition de liens, le code non associé manquant est marqué. Une deuxième compilation a lieu qui consiste à compiler le code marqué. Lors de la deuxième édition de liens, si à nouveau du code est manquant, une troisième compilation a lieu, et ainsi de suite. Cette façon de faire est implémentée dans certains environnements de développement intégrés mais ne peut plus être pleinement qualifiée de compilation séparée.

7.3.4 Limites de l'approche de C++

Le schéma de compilation utilisé par les compilateurs C++ possède un défaut intrinsèque puisqu'il se base sur des éditeurs de liens procéduraux qui ne prennent pas en compte les spécificités du modèle objet. Lors de la phase locale, ces mécanismes inadaptés imposaient donc des techniques d'implémentation complexes et coûteuses spatialement et temporellement (cf. chapitre précédent).

C'est pourquoi le modèle de compilation que nous proposons élargit le champ d'action de l'éditeur de liens de façon à améliorer l'implémentation des langages à objets en compilation séparée.

7.4 Compilation séparée et optimisations globales

Le schéma de compilation que nous présentons maintenant est un approfondissement de celui présenté dans [Privat et Ducournau, 2005a]. Il est compatible avec le schéma classique de compilation séparée et permet l'utilisation de trois techniques globales : l'analyse de types, la coloration et les arbres de sélections.

La principale différence avec les schémas utilisés pour les langages procéduraux se situe au niveau des informations inconnues dans les fichiers binaires intermédiaires. Celles-ci ne se réduisent pas à des adresses des fonctions et de variables statiques.

7.4.1 Phase locale

Afin d'être le plus général possible, nous considérons que la classe est l'unité de compilation. Bien évidemment, dans le cadre de la compilation du langage PRM, c'est le module qui est l'unité de compilation ; nous traitons le cas des modules et du raffinement dans la section 7.5.

La phase locale prend en entrée le code source d'une classe et produit en sortie trois niveaux différents de code (figure 7.2) :

- le *schéma externe* de la classe décrit son interface et ses super-classes directes, sans le code — le schéma externe est une instance du méta-modèle présenté chapitre 3 ;
- le *schéma interne* est la représentation des sites polymorphes et de la circulation des types dans les méthodes de la classe — le schéma interne est ce que [Agesen, 1996] appelé un *patron* (*template*, cf. section 7.6) ;
- le *code binaire* correspond à la sortie habituelle du compilateur, dans le langage cible de la compilation (généralement du code machine).

Ces trois parties peuvent être incluses dans le même fichier ou dans des fichiers distincts mais le schéma externe doit pouvoir être disponible séparément.

Schémas externes des classes

La compilation d'une unité est indépendante des autres unités jusqu'à un certain point seulement car elle doit connaître l'interface des classes dont l'unité est sous-classe ou cliente.

Cette interface peut être disponible, soit dans le schéma externe de la classe utilisée parce qu'elle a déjà été compilée ou parce que ce schéma a été fourni séparément, soit dans le code source de la classe utilisée. Dans ce dernier cas, une génération récursive du

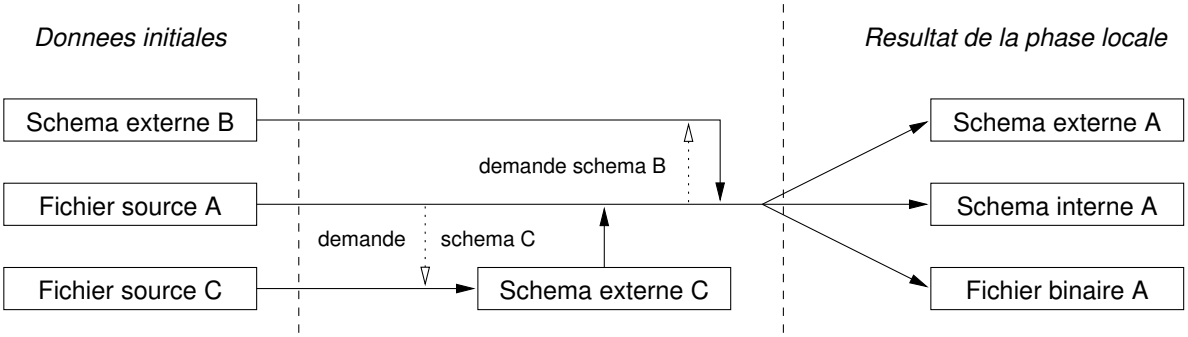


FIG. 7.2: Phase locale : compilation de la classe A

schéma externe sera nécessaire (figure 7.2). Toutefois, lorsque les unités de compilation sont les modules et que la dépendance entre module est hiérarchique (cf. section 2.7 page 27), la génération récursive du schéma n'est pas strictement nécessaire, une compilation récursive suffit.

Le schéma externe d'une classe est une instance du méta-modèle du langage (cf. chapitre 3). Son rôle est de fournir les briques élémentaires permettant à la phase globale de construire le modèle complet d'un programme.

Pour des raisons d'optimisation, le schéma externe d'une classe peut contenir des informations étendues à ses super-classes directes et indirectes. Ce genre d'optimisation est intéressant dans les langages où l'héritage est complexe (comme par exemple EIFFEL via les clauses de redéfinition), mais aussi pour la compilation de programmes utilisant des hiérarchies profondes. Ce dernier point peut s'illustrer en C++ par la technique de précompilation d'en-têtes qui peut réduire, dans des cas réels, de plus de la moitié la durée de compilation d'un programme.

Analyse de types et schéma interne

Le *schéma interne* d'une classe est un graphe qui représente la circulation des types dans les méthodes de la classe, entre leurs entrées et leurs sorties. Une entrée peut être un receveur, un paramètre, la lecture d'un attribut ou le résultat d'un appel de méthode et une sortie est le résultat de la méthode si c'est une fonction, l'écriture d'un attribut ou les arguments d'un appel de méthode. Les sommets du graphe sont les entrées, les sorties et des nœuds intermédiaires, et ses arcs sont des contraintes d'inclusion des types concrets associés à chaque sommet. Les sorties peuvent alors êtres vues comme des fonctions des entrées : [Boucher, 1999] en donne d'ailleurs une forme fonctionnelle. Les instanciations de classes sont explicitées dans le schéma interne : si la méthode est vivante, les classes qu'elle instancie le sont aussi. Une analyse de types intraclasse et intraprocédurale [Privat, 2002] permet de construire ces schémas internes en minimisant

leur taille.

Pour réduire encore celle-ci, l'analyse peut être limitée aux types polymorphes, le type concret des types non polymorphes, en particulier primitifs, étant connu statiquement. On notera qu'il s'agit de la partie la plus simple de l'analyse de types, qui est passée sous silence dans quasiment toute la littérature.

Génération du code

Le code généré contient des symboles. Comme dans les schémas standards de compilation, les symboles sont utilisés pour l'adresse des fonctions et des variables statiques. Dans notre proposition, nous introduisons également d'autres symboles : des symboles pour l'adresse de résolution des sites d'appel, des symboles pour la couleur des attributs, des symboles pour la couleur des classes, des symboles pour l'identifiant des classes, des symboles pour la taille des instances et des symboles pour l'adresse des tables de classe.

La phase locale affecte un symbole unique à chaque site d'appel. Ce symbole correspond à l'adresse de la résolution. Ainsi, pour un site d'appel `x.f(a)` dont le symbole associé serait $f12$, la phase locale génère un appel procédural direct :

```
call ⟨f12⟩
```

La coloration, cf. section 6.4.5 (page 137), garantit l'invariance de la position d'un attribut, l'adresse d'un attribut de couleur Δ_a est donc $objet + \Delta_a$. Ainsi, pour Δ_a le symbole représentant la couleur de l'attribut a, l'accès s'effectue par un accès direct dans l'objet, par exemple pour une lecture :

```
load [object + ⟨Δa⟩] → val
```

et pour une écriture :

```
store val → [object + ⟨Δa⟩]
```

De même, la coloration permet de compiler une vérification de type par un accès direct dans la table de la classe de l'instance. Par exemple, pour Δ_C et ID_C, les symboles représentant respectivement la couleur et l'identifiant de la classe C, vérifier que x est une instance directe de C ou une instance d'une sous-classe de C est compilé par :

```
load [x + #tableOffset] → table
load [table + ⟨ΔC⟩] → class
cmp class = ⟨IDC⟩
jne #false
; test OK
```

Lors de l'instanciation d'une classe, un objet doit être construit. Il est représenté en mémoire par un espace délimité contenant d'une part le pointeur vers la table des

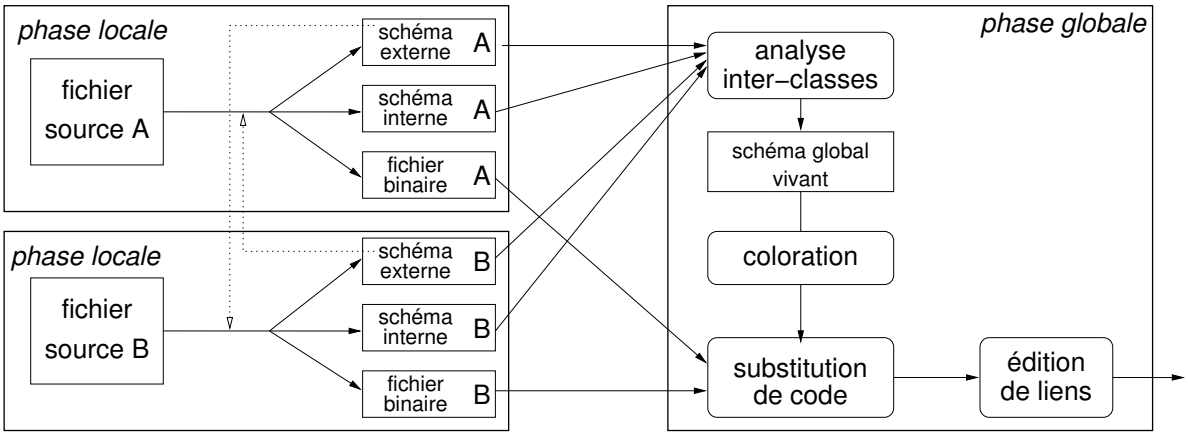


FIG. 7.3: Phase globale : compilation des classes A et B

méthodes à la position `#tableOffset` et d'autre part les attributs. Le code de l'instanciation doit donc réserver un espace correspondant à la couleur maximum de la classe et écrire l'adresse de la table à la position `#tableOffset`.

7.4.2 Phase globale

La phase globale comporte trois étapes : l'analyse de types qui détermine le code vivant, la coloration et la transformation du code (figure 7.3).

Analyse de types

L'analyse de types interclasse se base sur l'ensemble des schémas internes et externes des unités. Dans les techniques d'analyses à flot comme CFA, CPA ou SCS (cf. section 6.4.2 page 132), les schémas internes sont liés entre eux, éventuellement avec création de contours pour tenir compte des contextes d'appel, en connectant leurs entrées et sorties de façon à constituer un réseau global de contraintes d'inclusion de types.

En partant du point d'entrée du programme, les classes vivantes et leurs méthodes vivantes sont identifiées, ainsi que le type concret de toutes leurs expressions, c'est-à-dire de tous les sommets du graphe. Les classes mortes ne sont pas examinées et les méthodes et attributs vivants sont incorporés au graphe et au schéma global au fur et à mesure de la détection de leur caractère vivant.

On obtient un schéma global vivant, accompagné d'informations sur les types concrets des sites d'appel vivants. Le code mort peut-être considéré comme supprimé puisqu'il ne sera pas traité par les étapes suivantes ni inséré dans le programme compilé, réduisant ainsi sa taille.

Coloration

La coloration, cf. section 6.4.5 (page 137), s'applique à ce schéma global vivant, restreint, en ce qui concerne les méthodes, à celles qui sont utilisées de façon polymorphe. Une heuristique de coloration produit les valeurs des identifiants et couleurs de classes, méthodes et attributs, ainsi que la taille des tables, c'est-à-dire les valeurs qu'il faudra substituer aux symboles correspondants (Δ_m, Δ_a, Δ_c, Id_c, $Size_c$ et VFT_c). Une fois ces valeurs obtenues, on peut générer les données globales correspondant aux tables de méthodes et tables de sous-typage.

Substitutions et génération de code

Une fois les étapes précédentes effectuées, la substitution dans le code est proche de celle utilisée pour la résolution des adresses par l'éditeur de liens, puisqu'il s'agit de données numériques dans les deux cas. Le code compilé de chaque classe vivante est examiné séquentiellement : ni mémorisation ni retour en arrière ne sont nécessaires.

Les couleurs calculées pendant l'étape de coloration sont substituées aux symboles correspondants. Pour chaque site d'appel, l'ensemble des méthodes potentiellement appelées est connu. Le remplacement du symbole est alors fonction du polymorphisme :

- pour les sites monomorphes, le symbole est remplacé par l'adresse de l'unique méthode : la compilation de tels sites revient à un appel procédural (*call*) direct, le coût du polymorphisme est alors entièrement annulé ;
- pour les sites polymorphes, le symbole est remplacé par l'adresse d'un thunk dont le rôle est de sélectionner la bonne méthode et de faire un branchement dessus : la compilation de tels sites revient à un `call` direct, le code de sélection et un branchement indirect (`jmp`).

Remarque : La redéfinition non sûre du type peut nécessiter l'utilisation de thunks dans le cas monomorphe et compliquer le contenu des thunks dans le cas polymorphe, en particulier, en cas de redéfinition contravariante du type de retour (raffinement) où le `jmp` doit être remplacé par un `call`. Nous revenons en détail sur ce cas dans le chapitre suivant, section 8.3.5.

Enfin, le code correspondant aux tables de méthodes doit être généré.

7.5 Compilation séparée et raffinement de classes

L'approche proposée dans le chapitre 4, qui considère des modules indépendants combinés entre eux pour former un programme, est compatible avec le schéma de compilation que nous proposons dans ce chapitre — en réalité, elle est compatible avec tous les schémas de compilation séparée, du moment qu'ils permettent de ne prendre connaissance du méta-modèle complet qu'à l'édition de liens.

En fait, une dépendance hiérarchique des modules simplifie même le schéma de compilation puisque la relation de dépendance entre modules est la relation de clientèle : il n'y a plus de besoin intrinsèque d'avoir à extraire l'interface d'un programme sans pouvoir le compiler.

Après l'édition de liens, les différentes classes locales d'une classe globale ont disparu : seules restent les classes locales du module principal du programme. Le fait de retrouver des classes sans module et sans raffinement permet de prévoir que l'implémentation peut se réaliser sans surcoût spatial ou temporel à l'exécution par rapport au même langage sans raffinement.

Toutefois, le raffinement de classes est difficilement compatible avec le chargement dynamique, c'est-à-dire le raffinement de classes lors de l'exécution d'un programme. En effet, celui-ci pose des problèmes liés à la nécessité de modifier toutes les instances déjà créées avec d'une part les problèmes de cohérences intrinsèques (que faire des nouveaux attributs ?) et d'autre part des problèmes de performances puisque cela impose des techniques d'implémentation comparables à la migration d'instances — par exemple `change-class` en CLOS.

7.6 Compilation séparée optimisante : autres approches

L'idée de calculer la coloration à l'édition de liens avait déjà été avancée par [Pugh et Weddell, 1990] mais il ne semble pas qu'elle ait jamais été appliquée. Il n'est d'ailleurs pas certain que la coloration ait été pleinement utilisée à part dans un usage très restreint en YAFOOL [Ducournau, 1991].

La notion de *schéma interne* correspond à celle de *patron* (*template*) introduit par [Agesen, 1996] pour désigner le réseau de contraintes associé à une procédure dans les analyses polyvariantes, où un même patron peut être dupliqué suivant les sites d'appel. Mais les analyses intra et interprocédurales n'y étaient pas séparées dans le temps.

Les architectures proposées par [Fernandez, 1995] et [Boucher, 1999] sont assez proches de la nôtre, respectivement pour MODULA 3 (mais sans élimination du code mort) et pour les langages fonctionnels (dans le but de supprimer les tests de types et d'optimiser l'utilisation des variables de type fonctionnel). Dans les deux cas, la différence principale réside dans la compilation séparée qui produit du code dans un langage intermédiaire de haut niveau, ce qui nécessite une compilation plus ou moins légère de la totalité du programme en phase globale, après l'optimisation.

Dans [Privat et Ducournau, 2004], nous avons présenté un schéma de compilation voisin de celui présenté dans ce chapitre. La principale différence en est l'absence de génération de thunks lors de la phase globale : pour un même mécanisme (par exemple un envoi de message, la phase locale générait différentes implémentations (par exemple,

un appel direct et un accès à la table des méthodes). Chaque implémentation était isolée par un langage à balise (à la XML). Lors de la phase globale, après l'analyse de type et la coloration, seule l'implémentation la plus pertinente était conservée.

7.7 Perspectives et conclusion

Deux grandes améliorations pourraient être apportées à ce schéma de compilation. D'une part la mise en ligne des appels monomorphes et d'autre part la gestion des bibliothèques partagées.

7.7.1 Mise en ligne des appels monomorphes

Il y a intérêt à mettre en ligne les procédures simples (cf. section 6.4.3 page 135). Le surcoût de leur duplication est compensé par l'absence d'appel avec tout ce que cela implique (sauvegarde et restauration du contexte, sauts). Plusieurs types de procédures simples ont été identifiées [Zendra *et al.*, 1997] : les méthodes ne faisant rien, celles retournant une constante, le receveur ou un paramètre, les accesseurs d'attributs ou les méthodes mandataires appelant uniquement une autre méthode sur l'un de leurs paramètres.

Dans le cadre de notre schéma, la mise en ligne gardée semble être une solution possible [Detlefs et Agesen, 1999], toutefois, cette mise en ligne ne concernerait de des méthodes statiquement connues lors de la phase locale.

7.7.2 Bibliothèques partagées

Notre modèle de compilation n'est *a priori* pas plus compatible que SmartEiffel avec l'utilisation de bibliothèques partagées de classes Eiffel. Plusieurs directions sont tout de même envisageables. On peut d'abord considérer une bibliothèque compilée en totalité, sans code mort, ni détection d'appels monomorphes. La seule spécificité de notre approche est alors la coloration.

La solution la plus grossière consiste à faire la coloration et l'édition de liens au chargement. Lors du chargement, les bibliothèques sont chargées et adaptées pour chaque programme. Cette voie implique que le temps de chargement des programmes à objets augmente considérablement, d'une part à cause de la surcharge de travail liée à l'édition de liens et d'autre part en empêchant la liaison paresseuse (*lazy procedure linkage*) qui cherche, charge et lie une bibliothèque à sa première utilisation et non au début du programme. De plus, cette voie ne permet pas le chargement dynamique de bibliothèques, impose une duplication du code pour chaque processus et nécessite sans doute une adaptation du système d'exploitation pour sa mise en place.

Une seconde solution utilise des tables d'indirections : cette voie s'inspire des techniques utilisées pour les bibliothèques dynamiques des langages procéduraux. Les valeurs de la coloration seraient stockées dans les données des processus plutôt que dans le code. Ainsi le code serait partageable par tous les processus et ne nécessiterait pas de modification au chargement (et *a fortiori* de modification du système d'exploitation). En contrepartie, l'utilisation d'indirections aurait un impact sur les performances.

Une troisième solution consiste à effectuer la coloration lors de la compilation de la bibliothèque : les couleurs sont alors associées au schéma des classes. On peut alors, soit limiter le développement pour empêcher les conflits, soit les résoudre par un arbre de décision, suivant la technique décrite dans [Ducournau, 1997]. La technique s'applique parfaitement aux méthodes. Pour les attributs, l'indirection nécessaire peut être obtenue par la technique de simulation des accesseurs qui peut remplacer la coloration d'attributs dans le cadre de la coloration de méthodes [Ducournau, 2002b]. Dans ce cas, des balises intégreraient facilement la double compilation proposée par [Myers, 1995]. Il resterait à trouver une adaptation pour le test de sous-typage.

7.7.3 Conclusion

Nous avons présenté dans ce chapitre un schéma de compilation séparée dans lequel sont intégrées les techniques globales d'implémentation que sont l'analyse de types, la coloration et les arbres binaires de sélection.

De façon théorique, ce schéma de compilation conserve les nombreux avantages de la compilation par morceaux tout en offrant une implémentation des mécanismes objets plus efficace que celle des schémas séparés traditionnels comme celui de C++ (cf. section 6.3 page 121) : code mort éliminé, réduction d'un ordre de grandeur de la taille des tables de méthodes, abandon des sous-objets, détection des appels monomorphes, etc.

Par rapport à la stratégie adoptée par SmartEiffel (cf. section 6.4.6 page 140), les avantages sont forcément plus modestes : la qualité de l'analyse de types peut être identique dans les deux approches, mais théoriquement SmartEiffel devrait obtenir des performances temporelles un peu meilleures, grâce à son usage de la spécialisation de code — particularisation et mise en ligne.

Un autre avantage de ce schéma de compilation est parfaitement compatible avec notre approche des modules (cf. section 2.7 page 27) et supprime même tout surcoût apporté par le mécanisme du raffinement de classes que nous avons présenté dans le chapitre 4.

Afin de confronter la théorie et la pratique, le chapitre suivant est consacré à la mise en œuvre de ce schéma dans le cadre de `prmc`, le compilateur que nous avons développé pour le langage PRM.

Chapitre 8

prmc, le compilateur PRM

Préambule

Nous présentons dans ce chapitre `prmc`*, le prototype de compilateur que nous avons développé pour le langage* PRM*. Nous y détaillons également la mise en œuvre des techniques que nous avons exposées aux chapitres précédents et comparons l'efficacité des techniques d'implémentation au travers de différents benchmarks.*

8.1 Présentation du compilateur prmc

Le compilateur `prmc` permet de compiler un programme écrit en PRM afin de produire un exécutable. Pour rappel, un programme PRM est constitué d'un module principal et de ses super-modules, chaque module étant matérialisé par un fichier source.

Actuellement, nous disposons d'un prototype écrit en Ruby. L'objectif de ce prototype est double. D'une part il s'agit de pouvoir intégrer de nombreuses techniques différentes (Ruby étant un langage permettant le développement rapide de prototypes). En effet, en étant polyvalent, `prmc` permet de comparer l'efficacité de plusieurs techniques de compilation et de mesurer l'impact des différentes combinaisons. D'autre part il s'agit de répondre à la nécessité de disposer d'un compilateur PRM afin de bootstrapper le compilateur PRM autogène actuellement en cours de développement. En effet, pour que le développement d'un compilateur PRM écrit en PRM ait du sens, il est nécessaire de disposer d'un premier compilateur.

Toutefois, le prototype n'a pas pour vocation d'être performant sur le temps de compilation : Ruby est un langage interprété[1] et nous préférons passer du temps à ajouter différentes techniques d'implémentation plutôt qu'à optimiser l'implémentation

[1] Il s'agit d'un abus de langage, le langage Ruby n'étant qu'un langage de programmation. Toutefois, actuellement, le seul moyen d'exécuter un programme écrit en Ruby consiste à utiliser un interpréteur.

en Ruby de certaines d'entre elles. Il nous faudra attendre la finalisation du compilateur écrit en PRM, dans le cadre du master recherche en informatique de Floréal Morandat, pour évaluer de façon pertinente le temps de compilation nécessaire à l'utilisation des techniques d'implémentation retenues.

8.1.1 Programmes de test

Dans ce chapitre, nous discutons des différentes techniques de compilation au fur et à mesure et nous les illustrons par leur impact sur différents programmes de test. C'est pourquoi nous présentons dès maintenant la demi-douzaine de programmes de test que nous utilisons tout au long du chapitre.

Lors du développement du compilateur, de nombreux programmes furent écrit pour d'une part s'assurer du bon fonctionnement du compilateur et d'autre part effectuer des tests de non régression. Au total, plus de 200 programmes de tests furent écrits pour presque 7 000 lignes de code. Cependant, la plupart de ces programmes sont très courts et ne se concentrent que sur une toute petite partie de la spécification du langage PRM. Ainsi, afin de vérifier l'efficacité du compilateur sur de « gros » programmes, ces petits programmes de tests ne sont pas adaptés : il nous faut de « gros » programmes écrits en PRM. Or, bien évidemment, il n'existe pas de gros programme écrit en PRM.

Nous avons donc développé quelques petits programmes particuliers destinés à tester le compilateur `prmc`. Malgré tout, nous sommes conscients que ces programmes sont limités et ne couvrent que faiblement les caractéristiques du langage.

Les programmes

`bench_421` Calcule de façon itérative le nombre de sauts que met une suite pour converger. Ce programme utilise principalement des opérations sur les entiers, ne crée quasiment pas d'instance et la quasi-totalité des envois de message sont monomorphes.

`bench_netsim` Simule un réseau de communication dont les nœuds peuvent jouer des rôles variés. Un ordonnanceur s'occupe de coordonner les signaux entre les nœuds du réseau. Contrairement au précédent, ce programme manipule des objets complexes, construit beaucoup d'instances et profite du polymorphisme.

`bench_string_append` Construit une très longue chaîne de caractères par concaténations successives puis effectue des recherches de motifs dans cette grande chaîne en utilisant l'algorithme de Boyer-Moore [Boyer et Moore, 1977].

`shootout_bynarytrees` Ce programme a été conçu pour tester l'efficacité des ramasse-miettes (cf. section 8.1.4). Comme son nom l'indique, il construit des arbres binaires et les laisse se faire ramasser.

`bench_random_n_sort` Remplit des tableaux avec des entiers tirés aléatoirement puis trie ces tableaux. Ce programme teste l'utilisation de types primitifs dans des collections.

`bench_complex_sort` Remplit des tableaux avec des objets complexes puis les trie en fonction de différents critères. Ce programme contient beaucoup de polymorphisme puisque les objets dans les tableaux sont instances de classes variées.

Protocole de test

Pour chaque technique de compilation (caractérisée par une combinaison d'options du compilateur, cf section 8.1.6) et chaque programme de test, un exécutable est généré. Ensuite, celui-ci est exécuté au moins cinq fois d'affilée et le meilleur temps est conservé.

L'ordinateur que nous utilisons pour compiler et exécuter les programmes est un Bi-Xeon CPU 1.80GHz.

8.1.2 Modes de fonctionnement

Le compilateur possède trois modes de fonctionnement :

`prmc` *`files`* C'est le mode par défaut, ou mode séparé. Chacun des modules du programme dont le module principal est un module implicite qui dépend des modules `files` est compilé séparément (si ce n'est déjà fait) puis les modules compilés sont liés pour produire un exécutable.

`prmc -c` *`files`* C'est la restriction à la phase locale du mode séparé, chaque module est seulement compilé séparément. Aucune phase globale n'a lieu, et aucun exécutable n'est généré.

`prmc -g` *`file`* C'est le mode global, le programme dont le module principal est le fichier `file` est analysé dans sa totalité et un exécutable est généré. Contrairement au mode séparé, tous les modules du programme sont (re-)compilés à chaque fois.

Toutefois, que soit utilisé le mode séparé ou global, les techniques de compilation restent les mêmes : les analyses statiques sont identiques et le code généré est équivalent. La différence réside principalement dans la façon dont le code est généré (cf. section 8.2). Le tableau 8.1 montre que les différences entre le mode global et le mode séparé sont négligeables.

L'intérêt du mode global est double :

- il aide à déboguer le compilateur puisque le processus de compilation est plus simple (une seule phase de génération de code, cf section 8.2) ;
- il permet de compiler des programmes PRM sur des plates-formes qui ne sont pas supportées par le mode séparé (cf. section suivante). De plus, comme le mode global et le mode séparé génèrent des exécutables équivalents, le mode global pourrait

Programme	Séparée (s)	Globale `-g` (s)	Gain (%)
`bench_421`	5,54	5,5	0
`bench_netsim`	4,44	4,41	0
`bench_string_append`	8,29	8,61	-3
`shootout_binarytrees`	0,75	0,75	0
`bench_random_n_sort`	4,41	4,55	-3
`bench_complex_sort`	4,48	4,51	0

TAB. 8.1: Impact de la compilation en modes séparé et global

être utilisé pour estimer l'efficacité des différentes techniques d'implémentation sur les plates-formes non supportées.

8.1.3 Langage cible : C

Bien que `prmc` ait pour but de construire des exécutables écrits en langage machine, nous ne générons pas directement de tels fichiers mais nous avons choisi de générer principalement du C, éventuellement complété par de l'assembleur. Un compilateur C s'occupe alors de produire l'exécutable final.

Le choix de passer par un ou plusieurs langages intermédiaires est une pratique courante dans la réalisation de compilateurs. En particulier, le choix du langage C comme intermédiaire lors de la compilation de langage de haut niveau n'est pas original puisque celui-ci a déjà été employé avec succès, par exemple dans le compilateur SCHEME BIGLOO [Serrano, 1994] ou le compilateur EIFFEL SMARTEIFFEL (cf. section 6.4.6).

Avantages

Les avantages d'utiliser le langage C comme langage intermédiaire sont nombreux :

Portabilité Le langage C est un langage normalisé (ANSI puis ISO), il dispose de compilateurs sur presque toutes les architectures et sa durée de vie est sans doute bien plus grande que celle des langages machine.

Efficace Les compilateurs C modernes savent générer un code écrit en langage machine aussi efficace que celui que l'on aurait pu écrire à la main. Ils disposent de tout un attirail de techniques d'optimisation.

Procédural Le langage C possède quelques notions de haut niveau comme les variables, les fonctions ou les structures de contrôles. La compilation vers C d'un langage comme PRM qui possède, entre autres, les mêmes notions en est grandement simplifié.

Interopérabilité C est le langage de choix en ce qui concerne l'interopérabilité : il est facile d'implémenter l'interface avec le monde extérieur puisque d'une part l'accès

aux fonctions système passe par C, et que d'autre part la plupart des langages existants peuvent au moins interopérer avec C.

Développement Le langage C est un langage de programmation à part entière et dispose de nombreux outils de mise au point (vérification, profilage, débuggage). Ces outils peuvent alors être utilisés pour vérifier la qualité du code généré par notre compilateur.

Forme du code produit

Pour le développeur de compilateurs de langages de haut niveau, le langage C peut être vu comme un macro-assembleur. Pour les autres développeurs, C est surtout un langage de programmation, largement utilisé de surcroît.

Lors du développement d'un compilateur générant du code C, on est alors en droit de se poser deux questions :

- Le compilateur doit-il faire l'effort de générer du code lisible par un être humain ?
- Le code C généré doit-il ressembler à du code qui aurait pu être écrit par un être humain ?

Pour la première question, tout développeur de compilateur doit pouvoir déjà répondre à la question : oui, le code généré doit être lisible, et ceci dans le but d'aider le débuggage du compilateur. Cela inclut de choisir des noms explicites (pour les fonctions, types et variables), d'indenter et d'espacer le code, de générer des commentaires les plus précis, mais aussi de définir des entités (macros, types, fonctions) qui, bien que pas strictement nécessaires, facilitent la lecture du code.

La seconde question, correspond à ce que Manuel Serrano [Serrano, 1994] désigne par la génération de *code orthodoxe* (qui ressemble à du code « écrit à la main ») et de *code hétérodoxe* (le contraire). Le code orthodoxe essaye de se calquer sur les structures du langage source : une variable PRM est associée à une variable C, une fonction PRM est associée à une fonction C, une boucle est associée à une boucle, etc. Le code hétérodoxe peut revêtir de nombreuses formes : pas d'utilisation de structure de contrôle (ou simplement des `goto`), détournement de l'utilisation de la pile d'exécution, passage d'arguments via des variables statiques, etc. En général, les compilateurs qui génèrent du code hétérodoxe sont des compilateurs de langages éloignés des langages procéduraux (en particulier les langages fonctionnels comme Scheme ou Ml).

En ce qui nous concerne, nous nous contenterons de générer du code orthodoxe, principalement pour quatre raisons :

- les structures PRM sont proches de celles du C ;
- le code orthodoxe est plus facile à lire (puisqu'il ressemble à du code habituel) ;
- l'utilisation de C « normal » s'interface plus facilement avec les bibliothèques développées pour du C « normal » ;
- les outils manipulant du C sont prévus pour du code ressemblant à du code écrit à la main. C'est le cas des outils d'aide au développement comme les débuggueurs

et profileurs. C'est aussi le cas des compilateurs. En effet, les compilateurs se concentrent principalement sur les schémas classiques, c'est-à-dire ceux qui sont utilisés par le plus grand nombre de programmes. Tout code généré a donc plus de chance d'être compilé efficacement s'il ressemble à du code écrit à la main.

