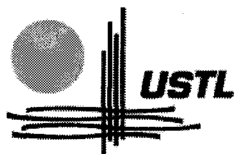
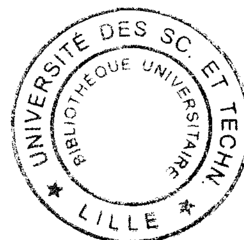
30376
2007
15

n° d'ordre : 3968



Sûreté et optimisation par les systèmes de types en contexte ouvert et contraint



mémoire présenté le 13 avril 2007

pour l'obtention du titre de

Docteur

en Sciences mathématiques (spécialité informatique)

par

Yann Hodique

Composition du jury

Président :	Jean-Marc TALBOT (Professeur - Université de Provence)
Rapporteurs :	Bertil FOLLIOT (Professeur - Université Pierre et Marie Curie de Paris VI) Thomas JENSEN (Directeur de recherche CNRS - IRISA, Rennes I) Xavier LEROY (Directeur de recherche INRIA - INRIA Rocquencourt)
Examinateur :	Ludovic HENRIO (Chargé de recherche CNRS - Université de Nice)
Directeur :	Isabelle SIMPLOT-RYL (Maître de conférences habilité - Université de Lille I)
Co-encadrant :	Gilles GRIMAUD (Maître de conférences - Université de Lille I)

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UMR 8022
UFR d'IEEA – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : 33 (0)3 28 77 85 41 – Télécopie : 33 (0)3 28 77 85 37 – email : direction@lifl.fr

Ce document a été rédigé avec GNU Emacs et mis en page avec $\LaTeX\,2_\varepsilon$.

Table des matières

Table des figures 9

Liste des tableaux 11

1 Préambule et conditions d'étude 15

1.1 Plates-formes embarquées . . . 15

1.1.1 Le système d'exploitation CAMILLE . . . 16

1.1.2 La plate-forme JITS . . . 17

1.2 Rôle des systèmes de types . . . 18

1.3 Structure du document . . . 19

1.3.1 Conception d'un système de types . . . 20

1.3.2 Typage et optimisation . . . 20

1.3.3 Le typage dans l'embarqué . . . 21

2 Introduction au typage 23

2.1 Notion générale . . . 23

2.2 Différentes approches . . . 24

2.2.1 Détermination du type . . . 24

Typage fort ou faible ? . . . 24

Typage statique ou dynamique ? . . . 24

Typage intrinsèque ou syntaxique ? . . . 25

2.2.2 Nature des types . . . 25

Étiquette constante . . . 25

Prédicats complexes . . . 26

2.3 Extensions des systèmes de types . . . 26

2.3.1 Structure des systèmes . . . 26

2.3.2 Modes d'extension . . . 27

2.3.3 Typages « exotiques » . . . 28

2.3.4 Composition de systèmes de types 29
2.4 Organisation des travaux 29

3 Exploitation d'un système de types **31**
3.1 Impact des conditions d'utilisation 31
3.2 Conception du système de types 32
3.2.1 Du choix d'un langage intermédiaire 33
3.2.2 Extensibilité et sécurité 34
3.2.3 Extensibilité et objets 35
3.2.4 Intégration des métatypes 37
3.2.5 Exemple de projection 39
3.2.6 Détails de conception 40
Structure des métatypes 40
Héritage 41
Correction par construction 42
3.3 Processus de compilation embarquée 43
3.3.1 Rôle de la compilation tardive 43
3.3.2 Flexibilité du mécanisme de liaison 44
Méthode de liaison 45
Accès maximal à la plate-forme 46
Extension de la projection Javacard 47
Exemple d'utilisation non-triviale 48
3.3.3 Intégration dans le processus de chargement 49
3.4 Liaison et sécurité 51
3.4.1 Attaques envisageables 51
3.4.2 Schéma de liaison 52
Typage 53
Droits d'accès 53
Liaison effective 56
3.4.3 Simulation de politiques de sécurité 56
3.5 Liaison et métatypes 57
3.5.1 Facteur de sécurité 57
3.5.2 Facteur d'optimisation 59
3.6 Résultats expérimentaux et possibilités d'extensions 59
3.6.1 CAMILLE et FAÇADE 59
3.6.2 Proposition d'héritage multiple 61

3.7 Consistance du mécanisme de liaison 64
3.7.1 Liaison 64
3.7.2 Preuve de cohérence vis-à-vis du système de types 66
Définitions préalables 66
Suffisance des contrôles 66
Invoke 67
Return 67
Existence des contrôles 67
3.8 Conclusion 68

4 Analyse d'alias 69
4.1 Analyse de code et interprétation 69
4.2 Présentation de l'analyse 70
4.3 Moteur d'alias 71
4.3.1 Alias et pointeurs 71
4.3.2 Définitions préliminaires 72
4.3.3 Abstraction 72
4.3.4 Algorithme 75
Analyse inter-méthodes 75
Analyse intra-méthode 77
4.4 Choix et approximations 79
4.4.1 Flot de contrôle 79
4.4.2 Problème de l'héritage 80
4.4.3 Types exacts 81
4.4.4 Dictionnaire et calcul des signatures 83
4.4.5 Objets et champs 84
4.4.6 Exceptions 85
4.5 Exemple 87
4.6 Vérification 89
4.7 Implantation 90
4.7.1 Généricité de l'implantation 90
4.7.2 Performances 91
4.7.3 Discussion sur la vérification 91
4.8 Conclusion 93

5 Analyse d'échappement 95

5.1 Présentation 95
5.2 Analyses d'alias et d'échappement 97
5.2.1 Exemple de résultat attendu 98
5.2.2 L'analyse d'échappement 99
Allocations en boucle d'objets capturés 100
Exceptions 101
5.2.3 Comparaison à d'autres analyses 102
Allocations cachées 102
Interférences avec d'autres optimisations 104
Chargement dynamique 105
5.2.4 Implantation 105
5.2.5 Résultats expérimentaux 107
5.3 Conclusion 109