Limites

Malgré tout, dans notre optique de compilation, le langage C possède quelques inconvénients :

- le langage C n'intègre pas de mécanisme de contrôle complexe comme les exceptions[2] ;
- la détection et le traitement des erreurs arithmétiques (division par zéro, débordement des entiers) sont inexistants ;
- l'utilisation de symboles, autres que ceux représentant des adresses de fonctions et de variables globales, est impossible. Cette limitation complique la mise en œuvre du schéma de compilation décrit dans le chapitre précédent.

Ainsi, l'implémentation du schéma de compilation séparée que nous avons présenté au chapitre précédent nous oblige à générer des fichiers écrits en langage d'assemblage en plus de ceux écrits en C. Comme le prototype `prmc` a été développé sur une plate-forme GNU/Linux PC, nous nous sommes contentés de générer de l'assembleur compatible avec cette plate-forme.

Toutefois, en mode global le compilateur `prmc` génère uniquement du C standard sans assembleur, c'est pour cette raison que l'on peut considérer que `prmc` est un compilateur relativement portable[3].

8.1.4 Gestion de la mémoire

Ramasse-miettes

Les langages modernes offrent au programmeur la possibilité de se dessaisir de la gestion de la mémoire : les zones de mémoire qui sont allouées mais qui ne sont plus utilisées sont automatiquement libérées. Ce récupérateur automatique de mémoire est appelé *ramasse-miettes*, *glaneur de cellules* ou *garbage collector* (GC).

Le gain pour le développeur est immédiat. Celui-ci peut se consacrer à sa tache de développement et le code source est plus lisible (donc plus maintenable).

Toutefois, la gestion automatique de la mémoire a un coût en terme de performances. Toute la difficulté d'un ramasse-miettes consiste à :

- nettoyer la mémoire inutilisée d'un programme. Toutefois, déterminer si une zone de mémoire peut être libérée est en toute généralité un problème indécidable. Les

[2]Bien que les exceptions n'aient pas encore été spécifiées dans le langage PRM, c'est prévu à terme.
[3]Il suffit alors de disposer d'un interprète RUBY et d'un compilateur C.

différentes heuristiques des ramasse-miettes peuvent être plus ou moins efficaces pour libérer la mémoire inutilisée ;
- ne pas altérer la mémoire utilisée par le programme. Bien que ce point puisse sembler évident, il n'est pas forcément trivial à garantir. D'ailleurs certains ramasse-miettes ne le garantissent même pas [Chailloux, 1992] ;
- ne pas avoir de performances temporelles prohibitives. C'est en partie le cas des ramasse-miettes modernes (dits *générationnels* [Lieberman et Hewitt, 1983; Wilson, 1992]) qui sont actuellement utilisés dans la plate-forme .NET de Microsoft ou la machine virtuelle JAVA de Sun.

Le compilateur `prmc` actuel n'impose pas un ramasse-miettes particulier. Actuellement, le seul ramasse-miettes utilisé est le ramasse-miettes conservatif de Boehm-Demers-Weiser [Boehm, 1993]. Celui-ci a été choisi puisqu'il offre deux avantages importants : il est portable et très simple d'utilisation. Ce ramasse-miettes a pour rôle de remplacer le `malloc` de la `libc`. Il permet d'allouer des zones mémoire aussi simplement que possible sachant que l'utilisation de desallocations explicites n'est pas nécessaire.

La force de ce ramasse-miettes est qu'il ne considère aucun pré-requis, ni sur la structure ou le contenu des zones de mémoire allouées, ni sur la forme des références sur ces zones manipulées par le programme. Toutefois, en tant que développeur de compilateur, il nous est possible de contrôler plus finement la structure des zones allouées et la forme des références. Ainsi, nous pensons que l'utilisation d'un ramasse-miettes spécialement adapté aux techniques de compilation que nous utilisons est plus efficace que l'utilisation d'un ramasse-miettes généraliste. En particulier, Nicolas Desnos [Desnos, 2004] a proposé un ramasse-miettes adapté à la coloration bidirectionnelle des attributs, qui reste à adapter à PRM et à lier au compilateur `prmc`.

Autres politiques

Toutefois, afin de mesurer et comparer les différentes techniques de compilation, il est nécessaire de pouvoir évaluer la part du ramasse-miettes. Afin de cerner son coût, le compilateur `prmc` dispose de deux politiques de gestion de la mémoire qui ne font pas appel à un ramasse-miettes. Dans les deux cas, une fois un bloc alloué, celui-ci ne sera jamais libéré. Dit autrement, la fuite de mémoire (*memory leak*) devient la règle. Bien sûr, ce genre de politique n'a de sens que dans notre étude et n'est pas adaptée pour de véritables applications — à part éventuellement certaines applications particulières dont la durée de vie des processus est très courte ou dont la mémoire est entièrement statique.

La première politique, et la plus aisée à mettre en œuvre, consiste à simplement utiliser la fonction `malloc` pour chaque allocation d'une zone mémoire.

La seconde politique consiste à allouer une grande zone de mémoire (avec un `malloc`) et l'utiliser au fur et à mesure des demandes d'allocation. La mise en œuvre se fait simplement en déplaçant un curseur (un pointeur) dans la grande zone mémoire tout en

Programme	large	stdlib		boehm		
	temps (s)	temps (s)	surcoût (%)	temps (s)	surcoûts (%)	
bench_421	5,54	5,49	0	5,49	0	0
bench_netsim	4,44	5,86	+31	8,84	+99	+50
bench_string_append	8,29	8,31	0	8,92	+7	+7
shootout_binarytrees	0,75	2,02	+169	5,0	+566	+147
bench_random_n_sort	4,41	4,4	0	4,4	0	0
bench_complex_sort	4,48	4,51	0	4,5	0	0

TAB. 8.2: Impact des politiques de gestion de la mémoire (option `--gc`)

considérant que ce qui est d'un côté correspond à la mémoire allouée par le programme et ce qui est de l'autre côté correspond à la mémoire encore libre. Si jamais la grande zone mémoire n'est pas suffisante, il suffit alors d'allouer une nouvelle zone (à nouveau avec un `malloc`) et de l'utiliser à la place de la précédente zone pour les futures allocations de la part du programme.

La fonction C suivante `alloc` correspond au code d'allocation de la seconde politique. Les variables `alloc_pos` et `alloc_size` sont statiques (donc initialisées à 0). Ainsi, c'est la première demande d'allocation effective qui construit la grande zone mémoire (puisque `alloc_size < s` sera vrai).

```
void * alloc(size_t s0)
{
        static char * alloc_pos; /* Cursor */
        static size_t alloc_size; /* "free memory" */
        void * res;
        size_t s = ((s0+3)/4)*4; /* Force alignement on 4B */
        if(alloc_size < s) {
                /* Allocate a new area */
                alloc_size = s + 1024*1024;
                alloc_pos = (char *)malloc(alloc_size);
        }
        res = alloc_pos;
        alloc_pos += s; /* move the cursor */
        alloc_size -= s; /* less free memory */
        return res;
}
```

Mesure des performances

Le tableau 8.2 compare les trois politiques de la gestion de la mémoire.

Lorsque qu'un programme fait peu d'allocations (c'est-à-dire peu de créations d'objets), la variation entre les différentes politiques est nulle, c'est le cas de `bench_421`, `bench_string_append`, `bench_random_n_sort` et `bench_complex_sort`.

Lorsqu'un programme fait beaucoup d'allocations, c'est le cas de `bench_netsim` qui construit beaucoup de petits objets, l'impact est important. Toutefois, il faut noter que les performances des politiques sans ramasse-miettes sont à considérer comme idéales[4] puisqu'elles ne font jamais de désallocation.

`shootout_binarytrees` est un cas particulier puisque c'est avant tout un programme pour tester l'efficacité des ramasse-miettes or ni `large`, ni `stdlib` ne ramassent quoi que ce soit.

8.1.5 Dépendance externe

Jusqu'à présent, nous avons à peine évoqué la problématique de la dépendance externe, principalement parce que ce mécanisme tel qu'il est spécifié et implémenté est temporaire. La nécessité de pouvoir faire des entrées-sorties, donc de pouvoir faire des `printf`, a créé l'organe.

Méthode externe

Ce mécanisme est actuellement mis en œuvre par le mot clé `extern` lors de la définition des méthodes. Les modules de base `io`, `exec`, `maths`, `net` et `sdl` utilisent de telles fonctions externes (cf. annexe A.3.5 page 257).

Par exemple dans le module `math`, on peut lire :

```
class Float
        def cos: Float as extern
        # Cosinus
end
```

Il s'agit bien sûr d'un raffinement, la classe `Float` étant introduite dans le module `kernel`.

Fichier de configuration externe

Lorsqu'une méthode d'un module est déclarée externe, un fichier de configuration externe (`.extern`) doit être disponible.

Un fichier de configuration externe est un fichier texte *clé=valeur*. Les clés actuelles sont :

– `cheader`, entête C utilisé lors de la compilation séparée du module pour satisfaire les déclarations des fonctions C associées aux méthodes externes ;

[4]Modulo les pertes des performances, dues aux défauts de cache et à la gestion de la mémoire virtuelle du système d'exploitation, lorsque la quantité de mémoire allouée par le processus devient très grande.

- `cbody`, fichier de fonctions C utilisées par le module (généralement de fonctions servant de wrappers entre des bibliothèques C et le module écrit en PRM), la phase locale se charge de compiler ces fichiers C, voire de les recompiler si c'est nécessaire ;
- `cflags`, flags `gcc` supplémentaires utilisés lors de la compilation séparée du module, généralement des `-I` pour indiquer les répertoires des fichiers d'entête ;
- `cflags_exec` flags `gcc` supplémentaires extraits du résultat d'une commande shell, par exemple la très commune commande `pkg-config`[5] ;
- `clibs` : bibliothèques C partagées utilisées lors de l'édition de liens des programmes incluant ce module ;
- `clibs_exec` : comme `cflags_exec` mais vis-à-vis de `clibs`.

Par exemple, `math.extern` est le fichier de configuration externe du module `math`. Il contient :

```
cheader=math_prm.h
clibs=-lm
```

8.1.6 Les différentes options

Ce qui fait l'intérêt du compilateur `prmc` actuel est sa polyvalence. Voici la liste des différentes options qui existent :

`-g`, `--global` Compile de façon globale le module passé en argument (cf. section 8.1.2). En l'absence de cette option, la compilation est faite de façon séparée. Cette option est implicite sur les plates-formes qui ne sont pas encore supportées par la compilation séparée.

Cette option est incompatible avec `-c`.

`-c`, `--compileonly` En compilation séparée, n'effectue que la phase locale pour chacun des modules passés en argument (cf. section 8.1.2).

Cette option est incompatible avec `-g`.

`-o`, `--output` *exec* Nomme `exec` l'exécutable généré. Par défaut, l'exécutable porte le nom du premier module de la ligne de commande.

Cette option est incompatible avec `-c`.

`-r`, `--recompile` Ré-analyse les modules et leurs super modules et re-génère tous les fichiers.

`--nocheck` Supprime les vérifications des instructions `check` (cf. section A.4.5 page 276) à l'exécution.

[5] `pkg-config` permet d'extraire des méta-informations de la partie de développement de bibliothèques installées sur le système, comme par exemple les flags gcc (`-I`, `-l`) nécessaires à leur bonne utilisation. Plus d'information sur `http://pkgconfig.freedesktop.org`.

Programme	Sans option	Avec `--nocheck`		Avec `--boost`		
	temps (s)	temps (s)	gain (%)	temps (s)	gains (%)	
`bench_421`	5,54	5,49	0	5,515	0	0
`bench_netsim`	4,44	3,61	18	2,58	41	28
`bench_string_append`	8,29	8,24	0	6,6	20	19
`shootout_binarytrees`	0,75	0,74	1	0,74	1	0
`bench_random_n_sort`	4,41	4,29	2	3,04	31	29
`bench_complex_sort`	4,48	4,43	1	3,37	24	23

TAB. 8.3: Impact des options `--nocheck` et `--boost`

Comme le montre le tableau 8.3, le gain apporté par l'option `--nocheck` est nul pour les programmes utilisant peu les bibliothèques de base mais existe pour les programmes se servant des collections.

`--boost` Supprime les vérifications à l'exécution. Cette option supprime aussi bien les vérifications générées (vérification de l'envoi de message sur `nil` par exemple) que celles du programmeur (`--boost` implique `--nocheck`).

Comme le montre le tableau 8.3, le gain apporté par l'option `--boost` est nul dans le cas de programmes manipulant principalement des types primitifs mais peut être important (entre vingt et trente pourcent) dès que les programmes utilisent beaucoup d'objets. En effet, l'option `--boost` rend superflue de nombreux thunks et les remplace par des appels directs (cf. section 8.3.5), elle désactive également les vérifications lors des accès aux attributs (cf. section 8.3.7).

`-I, --include` *`dir`* Ajoute `dir` à la liste des répertoires dans lesquels chercher les éventuels modules requis. (cf. section 8.2.1)

`--analysis` *`mode`* Sélectionne l'analyse de type à utiliser (cf. section 8.2.4). Les modes disponibles sont deux variantes de RTA : `rta` (par défaut) qui correspond au RTA de base, `rta0` qui correspond à RTA mais qui considère que toutes les classes sont vivantes (en pratique, au niveau des types concrets, le résultat de l'analyse correspond à celui de CHA).

Cette option est incompatible avec `-c`.

`--select` *`mode`* Sélectionne l'implémentation des envois de messages polymorphes à utiliser (cf. section 8.3.5). Les modes disponibles sont : `vft` pour l'implémentation par table de fonctions virtuelles combinée à la coloration des méthodes, `btd` pour l'utilisation d'arbres binaires de recherche équilibrés, `switch` pour l'utilisation d'un peigne (forme dégradée de l'arbre de recherche), `mix` (par défaut) pour l'utilisation de VFT pour les sites d'appel mégamorphes et de BTD pour les sites d'appel oligomorphes.

Cette option est incompatible avec `-c`.

`--select_limit` *`cardinality`* Détermine la limite entre un site d'appel mégamorphe et un site d'appel oligomorphe. La différence se faisant en fonction de la cardinalité du type concret du receveur. Par défaut, cette valeur est 8.

Cette option nécessite l'option `--select mix` et est incompatible avec `-c`.

`--attribute` *`mode`* Sélectionne l'implémentation de l'accès aux attributs à utiliser (cf. section 8.3.7). Les modes disponibles sont : `col` pour la coloration des attributs, `sim` pour la simulation des accesseurs.

`--contrattr` N'implémente jamais les attributs de façon native. L'objectif est de systématiquement permettre une redéfinition contravariante des attributs[6] (cf. section 8.3.7).

`--gc` *`mode`* Sélectionne le ramasse-miettes (ou à défaut la gestion de la mémoire) à utiliser (cf. section 8.1.4). Les modes disponibles sont : `boehm` pour utiliser le ramasse-miettes de Boehm, `stdlib` pour ne pas utiliser de ramasse-miettes, l'allocation de nouveaux objets se fait simplement par la fonction `malloc` de la libc, `large` (par défaut) pour ne pas utiliser de ramasse-miettes, un grand bloc de mémoire est alloué initialement.

Cette option est incompatible avec `-c`.

`--monodirect` Utilise une coloration monodirectionnelle (cf. section 8.2.4).

Cette option est incompatible avec `-c`.

`--inlinetables` Les tables des classes sont déclarées en C plutôt que dans un fichier assembleur séparé (cf. section 8.2.5)..

Comme pour l'option `-g`, cette option est implicite sur certaines plates-formes.

Cette option est incompatible avec `-c`.

`--notag` Ne pas étiqueter les types primitifs mais toujours les mettre dans des boîtes (cf. section 8.3.2).

`--class_identifier` Utiliser des petits entiers pour identifier les classes (cf. section 8.3.1).

`--no_inline_primitive` Ne pas court-circuiter l'appel aux méthodes primitives (cf. section 8.3.5)

`--noselect_sharing` Avec cette option, les thunks ne sont pas partagés : chaque site qui nécessite la création d'un thunk se voit attribuer le sien propre (cf. sections 8.3.5 et 8.3.6). Elle facilite le débuggage dans certains cas. Cette option est incompatible avec `-c`.

`--trace` Affiche de façon systématique chaque pas dans la progression de l'exécution. C'est généralement le dernier recours avant d'utiliser le débuggeur `gdb`.

`--profile` Compte les demandes d'allocations et les mises en boîtes (cf. section 8.3.2).

[6] « contrattr » est un mot valise formé à partir des mots « contravariant » et « attribut ».

`--prmc_check` Ajoute des vérifications systématiques superflues maintenant que le compilateur fonctionne. Ces vérifications permettent par exemple de contrôler que l'adresse de la méthode extraite de la table des méthodes n'était pas nulle avant d'effectuer un branchement.

`--error_code` En cas d'erreur, n'affiche que le code d'erreur plutôt que le message habituel. Cette option est utilisée principalement par les scripts automatiques qui vérifient la non régression du compilateur.

`--log` Génère des fichiers journaux intermédiaires et divers graphes (modules, spécialisation, graphe d'appels, etc.) pendant le processus de compilation. Ceux-ci permettent de contrôler le bon fonctionnement du compilateur.

`--noclean` Ne supprime pas les fichiers temporaires C et assembleur générés pendant le processus de compilation.

`-v, --verbose` Le compilateur fonctionne en mode bavard.

`-h, -?, --help` Affiche l'aide et quitte.

`--version` Affiche le numéro de version et quitte.

8.2 L'architecture de prmc

Avant d'entrer dans les détails des analyses et de la génération de code de `prmc`, nous allons exposer l'architecture du compilateur.

Comme nous l'avons déjà dit, `prmc` est développé en RUBY (historiquement pour sa capacité de prototypage). Les langages RUBY et PRM possèdent de nombreux points communs : modules hiérarchiques, définition incrémentale de classes pour l'un et raffinement de classes pour l'autre, syntaxe proche. Nous avons donc développé `prmc` en RUBY de la façon la plus proche possible de ce que nous aurions pu faire en PRM afin de faciliter le développement du compilateur autogène.

Le prototype du compilateur est composé de 19 modules (cf. figure 8.1) pour environ 160 classes et 11 000 lignes de code :

`tools.rb :` Diverses fonctions outils.

`metamodel.rb :` Implémentation du méta-modèle tel que présenté dans les chapitres 3 et 4.

`int.rb :` Extension du méta-modèle pour la gestion des types primitifs.

`inheritance.rb :` Mécanismes d'importation, de raffinement et d'héritage.

`prm.y :` Lecture des fichiers sources. Ce fichier est destiné à être transformé par l'utilitaire Racc[7] dans le but de produire un ficher source RUBY.

[7]Racc (*Ruby yACC*) est un générateur d'analyseur grammatical (*parser*) pour RUBY. Cf. `http://i.loveruby.net/en/projects/racc/`.

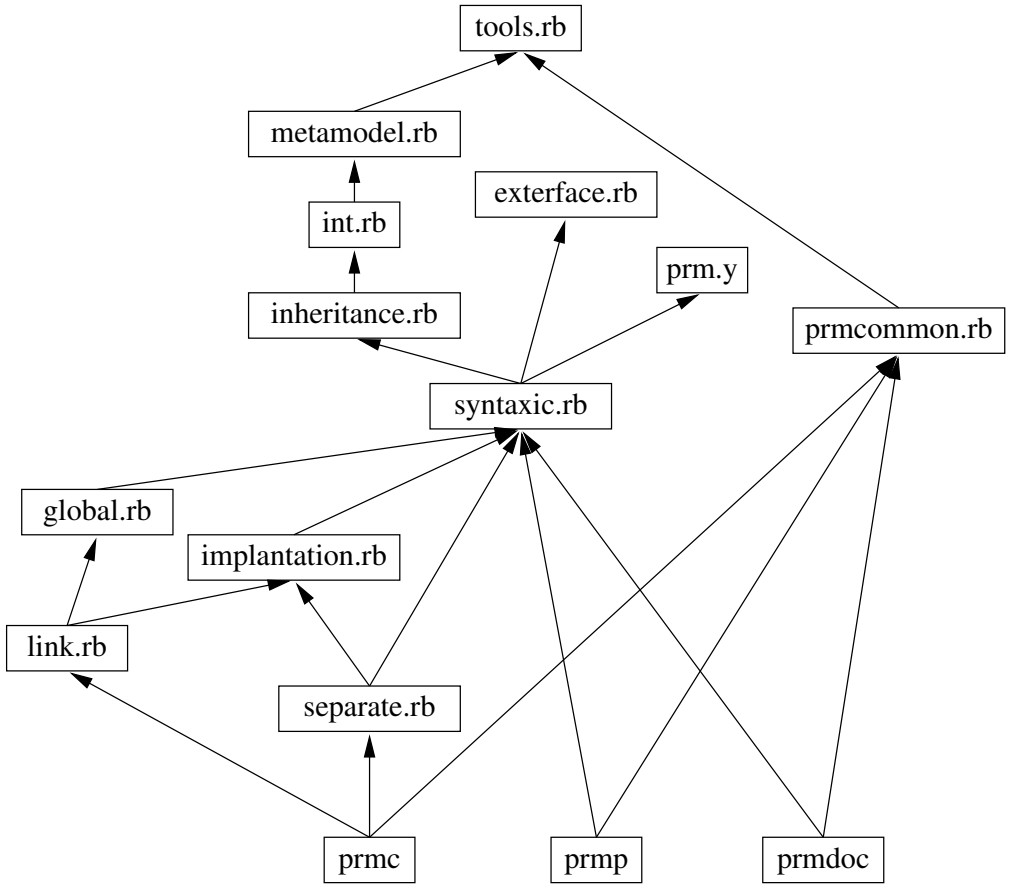


FIG. 8.1: Hiérarchie des modules de `prmc`

`exterface.rb :` Fichiers de configuration externe — « exterface » est un mot valise formé à partir d'« externe » et d'« interface ».

`syntaxic.rb :` Analyse syntaxique et vérification du typage.

`global.rb :` Analyse globale : analyse de type, coloration, etc.

`implantation.rb :` Génération de code en langage cible (langages C et d'assemblage).

`separate.rb :` Génération locale de code en langage cible.

`link.rb :` Génération globale de code en langage cible.

`prmcomon.rb :` Factorisation des modules outils (gestion des options de la ligne de commande, numéro de version, etc.).

`prmc :` Le compilateur.

`prmdoc :` L'extracteur de documentation à partir des sources. Actuellement, il génère de la documentation au format xhtml.

`prmp` : *Pretty printer* pour PRM. Un *pretty printer* est un outil qui reformate un fichier source de façon à ce qu'il respecte une convention donnée, en particulier vis-à-vis des espaces (sauts de ligne, indentation, etc.).

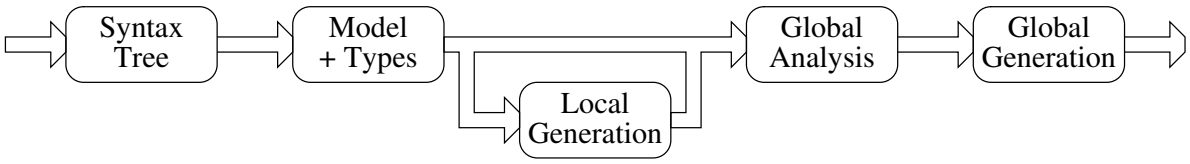


FIG. 8.2: Architecture du compilateur `prmc` : les étapes fondamentales.

La figure 8.2 présente les étapes fondamentales du mécanisme de compilation. Les sections suivantes détaillent chacune de ces étapes fondamentales.

8.2.1 Construction de l'arbre syntaxique

Cette première étape contient trois sous-étapes (figure 8.3). Elle se caractérise par le fait qu'elle travaille de façon indépendante sur les modules : chaque module est traité de façon totalement isolée (même de ses super-modules).

L'objectif de cette étape est de produire l'arbre syntaxique de chaque module. `prmp` ne nécessite que cette étape pour pouvoir fonctionner. Il utilise l'arbre syntaxique final pour réécrire un nouveau fichier source PRM.

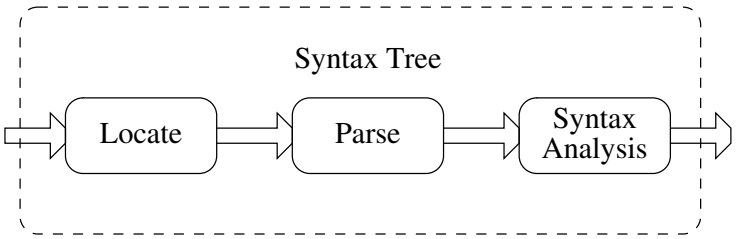


FIG. 8.3: Les étapes de la construction de l'arbre syntaxique.

Locate À moins qu'un chemin vers un fichier ne soit fourni, la première tâche consiste à trouver le fichier correspondant à un module sachant que seul son nom est connu. La recherche a lieu dans le répertoire courant, dans les répertoires systèmes et éventuellement dans les répertoires que l'utilisateur a précisés avec l'option `-I`.

Parse Cette tâche, inévitable quels que soient le compilateur et le langage, consiste à générer l'arbre syntaxique d'un module à partir d'un fichier source. L'arbre ainsi

fourni est brut et correspond exactement au code source. Aucune transformation, même triviale, n'a encore eu lieu.

Syntax Analysis L'analyse syntaxique effectue quelques transformations sur l'arbre, s'occupe des définitions des variables et de leur portée et signale quelques erreurs simples qui ne nécessitent pas une quelconque connaissance des autres modules comme l'absence de `return` dans le corps des fonctions.

8.2.2 Construction du modèle et vérification des types

Cette seconde étape a pour objectif de construire le modèle du module[8], c'est-à-dire identifier l'ensemble des classes (définies, importées et raffinées) et l'ensemble de leur propriétés (définies, redéfinies ou héritées). Elle comporte cinq sous-étapes, figure 8.4, et se caractérise par le fait de ne plus traiter les modules de façon totalement isolée : un module est analysé en regard de ses super-modules.

Le programme `prmdoc`, a seulement besoin d'arriver à la fin de cette étape pour pouvoir générer correctement la documentation.

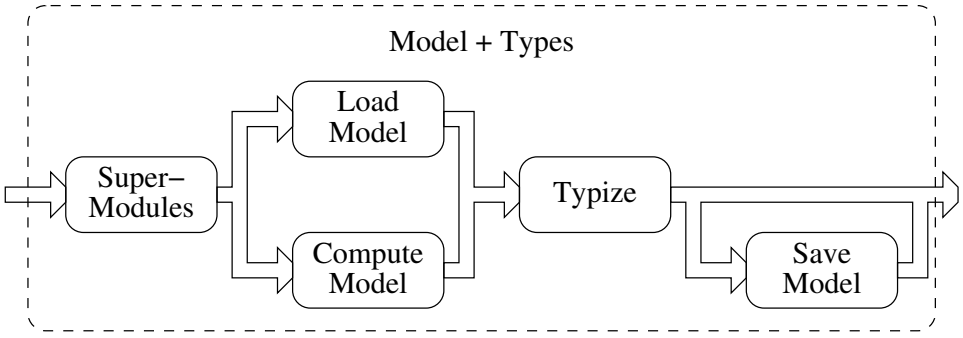


FIG. 8.4: Les étapes de la construction du modèle et de typage.

Supermodules À partir d'ici, les modules ne peuvent plus être considérés de façon isolée. En effet, pour construire le modèle d'un module, le modèle de ses super-modules est nécessaire.

Cette étape consiste à identifier les super-modules ce qui peut récursivement amener à charger et analyser de nouveaux fichiers. Elle vérifie également l'absence de dépendance circulaire entre modules (cf. section 4.3.1 page 86).

Load Model Si le modèle du module est disponible (provenant d'une compilation précédente), et que celui-ci n'est pas obsolète, celui-ci est chargé. Obsolète signifie soit que le fichier source du module ou de l'un de ses super-modules est plus récent que le fichier du modèle, soit que l'option `-r` est utilisée.

[8]Le modèle d'un module est une instance du méta-modèle.

Compute Model Si le modèle du module est indisponible ou obsolète, celui-ci doit être construit.

Le modèle d'un module se construit d'abord en important les classes des super-modules et en distinguant les raffinements des nouvelles définitions. Puis, en faisant l'héritage des propriétés.

C'est à ce moment-là que les différents conflits d'importation et d'héritage sont signalés et que les vérifications de conformité entre propriétés sont faites (cf. section 3.8 page 76).

Typize Maintenant que l'on connaît l'ensemble des classes et leurs propriétés, le corps des méthodes définies dans le module est analysé. Les classes, les types et les propriétés utilisés dans le corps des méthodes peuvent désormais être identifiés.

Cette étape signale également les erreurs de types et autres « messages inconnus » (cf. section 3.8).

Save Model Le modèle est connu et le module ne contient pas d'erreur de types. Son modèle peut être sauvegardé.

8.2.3 Génération locale

Le code destination est généré dans cette étape (figure 8.5). Celle-ci n'est pas systématique et peut être évitée dans deux cas : soit le code destination est déjà disponible et n'est pas obsolète (celui-ci provenant d'une précédente compilation), soit le compilateur fonctionne en mode global (option `-g`).

Avec l'option `-c`, le compilateur s'arrête à la fin de cette étape.

C'est dans cette étape que le langage cible (dans notre cas C) conditionne pour la première fois la compilation.

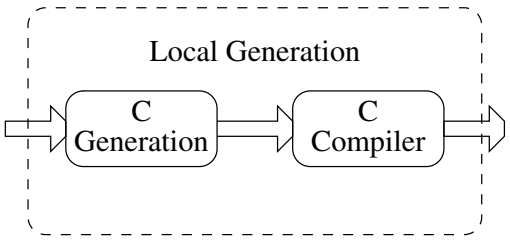


FIG. 8.5: Les étapes de la génération locale du code destination.

C Generation Le fichier C est généré en traduisant chaque méthode déclarée dans le module en C.

Bien que le résultat de cette phase soit indépendant de tout programme utilisant le module, celui-ci est dépendant d'une part des super-modules et d'autre part des options de compilation comme `--boost`, `--attribute` ou `--class_identifier`.

C Compilation Le compilateur C est invoqué.

Bien que celui-ci pourrait sans encombre aller jusqu'à la génération d'un fichier objet en langage machine, nous nous arrêtons à la génération d'un fichier en langage d'assemblage, ce dernier étant plus facile à manipuler par la suite.

Dans le cas où le module possède un fichier de configuration externe, la phase locale est mise à profit pour recompiler automatiquement tout fichier `cbody` si c'est nécessaire (cf. section. 8.1.5).

8.2.4 Analyse globale

L'ensemble du programme à compiler est désormais connu, les techniques globales peuvent s'appliquer. Toutefois, que l'on soit en compilation globale ou non (option `-g`), l'analyse globale reste exactement la même.

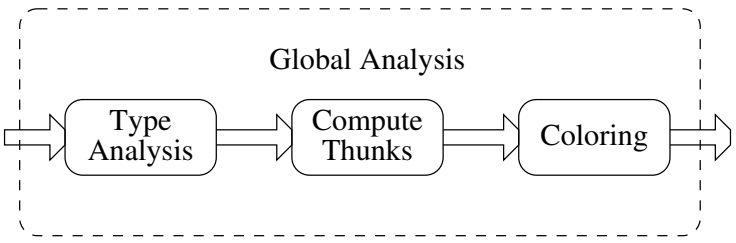


FIG. 8.6: Les étapes de l'analyse globale.

Type Analysis En utilisant les modèles et éventuellement les arbres syntaxiques de chaque module, l'analyse de types calcule les types concrets et le code vivant (modules, classes et propriétés).

Actuellement, seuls sont implémentés RTA et une variante qui de fait correspond plus ou moins à CHA (cf. sections 6.4.2 et 7.4.1). L'ajout d'analyses de types supplémentaires est bien sûr à l'ordre du jour, en particulier, l'ajout de techniques de flots et polyvariantes.

L'option `--analysis` permet de choisir le type d'analyse.

Compute Thunks Pour rappel, les thunks sont les fonctions qui seront utilisées lors des envois de messages et lors des invocations des constructeurs (cf. sections 8.3.5 et 8.3.6). Cette étape a pour objectif de les identifier.