6 Extensions de l'analyse d'échappement **111**
6.1 Analyse d'échappement et agrégation 111
6.1.1 Allocations dans les constructeurs 111
6.1.2 Solution 113
Objets membres 115
Cascade de constructeurs 117
Justification 118
6.1.3 Adaptation du support d'exécution 119
Comment empiler 119
Comment dépiler 120
6.1.4 Exemple détaillé 121
6.1.5 Apports de l'analyse 122
6.2 Analyse d'échappement et fabriques d'objets 123
6.2.1 Utilisation de fabriques 123
6.2.2 Solution 125
Déclenchement de la mise en pile 126
Propagation de la mise en pile 127
6.2.3 Transformation du ficher `.class` 128
6.2.4 Résultats expérimentaux 129
6.3 Conclusion 131

7 Incursion dans le monde fonctionnel **133**

7.1 Du non-fonctionnel au fonctionnel . . . 133
7.2 Cas d'étude . . . 134
7.2.1 Problématique . . . 134
7.2.2 Concrétisation du test exhaustif . . . 135
7.2.3 Utilisation des modes d'accès . . . 136
7.2.4 Détails de conception . . . 137
7.3 Résultats expérimentaux . . . 137
7.4 Conclusion . . . 138

8 Conclusion **141**
8.1 Contributions . . . 142
8.1.1 Design de systèmes de types . . . 142
8.1.2 Conception d'extensions . . . 142
Optimisation . . . 142
Fiabilité . . . 143
8.1.3 Bilan . . . 143
8.2 Perspectives . . . 144

Bibliographie **147**

Table des figures

2.1 Ensembles d'objets et treillis de types. 27

3.1 Génération des niveaux d'abstraction 36
3.2 Classification des objets 38
3.3 Transformation des systèmes de types et du code. 46
3.4 Invariant de classe. 49
3.5 Déroulement du processus de liaison 50
3.6 États de chargement. 51
3.7 Attaque basique. 52
3.8 Attaque indirecte. 53
3.9 Schéma de liaison. 54
3.10 Autorisation et liaison effective. 55
3.11 Hiérarchie des métatypes 61
3.12 Héritage problématique 63
3.13 Unification des types 64
3.14 Mécanisme de chargement 65

4.1 Suivi de liens 72
4.2 Chargement dynamique et héritage. 80
4.3 Exemple d'implantation. 84
4.4 Exemple de gestion d'exception. 85
4.5 Code « malicieux ». 86
4.6 Exemple d'analyse d'alias. 87
4.7 Exécution de l'algorithme. 88

5.1 Alias. 98
5.2 Allocation cachée. 102
5.3 Bytecode correspondant. 103

5.4 Expansion du code. 104

6.1 Exemple d'API : StringBuffer. 112
6.2 Exemple d'API : utilisation de StringBuffer. 113
6.3 Possibilité d'allocation en pile 114
6.4 Impossibilité d'allocation en pile 115
6.5 Exemple d'un conteneur. 116
6.6 Constructeurs imbriqués. 118
6.7 Exemple d'agrégation. 122
6.8 Exemple d'utilisation de fabrique. 124
6.9 Récupération d'objets en dehors des fabriques. 126
6.10 Exemple. 129
6.11 Labyrinthes. 130

7.1 Algorithme des calculs de signatures. 139
7.2 Algorithme de vérification. 140

Liste des tableaux

3.1 Règles d'exécution. 67

4.1 Extrait de sémantique pour le bytecode Java. 78
4.2 Sémantique du bytecode `invokeVirtual`. 82

5.1 Résultats obtenus sur les jeux de test. 110

Remerciements

Comme le savent ceux qui « en sont passés par là », la thèse de doctorat est un exercice propice au sentiment de solitude. Quelle que soit l'aide et les conseils reçus, en définitive on reste (le plus souvent) seul dans sa tête avec cette petite voix intérieure, bien trop souvent sarcastique, qui pousse à la productivité et s'offusque de tout immobilisme. Cette épreuve, pour personnelle qu'elle soit, a néanmoins le mérite de faire prendre conscience de l'extraordinaire importance de la myriade de petites choses qui aident à la surmonter. C'est tout le sens de ces remerciements qui, bien que relevant de la pratique courante, sont aussi sincères que primordiaux.

En premier lieu je tiens à remercier Bertil FOLLIOT, Thomas JENSEN et Xavier LEROY pour avoir accepté de rapporter ce travail et m'avoir permis d'y apporter une touche finale par de nombreuses remarques constructives et une extrême précision. Merci également à Ludovic HENRIO pour avoir accepté de prendre place parmi les membres du jury, et à Jean-Marc TALBOT pour avoir traversé la France à seule fin de présider ledit jury.

Je tiens également à exprimer toute ma gratitude à mes encadrants, Isabelle SIMPLOT-RYL et Gilles GRIMAUD, pour m'avoir fait confiance et m'avoir permis de jouir d'une liberté appréciable dans le choix de l'orientation de mes travaux. Merci également pour leur disponibilité, enviée par beaucoup, pour leurs suggestions, source éternelle de nouveaux développements, et simplement pour leurs qualités humaines qui m'ont permis d'effectuer cette thèse dans une ambiance aussi saine et détendue que possible.

Merci à toutes les personnes qui ont pris la peine de relire tout ou partie de ce document, à la recherche d'imprécisions orthographiques ou sémantiques, et tout particulièrement à Mireille CLERBOUT, qui a eu la gentillesse de mettre sa rigueur au service du contenu de cette thèse.

Mes divers collègues de bureau ne sont pas oubliés, eux qui ont eu à supporter les différentes phases de mon évolution, depuis l'enthousiasme des débuts jusque la déprime rédactionnelle, en passant par les moments de stress en période de soumission d'article ou d'euphorie consécutive à la traque couronnée de succès d'un « bug » particulièrement vicieux. Merci donc à Arnaud BAILLY, Mirabelle NEBUT et Dorina GHINDICI pour leur gentillesse et leur compréhension. Puissent-ils me pardonner un jour...