L'option `--select` permet de préciser la nature des thunks souhaités pour les envois de messages polymorphes (cf. sections 8.3.5).

Coloring Une fois le code vivant et les thunks identifiés, une coloration est effectuée. Les couleurs et identifiants sont calculés, le contenu des tables est alors connu. Toutefois, l'heuristique de coloration actuellement implémentée dans `prmc` est terriblement naïve[9] mais semble suffire pour les programmes actuels : aucun trou (ni dans les tables de classes, ni dans les instances) n'est à signaler pour les programmes de benchmark utilisés.

L'option `--monodirect` permet de préciser si la coloration doit être monodirectionnelle. Autrement, celle-ci est bidirectionnelle. Cette option a principalement un intérêt historique lorsque la manipulation de pointeurs qui ne pointaient pas au début des tables était encore hasardeuse. Toutefois, une fois que de vraies heuristiques de coloration et de vrais programmes écrits en PRM seront disponibles, cette option pourra reprendre du service.

8.2.5 Génération globale

La dernière étape a pour rôle de produire l'exécutable.

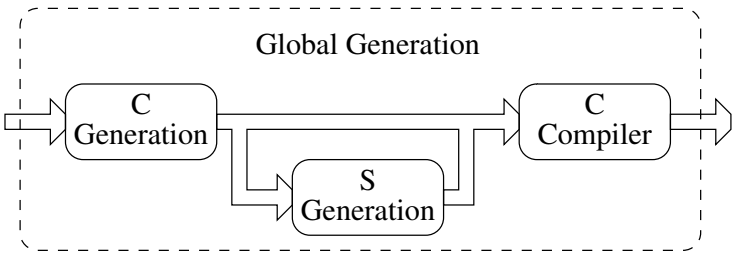


FIG. 8.7: Les étapes de la génération globale de l'exécutable.

C Generation Le fichier généré est ici spécifique au programme compilé. Il contient entre autre le point d'entrée du programme (`main`) et les différents thunks.

En mode global (option `-g`) ce fichier contient en plus la version compilée de l'ensemble des méthodes vivantes du programme.

Avec l'option `--inlinetables`, le fichier contient également la définition des tables des méthodes de chaque classe vivante. Autrement, ces tables sont écrites dans un fichier assembleur spécialement généré pour.

S Generation En mode séparé (sans l'option `-g`), on génère pour chaque module vivant du programme un fichier `.S` dont l'objectif est de faire la substitution des symboles utilisés dans chaque fichier assembleur généré localement.

[9]Elle n'a jamais été réécrite depuis sa première implémentation.

C Compiler Le compilateur C est invoqué sur les fichiers générés (C et assembleurs). Cette fois-ci, la compilation va à son terme (édition de liens procédurale incluse) et l'exécutable tant attendu est généré. La compilation n'oublie pas d'inclure les dépendances supplémentaires si un module du programme possède un fichier de configuration externe.

8.3 Génération de code C

8.3.1 Représentation des objets standards

Comme nous l'avons vu dans le chapitre précédent, la coloration (ou tout autre technique sans sous-objet) nous permet de représenter simplement un objet par une zone mémoire en mémoire dynamique contenant les attributs de l'objet et un pointeur vers la table de la classe qui l'a instancié.

Représentation des classes

Le contenu des tables de classe peut varier en fonction des techniques utilisées :
- identifiant de la classe courante à l'indice 0 (si l'identité des classes est assurée par des petits entiers, cf. section suivante) ;
- pointeurs vers le code des méthodes de la classe (pour les envois de messages qui ne sont pas implémentés par appel direct ou BTD, cf. section 8.3.5) ;
- identifiants de sous-classes (pour les tests de sous-typage) qui peuvent être des petits entiers ou des pointeurs vers des tables de classes (cf. section 8.3.8);
- indice de décalage (pour la simulation d'accesseurs d'attributs, c'est-à-dire avec l'option `--attribute sim`, cf. section 8.3.7).

Afin de regrouper toutes ces variations, nous déclarons les types C suivants :

```
typedef int (*fun_t)(int);
typedef int cid_t;
typedef union { int i; fun_t f; cid_t cid} classtable_elt_t;
typedef classtable_elt_t * classtable_t;
typedef union { classtable_elt_t * vft; } * obj_t;
```

- `fun_t` est un type fonctionnel générique qui nous permet de manipuler de façon uniforme les pointeurs sur fonctions ;
- `cid_t` est le type des identifiants de classes ;
- `classtable_elt_t` correspond à une entrée dans les tables de classes qui peut être soit un entier (indice de décalage), soit un pointeur vers une fonction, soit un identifiant de classe ;
- `classtable_t` correspond à une table de classe ;

Programme	sans (s)	avec (s)	gain (%)
`bench_421`	5,54	5,55	0
`bench_netsim`	4,44	4,45	0
`bench_string_append`	8,29	8,27	0
`shootout_binarytrees`	0,75	0,75	0
`bench_random_n_sort`	4,41	4,38	0
`bench_complex_sort`	4,48	4,45	0

TAB. 8.4: Impact de l'option `--class_identifier`

– `obj_t` correspond à un objet standard. On utilise une union à un seul élément pour pouvoir écrire des choses comme `o->vft` afin de désigner la table de la classe de l'objet `o` (on pose que la table de la classe est au décalage 0).

Identité des classes

En ce qui concerne l'identité des classes, deux choix s'offrent à nous : soit par un petit entier unique stocké dans la table, soit par l'adresse de la table elle-même.

L'inconvénient de l'utilisation d'un petit entier est la nécessité d'une case supplémentaire dans la table des méthodes et d'une indirection supplémentaire pour l'accès à cet entier à partir d'un objet, en contrepartie, l'arithmétique sur de très petits entiers est plus facilement optimisable. L'inconvénient de l'utilisation de l'adresse de la table est que la comparaison d'ordre entre les classes pose un problème de portabilité (du moins telles que les tables sont actuellement générées avec l'option `--inlinetables`).

L'identité des classes est utilisée dans plusieurs mécanismes : test de sous-typage (section 8.3.8), test d'égalité (section 8.3.3) et BTD (section 8.3.5).

Par défaut, c'est l'adresse des tables de classes qui représente l'identité de la classe, toutefois, l'option `--class_identifier` force l'utilisation de petits entiers en tant qu'identifiants de classe. La table 8.4 montre que l'effet de cette option est complètement négligeable.

8.3.2 Représentation des types primitifs

Un *type primitif* est un type nativement manipulé par l'architecture cible. À chaque type primitif on peut faire correspondre un type en C. On peut citer entre autres les petits entiers (`int`), les booléens (`int` aussi), les caractères (`char`) et les flottants (`float`).

Au niveau de la spécification du langage PRM, les données manipulées sont toutes des instances de classes. C'est également le cas pour les valeurs primitives puisque les types primitifs sont considérés comme des classes. Cela permet de manipuler les valeurs

de base comme des objets standards (envois de messages) et autorise le polymorphisme (la classe des petits entiers `Int` est sous-type de la classe racine `Any`).

Toutefois, au niveau de l'implémentation, nous ne pouvons pas nous contenter d'une représentation uniforme.

Le problème n'est pas nouveau. Les langages statiques comme C++ ou JAVA résolvent le problème au niveau de la spécification du langage en dissociant types primitifs et classes. Le prix de cette façon de faire est la perte du polymorphisme ce qui, par exemple en JAVA, oblige le programmeur à passer par des classes mandataires (comme `Integer`) dont le rôle est d'encapsuler une valeur primitive[10]. EIFFEL considère les types primitifs comme des classes expansées (*expanded classes*), une catégorie particulière de classes dont les instances ne sont pas soumis au polymorphisme (le type dynamique et le type statique sont identiques). Dans les deux cas les valeurs primitives sont représentés et manipulées de façon native.

Les langages à typage dynamique (comme LISP ou SMALLTALK) utilisent principalement deux approches pour implémenter le polymorphisme de types: le *boxing* (mise en boîte) et le *tagging* (étiquetage) [Leroy, 1997].

Représentation native

Lorsque le type statique des expressions et des variables est un type primitif, il est possible d'utiliser la représentation native. L'avantage de la représentation native est son efficacité : la représentation est optimale, c'est celle de la machine. Toutefois, comme nous l'avons vu au chapitre 6, avoir une représentation des objets dépendante des types statiques pose des problèmes : des conversions sont nécessaires lors des castings ascendants et descendants.

Mise en boîte (boxing)

Le *boxing* (ou *mise en boîte*) consiste à créer un conteneur pour y stocker une valeur primitive. Le conteneur est composé de deux champs, le premier permettant d'identifier le type dynamique et le second contenant la valeur primitive.

L'avantage de la mise en boîte est d'une part son universalité : tous les types primitifs peuvent avoir leur boîte, quel que soit la taille des valeurs primitives ou le nombre de types primitifs différents à représenter. De plus, si l'on place un pointeur vers une table de classe dans la première cellule, la représentation par boîte est strictement comparable à la représentation des objets standards :

Principe 12 (Boîtes à types primitifs) *En typage statique, les objets standards et les valeurs primitives dans des boîtes se manipulent de la même manière tant qu'on ne se pose pas la question de sortir la valeur primitive de la boîte.*

[10]En JAVA 1.5, ce mécanisme est devenu implicite, ce qui facilite le développement, mais il est toujours présent.

Ainsi, que cela soit pour l'envoi de message ou le test de sous-typage, boîtes et objets standards se comportent de la même façon puisque, au pire, seule la table de la classe entre en jeu. Comme les types primitifs et leurs super-types n'ont pas d'attributs, l'accès aux attributs n'a pas de sens.

Malheureusement, la mise en boîte d'une valeur primitive a un coût élevé puisque chaque mise en boîte requiert l'allocation d'une zone mémoire, ce qui peut rapidement se révéler prohibitif si les mises en boîte sont très nombreuses.

Étiquetage (tagging)

Le *tagging* (ou *étiquetage*) consiste à utiliser les bits de poids faible pour différencier dynamiquement le type des données. L'utilisation de ces bits est possible du moment que les pointeurs sont alignés, ce qui est une pratique de toute façon encouragée par l'architecture des microprocesseurs. Lorsqu'un pointeur est aligné, un certain nombre de bits de poids faible valent alors invariablement 0 et sont donc disponibles pour stocker et distinguer des types supplémentaires. Les autres bits servent alors à distinguer la valeur de la donnée.

Au final, sur des architectures où les pointeurs mesurent et sont alignés sur 32 bits, les deux bits de poids faible sont libres, on peut alors représenter quatre types différents :

- la première sorte, étiquetée 00, correspond aux pointeurs vers des représentations classiques ;
- la seconde sorte, étiquetée 01, correspond aux petits entiers (`Int` en PRM). Comme deux bits sont utilisés pour l'étiquette, la taille maximale des entiers représentables de cette façon est limitée à 30 bits ;
- la troisième sorte, étiquetée 10, correspond aux caractères (`Char` en PRM) ;
- la quatrième et dernière sorte, étiquetée 11, correspond aux booléens (`Bool` en PRM).

Contrairement à la mise en boîte, l'étiquetage ne permet pas de représenter toutes les valeurs primitives puisque le nombre de bits utilisables est limité. De plus, la manipulation de telles valeurs nécessite une implémentation particulière pour les mécanismes objets.

Mise en oeuvre

Afin de manipuler indifféremment les objets étiquetés des objets manipulés par un pointeur (objets normaux et boîtes), nous utilisons un type abstrait universel `val_t` :

```
typedef long int val_t; /* value (everything is a val_t) */
```

La mise en œuvre de l'étiquetage passe par des macros simples :

```
#define TAG(x) ((int)(x) & 3)
#define TAG_Int(x) ((val_t)(((x)<<2)|IntTAG))
```

```
#define UNTAG_Int(x) ((int)(x)>>2)
#define TAG_Char(x) ((val_t)((((int)(x))<<2)|CharTAG))
#define UNTAG_Char(x) ((char)((int)(x)>>2))
#define TAG_Bool(x) ((val_t)(((x)<<2)|BoolTAG))
#define UNTAG_Bool(x) ((int)(x)>>2)

#define ISOBJ(x) (TAG((x)) == OBJTAG)
#define VAL2OBJ(x) ((obj_t)(x))
#define OBJ2VAL(o) ((val_t)(o))
#define VAL2VFT(x) \
        (ISOBJ(x) ? VAL2OBJ(x)->vft : TAG2VFT[TAG(x)])
#define VAL2CID(x) \
        (ISOBJ(x) ? VFT2CID(VAL2OBJ(x)->vft) : TAG2CID(TAG(x)))
```

Le tableau `TAG2VFT` permet de retrouver la table des méthodes à partir d'une étiquette. Par exemple pour l'étiquette 1 on obtient l'adresse de la table de la classe des entiers. Lorsque les identifiants de classes sont des petits entiers (option `--class_identifier`), une petite astuce consiste à s'arranger à pouvoir déterminer directement l'identifiant de la classe en fonction de la valeur de l'étiquette.

Pour chaque type primitif mis en boîte, on construit un type C associé, la mise en boîte passe par une fonction et le déboitage passe par une macro :

```
struct TBOX_Float { classtable_t vft; float val;};
#define UNBOX_Float(x) (((struct TBOX_Float *)(VAL2OBJ(x)))->val)

val_t BOX_Float(float x) {
        struct TBOX_Float * box =
                alloc(sizeof(struct TBOX_Float));
        box->vft = VFT_Float;
        box->val = x;
        return (val_t)box;
}
```

8.3.3 Test d'égalité

Sans type primitif, deux objets ont la même identité si les deux pointeurs sont égaux. Or la mise en boîte casse ce principe puisque deux objets identiques peuvent être mis dans deux boîtes différentes.

Ainsi, le code suivant :

```
let a: Any := 1.1
let b: Any := 1.1
let x := (a == b)
```

ne peut pas être naïvement traduit en :

```
val_t a = BOX_Float(1.1);
val_t b = BOX_Float(1.1);
int x = (a == b);
```

L'égalité d'identité de deux objets `x` et `y` qui ne sont pas statiquement typés par un type primitif[11] est vérifié dans deux cas :

- `x` et `y` référencent la même zone mémoire (égalité des pointeurs). On a donc à faire au même objet : soit le même objet standard, soit la même boîte (donc forcément le même contenu) ;
- `x` et `y` ne référencent pas la même zone mémoire, mais référencent des boîtes de même type primitif avec le même contenu.

Il est donc nécessaire de pouvoir distinguer dynamiquement si l'on affaire à une boîte ou à un objet standard ce qui correspond à savoir si la classe dynamique de l'objet est primitive ou non. L'un des moyens consiste alors à simuler un test de type : le compilateur génère une classe *ad hoc* que spécialise seulement les classes primitives : un objet est donc une boîte si sa classe spécialise cette classe *ad hoc*. Toutefois, un moyen plus efficace consiste à ranger les classes primitives en bout de liste : par exemple en réservant les identités « minimales » pour les classes primitives. Ainsi, nous avons défini les macros suivantes :

```
/* with --class_identifier */
#define OBJ_IS_BOX(x) \
        ((VAL2OBJ(x)->vft->i) <= LAST_PRIMITIVE_ID)
/* without --class_identifier */
#define OBJ_IS_BOX(x) \
        ((VAL2OBJ(x)->vft) <= LAST_PRIMITIVE_VFT)
```

où `LAST_PRIMITIVE_ID` et `LAST_PRIMITIVE_ID` correspondent à la « plus grande » identité d'une classe primitive.

Au final, pour effectuer les comparaisons, nous utilisons les macros suivantes :

```
#define IS_EQUAL_OO(x, y) \
        ((x)==(y) || \
                (ISOBJ(x) \
                && OBJ_IS_BOX(x) \
                && IS_EQUAL_BOX((x), (y))))
#define IS_EQUAL_BOX(x, y) \
        (ISOBJ(y) && \
        (VAL2OBJ(x)->vft==VAL2OBJ(y)->vft)) && \
        (VAL2OBJ(x)[1].vft==VAL2OBJ(y)[1].vft)
```

[11] Du moment que l'un des deux objets est statiquement typé par un type primitif, le test d'égalité est facile puisque la représentation native est utilisée.

En réalité, `IS_EQUAL_*` est une famille de macros, où chaque macro de comparaison est adaptée en fonction de ce que le compilateur a pu déterminer statiquement.

Remarque : Dans la macro `IS_EQUAL_BOX`, nous nous permettons de tester le contenu de la boîte de façon universelle (c'est-à-dire sans tenir compte du type de la valeur) puisque dans l'implémentation actuelle, toutes les boîtes ont la même taille.

Performances

Le fait de ne pas étiqueter les types primitifs tels que les entiers fait généralement exploser la consommation mémoire dans le cas de politique sans ramasse-miettes.

La table 8.5(a) montre le nombre de mises en boîte et la mémoire totale allouée — c'est-à-dire la somme de la quantité de mémoire demandée par chaque allocation. La table 8.5(b) montre l'efficacité temporelle avec ramasse-miettes et la table 8.5(c) montre l'efficacité temporelle sans ramasse-miettes.

8.3.4 Corps des méthodes

En PRM, le code métier est toujours dans le corps des méthodes. Comme nous générons du C orthodoxe, compiler un programme PRM consiste en grande partie à traduire le corps de chaque méthode PRM en une fonction C.

Par exemple, la méthode PRM nommée `regarde` définie dans une classe nommée `Client`, elle-même définie dans un module nommé `galerie`, a le code suivant :

```
def regarde(t: Tableau)
        # Examine le tableau 't'
do
        # Achat coup-de-coeur
        if t.joli and @credit >= t.etiquette.prix then
                achete(t)
        end
end
```

Le résultat de la compilation de cette méthode est :

```
/* Class Client, method regarde(1) */
void I7galerie_6Client_7regarde_1(val_t c, val_t p0)
{
        if ((S1_joli(p0) &&
             (R2_credit(c) >= S4_prix(S3_etiquette(p0))))) {
                S6_achete(c, p0);
        }
}
```

Programme	Sans `--notag`		Avec `--notag`		Différence (%)
	mem. (o)	nbr. boîtes	mem. (o)	nbr. boîtes	mem.
`bench_421`	349	0	357	1	+2
`bench_netsim`	177 257 954	0	523 391 146	43 266 649	+195
`bench_string_append`	260 326 856	0	262 563 848	279 624	+1
`shootout_binarytrees`	203 076 910	0	203 076 918	1	0
`bench_random_n_sort`	8 004 717	0	32 005 397	3 000 085	+300
`bench_complex_sort`	13 877	0	21 885	1001	+58

(a) Consommation mémoire et nombre de mises en boîte

Programme	Sans (s)	Avec (s)	Surcoût (%)
`bench_421`	5,47	5,48	0
`bench_netsim`	8,84	20,95	+136
`bench_string_append`	8,92	8,97	0
`shootout_binarytrees`	5,0	5,02	0
`bench_random_n_sort`	4,4	10,03	+127
`bench_complex_sort`	4,5	4,48	0

(b) Impact de l'option `--notag` (avec `--gc boehm`)

Programme	Sans (s)	Avec (s)	Surcoût (%)
`bench_421`	5,54	5,495	0
`bench_netsim`	4,44	4,92	+10
`bench_string_append`	8,29	8,3	0
`shootout_binarytrees`	0,75	0,75	0
`bench_random_n_sort`	4,41	8,58	+94
`bench_complex_sort`	4,48	4,45	0

(c) Impact de l'option `--notag` (avec `--gc large`)

TAB. 8.5: Impact de l'option `--notag`

Le nom de la méthode a été mutilé afin de garantir l'unicité du nom de la fonction C correspondante : la mutilation inclut le nom du module, celui de la classe et celui de la méthode, de plus les caractères spéciaux comme les opérateurs sont échappés. Le schéma de mutilation utilisé est bien plus simple que celui de C++ puisqu'il n'y a pas d'information de type à intégrer ni d'espaces de noms imbriqués.

La signature de la fonction C est composée du receveur (`c`) suivi des paramètres éventuels de la méthode (`p0`, `p1`, etc.).

Les structures de contrôles (`if`, `while`, blocs, etc.) sont traduites en C de façon littérale.

8.3.5 Envoi de message

Phase locale

Selon le schéma de compilation du chapitre 7, les envois de messages sont directement traduits par des appels de fonctions C.

En compilation séparée et pendant la phase locale, on associe à chaque site d'appel un symbole unique. Ces symboles sont représentés par des fonctions considérées externes et pré-déclarées au début du fichier pour être acceptées par le compilateur C.

Dans l'exemple de la section précédente, le site d'appel `t.etiquette` est compilé par l'appel de fonction `S3_etiquette(p0)` dont la signature C est `val_t S3_etiquette(val_t)`. Ici le `S` indique qu'il s'agit d'un envoi de message (*send*), le numéro a pour rôle de garantir l'unicité dans le module et le reste est là pour faire joli et faciliter la lecture du code C par un être humain, et *a fortiori*, par un être humain déboguant le compilateur et devant analyser à la main les fichiers générés.

Méthodes primitives

Nous avons déjà évoqué les types primitifs qui peuvent être implémentés nativement. De la même façon, de nombreuses opérations sur ces types primitifs peuvent également être implémentées nativement. Nous appelons *méthode primitive* (ou *méthode interne*) les méthodes des types primitifs qui correspondent à ces opérations.

En PRM, la définition de ces méthodes se distingue par le mot clé `intern` et par l'absence de corps. Bien sûr, la totalité des méthodes primitives sont définies dans les modules de base du langage (cf. annexe A.3.5 page 257). Ainsi, dans la définition de la classe `Int` du module `kernel`, on peut lire :

```
class Int
        def +(o: Int): Int as intern
        # Addition
end
```

Dans la spécification du langage, les méthodes primitives ne peuvent pas être redéfinies[12] (que cela soit dans une sous-classe ou dans un raffinement). Il nous est donc possible d'implémenter de façon native les envois de message vers les méthodes primitives. Dans l'exemple de la section 8.3.4, c'est le cas de l'opérateur de comparaison `>=` qui a été directement traduit en C.

Remarque : Le fait de disposer d'un mot clé `intern` désignant à la fois une propriété primitive et non redéfinissable (puisque primitive), permet au langage de ne pas avoir besoin d'une directive de compilation comme `frozen` ou `final` (cf. chapitre 5).

Programme	Sites d'appel primitif		Option `--no_inline_primitive`		
	nbre.	%	avec (s)	sans (s)	surcoût (%)
`bench_421`	97	46	5,54	33,335	501
`bench_netsim`	116	36	4,44	5,74	29
`bench_string_append`	130	46	8,29	21,05	153
`shootout_binarytrees`	110	43	0,75	1,0	33
`bench_random_n_sort`	153	45	4,41	9,02	104
`bench_complex_sort`	79	45	4,48	8,06	79

TAB. 8.6: Impact temporel de l'implémentation native des méthodes primitives.

Le tableau 8.6 montre pour chaque programme de test :
- le nombre de sites d'appel vivants qui correspondent à une méthode primitive (et le pourcentage que cela représente par rapport à tous les sites d'appel vivants) ;
- le temps d'exécution avec l'implémentation native des méthodes primitives ;
- le temps d'exécution sans l'implémentation native des méthodes primitives (option `--no_inline_primitive`) ;
- le surcoût que cela représente.

Pour conclure sur les types primitifs, le fait d'implémenter les valeurs primitives et les fonctions primitives de façon native permet à `prmc` de compiler des programmes PRM n'utilisant quasiment que des types primitifs de façon optimale, c'est-à-dire de façon équivalente au même programme écrit en C et compilé par un compilateur C. Ainsi, nous avons réécrit le programme `bench_421` en C et obtenons des temps identiques entre la version PRM compilée par `prmc` et la version C compilée par `gcc`.

Phase globale

La phase globale substitue chaque symbole d'un site d'appel par l'adresse d'un thunk qui doit sélectionner la méthode à invoquer. Les thunks sont identifiés et générés pendant la phase globale.

[12]En PRM, toute méthode non primitive peut être redéfinie, que celle-ci appartienne à des classes primitives ou non.

Il existe plusieurs sortes de thunks :

Wrapper. Lorsque l'analyse de type détermine qu'un site d'appel invoque de façon monomorphe une unique méthode, le thunk est un wrapper, c'est-à-dire est une fonction qui fait un branchement vers la fonction C d'une méthode PRM. Le rôle d'un wrapper est de faire des vérifications et des conversions : vérifier que le receveur n'est pas `nil`, tester le type dynamique des paramètres (dans le cas de covariance), tester le type dynamique du résultat (en cas de contravariance) et éventuellement les convertir (dans le cas des types primitifs). De plus, dans l'éventualité de l'ajout de contrats (à la EIFFEL) à la spécification du langage PRM, ce serait aux wrappers de vérifier si ceux-ci sont respectés.

L'option de compilation `--boost` supprime la vérification que le receveur n'est pas `nil` et la vérification des types dynamiques. Bien évidement, l'option `--boost` ne supprime pas la conversion des types primitifs.

Appel direct. Lorsqu'aucune vérification ni aucune conversion n'est nécessaire, un appel direct est utilisé à la place d'un wrapper : le symbole associé au site d'appel est substitué par l'adresse de la fonction C résultat de la compilation de la méthode PRM invoquée.

Sélectionneur. Un sélectionneur a pour rôle de faire la sélection de la méthode dans le cas de sites d'appels polymorphes. Le rôle du sélectionneur est uniquement de déterminer la méthode appelée en fonction du type dynamique du receveur.

Il existe actuellement trois sortes de sélectionneurs mais dans tous les cas, ils se contentent de faire un branchement soit vers la fonction C de la méthode PRM sélectionnée, soit vers un wrapper qui s'occupera de faire des vérifications et des conversions si c'est nécessaire.

Sélectionneur VFT. La sélection de la méthode se fait par un accès à la table des méthodes. Si des vérifications ou des conversions sont nécessaires, c'est l'adresse d'un wrapper qui est dans la table des méthodes — on retrouve l'utilisation des thunks tels que nous les avons rencontrés la première fois dans la section 6.2.3 page 119.

Sélectionneur BTD. La sélection de la méthode se fait par un arbre binaire de sélection. Les classes des receveurs sont ordonnées par l'adresse de leur pointeur (ou par leur identifiant si l'option `--class_identifier` est présente).

Sélectionneur peigne. Il s'agit d'un BTD non équilibré (un simple `switch/case` en C). Ce sélectionneur n'existe que pour le débuggage lorsque pour une raison quelconque les deux types de sélectionneurs précédents sont cassés.

La table 8.7 compare les différentes implémentations des envois de messages polymorphes, le surcoût est comparé par rapport à `mix`. A notre grande surprise, l'impact sur nos programmes de tests des différentes techniques est majoritairement nul : en effet, après étude, la plupart des envois de messages sont détectés comme monomorphes

Programme	`mix`	`vft`		`btd`		`switch`	
	temps (s)	temps (s) & surcoût (%)		temps (s) & surcoût (%)		temps (s) & surcoût (%)	
`bench_421`	5,54	5,5	0	5,5	0	5,53	0
`bench_netsim`	4,44	4,46	0	4,4	0	4,45	0
`bench_string_append`	8,29	8,26	0	8,3	0	8,29	0
`shootout_binarytrees`	0,75	0,75	0	0,75	0	0,75	0
`bench_random_n_sort`	4,41	4,43	0	4,42	0	4,49	1
`bench_complex_sort`	4,48	5,7	27	4,46	0	4,48	0

TAB. 8.7: Impact de l'implémentation des envois de message polymorphes (option `--select`)

(donc implémentés par un appel direct). La seule exception est `bench_complex_sort` qui regorge d'appels oligomorphes (donc peu efficacement implémentés par les VFT).

Toutefois, à la lumière de ces résultats, deux options non exclusives sont possibles :
- nos programmes de tests sont trop simples pour illustrer de façon correcte les différences entre les techniques d'implémentation ;
- dans les *vrais* programmes de la *vraie* vie, les techniques d'implémentation des envois de message polymorphes n'ont qu'un faible impact sur l'efficacité générale du programme[13].

Afin d'éclaircir ce point, il nous faudrait disposer de *vrais* programmes écrits en PRM. Nous attendons donc avec impatience la première version du compilateur autogène.

Conversion de types primitifs

La conversion des paramètres a lieu lors des changements de représentation des types primitifs — (un)tagging et (un)boxing, cf. section 8.3.2. Ainsi, soit le programme PRM suivant :

```
class A
        def foo(a: Any): Any
        do
                return a
        end
end

class B
inherit A
```

[13]Dans la section 8.4.2, l'impact de ces techniques d'implémentation est clairement identifiée, toutefois, sur de *faux* programmes.

```
        def foo(i: Int): Int
        do
                return i
        end
constructor
        def init do end
end

let a: A
# ...
a := new B
# ...
let i := a.foo(5)
```

La phase locale générera le code C suivant :

```
val_t I4test_1A_3foo_1(val_t c, val_t p0)
{
        return p0;
}
int I4test_1B_3foo_1(val_t c, int p0)
{
        return p0;
}
void I4test_3Sys_4init_0(val_t c)
{
        val_t v_1; /* var 'a' : A */
        val_t v_2; /* var 'i' : Any */
        /* ... */
        v_2 = S2_foo(v_1, TAG_Int(5));
}
```

La phase globale (avec l'option `--boost`), produira le wrapper `W2_foo` suivant qu'il substituera à l'envoi de message `S2_foo`.

```
val_t W2_foo(val_t c, val_t p0) {
        return TAG_Int(I4test_1B_3foo_1(c,UNTAG_Int(p0)));
}
```

Mode global

En mode global (option `-g`), le code C est généré après l'analyse globale du programme.

De cette façon, lors de la génération du code des méthodes, plutôt que d'utiliser un symbole au niveau des sites d'appel, on peut directement utiliser l'adresse d'un thunk ou de la méthode invoquée.

8.3.6 Création d'instance

Les créations d'instances, c'est-à-dire les `new Toto` mais aussi les objets littéraux des chaînes de caractères des tableaux, sont implémentées de façon comparable aux méthodes.

Pendant la phase locale

La phase locale, on associe à chaque site d'instanciation un symbole unique. Par exemple, le code PRM suivant :

```
x := new String.with_capacity(5)
```

sera compilé en:

```
v2 = N9_String_with_capacity(5);
```

où `N9_String_with_capacity` est une fonction déclarée avec une signature C `val_t N9_String_with_capacity(int)`. Ici le `N` indique qu'il s'agit d'une instanciation (*new*), le numéro a pour rôle de garantir l'unicité dans le module et le reste est là pour faire joli.

Les méthodes qui ont un rôle de constructeur sont compilées de la même façon que les autres méthodes : il n'y a aucune particularité lors de la phase locale.

L'allocateur d'objet

Lors de la phase globale, chaque classe vivante se voit attribuer un `allocateur d'objet`[14] dont le rôle est d'allouer une zone mémoire et de la remplir avec les valeurs par défaut (par un `memcpy` brutal) :

```
val_t NEW_String(void) {
        obj_t obj;
        obj = (obj_t)alloc(sizeof(PAT_String));
        memcpy(obj, &PAT_String, sizeof(PAT_String));
        obj = (obj + 1);
        return OBJ2VAL(obj);
}
```

[14] Afin de distinguer les deux étapes de la création d'instance, l'*allocateur* (`allocate-instance` en CLOS) s'occupe de réserver et de préparer une zone mémoire (la location du terrain) et le *constructeur* (`initialize-instance` en CLOS) s'occupe de remplir la zone mémoire et d'effectuer les traitements idoines (la construction de la maison).

Dans le listing précédent, `PAT_String` est un patron de chaîne de caractères qui contient les valeurs par défaut des attributs et un pointeur vers la table de la classe dans la bonne case. De plus dans cet exemple, la phase globale a assigné à la classe `String` un attribut d'indice négatif : une instance de cette classe doit être représentée non pas par une référence pointant vers le début de la zone mémoire mais par un pointeur situé au milieu. La ligne `obj = (obj + 1);` effectue le décalage nécessaire (+1 dans ce cas là).

Résolution des symboles

Lors de la phase globale, chaque symbole associé à un site d'instanciation est substitué par l'adresse d'un thunk particulier, appelé *thunk d'instanciation*, chargé :

1. d'appeler l'allocateur et récupérer un objet ;
2. d'appeler la méthode constructeur sur cet objet ;
3. de retourner cet objet.