À la liste des personnes sans lesquelles cette thèse aurait été bien morne s'ajoutent bien entendu ceux que j'ai côtoyés durant ces quelques dernières années. Merci à David SIMPLOT-RYL pour m'avoir accueilli dans une équipe dynamique et sympathique alors

que j'étais un mathématicien repenti à la recherche d'une nouvelle patrie. Merci à Nadia BEL-HADJ-AISSA *(courage Nadia, c'est bientôt fini), Antoine* GALLAIS *(pas mieux), Kevin* MARQUET *(décidément, c'est une épidémie), Alexandre* COURBOT *(ah non, lui c'est fait), pour les moments de détente que nous avons partagés, les crises de fou rire au restaurant universitaire (qui ont dû mettre à mal l'image du personnel enseignant que nous représentions), et aussi pour le soutien moral que vous avez représenté.*

Une pensée et un grand merci à tous nos chers disparus... François INGELREST*, exilé en Suisse pour raisons post-doctorales, Damien* DEVILLE*, porté disparu dans le triangle de l'Industrie, Julien* CARTIGNY*, parti prêcher la bonne parole quelque part dans le Limousin... et j'en oublie très probablement.*

Peut-être un peu étrangement, je voudrais aussi remercier toutes les personnes qui n'ont jamais vraiment compris ce en quoi consistait la thèse en général, et la mienne en particulier, pour m'avoir fourni d'excellentes occasion de relativiser l'importance globale de ce qui a tout de même constitué une bonne approximation de ma vie pendant plus de trois ans.

Sur un plan plus personnel, je tiens à remercier ma famille pour son soutien indéfectible, et bien sûr Marion, pour avoir été le centre stable d'un univers pour le moins chaotique depuis bientôt huit ans. Je ne trouverai pas de mots pour exprimer ce que je ressens, mais le sentiment est plus que sincère.

Oh, et bien sûr... merci à vous, lecteur, d'avoir supporté jusqu'ici mon humour douteux et mes tournures alambiquées. Sachez néanmoins que le pire reste à venir...

Chapitre 1

Préambule et conditions d'étude

L'intérêt pour les systèmes embarqués a connu un net essor avec la généralisation de l'utilisation d'équipements électroniques dans les activités quotidiennes. Certains de ces systèmes sont spécifiquement conçus pour la gestion de plates-formes de petite (voire très petite) taille : cartes à puces, étiquettes électroniques, ordinateurs de poche... Ces divers matériels ont en commun une limitation significative des ressources disponibles, tant en énergie qu'en stockage ou en puissance. Les systèmes d'exploitation qui leur sont destinés se doivent donc de prendre en compte ces limitations et d'être les plus légers possibles. Parallèlement, l'évolution rapide des besoins des utilisateurs de ces équipements impose une capacité d'adaptation croissante des systèmes, et les domaines dans lesquels ils sont employés exigent bien souvent une attention importante quant à la sécurité des matériels et des données manipulées (téléphonie mobile, cartes bancaires...).

Des travaux récents [DGGJ03] mettent en avant la possibilité d'opter pour des modèles hérités de l'ingénierie logicielle (composants, évolution dynamique par chargement tardif) pour répondre au besoin d'extensibilité, en les adaptant pour les rendre compatibles avec le manque de ressources disponibles. Néanmoins, la sécurité des systèmes ne doit en aucun cas être remise en cause. Les travaux présentés dans cette thèse s'articulent autour de ces trois préoccupations : fiabilité, extensibilité et performance, en soulevant le problème de la conciliation de ces différents objectifs.

1.1 Plates-formes embarquées

Il existe de nombreux systèmes dédiés à l'embarqué, certains totalement dédiés au matériel qui les supporte (comme les systèmes pour carte à puce classiques), d'autres suffisamment génériques pour être déployés sur un éventail plus ou moins important de matériels. La plupart ont en commun une préoccupation importante concernant la sécurité, héritée directement du fait que ces systèmes sont généralement destinés à contenir des informations d'un niveau de confidentialité élevé. Par ailleurs, comme il existe peu de protection au niveau du matériel lui-même (MMU par exemple), toute la sécurité est

déportée au niveau du système, ce qui justifie un soin accru apporté à la conception de ces logiciels.

Deux plates-formes privilégiées sont employées dans cette étude, servant à illustrer des aspects complémentaires de la conception de systèmes et d'applications pour l'embarqué. Il s'agit d'une part du système d'exploitation CAMILLE, et d'autre part de l'environnement Java JITS, tous deux développés au sein du LIFL et visant un déploiement sur des matériels de petite taille comme les cartes à puce.

Le choix de ces plates-formes trouve naturellement une première justification dans l'accessibilité très aisée à ces systèmes du fait de leur création et développement au sein même de notre équipe de recherche. Mais surtout nous disposons par l'intermédiaire de ces deux plates-formes d'une combinaison intéressante de deux environnements très distincts et complémentaires :

- d'un côté JITS, un environnement Java, qu'on peut considérer comme incontournable dans le domaine de l'embarqué – il ne peut être qu'utile de tenter de tirer le meilleur parti d'une implantation d'une plate-forme aussi répandue ;
- de l'autre, le système d'exploitation CAMILLE, qui par son état de développement expérimental offre une grande liberté, et ce d'autant plus qu'il présente l'avantage d'être suffisamment ouvert pour permettre d'explorer des designs qui s'écartent des voies classiques.

Pour ces deux plates-formes, nous présenterons des extensions visant à améliorer un ou plusieurs de nos trois critères clefs. En ce qui concerne JITS, comme la compatibilité Java doit être assurée, les travaux présentés relèveront essentiellement de l'optimisation, puisque aussi bien la politique de sécurité de la plate-forme que ses possibilités d'extension sont globalement figées par les spécifications de Java. En revanche, pour CAMILLE, nous partirons d'un système conçu pour être performant, et verrons comment le modifier pour le faire bénéficier d'une grande extensibilité et d'une sécurité très stricte.

1.1.1 Le système d'exploitation Camille

CAMILLE est un système d'exploitation extensible [DGGJ03] spécifiquement conçu pour fonctionner sur des plates-formes aux ressources limitées, fortement axé sur les cartes à puce. L'architecture du système est basée sur un exo-noyau [Eng99], dont le principe fondateur est de ne pas imposer une quelconque abstraction du matériel au niveau du noyau, mais au contraire de conserver les spécificités apparentes à un niveau relativement haut, le noyau lui-même ne devant qu'assurer la gestion immédiate des ressources. CAMILLE procure un accès sécurisé à différentes ressources logicielles et matérielles utilisées par le système, telles que le processeur, l'ensemble des pages mémoire, la mémoire de code, *etc.* et autorise (ou non) les applications à gérer directement ces ressources par le biais de mécanismes d'une grande flexibilité.

D'autres systèmes offrent également un niveau élevé de flexibilité, à la manière de Think [FSLM02, Fas01, RS02] en autorisant une définition très générique et souple de ce qu'est la liaison entre deux composants d'un système, ou encore VVM [FPR98] en

permettant l'adaptation de tout ou partie de la chaîne d'exécution.

Toutefois le modèle d'architecture en exo-noyau est particulièrement bien adapté pour les plates-formes du type des cartes à puce. En effet, en partant d'un noyau minimal et en y adjoignant uniquement le minimum d'abstractions nécessaires il devient possible de construire des systèmes de très petite taille dont les services actifs sont constitués uniquement des éléments réellement utiles pour les applications. De plus, la flexibilité inhérente à ce modèle permet l'adjonction dynamique de nouvelles applications (et des services éventuellement manquants dont elles dépendent) non prévues ou écartées lors de la conception du système initial. Ainsi, il est plus simple de répondre à l'évolution des besoins de l'utilisateur, en ajoutant ce qui manque plutôt qu'en refondant un système complet à partir de rien. Par ailleurs, l'absence d'abstraction « standard » permet de spécialiser l'écriture des interfaces aux besoins particuliers des applications, et ainsi d'exploiter au mieux les ressources disponibles.

Les composants constitutifs de CAMILLE se distribuent entre le monde extérieur (hors-ligne) et le domaine propre du système embarqué (en ligne). Les composants système ainsi que les applications les utilisant peuvent être écrits dans un certain nombre de langages, dédiés ou génériques (Java, C...) suivant la disponibilité des compilateurs correspondants. Le code est ensuite compilé en une forme intermédiaire spécialisée appelée FAÇADE [GLV99] (par le biais d'un convertisseur de bytecode Java, ou d'une adaptation de GCC par exemple).

1.1.2 La plate-forme Jits

L'environnement JITS quant à lui, a pour objectif de fournir un environnement Java complet permettant le déploiement sur des systèmes très contraints. Au contraire de Javacard, qui fournit des services arbitrairement dégradés par rapport à Java de façon à se conformer aux exigences de taille de ces systèmes, JITS s'intéresse à la spécialisation tardive du code de façon à pouvoir ne déployer que le strict nécessaire tout en évitant d'avoir à décider *a priori* ce qui dans l'environnement Java standard est utile pour le système et ce qui ne l'est pas.

Dans le cadre de cette thèse, JITS est moins utilisé pour ses caractéristiques propres que pour l'accès qu'il fournit à un environnement Java adaptable aux besoins de l'étude. Néanmoins, le soin apporté à l'élaboration d'une gestion souple et optimisée de la mémoire nous permet de nous reposer sur des mécanismes bas-niveau existants qu'il serait fastidieux de transposer dans une autre machine virtuelle. De même, les spécificités du mécanisme de chargement (faisant appel lui-même à du code Java) ont contribué à rendre plus facile la mise en oeuvre des aspects étudiés.

Aussi, sans qu'ils soient véritablement propres à la plate-forme JITS, nos travaux sur les environnements Java bénéficient amplement des structures spécifiques introduites dans JITS.

1.2 Rôle des systèmes de types

Au cours de cette étude, nous avons choisi d'aborder les questions de sécurité, extensibilité et performance sous un angle commun. Traditionnellement ces trois questions sont traitées de façon séparée, pour ne pas dire antagoniste : en effet, dans de nombreux cas, la focalisation sur une partie de ces aspects s'effectue au détriment d'une autre :
- l'exemple des cartes bancaires pourrait être typique d'un système sécurisé, mais pas extensible ;
- les systèmes à base de micro-noyaux, réputés pour leur sécurité et leur extensibilité, sont souvent critiqués pour leurs faibles performance ;
- lorsque la priorité est donnée à l'efficacité, il est fréquent de recourir à des langages bas-niveau (tels que le C), qui rendent plus ardu le respect des contraintes de plus haut niveau telles que la sécurité.

On peut notamment voir à travers ces exemples l'illustration de ce qui est présenté dans [KdRB91] comme un paradoxe de la programmation de haut niveau : le fait d'exprimer à plus haut niveau la sémantique d'un programme a tendance à provoquer la génération d'un code moins efficace, alors même que les idées y sont exposées plus clairement, et qu'il devrait donc être possible, à l'aide de transformations suffisamment « intelligentes » de trouver le plus court chemin menant au résultat final.

Bien sûr l'introduction dans ces langages de concepts destinés à fiabiliser la conception des logiciels (contrôle d'accès à la mémoire, mécanismes de reprise sur erreur par exceptions,...) ajoute dans un premier temps une complexité de traitement non négligeable. Il n'en demeure pas moins que deux programmes *corrects* exécutant la même fonction devraient pouvoir afficher des performances comparables, indépendamment des langages utilisés. En d'autres termes, il serait souhaitable que les langages de haut niveau n'engendrent pas une dégradation de performance supérieure à celle qui est nécessaire à la prise en charge de la complexité intrinsèque des problèmes (le contrôle de la mémoire et la reprise sur erreur entrant dans cette dernière catégorie). Il s'agit donc là d'une distinction qu'il conviendrait de faire entre programmation de haut niveau et programmation défensive.