Par exemple, un tel thunk peut ressembler à :

```
val_t C8_String_with_capacity(int p0) {
        val_t c = NEW_String();
        I6string_6String_13with_capacity_1(c,p0);
        return c;
}
```

Toutefois, comme les constructeurs peuvent être raffinés et leur signature changer, les thunks d'instanciation peuvent être amenés à appeler un wrapper à la place du constructeur.

Dans le cas où le constructeur est une méthode vide sans paramètre (ce qui est souvent le cas des constructeurs `init`), aucun thunk n'est construit et le symbole est directement remplacé par l'adresse de l'allocateur d'objet.

8.3.7 Accès aux attributs

L'accès aux attributs nécessite l'utilisation d'un symbole représentant un indice : soit l'indice de l'attribut dans l'instance (pour l'implémentation par accès direct), soit l'indice de l'accesseur simulé dans la table des méthodes (pour l'implémentation par simulateur d'accesseur).

Malheureusement, C ne nous permet de manipuler que des symboles d'adresses : adresse de fonction ou adresse de variable statique. C'est pourquoi nous implémentons nos symboles d'attributs en les faisant passer pour des symboles d'adresses.

Contrairement aux méthodes, on utilise pour les attributs un symbole par propriété globale, et non pas un symbole par site d'appel. De plus, afin d'augmenter la lisibilité du code C produit, deux macro-définitions par propriété globale sont générées : une pour

la lecture, l'autre pour l'écriture. Ce sont ces deux macros qui sont utilisées dans les fonctions C correspondant au corps des méthodes PRM.

Ainsi, dans l'exemple de la section 8.3.4, l'accès en lecture à l'attribut `@credit` est compilé en `R2_credit(c)` où `c` est le receveur, c'est-à-dire l'objet à qui appartient l'attribut accédé. Ici, le `R` indique qu'il s'agit d'un accès en lecture (*Read*), le numéro est là pour l'unicité dans le module et le reste pour la lisibilité.

Pour l'attribut `@credit` de notre exemple, les déclarations générées sont les suivantes :

```
/* Attribute introduced in galerie::Client::@credit */
extern void A2_credit; /* Symbol placebo */
#define ATTR_INDEX_2_credit ((int)&A2_credit)
#define R2_credit(c) \
        (*(int*)ATTR_ADDR(c, ATTR_INDEX_2_credit))
#define W2_credit(c, v) \
        (*(int*)ATTR_ADDR(c, ATTR_INDEX_2_credit) = (v))
```

- `A2_credit` désigne une variable globale externe mais qui n'a pas d'existence réelle. En effet, lors de la phase globale, nous substituerons son adresse avec l'indice calculé ;
- `ATTR_INDEX_2_credit` permet de manipuler l'indice ;
- `R2_credit` et `W2_credit` sont les macros d'accès. Elles fonctionnent en deux étapes : la première étape récupère l'adresse de la valeur dans l'instance (macro `ATTR_ADDR`). La seconde étape extrait ou assigne la valeur à cette adresse ce qui implique une coercition vers le type C de l'attribut et un déréférencement (ici le type est `int` puisque l'attribut `@prix` est de type primitif `Int`).

Il ne reste qu'à montrer le code de la macro `ATTR_ADDR`. Celui-ci dépend de l'implémentation choisie des attributs (option de compilation `--attribute`) :

```
/* Acces direct */
#define ATTR_ADDR(c, index) \
        (((char*)(OBJ(c))) + (index))
/* Ou simulation d'accesseurs */
#define ATTR_ADDR(c, index) \
        (((char*)(OBJ(c))) + OBJ(c)->vft[index].i)
```

Vérification de types et conversion

L'accès aux attributs pose des difficultés dans le cas de politiques de typages non sûres, c'est-à-dire lorsque le type d'un attribut peut être redéfini (cf. section 6.2.3 page 120).

Soit le listing suivant :

```
class A
```

```
        def @a: T
        def toto(x: T): T
                let r := @a
                @a := x
                return r
        end
end
class B
inherit A
        def @a: U
        def tata(y: U): Bool
                if @a == y then
                        return true
                else
                        @a := y
                        return false
                end
        end
end
```

Nous considérons que la classe `B` est définie dans un sous-module du module de définition de classe `A` : donc, lors de la phase locale la méthode, `foo` est compilée sans savoir si l'attribut sera redéfini ou non. Sans perte de généralité, nous considérons également que la définition de la classe `B` pourrait être un raffinement de la classe `A`.

On peut alors distinguer 4 cas :

1. `U` est un sous-type de `T` et ni `T`, ni `U` ne sont des types primitifs. Les accès en écriture générés pendant la phase locale doivent préventivement vérifier qu'un objet affecté à un attribut est bien conforme au type statique de l'attribut tel qu'il est déclaré dans la classe du receveur. Cette vérification de sous-typage est dynamique puisque la classe du receveur est son type dynamique.

   La solution esquissée dans le chapitre 6 consiste à stoker le type statique de chaque attribut dans la table de la classe et d'utiliser cette donnée pour effectuer un test de sous-typage entre le type stocké de l'attribut et le type dynamique de l'objet affecté.

   Malheureusement, nous n'avons pas encore implémenté une telle vérification dans `prmc`, nous espérons combler cette lacune rapidement.

2. `U` est un sous-type de `T` et `U` est un type primitif. En plus de la vérification de sous-typage telle que décrite précédemment, il est nécessaire de faire une conversion de représentation de type `U` vers le type `T` : c'est-à-dire un casting ascendant et en l'occurrence un *boxing* ou un *tagging*. En effet, bien que le type statique de l'attribut soit un type primitif, celui-ci est potentiellement manipulé comme un objet standard. Par exemple, la méthode `toto` du listing précédent manipule

l'attribut `@a` comme étant un `val_t`. C'est pourquoi, nous considérons que le type C des attributs est celui associé au type statique de l'attribut qui introduit la propriété globale.

Ainsi, dans la méthode `tata` du listing précédent, l'instruction `@a := y` sera compilée en quelque chose qui ressemble à :

```
W5_a(c, BOX_U(p0));
```

3. `U` est un super-type de `T` et ni `T`, ni `U` ne sont des types primitifs.

   Une vérification de type est donc nécessaire lors des accès en lecture. Toutefois, contrairement au premier cas, il suffit de vérifier que le type dynamique de l'attribut est bien sous-type du type statique de l'expression. Le mécanisme habituel de vérification de type statique suffit.

4. `U` est un super-type de `T` et `T` est un type primitif (et forcément `U` n'en est pas un). Par défaut, ce cas n'est pas implémenté dans `prmc` et toute tentative de redéfinition d'un attribut introduit avec un type primitif par un type non primitif est refusée par le compilateur. En effet, le compilateur implémente les types primitifs et les méthodes primitives de façon directe ce qui est incompatible avec un tel raffinement.

   Il existe au moins deux solutions à ce problème :
   - la première solution consiste à toujours implémenter les attributs de façon non native (c'est-à-dire en utilisant le type statique C `val_t`). Cette solution est mise en œuvre par l'option `--contrattr` ;
   - la seconde solution consiste à utiliser systématiquement de vrais accesseurs, c'est-à-dire des thunks générés lors de la phase globale dont le rôle est d'effectuer les vérifications et conversions seulement nécessaires.

En plus de toutes ces vérifications de types et conversions, il est nécessaire de vérifier également que le receveur n'est pas `nil`. Toutefois, la nécessité d'une telle vérification est rare puisqu'elle ne concerne en PRM que les attributs exportés (cf. section A.2.5 page 239) dont l'analyse intra-procédurale n'a pas permis de déterminer que le receveur n'est jamais `nil`.

L'option `--boost` permet de supprimer toutes les vérifications (mais ne peut toucher aux conversions)

Performances

Le tableau 8.8 compare l'efficacité temporelle des différentes implémentations des attributs. Le faible surcoût de l'option `--contrattr` est à relativiser par le fait que dans les programmes, la plupart des attributs statiquement typés par un type primitif sont étiquetés et non pas mis en boîte (la mise en boîte est bien plus coûteuse car elle requiert un appel de fonction et une allocation).

Programme	`col` (coloration)	`--contrattr` + `col`		`sim` (simulateurs)	
	temps (s)	temps (s)	surcoût (%)	temps (s)	surcoût (%)
`bench_421`	5,54	5,52	0	5,515	0
`bench_netsim`	4,44	4,54	2	4,98	12
`bench_string_append`	8,29	8,97	8	9,89	19
`shootout_binarytrees`	0,75	0,75	0	0,83	10
`bench_random_n_sort`	4,41	4,64	5	4,92	11
`bench_complex_sort`	4,48	4,67	4	4,74	5

TAB. 8.8: Implémentation de l'accès aux attributs (option `--attribute`)

8.3.8 Test de types

Le test de type s'effectue à l'aide de la technique de la coloration (cf. section 6.4.5 page 137).

La famille de macros `ISA(e,c)` permet de tester que l'objet `e` est une instance directe ou indirecte de la classe `c` :

```
#define VALISA(e, c) \
        (VAL2VFT(e)[COLOR_ ## c].cid == (cid_t) ID_ ## c)
#define OBJISA(e, c) \
        ((e)->vft[COLOR_ ## c].cid == (cid_t) ID_ ## c)
```

Pour chaque classe locale `X` d'un module, un symbole représente la couleur (`COLOR_X`), c'est-à-dire l'indice dans la table de la classe, et un symbole représente l'identifiant (`ID_X`). Comme pour les attributs, nous simulons l'existence d'une variable globale pour manipuler les deux symboles.

La première macro (`VALISA`) est utilisée dans le cas général. La seconde macro (`OBJISA`) est utilisée lorsque que `e` est un objet normal ou une boîte.

Il n'y a pas besoin de test de *bord de table* puisque `prmc` implémente les tables de classes de façon contiguë et réserve un espace suffisamment grand rempli de « non identifiants de classes » (c'est-à-dire de valeurs qui ne peuvent pas être confondues avec un identifiant de classe) avant la première table et après la dernière.

On parle de collision lorsque l'on trouve dans une table de classe `U` l'identifiant `ID_T` d'une classe C au décalage `COLOR_T` sans que `U` soit sous-classe de `T`. L'absence de telles collisions est garantie par la technique de la coloration du moment qu'il n'y a pas de confusion possible entre la valeur `COLOR_T` et une autre information stockée dans une table de classes (un pointeur vers une fonction ou un indice de décalage dans le cas des attributs implémentés par simulation d'accesseur). Ainsi, lorsqu'une classe est identifiée par l'adresse de sa table, il n'y a pas de collision possible.

Remarque : Actuellement, le compilateur `prmc` ne permet pas de distinguer deux types génériques distincts d'une même classe paramétrée : comme en JAVA 5.0, les

paramètres formels sont effacés. Il s'agit d'une limitation actuelle importante du compilateur `prmc` mais qui sera, nous l'espérons, bientôt levée.

8.4 Comparaison de trois compilateurs

Cette section tente de comparer les techniques utilisées par `prmc` avec un compilateur purement global et avec un compilateur purement séparé vis-à-vis des trois mécanismes que sont l'envoi de message, l'accès aux attributs et le test de sous-typage.

Nous en profitons également pour comparer trois techniques d'implémentation de l'envoi de message : VFT avec sous-objets, VFT avec coloration et BTD.

8.4.1 Description

Langages et compilateurs

Nous utilisons trois langages et compilateurs différents :

g++ le compilateur C++ de GNU. Il utilise l'implémentation standard de C++ avec des VFT et des sous-objets. En tant que composante de la suite `gcc` (*GNU Compiler Collection*), il génère directement des exécutables écrits en langage machine.

Comme les benchmarks que nous utilisons contiennent de l'héritage multiple et de la liaison tardive, nous utilisons systématiquement le mot clé `virtual`. Ce qui est une pratique qui ne correspond pas forcément à celle des programmes C++ que l'on peut trouver dans la nature. Toutefois, notre objectif est de comparer l'implémentation par sous-objets avec les autres techniques.

SmartEiffel est le compilateur Eiffel de GNU (cf. section 6.4.6 page 140). Pour rappel, il utilise une analyse de type rapide (une variante de RTA) et implémente les mécanismes objets principalement avec des BTD. Il compile les programmes Eiffel en C puis invoque un compilateur C pour terminer la génération des exécutables.

`prmc` est notre compilateur pour PRM. Comme SmartEiffel il compile les programmes en C puis invoque un compilateur C.

Pour SmartEiffel et `prmc`, nous utilisons le compilateur C `gcc`. Ainsi, les trois compilateurs utilisent le même *back-end* pour la génération du code machine. Nous utilisons le protocole de test décrit dans la section 8.1.1 à la variation près que nous générons, compilons et exécutons plusieurs programmes différents pour une même mesure. La suite `gcc` est la version 4.0.3, de plus nous utilisons les deux options `-O2` and `-fomit-frame-pointer`. La version de SmartEiffel utilisée est la 1.1 (toutefois la version 2.2 semble donner les mêmes résultats) avec les options `-boost` et `-no_gc`. `prmc` est utilisé avec les options `--boost` et `--gc large`.

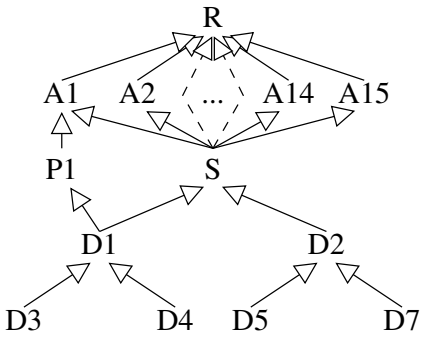


FIG. 8.8: Hiérarchie de classes utilisée dans les programmes de benchmark

Les programmes

De façon à garantir la pertinence des résultats, les programmes que nous allons utiliser pour comparer les implémentations des mécanismes objets doivent satisfaire les contraintes suivantes :

- le « même » programme doit être disponible pour chaque langage ;
- chaque programme doit se concentrer sur un seul des mécanismes objet ;
- un programme doit dépendre le moins possible de la bibliothèque standard du langage : nous souhaitons comparer l'efficacité de l'implémentation, et non pas l'efficacité des bibliothèques standards, de plus, utiliser trop de dépendances contreviendrait au point précédent.

À l'aide d'un programme de script nous générons des petits programmes équivalents pour chaque langage. Ces petits programmes sont basés sur une répétition d'action du même type et sur des receveurs de différents types statiques et dynamiques.

Chaque petit programme est divisé en trois parties : la définition des classes, la séquence d'initialisation et une boucle :

Génération des classes. De façon à éviter l'apparition d'artefacts, quatre hiérarchies de classes isomorphes sont générées.

Une hiérarchie, figure 8.8, est composée :

- d'une racine (`R`) ;
- de 15 sous-classes directes, les fournisseurs d'attributs (`A1` à `A15`) ;
- d'une sous-classe commune, le type statique manipulé (`S`) ;
- de sous-classes de `S` structurées sous forme d'un arbre binaire, leur nombre dépend du paramétrage du script (`D1`, `D2`, etc.) ;
- de classes cousines de `S`, c'est dire sans lien direct mais avec super-classes et sous-classes communes (`P1`, `P2`, etc.). En effet, avec une probabilité de 20 %, chaque classe `D` peut être sous-classe indirecte d'une classe `A` par l'intermédiaire d'une classe `P` — l'objectif est de faire un peu d'héritage multiple.

Chaque classe d'une hiérarchie introduit un nouvel attribut et définit (`R`) ou redéfinit (les autres classes) des méthodes.

Initialisation. Pour chaque hiérarchie, on déclare un tableau[15] dont les éléments sont statiquement typés par la classe `S` de la hiérarchie.

Le tableau est initialisé avec des instances de classes `D` choisies aléatoirement par le script, toutefois, les séquences d'objets sont identiques pour chaque langage.

Boucle. Une double boucle répète des actions sur chaque élément du tableau. Le code ainsi généré ressemble au listing suivant écrit dans un pseudo-langage :

```
for i from 0 to many do
        for j from 0 to arraylength-1 do
                action1 on array1[j]
                action2 on array2[j]
                action3 on array3[j]
                action4 on array4[j]
        end for
end for
```

En fonction du paramétrage du script, les actions sont soit des invocations de méthodes, soit des accès à des attributs soit des tentative de castings descendants. Nous avons choisi de prendre des tableaux de 400 éléments (`arraylength`=400). Bien sûr, nous nous arrangeons pour que la valeur `many` soit suffisamment grande de sorte que la durée d'une exécution soit suffisamment grande pour être pertinente et que la durée de l'initialisation soit négligeable comparé à la durée totale de l'exécution.

Pour chaque test, nous faisons varier le nombre de classes `D` des hiérarchies, ce qui a pour effet de faire varier le cardinal du type concret des expressions `arrayx[j]` et donc le degré de polymorphisme des mécanismes objet testés.

8.4.2 Résutats

Taille de l'exécutable

En premier lieu, nous nous intéressons à la taille des exécutables produits par la compilation. La figure 8.9 illustre la taille des exécutables générés en fonction du nombre de classes `D` de chaque hiérarchie — les courbes de SmartEiffel et `prmc` sont pratiquement confondues.

[15] `g++`, SmartEiffel et `prmc` permettent de manipuler des tableaux primitifs, c'est à dire une zone mémoire préalablement allouée. Nous ne pouvons pas utiliser les collections de haut niveau fournies par les langages (par exemple la classe `Vector` de STL de C++) puisque leur implémentation n'est pas équivalente pour chaque langage.

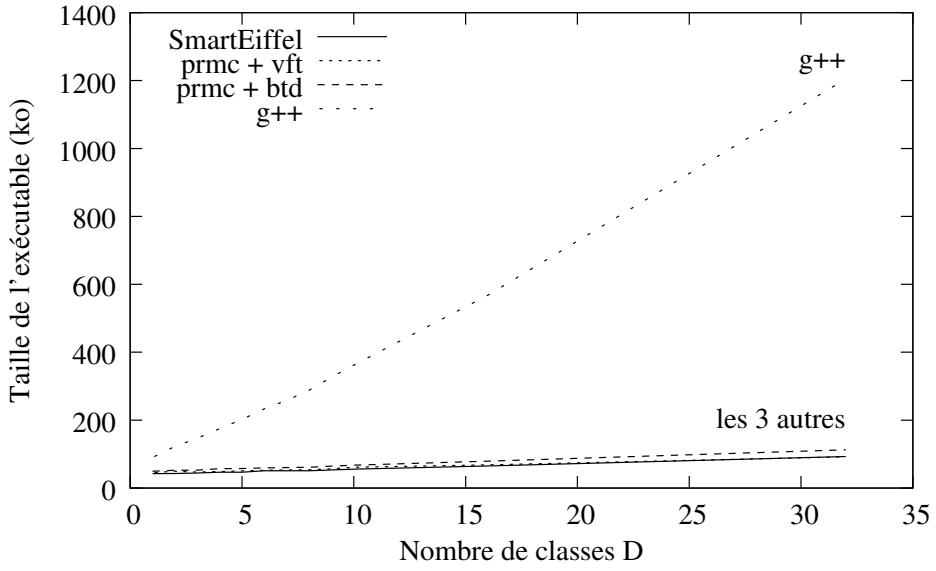


FIG. 8.9: Taille de l'exécutable *strippé*

Comme prévu (cf. chapitre 6), le compilateur C++ a généré de gros fichiers binaires : l'implémentation par sous-objets fait que les VFT occupent dans l'exécutable une place cubique (dans le pire des cas, ce qui n'est toutefois pas le cas ici).

Les arbres binaires et les VFT en coloration occupent une place bien moins importante (quadratique dans le pire des cas) comme le montrent les deux courbes de `prmc`.

SMARTEIFFEL, qui utilise également des arbres binaires, génère des exécutables de taille équivalente à ceux de `prmc`. Toutefois, du point de vue de la taille du code, les deux grosses différences entre SMARTEIFFEL et `prmc` ne s'appliquent pas dans ce benchmark :

- la particularisation de SMARTEIFFEL ne s'applique par car il n'y a rien à adapter : les méthodes sont toutes redéfinies dans les sous-classes ;
- la suppression du code mort, bien que plus fine en SMARTEIFFEL, ne s'applique pas non plus car les programmes tels qu'ils sont générés ne possèdent pas beaucoup de code mort.

Envoi de message

Le second benchmark, figure 8.10, illustre l'efficacité des envois de message en fonction de la taille du type concret de receveurs.

Pour les sites d'appel monomorphes, SMARTEIFFEL et `prmc` utilisent un appel direct. C++ utilise un accès à la table des méthodes mais le processeur semble le gérer efficacement.

De façon conforme aux études précédentes, l'implémentation par BTD est plus ef-

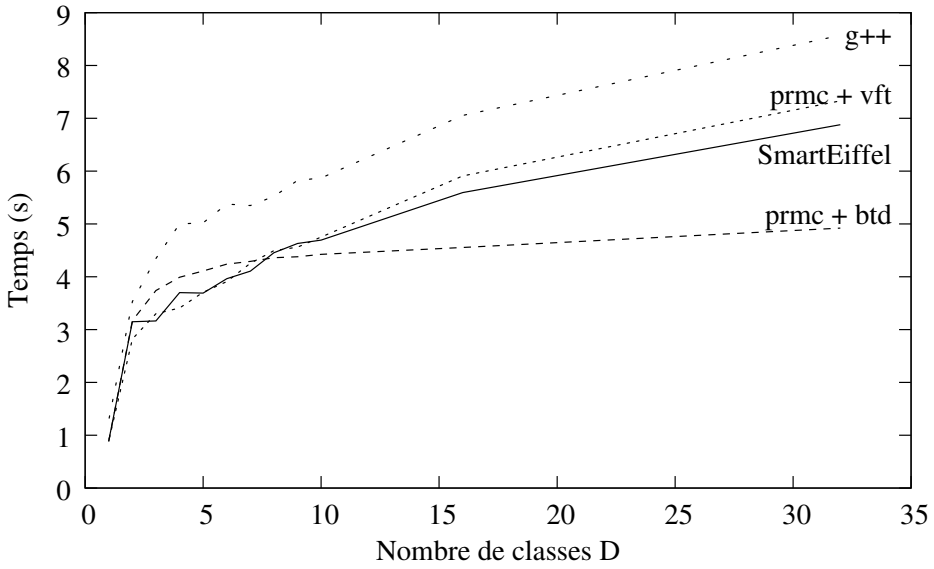


FIG. 8.10: 150 000 000 envois de messages

ficace que l'implémentation par VFT dans le cas de sites d'appel oligomorphes (grâce, vraisemblablement, aux prédictions du processeur) tandis que pour les sites d'appel mégamorphes, le rapport s'inverse (sans doute à cause de la complexité qui est en $\mathcal{O}(1)$ pour les VFT et en $\mathcal{O}(\log(n))$ pour les BTD). Ces résultats sont conformes à ceux de [Zendra et Driesen, 2002].

On peut également remarquer que SMARTEIFFEL et `prmc` + BTD ont des performances comparables, puisque, comme attendu, le schéma de compilation séparée que nous proposons n'introduit pas de surcoût.

Le surcoût de C++ est sans doute dû en partie à l'implémentation par sous-objets et en partie à des défauts de cache (les VFT de C++ occupent beaucoup de mémoire ce qui augmente le risque de défaut de cache). On observe également un léger impact des défauts de cache dans le cas de `prmc` + VFT.

Accès aux attributs

Pour la comparaison de l'accès aux attributs, figure 8.11, la coloration donne les meilleurs résultats à `prmc`, toutefois, ces résultats sont à mettre en perspective avec l'existence de trous dans les instances (en particulier dans les classes `A`).

Pour C++, le besoin systématique d'accès aux sous-objets rend l'accès aux attributs moins efficace.

Le cas de SMARTEIFFEL requiert une explication : lorsque les classes du type concret de receveur stockent les attributs aux mêmes décalages, l'accès aux attributs est un appel direct (comme avec la coloration). Dans le cas où l'indice d'un attribut varie en

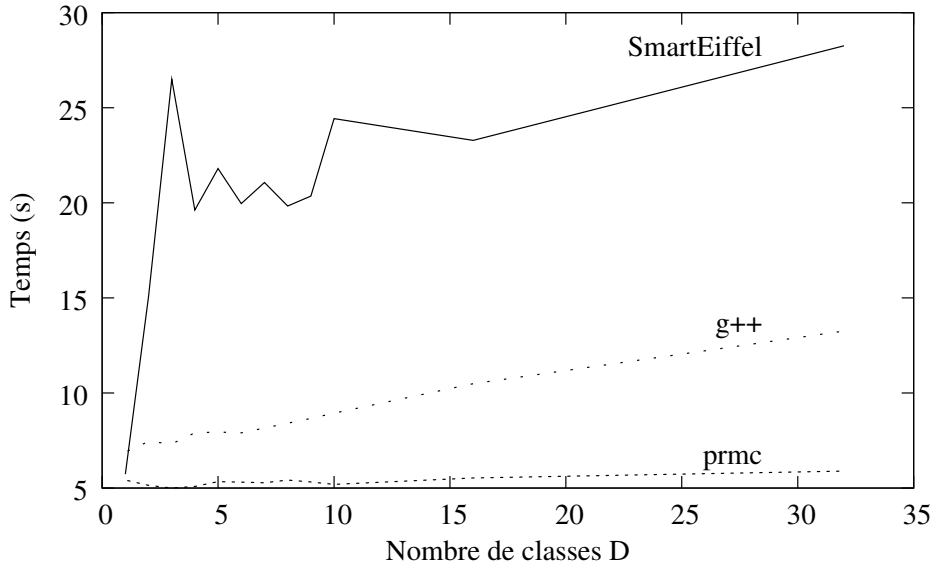


FIG. 8.11: 3 200 000 000 accès d'attributs

fonction des classes du type concret, SMARTEIFFEL implémente l'accès à cet attribut à l'aide d'un thunk et énumère les classes de chaque type du type concret à l'aide d'un arbre de sélection. Ainsi, l'accès à de tels attributs provoque un surcoût : branchement systématique vers le code du thunk (un appel de fonction) et coût de l'arbre de sélection qui est important lorsque le receveur est mégamorphe. Toutefois, dans les hiérarchies complexes, SMARTEIFFEL semble avoir du mal à garantir que l'indice des attributs soit invariant.

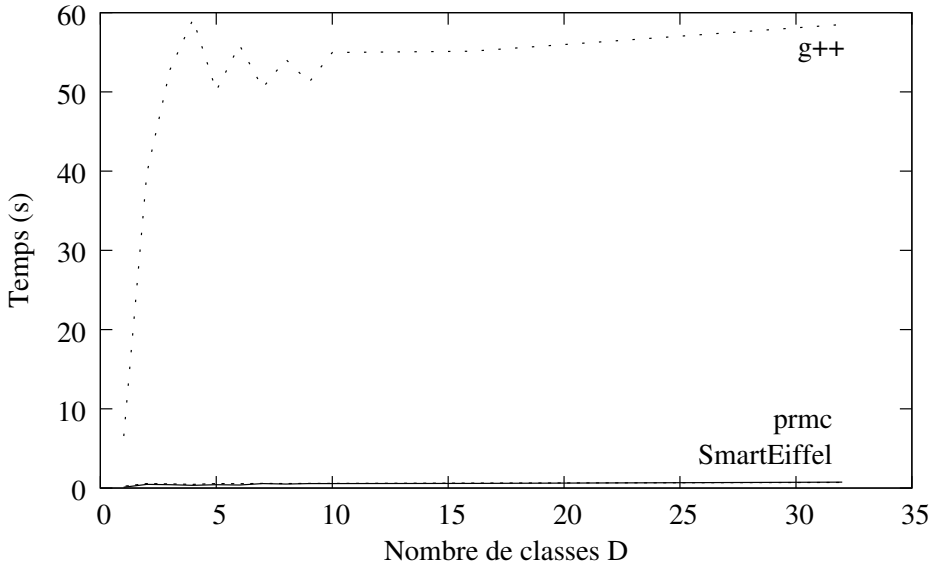
Remarque : Dans [Privat et Ducournau, 2005a], le surcoût de SMARTEIFFEL pour les attributs est important mais relativement moins que celui présenté en figure 8.11. La raison en est que dans le benchmark de l'époque (dont celui-ci n'est qu'une adaptation), `S` a beaucoup moins de super-classes et de nombreux attributs se sont retrouvés morts à notre insu.

Casting descendant

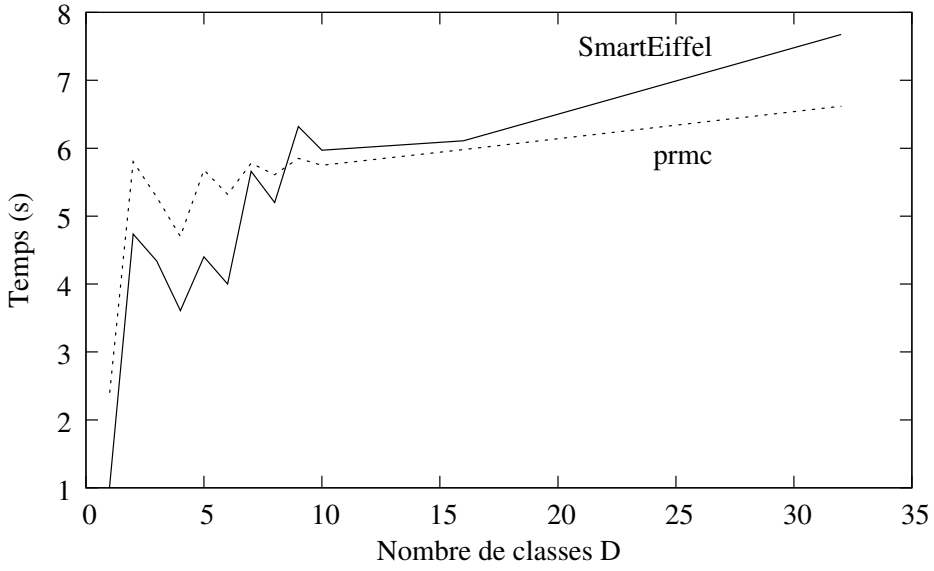
Dans ce dernier benchmark, nous testons le casting descendant ce qui se traduit au niveau des programmes par l'affectation d'un objet de type statique `S` à une variable d'un sous-type `D1` (cf. figure 8.8). Telle que la hiérarchie est construite, le casting descendant a en moyenne un peu plus de 50 % chance de réussir.

Nous utilisons l'opérateur `dynamic_cast` de C++, et l'instruction `?=` d'EIFFEL. Pour PRM, nous utilisons le `isa` pour effectuer la vérification de type suivie d'un `?=` pour affecter la variable (cf. annexe A).

Au niveau des résultats, figure 8.12(b), il apparaît que le coût du `dynamix_cast` de



(a) 50 000 000 castings descendants



(b) 500 000 000 castings descendants (sans C++)

FIG. 8.12: Casting descendant

`g++` est prohibitif (même s'il y a utilisation d'un appel de fonction, ceci n'explique pas tout).

SMARTEIFFEL et `prmc` ont tous les deux de bons résultats, même si SMARTEIFFEL est meilleur pour les cas oligomorphes et `prmc` meilleur pour les cas mégamorphes. Toutefois, contrairement à l'envoi de message, la différence est faible.

8.5 Conclusion

Le compilateur `prmc` que nous avons présenté dans ce chapitre permet de traduire des programmes écrits en PRM en des exécutables efficaces puisque `prmc` intègre des techniques d'implémentation efficaces présentées dans le chapitre 6. Il respecte également l'approche modulaire du langage PRM puisqu'il implémente le schéma de compilation présenté dans le chapitre 7.

Différents tests ont illustré que malgré le schéma de compilation séparée, `prmc` génère des exécutables performants, parfois même plus performants que le compilateur global SmartEiffel [16]

Actuellement, le compilateur `prmc`, bien qu'incomplet vis-à-vis de la spécification du langage (cf. annexe A), est suffisant pour compiler de nombreux programmes écrits en PRM mais également expérimenter et comparer de nombreuses techniques de compilation. La seule limitation est pour l'instant le très faible nombre de programmes écrits en PRM.

Une autre limitation de `prmc` est une gestion incomplète de la vérification dynamique du sous-typage :

- absence de l'implémentation de la vérification de type lors des redéfinitions covariante des attributs ;
- généricité implémentée de façon homogène : les types formels sont effacés et rien ne distingue dynamiquement une instance de `Array[Int]` d'une instance de `Array[Any]`.