L'opposition entre extensibilité et sécurité quant à elle est moins flagrante, mais il n'en demeure pas moins que la sécurité tend à cloisonner les systèmes, ce qui va rarement de paire avec une extensibilité non contrainte.

Nous constatons donc que dans l'état actuel des choses, il est pour le moins difficile de concilier les préoccupations fondamentales des systèmes contraints, et qu'il y a un véritable besoin, pour concevoir les logiciels destinés à ces systèmes, d'une plate-forme mettant en valeur à parts égales ces considérations. Nous abordons ici ce problème crucial à travers la conception et l'utilisation des propriétés de systèmes de types, car ces derniers sont fortement liés aux trois préoccupations majeures mentionnées précédemment : la fiabilité, l'extensibilité et la performance dans les systèmes contraints.

Les systèmes de types sont bien sûr primordiaux pour les aspects sécuritaires des plates-formes considérées, dans la mesure où la propriété de typage est généralement peu coûteuse

à établir tout en fournissant une garantie intéressante de sûreté des programmes. Ainsi, la plupart des systèmes requérant une certaine garantie de sécurité de la part des applications qu'ils exécutent imposent comme préalable à leur exécution la vérification d'un typage correct. Cela est notamment le cas pour les machines virtuelles comme Java.

De même, l'extensibilité des systèmes passe en partie par les possibilités d'extension des systèmes de types mis en jeu, par exemple à travers les modèles de typage basés sur une conception orientée objet de la programmation. Nous verrons comment tirer le meilleur parti de cette extensibilité et comment la rattacher à la notion de fiabilité afin qu'au lieu de s'opposer (il est généralement considéré comme plus aisé d'apporter des garanties sur un système « figé ») ces deux notions se renforcent.

Enfin, en ce qui concerne la performance, nous verrons que les systèmes de types ont également un rôle important à jouer, en tant que moyen de définir ou d'altérer la gestion des ressources des plates-formes. Dans ce domaine, les types manipulés sont quelques peu différents des types « structurels » communément employés dans les langages de programmation, et relèvent davantage de l'inférence, ce qui les rend peu visibles aux yeux du programmeur. Ces types seront essentiellement présentés à travers des représentants particuliers des « types d'effet ».

Dans un contexte extrêmement contraint ces notions prennent une connotation particulière, et nous verrons tout au long du document intervenir des considérations d'économie d'espace et de temps de calcul, qui imposerons une forme particulière aux systèmes de types que nous serons en mesure d'utiliser.

1.3 Structure du document

Dans la mesure où les types jouent un rôle central dans cette étude, il nous semble utile de revenir au chapitre 2 sur les différentes déclinaisons de la notion de type. Ce tour d'horizon des systèmes de types nous permettra de choisir en connaissance de cause les modèles de typage, avec leurs avantages et inconvénients, que nous adopterons pour aborder les problèmes propres aux systèmes embarqués contraints.

Par la suite, nous étudierons le typage selon deux points de vue complémentaires, se situant en amont et en aval de l'écriture des programmes. En effet le typage intervient doublement dans les programmes, en déterminant d'une part la manière de les écrire (ce qui est recouvre la notion familière au programmeur) et d'autre part celle de les interpréter (ce qui correspond au travail effectué par les compilateurs, interpréteurs et autres optimiseurs). Dans cette dernière approche, il est clair que la notion de typage peut s'écarter beaucoup du type structurel classique pour n'être en fin de compte qu'un moyen de regrouper différents objets suivant une ou plusieurs propriétés communes.

1.3.1 Conception d'un système de types

En amont (chapitre 3), nous nous intéresserons à la conception d'un système de types lié à un langage particulier (FAÇADE), que nous pensons adapté aux contraintes des petits systèmes embarqués ainsi qu'aux préoccupations d'expressivité propres aux logiciels que nous désirons déployer sur ces plates-formes.

L'emploi d'un mécanisme de liaison flexible mais néanmoins strictement contrôlé entre composants, combiné à un système de types fortement structuré, permet d'obtenir une extensibilité satisfaisante et une grande efficacité du code décrit en l'utilisant, sans pour autant laisser de côté les aspects sécuritaires incontournables. En ce sens, le langage FAÇADE constitue une base intéressante pour répondre à notre triple problématique.

Cette partie s'appuiera sur la plate-forme CAMILLE introduite en section 1.1.1 et présentera l'intégration de plusieurs niveaux d'abstraction permettant d'exprimer, à travers le typage des objets d'un système, le moyen de les utiliser de manière fiable et efficace. En outre, le système de types présenté sera fortement orienté vers l'extensibilité et la possibilité d'exprimer simplement différentes manières d'appréhender les modèles objet, telles qu'exposées précédemment.

1.3.2 Typage et optimisation

En aval, c'est un typage essentiellement *a posteriori* des applications qui nous intéressera, afin d'en apprendre autant que possible sur le code considéré, et de pouvoir adapter du mieux possible le support d'exécution à ce code. De la sorte, certains critères d'optimisation, dont l'écriture et/ou la détermination se révélerait trop complexe pour le programmeur, pourront être appliqués. Typiquement, ces critères sont différents de ceux pour lesquels il est possible de se fier à un compilateur extérieur : les optimisations en question touchent à des composants système bien trop centraux pour qu'il soit permis à un acteur externe de les prendre en charge.

On étudiera en particulier les exemples des régions d'allocation et des modes d'accès à un objet. La faculté d'allouer dans des zones proches (en mémoire) des objets présentant des propriétés similaires est particulièrement importante dans les systèmes embarqués présentant différents types de mémoires pour des usages particuliers. L'illustration d'une utilisation conjointe des régions d'allocation et des modes d'accès peut être par exemple l'allocation d'un objet dans un type de mémoire plus adapté à la lecture qu'à l'écriture, si l'on sait que cet objet est amené à rester constant (on pourra notamment considérer l'utilisation d'une mémoire de type EEPROM, voire FLASH).

Dans l'ensemble des analyses décrites dans ce document, certaines sont déjà bien connues et largement étudiées. Néanmoins l'approche suivie ici privilégie toujours l'économie maximale de calculs, à cause des contraintes avec lesquelles nous devons composer. C'est pourquoi toutes ces analyses sont avant tout conçues pour être très peu coûteuses, une fois que l'analyse commune sous-jacente (baptisée « analyse d'alias », *cf.* chapitre 4) est effectuée.

Le typage des objets suivant les régions dans lesquelles ils devraient être alloués nous fournit la possibilité d'exprimer simplement des propriétés particulièrement intéressantes concernant notamment l'analyse d'échappement, qui donne l'opportunité au système de regrouper géographiquement en mémoire des objets ayant des durées de vie similaires, de sorte que la suppression de ces objets à la fin de leur vie soit la moins coûteuse possible.

Dans le contexte des systèmes embarqués de petite taille, la gestion de la mémoire revêt une importance telle que certains travaux se concentrent sur la possibilité de borner statiquement la quantité de mémoire utilisée par un programme [CJPS05]. Bien évidemment, dans le cas général le calcul d'une telle borne est impossible, aussi est-il nécessaire de recourir à un gestionnaire interne de la mémoire. De ce fait, l'analyse d'échappement peut constituer un facteur non négligeable d'optimisation de l'organisation de la mémoire.

Nous verrons qu'il est possible, par des traitements très légers sur les résultats de l'analyse d'alias, d'obtenir des résultats intéressants d'analyse d'échappement (chapitre 5) et même d'étendre le champ d'action de cette dernière à des structures qui la rendent traditionnellement inopérante (chapitre 6).

Quant à la notion de mode d'accès, elle nous permet d'aborder le domaine particulier de la frontière entre un typage correct et une fonctionnalité correcte à travers l'exemple de la vérification d'un composant système central. Sur l'exemple de l'ordonnancement de tâches dans le domaine du temps réel, nous montrerons que la distance entre analyse statique de code et preuve de propriété fonctionnelle peut dans certains cas être comblée assez facilement (chapitre 7).

Bien que n'étant pas eux-mêmes directement axés sur la sûreté de fonctionnement du système, ces derniers travaux présenteront néanmoins la propriété fondamentale de ne pas remettre en cause la sûreté établie par ailleurs. Ainsi par exemple, dans le cas de la modification de la stratégie des objets en mémoire, il est bien évidemment d'une importance capitale que ce changement ne rende pas inconsistant l'état de la mémoire.

1.3.3 Le typage dans l'embarqué

Au cours de cette étude, nous fournirons un certain nombre d'éléments de réponse à la question problématique de la conciliation des préoccupations précédemment énoncées, dans le cadre des systèmes embarqués contraints. À travers l'utilisation de systèmes de types bien choisis, nous verrons qu'il est possible aussi bien de concevoir *a priori* que d'exploiter *a posteriori* les programmes de telle sorte que l'ensemble des contraintes imposées se résument globalement à un problème bien connu de typage.

Chapitre 2

Introduction au typage

La notion de typage est aujourd'hui centrale au sein des processus d'ingénierie logicielle. À l'exception notable des programmes développés en langage machine ou assembleur, il s'agit d'une abstraction basique employée dans la plupart des langages de programmation, et déclinée sous de nombreuses formes.

2.1 Notion générale

L'objectif du typage est de diviser en classes d'équivalences l'ensemble des structures manipulées par un programme : pour chaque objet[1] manipulé par le système et chaque type considéré, l'objet l'appartient à ce type ou pas. Du point de vue le plus général, chaque type définit donc deux classes d'objets pour l'ensemble du système.

Les types sont principalement employés pour contrôler l'utilisation des objets : en effet un type, lorsqu'il est associé à un objet, fournit une indication sur les opérations légalement applicables, ce qui autorise un contrôle grossier de la bonne marche d'un programme.

Ce contrôle est de nature non-fonctionnelle dans la mesure où les types représentent une abstraction des données véritablement manipulées par le programme, de sorte qu'ils ne peuvent suffire, dans le cas général, à garantir un fonctionnement correct vis à vis des résultats obtenus. Néanmoins, il s'agit d'un indice de fiabilité important, et son domaine d'utilisation est assez vaste.

Ce contrôle tire sa justification initiale dans les différences structurelles exposées par les objets du système, et notamment des différences de taille entre objets : on ne saurait *a priori* appliquer une opération manipulant des entiers codés sur 4 octets à des caractères codés sur 1 octet sans générer d'effets de bords potentiellement néfastes.

La notion de typage est naturellement liée à celle de vérification : les types calculés pour l'ensemble des données doivent être mis en cohérence pour obtenir la garantie de typage

[1] on considérera une acception simple du terme « objet », celui-ci désignant toute donnée simple ou complexe accessible à un programme.

correct au niveau du programme. Si, pour une raison quelconque, un objet ne respecte pas la contrainte de type imposée par le programme dans lequel il évolue, alors ce dernier est considéré comme erroné car sortant du cadre défini pour une exécution correcte. Aussi la vérification des types prélude-t-elle à l'exécution du code vérifié.

2.2 Différentes approches

Outre les programmes non typés, qui ne nous intéresseront pas ici (dans la mesure où l'absence totale de vérification représente nécessairement un niveau de sécurisation inférieur), il existe de nombreuses déclinaisons des systèmes de types, chacune adaptée à une préoccupation particulière. Le contexte de cette étude, fortement liée aux systèmes embarqués contraints, aura nécessairement une influence majeure sur les choix à faire pour concevoir un système de types adapté aux conditions souhaitées.

2.2.1 Détermination du type

Typage fort ou faible ?

La principale divergence au niveau des types se situe au niveau du choix entre un modèle de typage « fort » ou un modèle de typage « faible » [MMMP90]. Notons que les définitions associées à ces termes varient sensiblement d'un auteur à l'autre, probablement du fait de la connotation positive du typage fort par rapport au typage faible.