Ainsi, un programme non sûr au niveau des types (puisque la covariance des attributs fait partie de la politique de typage non sûre suivie par PRM) peut être compilée de façon non robuste (c'est-à-dire de façon « non sûre » selon la définition ancienne du terme, cf. section 2.6.1 page 23). De plus, l'implémentation homogène de la généricité a également un impact vis-à-vis de la performance dans le cas des types génériques paramétrés par des types primitifs.

Parmi les nombreuses perspectives du compilateur `prmc` lui-même, il y a bien sûr la levée de ces limitations. Il y a aussi l'amélioration des techniques d'implémentation existantes et l'ajout de techniques supplémentaires :

- au niveau des attributs : accesseurs simulés par classes et non pas par attributs ainsi que véritables accesseurs (c'est-à-dire des thunks générés lors de la phase globale) ;
- au niveau de la coloration : heuristiques de [Takhedmit, 2003] ;
- au niveau de la gestion mémoire : ramasse-miettes de [Desnos, 2004] ;
- au niveau de l'analyse de types : techniques de flots et polyvariantes en compilation séparée de [Privat, 2002] ;

[16] Toutefois, rien n'interdit l'implémentation dans SmartEiffel des techniques utilisées par `prmc`, comme la coloration par exemple.

Toutefois, ces améliorations seront sans doute implémentées non pas dans le prototype actuel mais plutôt dans le compilateur autogène en cours de développement.

Chapitre 9

Conclusion générale

Préambule

À la fin ce cette thèse, il m'apparaît que ma compréhension du paradigme objet et des mécanismes afférents a bien augmenté depuis mes débuts en DEA ; compréhension que je me suis efforcé de faire partager au lecteur au long de ce mémoire. Cette conclusion présente le travail fait, mais aussi ce qui reste à faire.

Le langage PRM

Dans le chapitre 2, nous avons présenté le langage PRM et nous avons essayé de justifier ses grandes caractéristiques au regard de principes qui permettent d'obtenir des langages de programmation de qualité, et donc des logiciels de qualité. Dans l'annexe A, nous avons présenté la spécification (quasi-)complète du langage.

L'originalité de PRM est sans doute d'être un langage concis et clair (que cela soit au niveau de la syntaxe ou au niveau des concepts) tout en appartenant à la famille des langage à objets statiquement typés. La simplicité d'utilisation et la rigueur apportée par le typage statique permettent au programmeur de se concentrer sur sa tâche (développer) sans avoir à se battre contre le langage (par manque d'expressivité ou lourdeur syntaxique) ni contre ses propres programmes (par manque de structuration).

On peut signaler trois utilisations concrètes du langage PRM :

- l'utilisation en tant que langage de support à l'apprentissage de la programmation et de l'algorithmique à l'IUT de Béziers. La capacité de PRM à « masquer » ses caractéristiques avancées (objets, covariance, raffinement, etc.) facilite grandement l'apprentissage ;
- le développement des modules de base (cf. section A.3.5 page 257) qui nous a servi à montrer l'intérêt du raffinement dans le cadre du développement de bibliothèques ;
- le développement actuel d'un compilateur autogène (c'est-à-dire un compilateur

PRM développé en PRM) dans le cadre du stage de master recherche de Floréal Morandat. Un compilateur est un logiciel complet et complexe qui permet de mettre à rude épreuve les spécifications d'un langage de programmation et la correction du compilateur qui le compile !

Spécialisation et raffinement

Dans les chapitres 3 et 4, nous avons modélisé les notions de modules, de classes et de propriétés ainsi que leurs relations. Nos propositions sont indépendantes des langages à objets mais expriment pleinement leur potentiel dans le cas de langages statiquement typés et en héritage multiple. Ces propositions sont basées sur une sémantique claire qui essaye, dans la mesure du possible, d'être la plus naturelle possible. En particulier, elle se base sur l'approche syllogistique qui guide la programmation et la représentation par objets.

Le futur du langage PRM

Actuellement, la spécification de PRM contient quelques manques importants pour un usage de tous les jours (constantes, variables statiques, énumération, etc.). La principale difficulté est simplement de l'ordre de la spécification : comment ajouter ces mécanismes afin qu'ils ne jurent pas avec le reste du langage ?

Deux autres caractéristiques importantes sont également manquantes : les exceptions et les contrats. L'intégration des exceptions à la spécification de PRM ne semble poser aucun problème particulier : les blocs `do/end` de PRM peuvent facilement être décorés de mots clés supplémentaires comme nous l'avons suggéré dans la section A.4.5 (page 270). De même pour les contrats, le bloc principal d'une méthode peut également accueillir de nouveaux mots clés :

```
class List[E: Any]
        def first: E
        # Get the first element
        require
                not_empty: not is_empty
        do
                ...
        end
end
```

Par contre, la question de la combinaison des contrats apparaît dans le cadre du raffinement de classes : il est possible que la politique d'Eiffel ne soit pas adaptée — pour rappel, en Eiffel, il y a disjonction des pré-conditions (`require` et `require else`) et conjonction des post-conditions (`ensure` et `ensure else`).

Comme nous l'avons vu dans le chapitre 4, le raffinement de classes rend la problématique de l'héritage multiple incontournable, en particulier, le statique appel à `super` tel qu'il existe dans le langage trouve vite sa limitation. L'idéal serait de disposer d'un mécanisme d'invocation plus souple des précurseurs comme CLOS et son `call-next-method`. La question qui se pose est alors celle de la bonne linéarisation, c'est-à-dire celle qui combine spécialisation et héritage de la meilleure façon[1].

Dans un objectif un peu plus lointain, nous pensons intégrer les multi-méthodes ou fonctions surchargées [Mugridge *et al.*, 1991; Castagna, 1997]. Bien que les multi-méthodes soient un changement relativement important en profondeur, en surface, ce changement devrait « presque » être transparent : les programmes PRM tels qu'ils existent actuellement devraient toujours fonctionner. Au niveau de la spécification du langage, cela reviendrait probablement à troquer la variance du type des paramètres par une sélection déterminée par les types dynamiques du receveur et des autres paramètres. Là aussi, la combinaison de raffinement et de spécialisation doit être tout particulièrement étudiée.

Un autre objectif tout aussi lointain concerne la mise en œuvre d'une couche d'introspection comparable au package `lang.reflect` de JAVA, voire par exemple, par une couche de méta-programmation à la compilation comme OpenC++ [Chiba, 1995].

La dernière extension du langage que l'on prévoit découle de la rencontre du raffinement de classes et de la généricité : que signifie raffiner une classe paramétrée si l'on change sa borne ? Nous avons commencé à chercher une réponse à cette question mais il semble qu'elle mérite une réflexion bien plus fine qu'il y paraît au premier abord. Il est même possible que cette question amène à se pencher sur des problèmes inédits. Toutefois, la question ne peut pas être ignorée car le besoin de pouvoir raffiner les classes paramétrées s'est présenté très tôt lors du développement des modules de base c'est pourquoi le compilateur `prmc` inclut déjà une spécification de ce mécanisme et permet par exemple d'écrire le module suivant :

```
class Array[E: Comparable]
# Refinement of the Array class of the standard module
        def maximum: E
                # Return the maximum element
                # of a non-empty array
        do
                check not is_empty
                let max := first
                for e in self do
                        if e > max then
                                max := e
                        end
                end
```

[1] En espérant que cette « meilleure » linéarisation existe.

```
                return e
        end
end
let a := [1, 5, 4, 9, 7]
print(a.maximum) # Output ''9''

let b: Array[Any]
print(b.maximum) # Compile-time error!
# 'maximum' is only for Array[Comparable]
```

Ici, la classe `Array` a été spécialisée par l'introduction d'une nouvelle méthode, `maximum`, avec la particularité que dans cette méthode, le paramètre formel est borné par `Comparable` (donc que `self` est de type statique `Array[Comparable]`), les autres méthodes de la classe demeurent inchangées. En effet, dans le corps de la méthode, l'opérateur $>$ est utilisé). Toutefois, afin d'éviter les erreurs de type, le compilateur doit statiquement vérifier que la méthode `maximum` est effectivement invoquée sur des receveurs statiquement sous-types de `Array[Comparable]`. Dans ce mémoire, nous avons fait l'impasse sur cette spécification car elle s'avère inconsistante dans certains cas complexes. Nous espérons toutefois arriver à un mécanisme qui soit cohérent dans tous les cas de figure.

Pour terminer cette section sur les évolutions du langage PRM, il nous faut bien évidemment signaler qu'il doit aller de pair avec l'enrichissement des modules de base et tout particulièrement de développement de couplages vers des bibliothèques en tous genre : XML, base de donnée, GCI, interface graphique, etc. Ceci va amener la question de la spécification des mécanismes d'externalisation :

- appel C par un programme PRM, ce mécanisme existe déjà mais n'est pas parfait ;
- appel de PRM par un programme C, ce mécanisme n'existe pas du tout.

Compilation séparée et techniques globales

Dans le chapitre 6, nous avons montré la difficulté, voire l'inefficacité, de l'implémentation de l'héritage multiple et l'existence de techniques globales (particularisation, analyse de types, coloration, arbre de sélection, etc.) permettant de réduire cette difficulté — voire de l'annuler complètement. Dans le chapitre 7, nous avons proposé un schéma de compilation séparée qui permet la mise en œuvre de certaines de ces techniques malgré le fait qu'elles soient globales, en effet, elles ont lieu lors de l'édition de liens. Dans le chapitre 8, nous avons proposé `prmc`, un compilateur performant pour la langage PRM qui intègre ce schéma de compilation.

Évolution du schéma de compilation

La principale limitation du schéma de compilation que nous avons proposé est son inadéquation avec le chargement dynamique. Une des perspectives de notre travail

consiste vraisemblablement à traiter ce problème de chargement dynamique. En particulier, une bonne solution serait de pouvoir proposer une machine virtuelle (éventuellement munie d'un compilateur juste-à-temps) qui permette la mise en œuvre d'un maximum de techniques globales non plus à l'édition de liens mais lors de l'exécution même des programmes.

Si l'on en reste au schéma de compilation séparée avec application de techniques globales à l'édition de liens, la perspective majeure est l'amélioration des techniques globales existantes et l'intégration de techniques globales supplémentaires (cf. section 6.4 page 130) :

Particularisation partielle. Même si la particularisation dans sa généralité va à l'encontre du principe même de la compilation séparée, rien n'empêche que la particularisation ne soit appliquée que partiellement, c'est-à-dire entre les classes d'un même module. Cela nécessiterait d'une part de compiler chaque méthode locale de façon spécifique pour les classes connues par le module ; et d'autre part de compiler une version plus générique destinée aux classes inconnues.

Mise en ligne. Lors de la phase locale il est difficile d'imaginer l'application de mise en ligne. Par contre, dans la phase globale, il est possible de mettre des choses en ligne au niveau des thunkss. Ainsi, une solution serait, lors de la phase locale, d'identifier les méthodes potentiellement mettables en ligne (en se basant par exemple sur la liste de [Zendra, 2000]) ; puis, lors de la phase globale, le branchement vers ces méthodes dans le corps d'un thunk sera remplacé par une mise en ligne.

Code mort. Actuellement, le traitement du code mort est loin d'être complet puisque du code détecté comme mort se retrouve tout de même dans l'exécutable final. En effet, du moment qu'une méthode d'un module est vivante, alors le résultat de la compilation de chacune des méthodes du module se retrouvera dans l'exécutable final. Toutefois, pouvoir n'intégrer dans l'exécutable que les méthodes vivantes ne semble pas impossible — au pire, il suffit de compiler chaque méthode dans un fichier C séparé.

Évolution de `prmc`

Si l'on se concentre plus particulièrement sur `prmc`, plusieurs points sont à travailler :

Coloration. L'heuristique naïve de coloration actuellement implémentée dans `prmc` est naïve[2], l'une des premières choses à faire serait d'intégrer celles de [Takhedmit, 2003].

Généricité. Les classes paramétrées sont implémentées de façon totalement homogène ce qui est inefficace dans le cas des types primitifs : un `Array[Int]` est rempli d'objets étiquetés, pire un `Array[Float]` est rempli de boîtes !

[2]Suffisante pour les petits programmes actuels mais insuffisante pour les gros programmes à venir.

De plus les types génériques ne sont pas distingués à l'exécution : comme en JAVA 5.0, le type formel est effacé, ce qui est problématique au niveau des vérifications dynamiques de types puisqu'à l'exécution, rien ne distingue un `Array[Int]` d'un `Array[Any]`. Une solution a été esquissée dans [Ducournau, 2002b].

Ramasse-miettes. Le ramasse-miettes généraliste de Bohem-Demers-Weiser [Boehm, 1993] est celui actuellement utilisé par `prmc`. Toutefois, il serait utile que le ramasse-miettes puisse profiter des techniques spécifiques d'implémentation de notre compilateur. Un objectif serait par exemple d'integrer le ramasse-miettes de [Desnos, 2004].

Le développement en cours par Floréal Morandat d'un nouveau compilateur écrit en PRM va nous permettre de disposer d'un produit utilisable par plus de personnes (le présent prototype en RUBY sera alors simplifié et servira à bootstrapper) mais aussi de pouvoir mesurer le processus de compilation lui-même : durée de compilation, durée de recompilation, durée de la phase globale, etc. Nous espérons disposer de ce compilateur le plus tôt possible. De plus, ce compilateur sera le premier gros programme écrit en PRM. Il permettra donc d'aider à mesurer, en tant que programme à compiler, les performances respectives des différentes techniques de compilation que nous avons considérées.

Annexes

Annexe A

Spécification complète du langage PRM: PRM—The Language

Préambule

Cette annexe est basé sur la documentation de PRM telle qu'elle est distribuée — c'est-à-dire en anglais [Privat, 2006]. Toutefois, nous avons choisi d'inclure cette documentation dans ce mémoire car elle n'est pas seulement une documentation technique du langage PRM :

- *elle justifie les choix du langage ;*
- *elle met PRM en perspective avec d'autres langages de programmation ;*
- *elle discute des évolutions potentielles du langage.*

A.1 A PRM Introduction

PRM stands for "Programming with Refinement and Modules". It is an open-source programming language that has a simple straightforward style and can usually be picked up quickly, particularly by anyone who has programmed before. It is object-oriented but allows a procedural style. `prmc` is a PRM compiler that produces efficient machine language executables.

The PRM website: `http://www.lirmm.fr/~privat/prm`

PRM is a language with a high degree of structure: it is statically typed and it allows programmers to easily produce readable source code. However it has two qualities that are mainly found in dynamically typed languages:

Concise but Clear Syntax. The syntax of the PRM language is clear and simple, without verbosity. A PRM program looks like a program written with a modern scripting language like Ruby or Python—in fact, the syntax mainly comes from

RUBY. The PRM syntax makes difficult to have one day an *Obfuscated PRM Code Contest*.

Small but Powerful Core. Even if the language proposes many features, the number of different core mechanisms of the language is small. It means that the language mechanisms are clean and easy to learn. It also means that in order to provide a clear syntax with a small core, the PRM language makes a great use of syntactic sugar. Many concise pieces of syntax are strictly equivalent to a more verbose one that directly uses the core mechanism. Therefore, the semantic of these small pieces of syntax can be deduced from the semantic of the more verbose one.

A.1.1 Three Simple Examples

Before starting with the full description of the PRM syntax and semantic, here can be found three simple PRM programs. One can have an idea of the light syntax of the language.

Remark: The objective here is not to show the most valuable PRM characteristics but to give an idea of the syntax that will help to understand the full PRM specification.

Hello World

One of the simplest programs:

```
print("Hello ", "World.\n")
```

This program simply outputs the text "Hello World."

Variables and Strings

A program with two local variables:

```
let s: String       # 's' is a local variable
s := "Hello"        # Assignment with a literal strings
s.append(" World") # Concatenate two string
let a := s + "."    # 'a' is another local variable
println(a)          # Output 'a'
```

This program also outputs the text "Hello World."

Subprograms

A program with the definition of a function and a procedure:

```
def square(i: Int): Int
# Return i²
```

```
do
        return i * i
end

def main
do
        println(square(5))
        println(square(-1))
end

main
```

This program outputs the numbers 25 and 1.

A.1.2 The PRM Syntax: A First Impression

A number of points can be seen about PRM clean style and modern syntax:

- → A.4.1: comments Comments are introduced by the `#` symbol and run to the end of line.
- → A.4.1: line structure Semicolons are superfluous—but these may optionally be placed between constructs.
- → A.4.5: statement block Blocks of statements start with the keyword `do` and are terminated by the keyword `end`.
- → A.2.3: procedure and function call Calls use the usual notation `foo(args)`, and without arguments there is no need of parenthesis. Method invocation[1] on objects uses the doted notation `x.foo(args)`.
- → A.2.3: `def` Procedures and functions are defined with the `def` keyword.
- → A.4.5: `let` Local variables are declared with the `let` keyword, and the static type may be inferred.
- → A.4.3: types Types in signatures (`def`) and in local variables definitions (`let`) use the PASCAL column notation "`x: Foo`".
- → A.4.2: names Type names start with an uppercase (`Int`, `String`) while the other names—variables, functions, procedures, etc.—start with a lowercase (`print`, `sqr`, `s`).

[1] In the current documentation, we use the terms "method invocation". In other OO languages, one can find the same idea under the terms "method call" or "send of message".

- → A.4.5: assignment Assignments use the `:=` notation.
- → A.3.6: literals Literals strings are enclosed within double quote.

A.1.3 Outline

The present manual is divided into three parts:

- Section A.2 is about object oriented programming: classes, properties, inheritance, etc.
- Section A.3 is about modules management, class refinement, and the base library (base modules and base classes).
- Section A.4 is about the base language: names, types, expressions, and statements.

A.2 Object-Oriented Programming

PRM is a pure object-oriented language. It means that:

- Each manipulated value is an object.
- Each object is an instance of a class.
- Each subprogram is a method defined in a class.
- Each method invocation corresponds to a message sending—or to a late binding i.e. depends on the dynamic type of the receiver.

→ A.3.4: procedural style However, it is possible to program software in PRM without explicitly defining classes and even subprograms: we call `procedural style` this way of programming. Procedural style is useful for simple programs or for teaching [2]. The current section focuses on "pure" object-oriented programming. The truth about procedural style is explained in a latter section.

A.2.1 Class Definition

A class represents entities, the attributes of those entities and the operations that those entities can perform. Classes can represent real world entities in a model, or more artificial artefacts which occur only in computer programs.

[2] PRM is used to teach algorithmic to students in first year of computer science.

`class`. Classes are defined with the usual `class` keyword. Here is an example of a simple PRM class definition:

```
class Car
end
```

→ A.2.4: object creation A class represents all objects of that type. For instance in the real world we have one concept of `Car`, but there are many instances of `Car`.

A.2.2 A Word about Class Specialisation

Classes are elements of a *specialisation* hierarchy. A class can have superclasses (i.e. class more general) and subclasses (i.e. class more specific).

→ A.3.6: `Any` The root of the hierarchy is a class named `Any`[3]. It means that `Any` is the most general class, and that each other class specialises the `Any` class.

→ A.2.6: class specialisation We talk more about specialisation, and especially inheritance, in a latter section.

A.2.3 Properties: Attributes and Methods

Each class has a set of properties which represents the attributes and the operations of its instances. Operations on an object typically alter its state changing the values of one or more of its attributes. In PRM, such operations are known as *procedures*. Computations that return a value to a query about the state of an object are known as *functions*. Functions and procedures are together known as *methods*.

Definition of Properties

`def`. In a class, the definition of a property starts with the `def` keyword followed by the name of the property→ A.4.2: names .

Attribute Definition. Attributes are identified with the first letter of their name, a `@` character[4]. It can be pronounced *at* and stands for *attribute*. → A.4.3: type Attributes must have a static type.

```
class Car
        def @speed: Int
        # The current speed of the car

        def @color: String
        # The color of the car
```

[3]The `Any` class corresponds to the `Object` class of some languages like JAVA.

[4]The `@` for attributes comes from RUBY.

```
end
```

Method Definition. Methods need a signature and a body. A signature is composed of some parameters (possibly none) and a return type for functions—procedures do not have return types. Bodies are blocks of statements → A.4.5: statement block . A function must return its result with a `return` statement → A.4.5: return .

```
class Foo
        def bar
        # a procedure without parameters
        do
                print("bar")
        end

        def baz: Int
        # a function without parameters
        do
                return 5
        end

        def foobar(i: Int): Int
        # a function with a parameter
        do
                return i + 1
        end

        def foobar(i: Int, j: Int): Int
        # a function with two parameters
        do
                return i + j + 1
        end
end
```

Some Remarks.

- Since attributes start with an `@` character, methods and attributes can therefore share the "same name":

```
class Foo
        def @foo: Int
        def foo: Int
        do
```

```
                ...
        end
end
```

→ A.2.5: accessors This characteristic it often used for attribute accessors.

- Methods can share the same name if they have a different number of parameters. Like ERLANG or to a lesser extent, SMALLTALK or SELF.

  → A.2.3: implicit parameter value This characteristic is used for implicit parameter values.

- There is no static overriding: two methods with the same name and the same number of parameters cannot coexist in a same class, even if they have distinct signatures.

  → A.4.3: covariant typing policy This is because PRM follows a covariant typing policy.

- → A.3.4: procedural style Methods can be defined outside class with the same syntax. Obviously, it is not possible with attributes.

Access to attributes

Objects directly access their attributes by their names:

```
class Foo
        def @bar: Int
        # an attribute

        def baz
        do
                @bar := 5    # write access
                print(@bar) # read access
                # output ``5''
        end
end
```

→ A.4.5: assignment Attribute write accesses follow the assignment rules.

→ A.2.5: exported attribute Except in some special cases, attributes can only be accessed by the objects that own them.

Invocation of Methods

Methods are usually invoked with the common doted notation `x.foo(args)` where `x` is the *receiver*, `foo` the name of a method and `args` some arguments. If there is no

arguments, parentheses are optional—providing superfluous ones may provoke a warning during compilation.

```
class Foo
        def bar
        do
                print("bar")
        end

        def baz: Int
        do
                return 5
        end

        def foobar(a: Int)
        do
                print("foobar", a)
        end

        def foobaz(a: Int): Int
        do
                return a + 1
        end
end
...
let x: Foo
...
x.bar                 # Output ``bar''
print(x.baz)          # Output ``5''
x.foobar(6)           # Output ``foobar6''
print(x.foobar(7))    # Output ``8''
```

`self`, the Current Receiver. If the receiver is the current receiver, called `self` in PRM[5], it can be implicit. Therefore, `self.foo(args)` is equivalent with `foo(args)`.

```
class Foo
        def bar
        do
                print("bar")
        end

        def baz
```

[5]In C++ and JAVA, the current receiver is called `this` ; in EIFFEL it called `Current`.

```
        do
                bar
                self.bar
        end
end
...
let x: Foo
...
x.bar   # Output ``bar''
x.baz   # Output ``barbar''
```

→ A.2.5: visibility It is important to distinct invocation on `self` with other invocations since visibility does not apply on invocation on `self`.

Some methods that have a special name are not invoked with the usual dotted syntax. These methods are *operators* and *assignment procedures*.

Operators

Operators are methods often used for mathematical operations. There are three kinds of operators: infix operators, prefix operators and bracket operators.

Infix Operator. Infix operators are: `+`, `-`, `*`, `/`, `%`, `=`, `!=`, `<`, `>`, `<=`, `>=`, `<=>`, `<<`, and `>>`. They are methods with one parameter and should be defined and invoked as follow:

```
# Definition signature
class C
        def -(o: T): U
        ...
end

# Invocation: x is the receiver and y the argument
let x: C
let y: T
let r: U
r := x - y
```

Prefix Operator. Prefix operators are: `+` and `-`. They are methods without parameters and should be defined and invoked as follow:

```
# Definition signature
class C
        def -: U
        ...
```

```
end

# Invocation: x is the receiver
let x: C
let r: U
r := -x
```

Bracket Operator. Bracket operators are methods mainly used for indexed access (arrays for instance). They are defined and invoked as follow:

```
# Definition signature
class C
        def [](o: T, p: U): V
        # Example with two parameters
        ...
end

# Invocation:
# * x is the receiver, y and z are the arguments
let c: C
let y: T
let z: U
let r: V
r := x[y, z]
```

Remark: Two remarks about operators:

- Invocation of operators requires an explicit receiver.
- → A.3.6: Booleans Some statements looks like operations but are not. For instance the Boolean's pseudo-operators: `and`, `or`, `not`.

Assignment procedures

→ A.4.5: assignment Their names are ended with `:=`. They must have at least one parameter and no return value. They are invoked with a syntax that looks like the assignment statement and follow its rules.

```
# Definition signature
class C
        def foo:=(o: T)
        # Example of the simple form
        # i.e. with one parameter
        ...
```

```
        def foo:=(o: T, p: U, q: V)
        # Example with three parameters
        ...
end

# Invocation
let x: C
let y: T
let z: U
let k: V

# * x is the receiver, y is the argument
x.foo := y

# * x is the receiver, y is the first argument,
#   z the second one, and k the third one
x.foo(y, z) := k
```

Bracket Assignment Procedure. There are also bracket assignment procedures:

```
# Definition signature
# * Example with three parameters
class C
        def []:=(o: T, p: U, q: V)
        ...
end

# Invocation
# * x is the receiver, y is the first argument,
#   z the second one, and k the third one
let x: C
let y: T
let z: U
let k: V
x[y, z] := k
```

Remark: Some remarks about assignment procedures:

- They also exist in RUBY in the simplest form (i.e. with exactly one parameter).
- → A.2.5: accessors With one parameter, they are mainly used to write attributes accessors.

- With more than one parameter, they are mainly used with indexed access when different kinds of indexes exist.
- Assignment procedure is different with the user-defined C++ assignment operator. In C++, "`x.a = 5`" may correspond to the invocation of the `operator=` method on the attribute `a` of the `x` object. In PRM "`x.a := 5`" corresponds to the invocation of the `a:=` method on the `x` object.

Implicit Parameter Value

Remark: The current prmc compiler does not yet implement this part of the specification.

PRM can yield a kind of implicit argument value:

```
class C
        def foo(a: Int, b: Int := 5, c: Int)
        do
                ...
        end
        ...
end
```

However, implicit argument is only syntactic sugar since the last code example is strictly equivalent to:

```
class C
        def foo(a: Int, b: Int, c: Int)
        do
                ...
        end

        def foo(a: Int, c: Int)
        do
                foo(a, 5, c)
        end
end
```

Multiple Implicit Parameter Values. A method can have multiple implicit parameter values:

```
class C
        def foo(a: Int := 5, b: Int := 6)
        do
                ...
```

```
        end
end
```

In order to avoid ambiguities, the first parameters are less implicit that the last parameters. Therefore the two following listings are equivalent:

```
class C
        def foo(a: Int, b: Int)
        do
                ...
        end

        def foo(a: Int)
        do
                foo(a, 6)
        end

        def foo
        do
                foo(5, 6)
        end
end
```

Comparison with Other Languages. Implicit parameter values exist in many other languages like C++, JAVA 5.0 or RUBY. However, their semantic differ in the way that there is one method defined, ant the implicit parameter values are integrated to the arguments when the method is invoked.

For instance, let "`void foo(int a, int b = 5)`" be a C++ method The two expressions `foo(1, 5)` and `foo(1)` invoke this method with the same arguments, since 5 is implicitly added in the last expression.

With PRM and the equivalent `foo` method "`def foo(a: Int, b: Int := 5)`", the two expressions `foo(1, 5)` and `foo(1)` invoke two distinct methods—respectively, `foo` with two parameters and `foo` with one parameter.

The PRM way offers two advantages:

- Implicit parameter values are only sugar syntax: it does not extent the PRM core mechanism.

- → A.2.6: redefinition Each method is independent and can be independently redefined. Thus more flexibility is offered to the programmer.

Variable Argument Number

Remark: This part of the specification is not considered as stable and may change in future version.

In PRM, some methods can be invoked with an unbounded number of arguments.

Definition. In those method definitions, one special parameter is declared as `t: T*` where `t` is the name of the parameter and `T` the type of arguments. In such definitions, the static type of `t` is `Array[T]` → A.3.6: `Array` . Example:

```
class C
        def foo(a: Int, b: Int*, c: Int)
        # The static type of 'b' is 'Array[Int]'
        do
                print(a, "-", b.length, "-", c)
        end
end
```

Invocation. In method invocation, the special parameter is associated with one or more arguments:

```
let c: C
c.foo(1, 2, 3)           # ''Output 1-1-3''
c.foo(1, 2, 3, 4, 5, 6) # ''Output 1-4-6''
c.foo(1, 2)      # compilation error, unknown foo method
                # with two parameters
```

→ A.3.6: `print` Without surprise, the standard `print` method, used in all those examples, accepts multiple arguments. Its signature is `print(a: Any*)`.

Passing Array. One can pass an array object instead of a list of elements with the `*a` notation– here, `*` is not an operator, it is just a notation:

```
let a: Array[Int]
let c: C
...
c.foo(1, *a, 3)
```

Passing array is often used to chain calls:

```
class C
        def printprint(a: Any*)
        # Remember, the static type of 'a' is 'Array[Any]'
        do
```

```
                print(*a)
                print(*a)
        end
end
let c: C
c.printprint("Hello") # Output ``HelloHello''
```

Remark: In a class, cannot coexist:

- Two methods with the same name that both accept a variable number of arguments:

```
class Foo # ERROR !
        def bar(a: Int, b: Int*, c: Int)
        do ... end

        def bar(d: Int*)
        do ... end
end
let f: Foo
f.bar(1, 2, 3) # Ambiguous
```

- Two methods with the same name, one accepts a variable number of arguments, and the other has more parameters than the minimal number of the first:

```
class Foo
        def bar(a: Int, b: Int*)
        do ... end

        def bar(c: Int, d: Int)
        do ... end
end
let f: Foo
f.bar(1, 2) # Ambiguous
```

However, the following listing is OK:

```
class Foo
        def bar(a: Int, b: Int*)
        # bar(2)
        do ... end

        def bar(c: Int)
        # bar(1)
```

```
        do ... end
end
let f: Foo
f.bar            # Not ambiguous, it is an error
f.bar(1)         # Not ambiguous, it is bar(1)
f.bar(1, 2)      # Not ambiguous, it is bar(2)
f.bar(1, 2, 3)   # Not ambiguous, it is bar(2)
```

A.2.4 Object Creation

new. Objects are created with the special `new` statement:

```
new Car
new Car("red")
new Car.with_color("blue")
```

The point to note is *constructors* need to be declared in classes in order to allow them to be instantiated. In PRM, constructors are a little different from those of languages like C++ and JAVA ; EIFFEL constructors are the closest.

Constructor. Constructors are procedures defined in a class after the `constructor` keyword. More than one procedure can be defined as constructors.

In the following listing, the two `init` procedures and the `with_color` one are constructors, but `paint` is a "normal" procedure—note that if a lot of code is duplicated it is only for the need of the illustration:

```
class Car
        def @color: String
        # The color of the car

        def paint(c: String)
        # Repaint the car
        do
                @color := col
        end

constructor
        def init
        do
                @color := "black" # Mr. Ford?
        end

        def init(col: String)
```

```
        do
                @color := col
        end

        def with_color(col: String)
        do
                @color := col
        end
end
```

Remark: The PRM naming convention is to reserve the methods named `init` or `with_something` to be constructor procedures.

→ A.2.5: Visibility It important to notice that constructors look like "normal" procedures. The visibility section will show the truth about the `constructor` keyword and status of constructors.

Implicit Constructor. Implicitly, the constructor named `init` is called on object instantiation. Thus `new Car` is equivalent to `new Car.init`, and `new Car("Blue")` is equivalent to `new Car.init("Blue")`

Remark: Since both object creation and method invocation use a dot in their notation, some cases should be disambiguated:

- `new Foo.bar` is always considered as the instantiation of a `Foo` object with a constructor named `bar`.
- `(new Foo).bar` and `new Foo.init.bar` are the instantiation of a Foo object with a constructor named `init`; and followed by the invocation of a property named `bar` on this newly created object.
- `new Foo(5).bar` and `new Foo.init(5).bar` are the instantiation of a Foo object with a constructor named `init` with 5 as argument; and followed by the invocation of a property named `bar` on this newly created object.

Abstract Classes

Abstract classes are classes that can not be instantiated. Classes that are not abstract are called *concrete classes*.