Dans certains travaux, le typage fort est considéré comme l'apport d'une garantie d'intégrité des données et des moyens d'y accéder. Le typage faible, quant à lui, recouvre une notion d'annotation de type assez arbitraire, et est très inadapté au calcul de propriétés d'un programme puisqu'il n'apporte quasiment aucune information fiable à son sujet.

Typage statique ou dynamique ?

Le typage statique privilégie une vérification de types se déroulant le plus tôt possible dans la chaîne de déploiement du programme (durant la phase de compilation), alors que le typage dynamique la déporte tardivement, près de l'exécution. Les termes choisis peuvent être source d'ambiguïté car le développement du chargement dynamique de code tend à introduire des opportunités de typage pendant l'exécution. Peut-être serait-il préférable de parler de typage au déploiement et de typage à l'exécution.

Schématiquement, on peut voir le typage statique comme une approche défensive visant à rejeter tout code présentant la moindre ambiguïté, alors que le typage faible laisse toujours une chance à la chaîne d'exécution de produire les conditions adéquates pour une exécution correcte, provoquant une erreur uniquement lorsque les conditions pour l'instruction courante ne sont pas remplies.

Toutefois, le typage statique n'est pas nécessairement facteur d'une fiabilité supérieure, certains langages statiquement typés (comme le C) autorisant le programmeur à servir de superviseur et à forcer le typage dans la direction qui lui convient (ce qui peut évidemment être en inadéquation avec les conditions réelles d'exécution). Néanmoins, dans une optique dans laquelle le déclenchement d'erreurs lors de l'exécution est inutile, celles-ci n'étant pas rattrapables, le choix d'un typage statique semble cohérent.

Typage intrinsèque ou syntaxique ?

Le typage d'une application peut être partiellement laissé à la discrétion du programmeur par l'intermédiaire d'annotations (lors des déclarations de variables, comme en Java) ou au contraire être inféré par un compilateur ou un interpréteur. Dans le premier cas nous parlerons de type syntaxique, et dans le second de type intrinsèque.

Notons que l'utilisation de types intrinsèques est naturellement incontournable pour les langages faiblement typés, tels Smalltalk [GR83] ou Python. À l'inverse, comme évoqué auparavant, de nombreux langages fortement typés font usage de typage explicite, tels que le C et ses dérivés. Néanmoins, avec le typage explicite vient la possibilité d'autoriser certaines formes de coercition de types (transtypage) qui peuvent affaiblir considérablement la robustesse du système. Il existe cependant quelques langages fortement typés faisant appel à l'inférence de types, comme Caml [WL99, LDGV07] : pour ceux-ci, l'inférence de types est intégralement menée avant l'exécution du programme et le programmeur ne doit pas intervenir dans son calcul.

Il existe enfin la possibilité de combiner les deux approches pour obtenir le meilleur compromis suivant la situation : le langage Lisp permet par exemple de donner des indications aux compilateurs, qui soient plus précises que celles qu'il est en mesure d'inférer, ceci à des fins d'optimisation. De fait il est même possible d'avoir recours à un typage fort, voire même coercitif par endroits, en désactivant les contrôles dynamiques.

2.2.2 Nature des types

Outre la façon de garantir un typage correct pour une application, la question se pose de savoir véritablement quelle forme prend la relation entre un type et l'objet qu'il représente.

Étiquette constante

Une première possibilité consiste en l'attribution d'un type unique à chaque objet, en général lors de sa création. Tout objet est construit à partir d'un type donné, et c'est ce type qu'il conservera jusqu'à sa destruction. Toute conversion d'un type vers un autre donne en fait lieu à la création d'un nouvel objet du type voulu, exception faite des mécanismes généralement autorisés par l'héritage, qui permettent de considérer un objet comme représentant d'un super-type.

Cette approche est caractéristique des langages orientés objet, puisque le type est alors fondamentalement lié à la structure de l'objet par construction (typage par classe). Il demeure néanmoins possible dans certains langages de considérer un objet comme représentant de plusieurs types, par exemple à l'aide du typage par interface [Dah92] – notamment possible en Java –.

Dans les deux cas, le type joue pour l'objet considéré le rôle d'une étiquette, laquelle n'est pas amenée à évoluer au cours de la vie de l'objet.

Prédicats complexes

Une vision plus générale et complexe des types peut être de considérer ceux-ci comme ayant pour seule contrainte de remplir leur rôle de discriminants sur l'ensemble des objets du système. Un type équivaut alors simplement à un prédicat séparant les objets en deux classes d'équivalences.

Dans une telle vision, rien n'empêche un objet de remplir les conditions requises par un nombre quelconque de types, ce qui nous rapproche de la notion d'interface évoquée précédemment. En revanche, rien n'oblige *a priori* un objet à être construit sur la base de la connaissance des types qu'il représente, et la « création » d'un type (ou plutôt la prise en compte de celui-ci) n'est absolument pas en corrélation avec la création des objets.

Cette approche correspond globalement aux habitudes des langages fonctionnels, pour lesquels l'usage est davantage de vérifier qu'un objet possède certaines propriétés plutôt que de les lui garantir par construction.

Les prédicats de typage peuvent prendre également différentes formes. Pour exprimer l'équivalent des types considérés dans les systèmes à objets, des prédicats portant sur la structure des objets (nombre et nature des champs, opérations disponibles) sont suffisants. Mais dans le cas général, il n'y a pas de raison particulière d'imposer des contraintes sur les éléments constitutifs du prédicat, aussi est-il tout à fait envisageable d'écrire un prédicat de type qui tienne compte d'une valeur par exemple.

2.3 Extensions des systèmes de types

La plupart des langages de programmation modernes offrent la possibilité au programmeur d'étendre le domaine des types utilisables pour que ceux-ci correspondent mieux à ses besoins. Bien que ces extensions puissent prendre des formes différentes suivant les langages, les mécanismes de base sont toujours assez similaires.