Remark: In PRM, Abstract classes are simply classes without constructors. In corollary, classes without constructor are abstract, therefore can not be instantiated.

Empty Constructor. In comparison with other languages, there are no default constructors since their use is marginal, even if they are the cause of many errors. However,

sometimes, programmers need to define concrete classes with empty constructors. In PRM they just have to explicitly do it:

```
class Foo
constructor
        def init
        do
        end
end
```

Garbage Collector

PRM has no delete operator. This is because, as many other modern languages, PRM is garbage collected. Garbage collection is known to completely cure the programming ills of dangling pointers and memory leaks. This greatly simplifies the programming effort by removing one of the largest bookkeeping headaches for programmers. Garbage collection has also proved to be very efficient in modern implementations.

A.2.5 Visibility

C++ and JAVA programmers might be wondering how to make methods public, protected and private. With PRM you have far more control: as in EIFFEL, any set of methods can be exported to all, to none or to some specific classes. Thus you have the possibility of many shades of grey between public and private. You might want a method to be public to some specific classes, but private to others. Moreover, method visibility and constructors are related together in a nice original way.

Remark: Visibility is not related to method invocation on `self`. Therefore, properties are always accessible to the current receiver.

Method Visibility Blocks

Visibility is controlled by three keywords that delimit *visibility blocks*: `public`, `private`, and the already known `constructor`. Method defined after such a keyword belong the corresponding visibility block.

A class definition can contain any number of blocks, in any order:

```
class Foo
        ...
public
        ...
public
        ...
private
```

```
        ...
public
        ...
constructor
        ...
private
        ...
end
```

Remark: It is recommended to regroup related methods with the same visibility in the same block. And it is also recommended to put two unrelated sets of methods in two different visibility blocks, even if they share the same visibility.

Public Method Visibility

Methods defined in a `public` block are exported and can be used by other classes. If the name of a class is added after the `public` keyword, methods are only exported to this class and to its subclasses. If there is not such a class name, method are exported to any classes—in fact, they are exported to the `Any` class → A.3.6: `Any` and to its subclasses.

Example:

```
class Car
...
public
        def speed: Int
        # Get the speed of the car
        do
                return @speed
        end

public Driver
        def stop
        # Stop the car
        do
                @speed := 0
        End
...
end
```

Let `c` be a variable statically typed by a `Car`. Here the function `speed` is exported to any class, therefore `c.speed` is valid in any class. The procedure `stop` is exported to the class `Driver` (and all its subclasses), therefore `c.stop` is only valid in the class `Driver` and in any subclasses of `Driver`.

Implicit Visibility Block. The *implicit visibility block* (i.e., the visibility block above the first visibility keyword) is a `public` one. For example, the three following listings are equivalent:

```
class Foo
public Any
        def bar
        do
                print("baz")
        end
end
```

```
class Foo
public
        def bar
        do
                print("baz")
        end
end
```

```
class Foo
        def bar
        do
                print("baz")
        end
end
```

Private Method Visibility

Method defined in a `private` block are not exported. Therefore, private methods are only accessible to the current receiver.

```
class Driver
        def @car: Car
        # The driven car

private
        def stop_car
        # Stop the driven car
        do
                @car.stop
        end
end
```

Let `c` be a variable statically typed by a `Car`. The procedure `stop_car` is exported to nobody, therefore `c.stop_car` is valid nowhere. The only way to invoke such a method is to use the current receiver.

PRM private vs. C++ private. In PRM, private methods are usable only by self—it is an instance visibility. In C++, private methods are usable only by instances of the current classes—it is a class visibility.

The following listing will try to illustrate de difference:

```
class Foo
public Foo
        def bar
        ...
private
        def baz
        ...
...
public
        def test
        do
                bar # OK, the receiver is self
                baz # OK, the receiver is self

                let f: Foo

                f.bar # OK, bar is public Foo
              # and I am Foo

                f.baz # Error, baz is private
                # and the receiver is not self
        end
end
```

Constructor Method Visibility

→ A.2.4: object creation Methods defined in a constructor block are usable as a constructor method. As with the `public` keyword, `constructor` can be used to control the visibility of constructors: if the name of a class is added after the `constructor` keyword, methods are only exported as constructor to this class and to its subclasses. If there is not such a name, the `Any` class is considered.

```
class Car
...
```

```
constructor CarFactory
        def init
        do
                @color := "black"
                @speed := 0
        end
...
end
```

The statement `new Car` is only valid in the class `CarFactory` and in its subclasses.

Remark: Even if some procedures have a status of constructor they can be invoked on the current receiver—constructor status is only a matter of visibility, and is not related with invocation on `self`. Such invocations allow a better factorisation:

```
class Car
        def @color: String

constructor
        def init
        do
                with_color("black")
        end

        def with_color(col: String)
        do
                @color := col
        end
end
```

Remark: Methods can not be both public and constructor since from a user point of view, object creation and send of message correspond to two different needs. Allowing exporting a procedure public and constructor will be a reusability limitation because of class refinement → A.3.3: class refinement . However, code duplication should be avoided:

```
class Car
        def @color: String

        def paint(col: String)
        do
                @color := col
        end

constructor
        def with_color(col: String)
        do
```

```
                paint(@col)
        end
end
```

Attribute Accessor

As in SMALLTALK, attributes are "private": they can only be accessed by the objects that own them. Therefore, some methods should be defined in order to access attributes. Methods that play this role are called *accessors*.

Usually, there is the need of two accessors, one for the read access and one for the write access. In PRM, you can use the same name for the attribute and for the two accessors: the attribute is distinguished with the `@` and the write accessor is usually an assignment procedure → A.2.3: assignment procedure, therefore distinguished with the `:=`.

Example:

```
class Car
        def @speed: Int
        # Attribute

        def speed: Int
        # Read accessor
        do
                return @speed
        end

        def speed:=(s: Int)
        # Write accessor
        do
                @speed := s
        end
...
end
```

In this example, let `c` be a `Car`. `c.speed` returns the value of the attribute `@speed` and `c.speed := 5` assigns 5 to the attribute `@speed`:

```
let c := new Car
c.speed := 5
print(c.speed) # Output ``5''
c.speed := 10
print(c.speed) # Output ``10''
```

Automatic Accessor. The keywords `def_read` and `def_write` can be used to simplify the declaration of such accessors. On attribute definition, `def_read` automatically generates a read accessor and `def_write` automatically generate a write accessor.

The following example is equivalent to the previous one.

```
class Car
        def @speed: Int def_read def_write
...
end
```

Automatic Accessor Visibility. Since `def_read` and `def_write` only correspond to syntax sugar, the visibility of automatic accessors is the one of the current visibility block.

```
class Car
public
        def public_price: Int
        do
                return @cost + @margin
        end

public CarSeller
        def @cost: Int def_read
        def @margin: Int def_read def_write
end
```

In this example, only a car seller can access the real price of a car.

Pseudo-accessor. Accessors are just a role playing by some methods. It is possible to define "pseudo-accessors", i.e. methods that act like accessors from the user point of view. The following example defines two pairs of accessors on the speed attribute of a `Car` class but with different speed units, one in kilometre per hour and the other in miles per hour:

```
Class Car
        def @speed_kmph: Int def_read def_write
        # Speed in kmph

        def speed_mph: Int
        # Speed in mph
        do
                return @speed_kmph * 63 / 100
        end
```

```
        def speed_mph:=(s: Int)
        # Speed in mph
        do
                @speed_kmph := s * 100 / 63
        End
...
end
```

Thus, from a user point of view, it is not possible to distinguish the "true" accessor from the pseudo-accessor:

```
let c := new Car
c.speed_kmph := 80
print(c.speed_kmph, " ", s.speed_mph)
# Output ``80 50''
c.speed_mph := 63
print(c.speed_kmph, " ", s.speed_mph)
# Output ``100 63''
```

Exported Attribute

Remark: The specification is not considered as stable and may change. Moreover, the current `prmc` compiler does not yet implement it.

→ A.2.3: attribute access Software engineering considers that attribute should be accessible only for the current receiver (`self`). However, in some exceptional case, attributes need to be directly accessed by different objects.

`export`. The keyword `export` permits to change the visibility of an attribute. The visibility granted is the one of the current visibility block.

In the following example, the attribute `@baz` is visible in the class `Bar` and in all its subclasses:

```
class Foo
public Bar
        def @baz: Int
        export @baz
end
```

Exported Attribute Access. Exported attributes are accessed with the doted notation `x.@baz` where `x` is the receiver (i.e. the instance that owns the attribute) and `@baz` the name of the attribute.

→ A.4.5: assignment Exported attributes can be used as an expression or as the left part of an assignment:

```
f.@baz := f.@baz + 1
```

A.2.6 Class Specialisation

Specialisation has three main uses:

- → A.2.6: property inheritance Build new classes out of existing classes since classes inherit properties defined their superclasses.
- → A.2.5: visibility Gain property visibility since properties exported to a class (`public` and `constructor`) are visible to their subclasses.
- → A.4.3: type Permit subtyping since objects of a class can be used where objects of the superclasses are expected.

Remark: In many object-oriented languages, inheritance is mainly a way to reuse property already defined. The semantic of inheritance of the PRM language is a bit different since it strictly corresponds to the natural semantic of specialisation: *If A is a superclass of B then each instance of B is also an instance of A.* The three uses of specialisation are simply corollaries of this strict semantic. It also means that two uses of specialisation, frequent in some OO languages, are forbidden in PRM: inheritance of implementation and repeated inheritance.

`inherit`. The `inherit` keyword is used to declare the superclass of the class. This keyword must be used before any property declarations.

The following listing is a very simple example of inheritance where a `Car` class is a subclass of a `Vehicle` class:

```
class Vehicle
end

class Car
inherit Vehicle
end
```

Multiple Class Specialisation. With multiple specialisation, the `inherit` keyword is repeated:

```
class Drake
inherit Duck
inherit Male
end
```

Transitive Specialisation. In PRM, transitive specialisation relation links are ignored. Therefore, the two following listings are equivalent:

```
class Ambulance
inherit Car
end
```

```
class Ambulance
inherit Car
inherit Vehicle
end
```

Moreover, the last one may produce a warning during compilation because of the superfluous `inherit Vehicle`.

Property Inheritance and Redefinition

Properties Inheritance. Subclasses inherit the properties—attributes and methods—of their superclasses.

```
class Car
        def @color: String def_read

        def sound: String
        do
                return "vroom"
        end
end

class Convertible
inherit Car
        def @roof_is_open: Boolean
end
```

This example shows a superclass `Car` and a subclass `Convertible`. The `Convertible` class inherits the following properties: the attribute `@color`, the automatic `color` accessor → A.2.5: accessors , and the `sound` function. It also defines a new property, the attribute `@roof_is_open`.

Properties Redefinition. Subclasses can redefine some inherited properties by providing a new definition of a property.

The following example shows the redefinition of the `sound` function inherited from the `Car` class:

```
class Ambulance
inherit Car
        def sound: String
        do
                return "wo-wo"
        end
end
```

Precursor. With the EIFFEL terminology, we says that the `sound` method of the `Car` class is a *precursor* of the `sound` method of the `Ambulance` class.

Global Property. In the previous example, the `sound` method of the `Car` class and the `sound` of the `Ambulance` class are two different methods. However, they belong to a "same property idea", here the idea is something like "sound of cars". We call *global property* this "same property idea".

Remark: Global properties are introduced when its first property is defined. Example, the global property "sound of cars" is introduced in the `Car` class by the `sound` method.

In PRM, global properties are not strictly related to properties names. For example, in the following listing, the two properties `@height` belong to distinct global properties:

```
class Person
        def @height: Int # in cm
        def @weight: Int # in kg
end

class Button
# A button for a graphical user interface
        def @height: Int # in pixels
        def @width: Int  # in pixels
end
```

Remark: This notion of *global property* is one of the PRM exclusivity. In great majority of other OO languages, the absence of this notion yields quantities of problems.

Attribute Redefinition. Obviously, redefinition is majority used for methods. It is also possible to redefine an attribute by specialising its static type:

```
class Car
        def @driver: Person
end

class PoliceCar
        def @driver: Policeman
end
```

→ A.4.3: covariant typing policy This is because PRM has a covariant typing policy.

Deferred Method

Deferred methods (called *pure virtual method* in C++) are methods without implementation. A deferred method is declared without a body, instead it has the `as deferred` keywords.

```
class Car
        def has_priority: Bool as deferred
end

class Ambulance
inherit Car
        def has_priority: Bool
        do
                return false
        end
end
```

Remarks:

- → A.2.4: abstract classes Usually, classes that contain deferred methods are mainly abstract classes—i.e. do not have constructors.
- → A.3.3: refinement Concrete classes with deferred methods cans be useful with refinement.

Multiple Inheritance

When a class has only one superclass, inheritance and redefinition are quite intuitive mechanisms. PRM multiple inheritance mechanism is also intuitive.

Which Properties to Inherit? The inherited properties are the most specific ones—i.e. the properties **defined** in the most specific classes. This base behaviour is quite simple but slightly differs from the majority of OO languages.

Example:

```
class A
        def foo
        do
                print("fooA")
        end
end

class B
inherit A
end

class C
inherit A
        def foo
        do
                print("fooC")
        end
end

class BC
inherit B
inherit C
end
```

In the `BC` class, there are two potential inherited methods: `foo"` defined in the `A` class, and `fooC` defined in the `C` class; the second is the most specific because `C` specialise `A`; therefore the `BC` class inherit the `"foo"` method defined in `C`.

Multiple Precursors. A property can redefine more than one property inherited from superclasses:

```
class D
inherit A
        def foo
        do
                print("fooD")
        end
end

class CD
inherit C
inherit D
```

```
        def foo
        do
                print("fooCD2")
        end
end
```

Here, the `foo` method of the `CD` class has two precursors since it redefines the `foo` methods of the classes `C` and `D`.

Property Conflict. A property conflict occurs when the most specific property to inherit is not unique:

```
class CD2
inherit C
inherit D
end
```

The solution to avoid them is to redefine the conflicting property.

→ A.2.6: deferred method When all properties but one are deferred, the conflict is automatically resolved: the one that is not deferred is inherited.

Global Property Conflict. A global property conflict occurs when a class inherits homonym properties that belong to distinct global properties:

```
class O
        def foo
        do
                print("fooO")
        end
end

class AO
inherit A
inherit O
end
```

`rename`. The solution to avoid them is to rename at least one of the two conflicting method with the `rename` keyword:

```
class AO2
inherit A rename foo(0) as fooA
inherit O
constructor
```

```
        def init do end
end
let x := new A02
x.foo   # Output `foo0`
x.fooA  # Output `fooA`
```

Remark: When renaming methods, the first name has to precise between parentheses the number of parameters.

One can rename more than one property

```
class Y
        inherit X rename foo(0) as fooX,
                @bar as @barX, -(1) as minus
end
```

The PRM renaming differs from the EIFFEL one and is slightly simpler and more coherent:

- One renaming per global property is enough, even if the global property comes from many superclasses:

```
class AC2 # WARNING: Superfluous renaming.
inherit A rename foo as foo2
inherit C rename foo as foo2
end
```

- A global property can not have two names in a same class:

```
class AC2 # ERROR: Multiple renaming.
inherit A rename foo as fooA
inherit C rename foo as fooC
end
```

- Two distinct global properties cannot be renamed to have the same name:

```
class P
        def bar
        do
                print("barP")
        end
end
class AP # ERROR: Global property conflict.
inherit A
inherit P rename bar as foo
end
```

Visibility

Visibility Inheritance. The visibility `public` and `private` of method inherited. However, constructors are inherited as private methods. This is because constructors of a class are not adapted to its subclasses.

In the following listing, `Ambulance`, a subclass of a `Car` class inherit the `with_color` as a private method. Therefore, `Ambulance` defines `init`, a new specific constructor.

```
class Car
        def @color: String def_read

constructor
        def with_color(col: String)
        do
                @color := col
        end
end

class Ambulance
inherit Car
constructor
        def init
        do
                with_color("white")
                # OK, since with_color is inherited
        end
end
```

Here some uses of the two classes:

```
let c := new Car.with_color("black")
print(c.color)  # Output ``black''
let a1 := new Ambulance
print(a1.color) # Output ``white''
let a2 := new Ambulance.with_color("blue") # Error!
# -> `with_color' is not a constructor,
#    it is a private method
```

Visibility Redefinition. One can redefine the visibility of inherited method by redefining the method in the wanted visibility block. The visibility of inherited method can also be redefined without having to redefine the whole method.

`export`. The `export` keyword enables to change the visibility of inherited method to the one of the current block:

```
class Car
constructor
        def init
        do
                @speed := 0
        end
end

class Ambulance
inherit Car
constructor
        export init(0)
end

let c := new Ambulance # OK
```

Remark: As for renaming, the number of parameters has to be indicated between parentheses.

Multiples methods can be exported at the same time:

```
class Bar
inherit Foo
public Baz
        export foo(1), bar(0), baz:=(1), +(1)
end
```

Call to Super

`super`. In a method redefinition, the programmer can refer to the previous property with the `super` keyword:

```
class Foo
        def foo(i: Int): Int
        do
                return i + 1
        end
end

class Bar
inherit Foo
        def foo(i: Int): Int
        do
                return super(i*2) * 2
```

```
        end
constructor
        def init do end
end

let b := new Bar
print(b.foo(2))  # Output ``10''
```

Implicit Super Arguments. Arguments of a super call are implicitly the parameters of the method. Therefore, the two following listings are equivalent:

```
...
        def foo(a: Int, b: String)
        do
                ...
                super(a, b)
                ...
        end
...
```

```
...
        def foo(a: Int, b: String)
        do
                ...
                super
                ...
        end
...
```

Multiple Precursor. When a method has more than one precursor, any call to super must be prefixed with a class name in order to remove the ambiguity. Such prefixes use the :: notation:

```
class CD3
inherit C
inherit D
        def foo(a: Int)
        # This method redefines the ones
        # of the classes C and D
        do
                C::super(a+2)
                D::super(a-1)
```

```
        end
end
```

A.2.7 Genericity

Inheritance is one of the fundamental mechanisms for reuse; so is genericity. Genericity is also important in making programs type safe without resorting to type casts. Java 5.0 introduces genericity, in previous version, many type casts where needed to make up for this deficiency. C++ has genericity in the form of template classes. If you have had problems understanding C++ templates, don't worry, PRM's generic syntax is much easier, and more powerful, as it also allows generic parameters to be bounded; this is known as bounded genericity.

In PRM, genericity is mainly the one of the EIFFEL language, please refers its specification to know more about genericity.

Generic Class Definition. In order to use genericity, you create a *generic class* with `formal generic parameters`. In the following listing, `Pair` is a generic class with one formal parameter bounded by `Any` and `VehiclePark` is a generic class with one formal parameter bounded by `Vehicle`.

```
class Pair[E: Any]
end
class Vehicle
end
class VehiclePark[E: Vehicle]
end
class Car
inherit Vehicle
end
```

Generic Type Construction. In program, the generic class can be used to construct many kinds of *generic types*:

```
let x: Pair[Int]           # x is a pair of integers
let y: Pair[Pair[String]] # y is a pair of pairs of strings
let z: VehiclePark[Car]    # z is a car-park
let t: VehiclePark[Int]    # Error since integers are not vehicles
```

Formal Generic Parameter Use. Inside the class definition, the formal generic parameter can be used as a type:

```
class Pair[E: Any]
        def @first: E def_read def_write
        def @second: E def_read def_write

        def switch
        do
                let t: E
                t := @first
                @first := @second
                @second := t
        end

        def display
        do
                print(@first, " ", @second)
        end

constructor
        def init(f: E, s: E)
        do
                @first := f
                @second := s
        end
end
```

Here some examples of use:

```
let pi := new Pair[Int].init(5, 4)
pi.display  # Output ``5 4''
pi.switch
pi.display  # Output ``4 5''

let ps := new Pair[String]("Hello", "Town")
ps.second := "World"
ps.display  # Output ``Hello World''
```

Generic Types and Subtypes. Genericity yields a kind of subtyping. For example, `VehiclePark[Car]` is a subtype of `VehiclePark[Vehicle]`:

```
let vehiclepark: VehiclePark[Vehicle]
let carpark: VehiclePark[Car]
vehicle := carpark # OK
```

→ A.4.3: covariant typing policy This is because PRM follows a covariant typing policy.

A.3 Modules

In the PRM language, classes are organised into *modules*. A modules corresponds to a source file and contain class definitions. They are also the compilation units: each module can be separately compiled then linked to produce an executable.

Filename. PRM source files should follow a strict naming scheme. They must be named `foo.prm` where `foo` is the name of the module.

A.3.1 Module Structure

A PRM source file (a module) is divided into four parts:

1. → A.3.2: module dependence Importation of modules.
2. → A.2.1: class definition Definition of classes.
3. → A.3.4: outside method Definition outside classes of procedures and functions.
4. → A.3.4: main statements Definition of the module main statements.

Each part is optional but the order must be respected.

A.3.2 Module Dependence

Modules can *depend* on others modules and *import* their classes. With analogy with the class terminology, we call *supermodules* of a module `m` the modules it depends on, and *submodules*[6] the modules that depend on `m`.

`import`. The dependence between modules is declared with the `import` keyword followed by the name of the module:

```
import crypt
import http
```

Remark: As in class hierarchy, cycles are forbidden in the module dependency! A module `m` cannot require itself nor require a module that requires `m`.

Implicit Dependence. → A.3.5: `standard` Implicitly, modules depend on the module `standard` that contains standard classes.

[6] In some language, "submodules" refers to nested modules (i.e. modules defined into a modules). Since there is no module nesting in PRM, there are no ambiguities.

Class Conflict. There is a class conflict when a module imports two homonymous distinct classes.

Such a conflict can be resolved with the `rename` keyword:

```
import automobile # import Car
import tramway rename Car as Tramcar
```

→ A.2.6: property renaming Class renaming follows and property renaming follows the same rules.

A.3.3 Class Refinement

PRM modules can extend imported classes, this is called *class refinement*:

```
# File m1.prm
class Foo
        def bar
        do
                print("before")
        end
constructor
        def init do end
end
```

```
# File m2.prm
import m1
class Foo
        def bar
        do
                print("after")
        end
end

(new Foo).bar  # Output ``after''
```

Properties: Definition and Redefinition. The main usage of refinement is to add new properties (methods and attributes) or to redefine them.

Remark: It's important to note that refinement is not specialisation: if you specialise a class, you have two classes, if you refine a class, you still have one class.

Class refinement is one of the greatest PRM features. It improves the reusability of OO software since it provides an answer to the separation of concern problem: a module can adapt existing classes to new concerns. Class refinement is clearly not a new OO

feature and exists in many dynamically typed languages (like RUBY or LISP) and in some statically typed language (like OBJECTIVE-C).

Multiple Refinement. Refinement can be also combined without difficulties:

```
# File m3.prm
import m1
class Baz
        def @foo: Foo

        def bar
        do
                @foo.bar
        end
constructor
        def init
        do
                @foo := new Foo
        end
end
```

```
# File m4.prm
import m2
import m3
(new Baz).bar  # Output ``after''
```

Class refinement works for any classes; even with build-in ones:

```
class Int
        def fib: Int
        # Fibonacci numbers
        # The inefficient recursive algorithm
        do
                if self <= 0 then
                        return 0
                elsif self <= 2 then
                        return 1
                else
                        return  (self - 1).fib +
                                (self - 2).fib
                end
        end
end
```

```
print(6.fib) # Output ``8''
```

Addition of Superclasses. It is also possible to refine a class by giving it new superclasses. This kind of refinement is quite rare, even in dynamically typed languages:

```
class Foo
        def foo
        do
                print(self, "-foo")
        end
end

class String
inherit Foo
end

"Hello".foo  # OK since String inherit the foo method
             # Output ``Hello-foo''

let f: Foo
f := "World" # OK since Strings are now Foos
f.foo        # Output ``World-foo''
```

Refinement, Specialisation and Multiple Inheritance. For property inheritance, refinement behaves like specialisation.

A.3.4 Procedural style

From a programmer point of view, PRM can be used as a procedural language:

- Some methods, like `print` seems to not have receiver.
- Procedures and functions can be defined outside classes.
- The main statements can be written outside procedures and functions.

However, PRM is a pure object-oriented language: procedures and functions are always methods, and statements always belong to method bodies.

Method Without Receiver

→ A.2.3: `self` Since each method invocation has a receiver, it means that each `print("Hello World")` use `self` as receiver.

In fact, `print` is a private method defined in `Any`, thus inherited as a private method in any other classes:

```
print("Hello ")      # Output ``Hello ''
self.print("World") # Output ``World''
5.print(".")         # Error
# -> 'print' is a private method
```

Method Definition outside Classes

Procedures and functions defined outside classes are implicitly defined as private methods of the class `Any`. Therefore, as `print`, they can be used everywhere.

Remark: Methods defined outside classes correspond to an implicit refinement of the class `Any`. Example: the two following listings are equivalent:

```
def foo
do
        print("hello world")
end
```

```
class Any
private
        def foo
        do
                print("hello world")
        end
end
```

Module Main Statement

The module main statements belong to the body of an implicit private `main` procedure defined in an implicit `Sys` class.

→ A.3.6: `Sys` class The `main` method of the `Sys` class corresponds to the entry point of programs.

Remark: Main statements correspond to an implicit refinement of the class `Sys`. The two following listings are equivalent:

```
print("Hello world")
```

is syntactically equivalent to

```
class Sys
private
        def main
```

```
        do
                print("Hello world")
        end
end
```

A.3.5 Base Modules

The PRM standard library is made of 6 modules:

standard → A.3.6: base classes The standard module is the implicitly required by other modules. It contains all the necessary base classes.

net implicitly depends on `standard`.

It specialises some IO classes in order to provide networking socket communication.

http depends on `net`.

It defines classes related to basic HTTP communication.

exec implicitly depends on `standard`.

It defines classes used to execute arbitrary commands of the shell system.

sdl implicitly depends on `standard`.

It defines wrapper classes around the Simple DirectMedia Layer C library[7]. It is primarily defined to experiment the feasibility of wrapping C libraries.

opts implicitly depends on `standard`.

It defines classes related to the parse and the analysis of command line options.

Remark: This base module is the only one developed by someone else: Floréal Morandat.

Standard Module

In fact, the `standard` module is not the root of the module hierarchy. `standard` is an empty module that only require base classes from some "super-standard" modules.

Remark: This section is mainly an illustration to how module hierarchy and class refinement can be used to develop modular applications—i.e. where modules are clearly "concern" units.

The PRM `standard` module depends on 11 supermodules:

[7] http://www.libsdl.org/

kernel depends on nothing.

It is the root of the module hierarchy and minimally defines the most basic classes like `Any`, `Sys`, `Int`, `Float`, `Bool` or `Char`.

math depends on `kernel`.

It refines the `Int` and `Float` classes with useful mathematical methods; for instance trigonometry. In a near future it will also define some class like complexes or big numbers.

abstract_collection depends on `kernel`.

It defines the PRM collection abstract generic class hierarchy. From the general `Collection[E: Any]` and `Iterator[E: Any]` classes to more specific like abstract set or abstract maps (i.e. associative arrays).

range depends on `abstract_collection`.

It defines the `Range` class and related ones.

array depends on `abstract_collection`.

It defines the `Array` class and related ones. It also specialises many abstract collection classes into concrete ones implemented with arrays—`ArrayMap`, `ArraySet`, etc.

sorter depends on `array`.

It refines the classes of the array module by the addition of sort method.

list depends on `abstract_collection`.

It specialises some collection classes—i.e. like the `array` module but with linked lists.

string depends on `array`.

It defines the `String` class and related ones. It also refines many classes by adding a `to_s` method used to convert any objects to a human readable representation.

hash depends on `string`.

It refines classes with a `hash` function and implements some hashes collections—`HashMap`, `HashSet`, etc.

io depends on `string`.

It define input/output related classes likes `File`. It also refines the `Any` by adding the `print` method.

string_search depends on `string`.

This module is about string searching and matching. It also implements the Boyer-Moore fast string searching algorithm.

A.3.6 Base Classes

Kinds of Classes. Classes can be classified into 4 categories:

Primitive Classes: They correspond to the primitive values of the computer. They are mainly defined in the `kernel` module. Primitive classes and their superclasses can have some restrictions.

Built-in Classes: They are known by the compiler since they have literals representation or are needed in some particular statements or expressions. Obviously, they include primitive classes.

Base Classes: They are the classes defined or imported by the `standard` module. They include Built-in classes.

User Classes: They are the other classes and are defined by PRM programmers.

Literal Value. Some built-in classes have literate value: programmers can create objects without explicitly instantiate them. The table A.1 summarises build-in classes and gives example of literals values.

	examples	**PRM types**
integers	`51`, `-85`	`Int`
floats	`5.5`, `-.05`, `8.0`	`Float`
characters	`'a'`, `'n'`	`Char`
strings	`"hello!"`, `"I"`	`String`
Booleans	`true`, `false`	`Bool`
range	`[1..5]`, `['a'..'b'[`	`Range[Int]`, `Range[Char]`
arrays	`[1,5,6]`, `['a','b','c']`	`Array[Int]`, `Array[Char]`
void	`nil`	`None`

Table A.1: The Basic Types

Any

The `Any` built-in class is the root of the class hierarchy.

Here some notable properties that will be inherited or redefined in other classes:

```
Class Any
public # Equality tests
        def ==(a: Any): Bool
        # The identity equality
        # Return 'true' if 'self' and 'a'
        # are the same object
        # False otherwise
        # /!\ This method cannot be redefined

        def !==(a: Any): Bool
        # Return 'not self == a'
        # /!\ This method cannot be redefined

        def =(a: Any): Bool
        # The value identity
        # Return 'true' if 'self' and 'a'
        # have the same ''contents''

        def !=(a: Any): Bool
        # Return 'not self = a'

public # String
        def to_s: String
        # Convert 'self' to a human readable form

private # Basic IO
        def print(a: Any*)
        # For each argument, output the
        # human readable form ('to_s')
end
```

Int

The `Int` primitive class represents internal machine integers. Literals are sequence of digits.

→ A.2.3: operator Notable `Int` properties are their operators. Many of them cannot be redefined.

Float

The primitive `Float` class represents internal machine float numbers.

Literals are sequence of digits a dot and another sequence of digit.

Character

The `Char` primitive class represents characters.

Literals are delimited with single quotes. Characters can include escaped sequence—Table A.2.

Escape sequence	Meaning
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\0`	ASCII Nul (NUL)
`\\`	Backslash (\)
in strings only...	
`\"`	Double quote (")
`\#`	Hash (#)

Table A.2: Char and String Escape Sequences

String of Characters

The `String` built-in class represents strings of characters—i.e. pieces of text.

Literals are delimited with double quotes. As with character, some escape sequence can be used to represent some characters—Table A.2.

Extended String Literal. String literal can contain embedded expression using `#{}`[8]:

```
print("4 + 8 = #{4+8}")
# Output ''4 + 8 = 12''

let h := "Hello"
let w := "World"
let hw := "#{h} #{w}"
print(hw)
# Output ''Hello World''
```

Programmers are encouraged to use extended strings because they are better for internationalisation since only entire sentences should be presented to the translator. Therefore having one string `"Replace #{obj1} with #{obj2}?"` is obviously better than having the three strings `"Replace "`, `" with "`, and `"?"`.

[8]The `#{}` notation comes from RUBY

Boolean

The `Bool` primitive class represents the two Booleans `true` and `false`.

Pseudo-operators `and`, `or` and `not`. There is also three special operators on Booleans that are not operators of the `Bool` class: `and`, `or`, and `not`. These keyword have a special status since `and`, `or` are lazy and will return the value, as soon as falsity (`and`) or truth (`or`) is established.

Booleans are used in many places, especially:

- → A.4.5 Conditional statement.
- → A.4.5 While loop.
- → A.4.5 Check statement.

Collection

The `Collection` built-in generic abstract class is the root of the collection class hierarchy.

Here some notable deferred method:

```
class Collection[E: Any]
        def is_empty: Bool
        # Is there no item in the collection ?

        def length: Int
        # Number of items in the collection.

        def has(item: E): Bool
        # Is 'item' in the collection ?
        # Comparisons are done with =

        def iterator: Iterator[E]
        # Get a new iterator on the collection.
end
```

Iterator

The `Iterator` built-in generic abstract class is mainly used with collections. Instances of the `Iterator` class generates a series of elements, one at a time.

Here some notable deferred method:

```
class Iterator[E: Any]
        def item: E
```

```
        # The current item.

        def next
        # Jump to the next item.

        def is_ok: Bool
        # Is there a current item ?
end
```

Array

→ A.2.7: genericity The `Array` built-in generic class is a subclass of `Collection`. It is also the preferred representation form of collections of items.

Array Literal. Literals representation use brackets and elements are separated with comma. Example:

```
let ai := [5, 4, 6, 1]
print(ai.length)      # Output ``4''
print(ai.has(5))      # Output ``true''
print(ai.has(9))      # Output ``false''
print(ai)             # Output ``5461''

let ai := ["Hello", " ", "World"]
print(ai.length)      # Output ``3''
print(ai)             # Output ``Hello World''
```

Static Type of Array Literal. Since `Array` is generic, the type of literal should be computed. Literals expressions are valid if and only if there is a unique more general static type. The static type of the literal expression is build with this type.

```
let ai := [4, 5]       # more general type: Int
                       # Therefore ai is an Array[Int]
let ae := [4, '5']     # more general types: Int and Char
                       # Therefore, it is an error
let x: Any
let aa := [4, '5', x] # more general type: Any
                       # Therefore ai is an Array[Any]
```

Array Constructor. When literals are invalid, one can use the `init` array constructor therefore explicitly precise the desired static type:

```
let ae := new Array[Any](4, '5') # OK
```

→ A.2.3: multiple arguments Arrays are used in multiple argument procedures. The `Array init` constructor uses multiple arguments.

Range

The built-in a generic class `Range` is for `Discrete` elements. It represents intervals between a first element and a last element.

There are two kinds of ranges, inclusive ranges and exclusive ranges.

Inclusive Range. They include the first element, the last element and each element between them. Their literals use the `[a..b]` notation (`a` and `b` can be any expression): Literals are valid if first element and the last element have the same static types.

```
let x := [1..5] # x is Range[Int]
print(x.length) # Output ``5''
print(x.has(0)) # Output ``false''
print(x.has(1)) # Output ``true''
print(x.has(4)) # Output ``true''
print(x.has(5)) # Output ``true''
print(x.has(6)) # Output ``false''
```

Exclusive Range. Like inclusive range but they exclude the last element. Their literals use the `[a..b[` notation:

```
let x := [1..5[ # x is Range[Int]
print(x.length) # Output ``4''
print(x.has(5)) # Output ``false''
```

→ A.4.5: for loop Ranges are often used in for loops.

None

`None` is the absurd class, it is the class that specialise each other except primitives classes. It is not a "real" class since it does not have a definition. It is also the only class that cannot be specialised or refined.

`nil`. `None` has only one instance `nil`, often called the *void object*. `nil` correspond to the `null` constant of JAVA or the `Void` object of EIFFEL.

Each method invocation on `nil` will fail. The only exception are equality operators `=`, `==`, `!=`, and `!==`. Therefore, it is frequent to verify if potential receiver is nil before sending a message:

```
def safe_array_length(a: Array[Int]): Int
do
        if a = nil then
                return 0
        else
                return a.length
        end

end
```

Sys

The `Sys` built-in class

Program Start. When the program starts, it instantiates the `Sys` class then invoke the `init` procedure.

`Sys` **Definition.** The `Sys` class is defined in the `kernel` module as follow:

```
class Sys
private
        def init
        # The entry point of the program
        do
                init_begin
                main
        end

        def init_begin
        # Initialisation of library objects
        do
        end

        def main
        # The main part of the program
        do
        end
end
```

The `init_begin` procedure is used to initialise library object. For instance, the module `io` build the standard IO file objects in `init_begin`

→ A.3.4: procedural style The `main` procedure corresponds to the main statements of the module.

A.4 The Base Language

This section is dedicated to the base language programming.

A.4.1 Source Structure

PRM is a line-oriented language. PRM statements are terminated at the end of a line unless the statement is obviously incomplete—for example if the last token on a line is an operator or comma. A semicolon can be used to separate multiple expressions on a line.

Comment. Comments start with '`#`' and run to the end of the physical line. They are ignored during compilation. Currently, there are no multi-line comments.

```
# One statement, one line
a := 1

# Two statements, one line
b := 2; c := 3

# One statement, two lines
d := 4 + 5 +
     6 + 7

# Two statements, two lines
# But the second one clearly do nothing
# and may provoke a warning during compilation
e := 8 + 9
     + 10
```

A.4.2 Name

PRM names are used to refer to variables, properties (methods and attributes), classes, and modules. The first character of a name helps PRM to distinguish its intended use.

Reserved Names. Certain names, listed in Table A.3, are reserved and should not be used as variable, property, class, or module names.

and	as	check	class	constructor	def	def_attr
def_read	deferred	do	else	elsif	end	extern
false	for	if	import	in	inherit	intern
isa	let	let	new	nil	not	not
once	or	private	public	rename	return	self
then	true	until	while			

Table A.3: Reserved Names

In these descriptions, a lowercase letter means the characters "a" though "z", as well as "_", the underscore. An uppercase letter means "A" though "Z", and digit means "0" through "9". Name characters mean any combination of upper- and lowercase letters and digits.

A local variable name, a method name, or a module name consists of a lowercase letter followed by name characters. Examples: `foo`, `foo_bar_baz`, `_x`.

A class name starts with an uppercase letter followed by name. Examples: `Int`, `Any`.

An attribute name starts with an "at" sign ("@") followed by a lowercase letter, followed by any name characters. Examples: `@name`, `@x`, `@_`.

A.4.3 Type

PRM is a statically typed language, it's mean that "things" should have a static type.

Type Annotation. Type annotation use the PASCAL notation style: `thing: Type`. Such type annotations are used in the following places:

- → A.4.5 Local variable declarations.
- → A.2.3 Method signatures—for parameters, and in functions for the return type.
- → A.2.7 Generic classes—for formal generic parameters.

Type Language. A type is:

- → A.2.1 A non generic class—e.g. `Int`, `Car`...
- → A.2.7 A generic type—e.g. `Array[Int]`, `Iterator[Car]`
- → A.2.7 A formal generic parameter.

Covariant Type Policy. As EIFFEL, PRM uses a covariant typing policy.

The covariant typing policy allows the programmer to redefine properties with a more specific signature:

```
class Food
end
class Grass
inherit Food
end
class Animal
        def eat(f: Food)
        ...
end
class Cow
inherit Animal
        def eat(g: Grass)
end
```

However, such a typing policy is unsafe. In concrete term it means that in some case, type error may occur at runtime and stop the program execution[9].

A.4.4 Expression

There are ten kinds of expressions:

- → A.2.3 The current receiver `self`.
- → A.3.6 Literal values.
- → A.2.3 Function invocation.
- → A.4.5 Variable read.
- → A.2.3 Attribute read.
- → A.2.5 Exported attribute read.
- → A.2.4 Object creation.
- → A.3.6 Boolean pseudo operators.
- → A.4.4 Type checks.
- → A.4.4 Once expressions.

[9] In a future version, runtime type error will raise an exception.

Type Checks

Type check can be used to test if an object is an instance of a giver class (or an instance of a subclass). However, since PRM uses bounded genericity and a covariant typing policy, there is a very few need of such type checks.

→ A.4.5: assertions; A.4.5: assignment attempt The majority of their use corresponds to assertions and to assignment attempts.

`isa`. Type checks can be explicitly performed with the `isa` keyword:

```
let a: Any
a := 5
print(a isa Int)  # Output ``true''
print(a isa Any)  # Output ``true''
print(a isa Bool) # Output ``false''
```

Once Expression

Remark: This part of the specification is not stable and may change in a future version.

This expression is constituted by the `once` keyword followed by another expression called *sub-expression*:

```
let x := once "Message"
let y := once new Car
```

The semantic of the once expression is to evaluate the sub-expression only one time during the execution of the program. Successive evaluations of the once expression will return the first evaluated value.

Once expressions are mainly used to create singletons and to perform some local optimisation—it is often used with literals string and arrays.

Examples:

```
def only_one(i: Int): Int
do
        return once i
end
print(only_one(1)) # Output ``1''
print(only_one(2)) # Output ``1''
print(only_one(3)) # Output ``1''
```

```
class Person
   ...
end
```

```
def immortal: Person
# There can be only one
do
        return once new Person
end
```

The once expression is a generalisation of the EIFFEL `once` keyword.

A.4.5 Statement

There are nine different statements in PRM: statement block, local variable declaration, assignment, procedure invocation, conditionals, while loop, for loop, return and check.

Statements are always defined in a statement block or belong to the main module statements.

Statement Block

Blocks of statements often start with the `do` keyword and are ended with the `end` keyword:

```
# outside
do
        # inside
        do
                # more inside
        end
        # inside again
end
# outside again
```

→ A.3.4: main → A.4.5: `if` The only exceptions are the main statements of the program and the statements of the `if` statement.

PRM statement blocks are slightly different from other language ones:

- `do/end` differs from usual `begin/end` of others PASCAL-style languages. In fact, it is almost a 50% less characters[10].
- Curly brackets from C-influenced languages do not fit with the overall PRM PASCAL style.

[10] `do/end` statement blocks are used in some languages (PL/1, REXX). In RUBY, `begin/end` corresponds to statement blocks, although `do/end` and curly brackets correspond to closures.

In a near future version, statement block will be extended to allow exception management. A potential syntax can be:

```
do
        ...
rescue e: IOException
        ...
rescue e: EmptyListException
        ...
rescue
        ...
finally
        ...
end
```

Local Variable Declaration

`let`. The `let` keyword is used to declare local variables:

```
let i: Int                  # i is an integer
let j, k: String            # j and k are strings
```

Initial Value and Type Inference. An initial value can be directly assigned with the local variable. If the initial value is present and the static type absent, the static type of the local variable is implicitly the static type of the initial value. Examples:

```
let j: Int := 5 + 3         # an integer with the value 8
let k := j + 1              # an integer with the value 9
let c := new Car("Blue") # a blue car
```

Default Value. Without an explicit initial value, local variables are initialised at 0 for `Int`, `'0'` for `Char`, `false` for `Bool`, and `nil` for the other types.

Visibility. The visibility of local variable runs from its declaration until the end of the current block.

```
do
        # 'i' is not yet known
        let i: Int
        # 'i' is known
        do
                # 'i' is still known
```

```
        end
        # 'i' is still known
end
# 'i' is no more known
```

Remark: One can declare in a same block two local variables with the same name. The last declared will mask the others. However, the compiler may produce a warning.

```
let i: String
i := "foo"
do
        let i: Int          # Warning !
        i := 5              # Correct, the Int variable
                            # masks the String variable
end
i := "bar"                  # Correct, the Int variable
                            # is no more known
```

PRM encourages the use of local variables to store intermediate results. Therefore, it allows the programmer to have a liberal use of local variables:

- New local variables can be declared when they are needed. Some languages like Eiffel or Smalltalk only allow local variable declaration at begin of subprograms. Some other languages, like Ada, Modula 3, C, or Lisp, only allow them at begin of statement blocks.
- The static type is optional any can be inferred from the initial value. This feature is quite rare in statically typed languages even if it was one of the first that appears in during the PRM specification development. Modula-3 has it and it is planned for the future C# 3.0.

Assignment Statement

The assignment statement uses the quite common `:=` and is widely used in PRM programs:

- Local variable assignment.
- → A.2.3 Attribute assignment.
- → A.2.3 Assignment procedure.
- → A.2.3 Implicit parameter declaration.
- → A.2.5 Exported attribute assignment.
- → A.4.5 Initial local variable value.

Conformance. An assignment `a := b` is statically valid if the static type of the left-value `a` is a supertype of static type of the right-value `b`:

```
let x: Int
x := 4        # OK
x := 'a'      # Error
let y: Any
y := x        # OK
y := 'a'      # OK
```

Assignment Attempt. The assignment attempt use the EIFFEL `?=` notation. It works exactly like the assignment, except that conformance is not checked statically but at runtime. If an assignment attempt fails, the program execution will stop[11].

Example:

```
let x: Any
let y: Int
let z: Char
x := 5
y ?= x # OK
z ?= x # Error at run-time
```

→ A.4.4: type check In order to avoid run-time error, dynamic types can be checked before any assignment attempts:

```
let x: Any
let y := 0
...
if x isa Int then
        y := x
end
```

Conditional Statement

Conditionals use the standard `if then elsif else` keywords. The `elsif` and `else` parts are optional, and there can be more than one `elsif` part.

```
def game(guess: Int, solution: Int): String
# A simple game
do
        if guess > solution then
                return "It's less"
```

[11] In a future version, it will raise an exception.

```
        elsif guess < solution then
                return "It's more"
        else
                return "Correct"
        end
end
```

The question about the one-liner `if` was raised but we did not find a clear and concise syntax. We find only the PERL-ish post-test `if`:

```
instr if expr
```

but it does not satisfy us.

While Loop

The while loop is the main PRM loop structure. It is constituted with a Boolean expression and a statement block:

```
def gcd(x, y: Int)
# The greatest common divisor between x and y
# using the Euclid's algorithm
do
        while y != 0 do
                let t := y
                y := x % y
                x := t
        end
        return x
end
```

For Loop

The `for` loops are used for collection traversal. It is quite different of the C or C++ `for`. In fact it is comparable to the PERL `foreach` or to the new JAVA 5.0 `for/in` loop

This loop can be used with any expression subtype of the built-in class `Collection`, even those defined by the programmer. Since `Array` and `Range` are subclasses of `Collection`, here are two examples:

```
let pricelist := [34.50, 21.95, 4.95, 8.45]
for price in pricelist do
        let gstprice := price * 1.1
        print("Price is ", gstprice, " including GST.\n")
end
```

```
for i in [0..10] do
        print("Value of i: ", i, "\n")
end
```

In fact, the `for` loops are no more than a while loops adaptation. The last example with a range is equivalent with:

```
do
        let x := [0..10].iterator
        while x.is_ok do
                let i := x.value
                print("Value of i: ", i, "\n")
                x.next
        end
end
```

Return Statement

This statement has two usages according to the kind of method it is used: in a procedure or in a function. In both case, it terminate the method.

In a function, the return statement is mandatory and must provide a result value that is conform to the declared result type in the signature of the function:

```
def sign(i: Int): Int
do
        if i > 0 then
                return 1
        elsif i < 0 then
                return -1
        else
                return 0
        end
end
```

In a procedure, the return statement is optional and must not provide a value:

```
def stars(nb: Int)
do
        if nb <= 0 then
                println("I want stars.")
                return
        end

        println("A star: *")
```

```
        for nb in [2..i] do
                println("Another star: *")
        end
end
```

Check Statement

The check statement is about correctness. It corresponds to assertions and helps to check validity of programs and identify bugs.

→ A.3.6: Bool A check statement is constituted by the `check` keyword, optionally an assertion label followed by a colon, then a Boolean expression.

```
def hello(name: String)
        check correct_name: name !== nil and
                not name.is_empty
        print("Hello ", name)
end
```

During its execution, the program will halt on the check if the expression is evaluated to `false`[12].

A.5 A PRM Conclusion

The PRM language focuses expressive, clear, simple, and coherent concepts in a statically typed object-oriented language whereas the other languages of the same family rarely focus simplicity.

Currently, the PRM specification is almost complete however some characteristics are currently instable and others are missing like constant values, enumeration types, introspection, module visibility (import/export), exceptions, contracts, regular expressions...

The standard module hierarchy needs also to be extended. Actually there is less than 6000 line of code in the base modules—that is not a lot even if PRM has a concise syntax.

The last work is about the compiler and other tools. Actually, the PRM compiler, `prmc`, is just a prototype and does not yet implement entirely the current specification. However, it produces efficient executable.

[12] In a future version, it will raise an exception.

A.6 Index

`*`
    variable argument number, 228
.
    `Float` literal, 260
    method invocation, 221
::, 249
`:=`, *see* Assignment
`?=`, 273
`#`
    comment, 266
    extended string literal, 261

Abstract class, 231
`abstract_collection`, 258
Accessor, 237
    automatic, 238
    pseudo-accessor, 238
`and`, 262
`Any`, 259
`Array`, 263
`array`, 258
Assignment, 272
    assignment procedure, 224
    attempt, 273
    attribute access, 221
    bracket assignment procedure, 225
    conformance, 273
    implicit parameter value, 226
    local variable initial value, 271
Attribute
    access, 221
    accessor, *see* Accessor
    definition, 219
    exported access, 239
    redefinition, 242
    visibility, 239

Block
    statement, 270
    visibility, 232

`Bool`, 262
    pseudo-operator, 262
Boolean, *see* `Bool`
Bracket
    array literal, 263
    assignment procedure, 225
    generic class, 250
    generic type, 250
    operator, 224
    range literal, 264

`Char`, 261
Character, *see* `Char`
Character string, *see* `String`
`check`, 276
Class, 218
    abstract, 231
    concrete, 231
    conflict, 253
    definition, 218
    generic, 250
    instantiation, 230
    refinement, 253
    specialisation, 240
`class`, 219
`Collection`, 262
Comment, 266
Concrete class, 231
Conflict
    global property, 245
    property, 245
`constructor`, 230, 235
    empty, 231
    implicit, 231
    visiblity, 235

`def`, 219
`deferred`, 243
`def_read`, 238
`def_write`, 238

`else`, 273
`elsif`, 273
Escape sequence, 261
`exec`, 257
`export`, 239, 247
Expression, 268

`false`, 262
`Float`, 260
`for`, 274

Garbage Collector, 232
Genericity, 250
Global property, 242

`hash`, 258
`http`, 257

`if`, 273
Implicit
    constructor, 231
    module dependence, 252
    parameter value, 226
    receiver, *see* `self`
    super arguments, 249
    visibility block, 234
`import`, 252
Infix operator, *see* Operator
`inherit`, 240
`Int`, 260
Integer, *see* `Int`
`io`, 258
`isa`, 269
`Iterator`, 262

`kernel`, 258
Keyword
    reserved names, 266

`let`, 271
`list`, 258
Literal value, 259
Local variable
    declaration, 271
    implicit type, 271
    initial value, 271
    visibility, 271
Loop
    for, 274
    while, 274

`math`, 258
Method
    assignment procedure, 224
    bracket assignment procedure, 225
    deferred, 243
    definition, 220
    implicit parameter value, 226
    invocation, 221
    operator, *see* Operator
    super call, 248
    variable argument number, 228
    visibility, *see* Visibility
Module, 252
    dependence, 252
    implicit dependence, 252

Name, 266
    reserved, 266
`net`, 257
`new`, 230
`nil`, 264
`None`, 264
`not`, 262

Object
    creation, 230
`once`, 269
Operator, 223
    bracket, 224
    infix, 223
    prefix, 223
`opts`, 257
`or`, 262

Parameter
implicit value, 226
variable argument number, 228
Prefix operator, *see* Operator
`print`, 259
`private`, 234
Procedural style, 255
Property
conflict, 245
definition, 219
global, 242
global conflict, 245
inheritance, 241
redefinition, 242
rename, 245
`public`, 233

`Range`, 264
`range`, 258
Refinement, *see* Class refinement
`rename`
class, 253
property, 245
`return`, 275

`sdl`, 257
`self`, 222
`sorter`, 258
`standard`, 257
Statement, 270
assignment, 272
block, 270
check, 276
conditional, 273
for loop, 274
return, 275
while loop, 274
`String`, 261
extended, 261
`string`, 258
`string_search`, 259
`super`, 248
`Sys`, 265

`then`, 273
`true`, 262
Type, 267
check, 269
generic, 250

Variable
instance, *see* Attribute
local, *see* Local variable
Visibility, 232
attribute, 239
block, 232
constructor, 235
implicit block, 234
inheritance, 247
local variable, 271
private, 234
public, 233
redefinition, 247

`while`, 274

Annexe B

Index des langages

Préambule

Dans le présent manuscrit, nous faisons référence à de nombreux langages. Nous listons ici ceux auquels nous faisons référence de manière non anecdotique.

B.1 ADA

Ada [Taft et Duff, 2001] est un langage de programmation impératif, structuré, statiquement typé conçu par Jean Ichbiah et son équipe entre 1977 et 1983. En 1995, le langage est étendu et intègre quelques spécificités de la programmation par objets. Le langage est fortement orienté sur la structuration, la fiabilité et la robustesse. Pour cette raison, ce langage est utilisé par l'industrie dans les systèmes critiques comme l'aéronautique ou l'armement.

Parmi les caractéristiques notables du langage, on peut citer la généricité et un vrai système de modules (*package*) hiérarchiques.

Le nom du langage vient de Ada Lovelace[1], souvent créditée comme la première personne à avoir écrit un programme informatique.

De par sa forte structuration, le langage Ada est souvent utilisé dans les milieux universitaires comme langage de support à l'apprentissage de la programmation.

B.2 C

C [Kernighan et Ritchie, 1988] est un langage de programmation généraliste, procédural et impératif développé par Dennis Ritchie et Brian Kernighan.

[1] Il y aurait même des rues à son nom.

Il a été développé dans les laboratoires Bell à partir du début des années 70 pour être utilisé sur le système d'exploitation UNIX. Il s'est depuis diffusé sur d'autres systèmes d'exploitation et est maintenant l'un des langages de programmation les plus couramment employés.

C n'est pas un langage destiné à l'apprentissage de la programmation et de l'algorithmique, principalement à cause de certaines caractéristiques syntaxiques et sémantiques impitoyables pour le programmeur débutant. Malgré tout, il est couramment utilisé dans les milieux universitaires, principalement comme langage de support aux enseignements de système d'exploitation et de réseaux, domaines dans lesquels il est le langage dominant.

Le langage C est souvent critiqué par sa propension à permettre au programmeur d'écrire n'importe quoi, en particulier en ce qui concerne la gestion des pointeurs, de la mémoire et des tableaux.

B.3 C++

C++ [Stroustrup, 2000] est un langage de programmation généraliste développé par Bjarne Stroustrup comme une évolution du langage C. Il fut développé à partir du début des années 80 dans les laboratoires Bell.

La première évolution fut l'ajout de classes, vinrent ensuite les fonctions virtuelles, la surcharge d'opérateurs, l'héritage multiple, la généricité (*template*) et la gestion des exceptions.

C++ est l'objet de nombreuse critiques, en particulier celles qui étaient déjà adressées au langage C : « C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off » (Bjarne Stroustrup). Les autres critiques concernent plus sa spécification des mécanismes objets.

B.4 Clos

CLOS [Steele, 1990] (Common Lisp Object System) est la sur-couche objet de COMMON LISP [2].

Par rapport aux autres langages présentés dans cette annexe, CLOS possède trois grandes particularités :

Sélection multiple. Les méthodes ne sont pas définies *dans* les classes mais appartiennent à des *fonctions génériques*. Une fonction générique est un ensemble de méthodes partageant le même nom mais différenciées par le type statique de leurs

[2] COMMON LISP est un dialecte de LISP créé pour unifier et standardiser les nombreuses variantes de LISP.

paramètres. Les envois de messages sont résolus en prenant en compte l'ensemble des types dynamiques des arguments.

Linéarisation. Les techniques de linéarisation [Ducournau *et al.*, 1994; Barrett *et al.*, 1996] sont à la base du traitement de l'héritage multiple en CLOS (et dans la plupart des autres langages à objets basés sur LISP).

Protocole à méta-objet. le MOP [Kiczales *et al.*, 1991] de CLOS, lié à l'existence de méta-classes, et permet de modifier la sémantique et le comportement du système (toutefois, la méta-programmation n'est pas du tout abordé dans cette thèse).

B.5 Eiffel

EIFFEL [Meyer, 1997] fut créé en 1985 par Bertrand Meyer et développé par son entreprise, Interactive Software Engineering (ISE) (Goleta, Californie, USA).

Le nom EIFFEL est un hommage à Gustave Eiffel, ingénieur qui conçut la tour éponyme en respectant les délais et le coût prévu, et dont la structure est constituée de nombreux éléments modulaires. Pour Bertrand Meyer, aucun des langages existants n'était totalement satisfaisant du point de vue du génie logiciel. Ainsi EIFFEL n'est pas une sur-couche ou une variation d'un langage existant : d'une part, il propose une approche complète d'ingénierie objet [Meyer, 1988] ; d'autre part, il intègre un modèle de contrats [Meyer, 1991] — *Design by Contract* — qui vise à documenter et garantir les spécifications par la définition explicite dans le code source de préconditions, postconditions et invariants.

Depuis sa création et sa présentation, le langage n'a cessé d'évoluer. La définition du langage est dans le domaine public où elle est contrôlée par NICE, le *Non-profit International Consortium for Eiffel* fondé en 1991.

B.6 Java

JAVA [Gosling, 2000] est un langage de programmation objet développé par James Gosling à Sun Microsystems à partir du début des années 90.

Le langage hérite sa syntaxe de C et de C++ mais avec une épuration des concepts — la différence généralement signalée en premier est l'absence de pointeur explicite. Les autres différences signalées par rapport à C++ sont l'absence d'héritage multiple (compensée par un sous-typage multiple) et jusqu'a récemment (c'est-à-dire jusqu'à la version 5 du langage sortie en 2004) l'absence de généricité.

B.7 Objective Caml

Objective-Caml [Chailloux *et al.*, 2000] est un langage de programmation généraliste bâti sur un noyau fonctionnel et impératif (Caml qui descend de Ml) mais proposant une sur-couche objet. Il fut développé par, entres autres, Xavier Leroy, Jérôme Vouillon, Damien Doligez et Didier Rémy à partir de 1996. Objective-Caml est un projet libre géré et maintenu principalement par l'INRIA.

Le langage se caractérise par une inférence de types : bien que statiquement typé, les annotations de types ne sont jamais explicitées par le programmeur mais inférées par le système.

B.8 Pascal

Pascal [Wirth, 2002], est un langage de programmation impératif statiquement typé créé par Niklaus Wirth en 1970 à l'université de Zurich. Il a été conçu à l'origine pour servir à l'enseignement de la programmation de manière rigoureuse mais simple.

Actuellement, le langage ne semble plus vraiment utilisé, ni dans les milieux industriels, ni dans les milieux universitaires.

B.9 Perl

Perl [Wall *et al.*, 2000] (*Practical Extraction and Report Language*) est un langage de script originellement dédié aux tâches d'administration système. Il fut développé par Larry Wall à partir de la fin des années 80. Perl inclut des caractéristiques de nombreux langages de programmation et d'outils d'administration système, en particulier C, scripts shell (sh), awk et sed.

Contrairement aux autres langage de programmation exposés dans cette annexe, Perl n'est pas développé autour d'une spécification mais vis-à-vis de son interpréteur `perl` dont la page de manuel précise : « The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). », ce qui nous apparaît comme étonnamment contradictoire.

B.10 PRM

PRM [Privat, 2006] est le langage de programmation orienté objet et statiquement typé présenté dans ce mémoire.

B.11 Python

PYTHON [Rossum et Fred L. Drake, 2003] est un langage de programmation appartenant à la famille des langage de scripts (typage dynamique, prototypage rapide, faible structuration, majoritairement interprété) développé par Guido van Rossum à partir de 1990.

C'est un langage de programmation multi-paradigme, en particulier il permet une programmation par objets.

B.12 Ruby

RUBY [Thomas et Hunt, 2000] est un langage de script orienté objet développé dès 1993 par Yukihiro Matsumoto avec un objectif premier de cohérence et d'intelligibilité.

Comme de nombreux langages de script, il permet entre autres de manipuler facilement les fichiers textes (les expressions rationnelles sont natives), de faciliter la programmation web (gestion avancée des CGI, Web2.0 avec RUBY On Rails[3]) et d'aider à effectuer des tâches d'administration système.

Contrairement à la majorité des autres langages de script, RUBY est un langage pleinement orienté objet.

B.13 Simula

SIMULA [Birtwistle *et al.*, 1973] est un langage de programmation développé dans les années 60 par Ole-Johan Dahl et Kristen Nygaard. C'est le premier langage à avoir introduit les notions de *classe*, *objet*, *sous-classe* et *méthode (virtuelle)*.

Comme son nom l'indique, SIMULA fut créé pour effectuer des simulations.

B.14 Smalltalk

SMALLTALK [Goldberg et Robson, 1983] est un langage de programmation orienté objet, dynamiquement typé développé au Xerox PARC par Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg, et d'autres pendant les années 1970. Influencé par SIMULA, il est sans soute le premier langage de programmation à avoir conceptualisé la notion de programmation par objets.

SMALLTALK est un langage minimaliste qui se concentre sur quelques concepts et dont la syntaxe est étonnamment épurée — il possède seulement 5 mots clés `true`, `false`, `nil`, `self` et `super`.

[3] `http://www.rubyonrails.org/`

Encore aujourd'hui, il est toujours considéré comme l'un des langage de programmation par objets les plus *purs*.

Annexe C

Spécification du pseudo-langage d'assemblage

Préambule

Dans les chapitres 6 et 7 nous utilisons un pseudo-langage d'assemblage pour une pseudo-machine. Ce langage est une adaptation de celui utilisé dans [Driesen et Hölzle, 1995; Driesen, 1999; Driesen, 2001].

C.1 Instructions et arguments

Chaque ligne débute par un mot clé, l'instruction, et est suivi par un ou plusieurs arguments. Les commentaires commencent par un point-virgule. Une ligne peut également correspondre à une étiquette, c'est-à-dire une valeur immédiate suivie d'un deux-points.

C.1.1 Arguments

Il y a trois sortes d'arguments :
- les registres qui sont identifiés par un symboles simple (lettres et chiffres) : `R1` ;
- les valeurs immédiates qui commencent par un `#` : `#imm` ;
- les symboles identifiés par des chevrons : $\langle sym \rangle$.

Les symboles s'utilisent comme une valeur immédiate puisque lors de l'édition de liens, les symboles sont substitués par des valeurs.

C.1.2 Mémoire : instructions `load` et `store`

L'instruction `load` charge dans un registre (le deuxième argument) une valeur stockée dans la mémoire à une adresse donnée par le premier argument. L'instruction `store`

stocke la valeur d'un registre (le premier argument) dans la mémoire à une adresse donnée par le deuxième argument. Une adresse est une constante immédiate `[#imm]`, la valeur d'un registre `[R1]` ou la somme de la valeur d'un registre et d'une valeur immédiate `[R1 + #imm]`.

Exemples :

```
load [R1 + #imm] → R2
load [R1] → R2
load [#imm] → R2

store R1 → [R2 + #imm]
store R1 → [R2]
store R1 → [#imm]
```

C.1.3 Calculs : instructions `add` et `cmp`

L'instruction `add` additionne la valeur d'un registre avec une valeur immédiate ou avec la valeur d'un autre registre. Le résultat est stocké dans un troisième registre.

```
add R1 + #imm → R3
add R1 + R2 → R3
```

Les autres instructions arithmétiques ne sont pas utilisées.

L'instruction `cmp` compare la valeur d'un registre avec une valeur immédiate ou avec la valeur d'un autre registre :

```
cmp R1 = #imm
cmp R1 = R2
```

C.1.4 Branchements : instructions `call` et `j*`

L'instruction `jmp` effectue un branchement vers une adresse. Une adresse est soit une valeur immédiate (on parle de *branchement direct*) soit la valeur d'un registre (on parle de *branchement indirect*).

```
jmp #imm
jmp R1
```

L'instruction `call` fonctionne comme `jmp` mais sauve l'adresse de retour dans la pile.

```
call #imm
call R1
```

Les instructions `jeq`, `jne`, `jlt jgt`, `jle`, `jgt` effectuent ou pas un branchement en fonction du résultat de la dernière instruction `cmp` exécutée (on parle de *branchement conditionnel*).

```
cmp R1 = R2
jeq #imm          ; jump if R1 and R2 are equal
jne #imm          ; jump if R1 and R2 are not equal
jlt #imm          ; jump if R1 < R2
jgt #imm          ; jump if R1 > R2
jle #imm          ; jump if R1 ≤ R2
jge #imm          ; jump if R1 ≥ R2
```

C.2 Processeur

La pseudo-machine considérée correspond au processeur P95 de [Driesen et Hölzle, 1995; Driesen, 1999; Driesen, 2001].

À chaque séquence de code assembleur, on peut estimer le nombre de cycles nécessaires pour son exécution sur une machine. Le nombre de cycles d'une séquence dépend de chaque instruction :

- les calculs (`add` et `cmp`), le stockage (`store`) et le branchement direct (`j*` et `call` avec une valeur immédiate) coûtent un seul cycle ;
- le chargement (`load`) a une latence de 2 ou 3 cycles, nous notons L cette valeur ;
- le branchement indirect (`jmp` et `call` avec un registre) a une latence de 3 à 15 cycles, nous notons B cette valeur.

Toutefois, deux instructions indépendantes peuvent être exécutées en parallèle alors que deux instructions dépendantes doivent être exécutées en série.

Ainsi, la séquence suivante ne coûte que 1 car les deux instructions peuvent être exécutées en parallèle :

```
add R1 + R2 → R2                                                    1
add R1 + R3 → R3
```

alors que ces deux autres doivent être exécutées en série :

```
add R1 + R2 → R2                                                    2
add R2 + R3 → R3
```

La seule exception est le chargement (`load`) puisque deux chargements ne peuvent jamais être exécutés entièrement en parallèle et doivent être décalés au moins d'un cycle :

```
load [R1] → R2                                                    L+1
load [R3] → R4
```

cela ne change rien au niveau du chargement en série :

```
load [R1] → R2                                                    2L
load [R2] → R3
```

Toutefois, la principale limite de ce modèle de machine est la non prise en compte du cache — ou plutôt des défauts de cache (*cache miss*). En effet, lors d'un chargement, d'un stockage ou d'un branchement, l'adresse peut être absente du cache du processeur le plus bas (appelé L1) et donc nécessiter un nombre supplémentaire de cycles pour accéder à la mémoire : cache processeur plus haut (L2, L3, etc.) de l'ordre de la dizaine de cycles, à la mémoire vive (RAM) de l'ordre de la centaine de cycles, voire au périphérique de stockage (swap) de l'ordre plusieurs millions de cycles (millisecondes au lieu de nanosecondes).

Certains benchmarks au chapitre 8 montre clairement cette influence du cache sur des mécanismes théoriquement en temps constants.

Bibliographie

[Agesen, 1994] O. Agesen. « Constraint-based type inference and parametric polymorphism ». In *First International Static Analysis Symposium*, pages 78–100, 1994.

[Agesen, 1995] O. Agesen. « The cartesian product algorithm - simple and precise type inference of parametric polymorphism ». In *Proc. ECOOP'95*, éditeur W. Olthoff, volume 952 de *LNCS*, pages 2–26. Springer-Verlag, 1995.

[Agesen, 1996] O. Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, 1996.

[Ancona *et al.*, 2000] D. Ancona, G. Lagorio et E. Zucca. « JAM — a smooth extension of Java with mixins ». In *Proc. ECOOP'2000*, éditeur E. Bertino, volume 1850 de *LNCS*, pages 154–178. Springer-Verlag, 2000.

[Ancona et Zucca, 1999] D. Ancona et E. Zucca. « A primitive calculus for module systems ». In *Principles and Practice of Declarative Programming*, pages 62–79, 1999.

[Andersen et Reenskaug, 1992] E. P. Andersen et T. Reenskaug. « System design by composing structures of interacting objects ». In *Proc. ECOOP'92*, éditeur O. L. Madsen, volume 615 de *LNCS*, pages 133–152. Springer-Verlag, 1992.

[André et Royer, 1992] 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.

[Apel *et al.*, 2005] S. Apel, T. Leich, M. Rosenmüller et G. Saake. « Combining feature-oriented and aspect-oriented programming to support software evolution ». In *Proc. of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, 2005.

[Ardourel, 2002] G. Ardourel. *Modélisation des mécanismes de protection dans les langages à objets*. Thèse d'informatique, Université Montpellier II, 2002.

[Bacon *et al.*, 1996] D. F. Bacon, M. Wegman et K. Zadeck. « Rapid type analysis for C++ ». Rapport technique, IBM Thomas J. Watson Research Center, 1996.

[Bacon et Sweeney, 1996] 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.

[Bacon, 1997] D. F. Bacon. *Fast and effective optimization of statically typed object-oriented languages*. PhD thesis, University of California, Berkeley, 1997.

[Barrett *et al.*, 1996] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford et P. T. Withington. « A monotonic superclass linearization for Dylan ». In *Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pages 69–82. ACM Press, 1996.

[Bergel *et al.*, 2003] A. Bergel, S. Ducasse et R. Wuyts. « Classboxes: A minimal module model supporting local rebinding ». In *Proc. of Joint Modular Languages Conf. (JMLC'03)*, volume 2789 de *LNCS*, pages 122–131. Springer, 2003.

[Bergel *et al.*, 2005] A. Bergel, S. Ducasse et O. Nierstrasz. « Analyzing module diversity ». *Journal of Universal Computer Science*, 11(10):1613–1644, 2005.

[Birtwistle *et al.*, 1973] G. Birtwistle, O. Dahl, B. Myhrhaug et K. Nygaard. *SIMULA Begin*. Petrocelli Charter, New York (NY), USA, 1973.

[Boehm, 1993] H. Boehm. « Space efficient conservative garbage collection ». In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28(6), pages 197–206. SIGPLAN Notices, 1993.

[Bono *et al.*, 1999] V. Bono, A. Patel et V. Shmatikov. « A core calculus of classes and mixins ». In *Proc. ECOOP'99*, éditeur R. Guerraoui, volume 1628 de *LNCS*. Springer-Verlag, 1999.

[Boucher, 1999] D. Boucher. *Analyse et Optimisations Globales de Modules Compilés Séparément*. PhD thesis, Université de Montréal, 1999.

[Boyen *et al.*, 1994] N. Boyen, C. Lucas et P. Steyaert. « Generalised mixin-based inheritance to support multiple inheritance ». Rapport Technique vub-prog-tr-94-12, Vrije Universiteit, Brussel, Belgium, 1994.

[Boyer et Moore, 1977] R. Boyer et J. Moore. « A fast string searching algorithm ». *Communications of the Association for Computing Machinery*, 20(10):762–772, 1977.

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

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

[Bracha, 1992] G. Bracha. *The Programming Language Jigsa: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[Calder et Grunwald, 1994] B. Calder et D. Grunwald. « Reducing indirect function call overhead in C++ programs ». In *ACM Symposium on Principles of Programming Languages*, pages 397–408, 1994.

[Castagna, 1997] G. Castagna. *Object-oriented programming: a unified foundation*. Progress in Theoretical Computer Science Series. Birkhaüser, 1997.

[Chailloux *et al.*, 2000] E. Chailloux, P. Manoury et B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, 2000.

[Chailloux, 1992] 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.

[Chambers *et al.*, 1997] 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.

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

[Chiba, 1995] S. Chiba. « A metaobject protocol for C++ ». In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 285–299. ACM Press, 1995.

[Chiba, 1998] S. Chiba. « Javassist — a reflection-based programming wizard for Java ». In *Proc. of ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.

[Clark et Horning, 1971] B. L. Clark et J. J. Horning. « The system language for project SUE ». In *Proceedings of the SIGPLAN symposium on Languages for system implementation*, pages 79–88, 1971.

[Clifton *et al.*, 2000] 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.

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

[Cointe, 1987] P. Cointe. « Metaclass are first class: the ObjVlisp model ». In *Proc. OOPSLA'87*, éditeur N. Meyrowitz, pages 156–167, Orlando, 1987. ACM Press.

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

[Dean *et al.*, 1995] J. Dean, D. Grove et C. Chambers. « Optimization of object-oriented programs using static class hierarchy analysis ». In *Proc. ECOOP'95*, éditeur W. Olthoff, volume 952 de *LNCS*, pages 77–101. Springer-Verlag, 1995.

[Dean et Chambers, 1994] 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.

[Desnos, 2004] N. Desnos. « Étude d'un GC pour la technique de coloration bidirectionnelle ». Mémoire de DEA, Université Montpellier II, 2004.

[Detlefs et Agesen, 1999] D. Detlefs et O. Agesen. « Inlining of virtual methods ». *Lecture Notes in Computer Science*, 1628:258–277, 1999.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

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

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

[Driesen et Hölzle, 1995] K. Driesen et U. Hölzle. « Minimizing row displacement dispatch tables ». In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 141–155. ACM Press, 1995.

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

[Driesen, 2001] K. Driesen. *Efficient Polymorphic Calls.* Kluwer Academic Publisher, 2001. Book version of [Driesen, 1999].

[Ducasse *et al.*, 2006] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts et A. P. Black. « Traits: A mechanism for fine-grained reuse ». *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[Ducournau *et al.*, 1994] R. Ducournau, M. Habib, M. Huchard et M.-L. Mugnier. « Proposal for a monotonic multiple inheritance linearization ». In *Proc. OOPSLA'94*, pages 164–175. ACM Press, 1994.

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

[Ducournau et Pavillet., 2001] R. Ducournau et G. Pavillet. « Langage à objets et logique de descriptions : un schéma d'intégration ». In *Actes LMO'2001* in *L'Objet vol. 7*, éditeurs I. Borne et R. Godin, pages 233–249. Hermès, 2001.

[Ducournau, 1991] R. Ducournau. Y3 *: le manuel de référence.* Sema Group, Montrouge, France, 1991.

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

[Ducournau, 2001a] R. Ducournau. « La coloration : une technique pour l'implémentation des langages à objets à typage statique ». Rapport Technique 01-225, LIRMM, 2001.

[Ducournau, 2001b] R. Ducournau. « Spécialisation et sous-typage : thème et variations ». Rapport Technique 01-013, LIRMM, 2001.

[Ducournau, 2002a] R. Ducournau. « "Real World" as an argument for covariant specialization in programming and modeling ». In *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc*, éditeurs J.-M. Bruel et Z. Bellahsène, volume 2426 de *LNCS*, pages 3–12. Springer-Verlag, 2002.

[Ducournau, 2002b] R. Ducournau. « Implementing statically typed object-oriented programming languages ». Rapport Technique 02-174, LIRMM, Montpellier, 2002.

[Ducournau, 2003] R. Ducournau. « La coloration : une technique pour l'implémentation des langages à objets à typage statique. ii. la coloration de méthodes et d'attributs ». Rapport de recherche 03-036, LIRMM, Montpellier, 2003.

[Ducournau, 2006] R. Ducournau. « Coloring, a versatile technique for implementing object-oriented languages ». Rapport Technique 06-001, LIRMM, Montpellier, 2006.

[Eckel et Gil, 2000] N. Eckel et J. Gil. « Empirical study of object-layout and optimization techniques ». In *Proc. ECOOP'2000*, éditeur E. Bertino, volume 1850 de *LNCS*, pages 394–421. Springer-Verlag, 2000.

[ECMA, 1999] ECMA. « ECMAScript language specification ». Standard ECMA-262, ECMA, 1999.

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

[Ernst, 2003] E. Ernst. « Higher-order hierarchies ». In *Proc. ECOOP'2003*, éditeur L. Cardelli, volume 2743 de *LNCS*, pages 303–329. Springer-Verlag, 2003.

[Fernandez, 1995] M. F. Fernandez. « Simple and effective link-time optimization of Modula-3 programs ». In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–115, 1995.

[Findler et Flatt, 1998] R. B. Findler et M. Flatt. « Modular object-oriented programming with units and mixins ». In *International Conference on Functional Programming (ICFP)*, 1998.

[Findler et Flatt, 1999] R. B. Findler et M. Flatt. « Modular object-oriented programming with units and mixins ». In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1999.

[Fred P. Brooks, 1995] J. Fred P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1995.

[Gamma *et al.*, 1995] E. Gamma, R. Helm et R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

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

[Goldberg et Robson, 1983] A. Goldberg et D. Robson. *Smalltalk: the language and its implementation.* Addison-Wesley, Reading, MA, 1983.

[Gosling, 2000] J. Gosling. *The Java Language Specification.* Addison-Wesley, Boston, 2000.

[Grove *et al.*, 1997] 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.

[Grove et Chambers, 2001] D. Grove et C. Chambers. « A framework for call graph construction algorithms ». *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[Grove, 1995] D. Grove. « The impact of interprocedural class inference on optimization ». In *CASCON'95 Centre for Advanced Studies Conference*, pages 195–203, 1995.

[Hölzle *et al.*, 1991] U. Hölzle, C. Chambers et D. Ungar. « Optimizing dynamically-typed object-oriented languages with polymorphic inline caches ». In *Proc. ECOOP'91*, éditeur P. America, volume 512 de *LNCS*, pages 21–38. Springer-Verlag, 1991.

[Horning, 1976] J. J. Horning. « Some desirable properties of data abstraction facilities ». In *Proceedings of the 1976 conference on Data: Abstraction, definition and structure*, pages 60–62, New York (NY), USA, 1976. ACM Press.

[Huchard, 2003] M. Huchard. « Classification de classes dans les approches á objets, algorithmes et applications ». Habilitation á Diriger des Recherches. LIRMM, Université Montpellier II, 2003.

[Ichisugi et Tanaka, 2002] Y. Ichisugi et A. Tanaka. « Difference-based modules: A class-independent module mechanism ». In *Proc. ECOOP'2002*, éditeur J. M. Troya, volume 2374 de *LNCS*, pages 62–88. Springer-Verlag, 2002.

[Kernighan et Ritchie, 1988] B. W. Kernighan et D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988.

[Kiczales *et al.*, 1991] G. Kiczales, J. des Rivières et D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[Kiczales *et al.*, 1997] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier et J. Irwin. « Aspect-oriented programming ». In *Proc. ECOOP'97*, éditeurs M. Aksit et S. Matsuoka, volume 1241 de *LNCS*, pages 220–242. Springer-Verlag, 1997.

[Kiczales *et al.*, 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm et W. G. Griswold. « An overview of AspectJ ». In *Proc. ECOOP'2001*, éditeur J. L. Knudsen, volume 2072 de *LNCS*, pages 327–355. Springer-Verlag, 2001.

[Leroy, 1997] X. Leroy. « The effectiveness of type-based unboxing ». In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, Juin 1997.

[Leroy, 2000] X. Leroy. « A modular module system ». *Journal of Functional Programming*, 10(3):269–303, 2000.

[Levine, 1999] J. R. Levine. *Linkers and Loaders*. Morgan-Kauffman, October 1999.

[Lieberman et Hewitt, 1983] H. Lieberman et C. Hewitt. « A realtime garbage collector based on the lifetimes of objects ». *Communication of the ACM*, 26(6):419–429, 1983.

[Limberghen et Mens, 1996] M. V. Limberghen et T. Mens. « Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems ». *Object Oriented Systems*, 3(1):1–30, 1996.

[Liskov et Wing, 1993] B. Liskov et J. Wing. « Family values: A behavioral notion of subtyping ». Rapport Technique MIT/LCS/TR-562b, MIT, 1993.

[Madsen et Møller-Pedersen, 1989] O. Madsen et B. Møller-Pedersen. « Virtual classes. a powerful mechanism in object-oriented programming ». In *Proc. OOPSLA'89*, pages 397–406, New Orleans, 1989. ACM Press.

[Meyer, 1988] 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.

[Meyer, 1991] B. Meyer. « Programming as contracting ». In *Advances in Object-Oriented Software Engineering*, éditeurs D. Mandrioli et B. Meyer, pages 1–15. Prentice Hall International, 1991.

[Meyer, 1995] 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.

[Meyer, 1997] B. Meyer. *Eiffel - The language*. Prentice-Hall, 1997.

[Meyer, 2001] B. Meyer. « Overloading vs. object technology ». *Journal of Object-Oriented Programming*, 14(5):3–7, October/November 2001.

[Mugridge *et al.*, 1991] W. B. Mugridge, J. Hamer et J. G. Hosking. « Multi-methods in a statically-typed programming language ». In *Proc. ECOOP'91*, éditeur P. America, volume 512 de *LNCS*, pages 307–324. Springer-Verlag, 1991.

[Myers, 1995] A. Myers. « Bidirectional object layout for separate compilation ». In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 124–139. ACM Press, 1995.

[Napoli, 1992] A. Napoli. *Représentations à objets et raisonnement par classification en intelligence artificielle*. PhD thesis, Université Henri Poincaré Nancy 1, 1992.

[Nystrom *et al.*, 2004] N. Nystrom, S. Chong et A. C. Myers. « Scalable extensibility via nested inheritance ». In *Proc. OOPSLA'04*, SIGPLAN Notices, 39(10), pages 99–115. ACM Press, 2004.

[Nystrom *et al.*, 2005] N. Nystrom, X. Qi et A. C. Myers. « Software composition with multiple nested inheritance ». `http://www.cs.cornell.edu/nystrom/papers.html`, 2005.

[Odersky et Wadler, 1997] M. Odersky et P. Wadler. « Pizza into Java: Translating theory into practice ». In *Proc. POPL'97*, pages 146–159. ACM Press, 1997.

[OMG, 2004] OMG. « Unified Modeling Language 2.0 superstructure specification ». Technical report, Object Management Group, 2004.

[Ossher et Tarr, 2001] H. Ossher et P. Tarr. « Using multidimensional separation of concerns to (re)shape evolving software ». *Commun. ACM*, 44(10):43–50, 2001.

[Oxhoj *et al.*, 1992] N. Oxhoj, J. Palsberg et M. I. Schwartzbach. « Making type inference practical ». In *Proc. ECOOP'92*, éditeur O. L. Madsen, volume 615 de *LNCS*, pages 329–349. Springer-Verlag, 1992.

[Palsberg et Schwartzbach, 1991] J. Palsberg et M. I. Schwartzbach. « Object-oriented type inference ». In *Proc. OOPSLA'91*, SIGPLAN Notices, 26(10), pages 77–101. ACM Press, 1991.

[Palsberg et Schwartzbach, 1994] J. Palsberg et M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[Parnas, 1972] D. L. Parnas. « On the criteria for decomposing systems into modules ». *Communication of the ACM*, 15(12):1053–1058, dec 1972.

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

[Pavillet, 2000] G. Pavillet. *Des langages de programmation par objets à la représentation des connaissances à travers le MOP : vers une intégration.* Thèse d'informatique, Université Montpellier II, 2000.

[Phillips et Shepard, 1994] G. Phillips et T. Shepard. « Static typing without explicit types ». Rapport technique, Dept. of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada, 1994.

[Plevyak et Chien, 1994] J. Plevyak et A. A. Chien. « Precise concrete type inference for object-oriented languages ». In *Proc. OOPSLA'94*, SIGPLAN Notices, 29(10), pages 324–340. ACM Press, 1994.

[Privat et Ducournau, 2004] J. Privat et R. Ducournau. « Intégration d'optimisations globales en compilation séparée des langages à objets ». In *Actes LMO'04* in *L'Objet vol. 10*, éditeurs B. Carré et J. Euzenat, pages 61–74. Hermès, 2004.

[Privat et Ducournau, 2005a] J. Privat et R. Ducournau. « Link-time static analysis for efficient separate compilation of object-oriented languages ». In *Workshop on Program Analysis for Software Tools and Engineering PASTE'05*, éditeurs M. Ernst et T. Jensen, pages 29–36, 2005.

[Privat et Ducournau, 2005b] J. Privat et R. Ducournau. « Raffinement de classes dans les langages à objets statiquement typés ». In *Actes LMO'05* in *L'Objet vol. 11*, éditeurs M. Huchard, S. Ducasse et O. Nierstrasz, pages 17–32. Hermès, 2005.

[Privat et Ducournau, 2006] J. Privat et R. Ducournau. « Multiple inheritance, class refinement and modules, at the light of meta-modeling ». Rapport Technique 06-010, LIRMM, Montpellier, 2006.

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

[Privat, 2006] J. Privat. « PRM—the language. 0.2 ». Rapport Technique 06-029, LIRMM, Montpellier, 2006.

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

[Pugh et Weddell, 1993] W. Pugh et G. Weddell. « On object layout for multiple inheritance ». Rapport Technique CS-93-22, University of Waterloo, 1993.

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

[Raynaud et Thierry, 2001] O. Raynaud et E. Thierry. « A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests ». In *Proc.*

ECOOP'2001, éditeur J. L. Knudsen, volume 2072 de *LNCS*, pages 165–180. Springer-Verlag, 2001.

[Rossie et Friedman, 1995] 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.

[Rossum et Fred L. Drake, 2003] G. van Rossum et J. Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, Sept 2003.

[Ryder, 1979] B. Ryder. « Constructing the call graph of a program ». *IEEE Transaction on Software Engineering*, 5(3):216–225, 1979.

[Saint-Exupéry, 1939] A. de Saint-Exupéry. *Terre des hommes*. Gallimard, Paris, 1939.

[Serrano, 1994] M. Serrano. *Vers une compilation portable et performante des langages fonctionnels*. PhD thesis, Universitè Paris 6, 1994.

[Shivers, 1988] O. Shivers. « Control-flow analysis in scheme ». In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, 1988.

[Shivers, 1991] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[Silberschatz et Peterson, 1988] éditeurs A. Silberschatz et J. L. Peterson. *Operating systems concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[Sonntag et Boutet, 2004] B. Sonntag et J. Boutet. *Lisaac V.0.2 - Programmer's Reference Manual*. LORIA, 2004.

[Srivastava, 1992] A. Srivastava. « Unreachable procedures in object-oriented programming ». *ACM Letters on Programming Languages and Systems*, 1(4):355–364, 1992.

[Steele, 1990] G. Steele. *Common Lisp, the Language*. Digital Press, second édition, 1990.

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

[Stroustrup, 2000] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, Reading (MA), USA, 2000.

[Szyperski, 1992] C. A. Szyperski. « Import is not inheritance — why we need both: Modules and classes ». In *Proc. ECOOP'92*, éditeur O. L. Madsen, volume 615 de *LNCS*, pages 19–32. Springer-Verlag, 1992.

[Szyperski, 2002] C. Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002.

[Taft et Duff, 2001] T. S. Taft et R. a. Duff. *Consolidated Ada Reference Manual: Language and Standard Libraries*. Springer-Verlag Telos, 2001.

[Takhedmit, 2003] P. Takhedmit. « Coloration de classes et de propriétés : étude algorithmique et heuristique ». Mémoire de DEA, Université Montpellier II, 2003.

[Thomas et Hunt, 2000] D. Thomas et A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[Tip et Sweeney, 2000] F. Tip et P. F. Sweeney. « Class hierarchy specialization ». *Acta Informatica*, 36(12):927–982, 2000.

[Torgersen, 2004] M. Torgersen. « The expression problem revisited—four new solutions using generics ». In *Proc. ECOOP'2004*, éditeur M. Odersky, volume 3086 de *LNCS*, pages 123–143. Springer-Verlag, 2004.

[Ungar et Smith, 1987] D. Ungar et R. Smith. « SELF: The power of simplicity ». In *Proc. OOPSLA'87*, éditeur N. Meyrowitz, pages 227–242, Orlando, 1987. ACM Press.

[Vitek *et al.*, 1992] J. Vitek, R. N. Horspool et J. S. Uhl. « Compile time analysis of object-oriented programs ». In *Fourth International Conference on Compiler Construction*, pages 237–250, 1992.

[Vitek *et al.*, 1997] J. Vitek, R. N. Horspool et A. Krall. « Efficient type inclusion tests ». In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 142–157. ACM Press, 1997.

[Wall *et al.*, 2000] L. Wall, T. Christiansen et J. Orwant. *Programming Perl, Third Edition*. O'Reilly, 2000.

[Wang et Smith, 2001] T. Wang et S. Smith. « Precise constraint-based type inference for java ». In *Proc. ECOOP'2001*, éditeur J. L. Knudsen, volume 2072 de *LNCS*, pages 99–117. Springer-Verlag, 2001.

[Wilson, 1992] P. R. Wilson. « Uniprocessors garbage collection techniques ». In *Int. Workshop on Memory Management*, volume 637 de *LNCS*, 1992.

[Wirth, 2002] N. Wirth. *The Programming Language Pascal*. Springer-Verlag, New York (NY), USA, 2002.

[Zendra *et al.*, 1997] 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.

[Zendra et Driesen, 2002] 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, Aôut 2002.

[Zendra, 2000] O. Zendra. *Traduction et optimisation globale dans les langages de classes*. PhD thesis, Université Nancy 1, 2000.

[Zenger et Odersky, 2004] M. Zenger et M. Odersky. « Independently extensible solutions to the expression problem ». Rapport Technique IC/2004/33, EPFL Lausanne, Switzerland, 2004.

Index

Ada, 25, 30, 281
    package, 28
Analyse de types, 132
    séparée, 152, 154
Appel direct, 135
Apprentissage, 11
Arbre de décision, voir BTD
Aspect, 33
Assistance, 13

BTD, 136
Boxing, voir mise en boîte

C, 12, 162, 281
C++, 25, 30, 55, 282
    édition de liens, 146
    héritage, 68
    mutilation de nom, 147
    protection, 75
CHA, 134, 169
Cache (défaut de), 167, 201, 290
Cartesian Product Algorithm, 135
Casting, 115
Class Hierarchy Analysis, voir CHA
Classe, 19, 41
    extension, voir extension de classes
    globale, 81, 87
    locale, 81, 88
    raffinement, voir raffinement
    spécilisation, 45
Clos, 34, 42, 58, 282
    héritage, 69
Code mort, 133
Coloration, 137, 153
Compilateur, 109
Compilation, 109
    indépendante, 144
    juste-à-temps, 110
    séparée, 145
Conflit
    de classes globales, 87
    de propriétés globales, 56, 92
    de propriétés locales, 66, 94
    de spécialisation, 90
Contrat, 17, 37
Contrats, 76
Contravariance, 25, 77
    des intentions, 53
Control-Flow Analysis, 134
Covariance, 25, 77
    des extensions, 46
Customization, voir Particularisation

Définition incrémentale, 85
Dépendance
    de modules, 29, 86
    externe, 167
Désignation explicite, 57, 87

Édition de liens, 146
Eiffel, 17, 25, 59, 283
    cluster, 28
    contrat, 76
    héritage, 68

protection, 73
renommage de classes, 87
renommage de propriétés, 57
Envoi de message, 20, 42, 54, 186
Espace de nom, 30
Étiquetage, 181
Exceptions, 37
déclaration, 75
Expressivité, 12
Extension de classes
ensemble des instances, 46
raffinement, voir raffinement

Facilité d'apprentissage, 11
Fichier binaire, 146

Généralisation
de classes, voir spécialisation
de propriétés globales, 93
Généricité, 25
Garbage collector, 164

Héritage, 20, 45, 91
d'implémentation, 50
d'interfaces, 112
de mixins, voir mixin
multiple, 20, 49, 65
non conforme, 50
privé, 50
répété, 129
Hiérarchie
de classes, 43
de classes d'un module, 84
de classes d'une hiérarchie de modules, 91
de modules, 83
modèle, 44

Intension, 53
Interpréteur, 109

Java, 25, 30, 31, 34, 59, 283
exceptions déclarées, 75
namespace, 28
package, 28

Liaison tardive, voir envoi de message
Ligne de produit, 27
Linéarisation, 69, 283
Lisibilité, 12

MOP, voir protocole à méta-objet
Méta-modélisation, 39
Méthode
abstraite, 73
externe, 167
interne, 186
primitive, 186
Machine virtuelle, 110
Migration d'instance, 116
Mise en boîte, 180
Mise en ligne, 135, 157
Mixin, 51
Modula, 30
Module, 3, 27, 79
principal, 85
Mutilation de nom, 147
C++, 147
PRM, 186

Name mangling, voir mutilation de nom

Objective Caml, 284
héritage, 70
Objet, 2, 19

Particularisation, 131, 135
Pascal, 17, 284
Performance, 110
Perl, 12, 284
Phase locale, 151
Polymorphisme paramétrique, voir généricité
`prmc`, 159

Propriété, 41
    générique, 42, 59
    globale, 42, 53
    locale, 42, 59
Protection, 73
    module, 30
    propriété, 73
Protocole à méta-objet, 34, 283
Python, 285
    héritage, 70

RTA, 134, 169
Réflexivité, 37
Résolution par défaut, 64
Raffinement, 33, 79
Ramasse-miettes, 164
Rapide Type Analysis, voir RTA
Reachability Analysis, 134
Redéfinition
    de propriétés, 62
    de types, 118, 127
Renommage
    de classes, 87
    de propriétés, 57
Ruby, 16, 34, 285

Sélection, 63
Sélection multiple, 37, 58, 282
Sémantique additionnelle, 53, 67, 73, 77, 95
Scripts, 16
Simple Class Set, 135
Simula, 285
Site d'appel, 54
Site d'instanciation, 81
Smalltalk, 58, 285
    Catégories de classes, 28
SmartEiffel, 140
Souplesse, 13
Sous-objet, 122
Spécialisation, 45, 89
    multiple, 49
spacename, voir espace de nom
Surcharge statique, 63

Table
    d'instance, 113
    de classe, voir VFT
    des attributs, voir table d'instance
    des méthodes, voir VFT
Tagging, voir étiquetage
Test d'égalité, 182
Thunk, 119, 176
Trait, 51
Transitivité, 48, 51
Typage, 22, 95
    dans `prmc`, 175
Type primitif
    implémentation, 179

UML, 59
Unification
    de classes globales, 90
    de propriétés globales, 57
Unique Name, 134

VBPTR, 129
VFT, 113

Inlining, voir Mise en ligne

Résumé : Cette thèse se place dans le contexte des langages à objets statiquement typés en héritage multiple à la C++ ou à la Eiffel. Elle est divisée en deux parties.

La première concerne la spécification des langages. Elle aborde et formalise les relations entre classes et propriétés en insistant sur la problématique de l'héritage multiple : *une classe peut redéfinir les méthodes héritées des classes qu'elle spécialise*. Elle propose également le mécanisme du *raffinement de classes* qui permet d'ajouter *a posteriori* de nouvelles propriétés à des classes existantes. Ce mécanisme, couplé avec une notion de *hiérarchie de modules*, est basé sur une analogie structurelle avec les relations entre classes et propriétés : *un module peut raffiner les classes importées des modules dont il dépend*. Son originalité concerne la prise en compte des dépendances multiples, des raffinements multiples et des combinaisons entre raffinement et spécialisation. Cette première partie présente également PRM, un langage qui valide notre approche de l'héritage multiple, des modules et du raffinement de classes.

La seconde partie concerne la compilation des langages à objets. Elle propose un schéma original de compilation pour les langages à objets qui est à la fois séparé — les modules sont compilés indépendamment de leurs utilisations finales — et intègre des techniques d'implémentation globales qui nécessitent la connaissance du programme dans son ensemble : analyse de types, suppression du code mort, coloration et arbres binaires de sélection. Ces techniques annulent le coût lié à l'héritage multiple et au raffinement de classes et réduisent fortement le coût du polymorphisme inhérent à tout langage à objets. L'originalité d'une telle approche réside dans l'application des techniques globales après la compilation des modules. `prmc`, le compilateur pour PRM est ensuite présenté et le schéma de compilation proposé est validé par différents benchmarks.

Mots clés : Langages orientés objets, spécialisation, héritage multiple, raffinement de classes, modules, compilation séparée, optimisation, analyse de types, coloration, arbres binaires de sélection, PRM, prmc.

Abstract: This thesis is placed in the context of statically typed languages object-oriented languages as C++ or Eiffel. It is divided into two parts.

The first one relates to language specification. It approaches and formalizes the relations between classes and properties while insisting on the problems of the multiple inheritance: *a class can redefine methods inherited from classes it specializes*. It also proposes the class refinement mechanism which makes it possible to add *a posteriori* new properties to existing classes. This mechanism, coupled with a concept of *module hierarchy*, is based on a structural analogy with the relations between classes and properties: *a module can refine imported classes from modules on which it depends*. Its originality is to take into account multiple dependencies, multiple refinements, and combinations of refinement and specialization. This first part also presents PRM, a language which validates our approach of multiple inheritance, modules, and class refinement.

The second part relates to the compilation of object-oriented languages. It proposes an original schema of compilation for object-oriented languages which is separate—the modules are compiled independently of their final uses—and includes global implementation techniques which require the knowledge of the whole program: type analysis, dead code suppression, coloring and binary tree dispatch. These techniques cancel the cost of multiple inheritance and class refinement and strongly reduce the inherent cost of polymorphism of any object-oriented language. The originality of such an approach is the application of global techniques after the compilation of modules. `prmc`, the PRM compiler, is then presented, and the schema of compilation we propose is validated by different benchmarks.

Keywords: Object-oriented languages, specialization, modules, separate compilation, optimization, type analyze, coloring, binary tree dispatch, PRM, prmc.

LIRMM — 161, rue Ada — 34392 Montpellier Cedex 5 — France