2.3.1 Structure des systèmes

Il convient tout d'abord de noter que la structure des systèmes de types est très particulière. En effet, parce qu'ils opèrent une séparation sur l'ensemble des objets, les types

s'organisent naturellement en treillis suivant la relation d'ordre partielle induite par l'inclusion ensembliste comme illustré en figure 2.1.

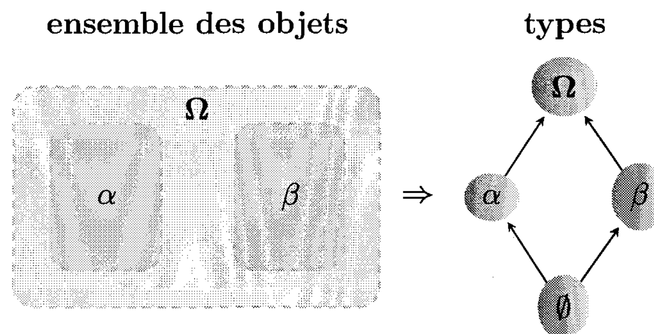


FIG. 2.1 – Ensembles d'objets et treillis de types.

Cette propriété commune aux systèmes de types permet la mise en œuvre d'algorithmes très génériques pour raisonner sur le typage d'un programme. Il est notamment commode, dans l'optique d'une vérification de typage, de s'appuyer sur les caractéristiques des treillis pour effectuer les calculs rapidement à l'aide d'indications bien choisies. C'est tout le sens des travaux autour de la vérification légère de bytecode[2] introduite par Eva Rose [RR98], sur laquelle nous reviendrons dans ce document.

2.3.2 Modes d'extension

Suivant l'approche majeure suivie par le langage (les langages se divisent principalement entre ceux qui suivent une approche objet, et ceux qui suivent une approche fonctionnelle, même si de nombreux langages présentent des aspects issus de ces deux branches), il existe différentes manières d'enrichir le système de types utilisé pour un programme.

La manière la plus immédiate d'ajouter de nouveaux types consiste à organiser des types existants en structures plus complexes, par composition : il s'agit de la méthode employée dans le langage C, par l'intermédiaire des données structurées (`struct` ou `union`). Les types peuvent alors être simplement des raccourcis syntaxiques pour désigner un arrangement particulier de types basiques. L'opérateur de « définition de type » `typedef` illustre bien ce phénomène, puisqu'il ne fait qu'attribuer un nom équivalent à une déclaration potentiellement complexe (les opérateurs `struct` et `union` servant eux-mêmes à distinguer un arrangement particulier de champs).

Pour les langages orientés objet, la création de nouveaux types a des répercussions plus subtiles par le biais de la relation d'héritage [Dah68] : celle-ci permet de réutiliser de manière transparente (au niveau où l'on se trouve) et d'enrichir aussi bien une structure qu'un certain nombre d'opérations associées à cette structure.

[2]lightweight bytecode verification

Dans un ordre d'idées comparable, la définition et l'utilisation d'interfaces permet de composer des types à partir de comportements élémentaires que devront respecter leurs représentants.

On pourra également mentionner la notion de type dépendant, sorte de type paramétré par un autre type, et donc définissant un nombre quelconque de nouveaux types. L'exemple canonique des types dépendants est l'ensemble des conteneurs d'objets (tableaux, vecteurs, ensembles...) et leurs concrétisations diverses suivant les langages : templates C++, types génériques d'Ada ou de Java....

En ce qui concerne les langages fonctionnels, les extensions de types caractéristiques de ce paradigme seront plutôt le raffinement de prédicats (correspondant globalement à la notion d'héritage) et leur composition (structuration), bien que pour la plupart de ces langages, l'approche structurelle et l'approche objet soient également disponibles.

2.3.3 Typages « exotiques »

Bien que la finalité la plus courante des types soit de décrire l'état d'un objet au sens de l'utilisation qu'il est possible d'en faire à un instant donné (le type « entier » caractérise bien l'ensemble des valeurs entières par exemple et donc l'ensemble des opérations applicables), il est tout à fait possible de faire intervenir des notions sans relation avec cet état.

La notion de « type d'effet »[3] [TJ92, Luc87] introduit l'historique des objets et de la manipulation de ceux-ci comme une donnée utilisable au même titre que leur état. Ces informations supplémentaires recouvrent notamment les conditions de création des objets, mais aussi l'ensemble des accès qui sont effectués sur eux.

Les nouvelles abstractions nécessaires pour manipuler ces types d'un genre particulier présentent des propriétés identiques à celles, plus habituelles, des objets abstraits, et s'organisent en treillis.

Certains types d'effet sont aujourd'hui assez répandus. Un exemple basique pourrait être celui de la propriété de constance des objets. Nombreux sont les langages incorporant dans leur hiérarchie de type une notion similaire au `const` du C++, qui garantit que l'objet ainsi désigné n'est jamais modifié. Des langages comme $\mathcal{F}$*ickle* [DDDCG01] introduisent de tels types d'effet de façon plus générique en ajoutant à un langage proche de Java une notion de « reclassification » des objets au cours de l'exécution, ce qui revient à enrichir les types structurels d'un état.

Un autre exemple, non encore répandu dans les langages populaires, concerne l'allocation des objets : l'allocation par région vise à simplifier la gestion automatique de la mémoire en allouant dans des même zones des objets présentant des propriétés similaires ou une durée de vie équivalente. Au contraire de la propriété de constance mentionnée précédemment, cette allocation par région est un typage rarement explicite ; celui-ci est laissé à la plate-forme à travers une inférence (éventuellement aidée d'annotations).

[3] « effect types » dans la littérature anglo-saxonne.

Il est clair que ces méthodes de typage sont parfaitement orthogonales aux méthodes de typage « classiques », quasi structurels. Néanmoins, les procédés qui permettent de les calculer, vérifier et utiliser, sont strictement semblables.

2.3.4 Composition de systèmes de types

Les systèmes de types ayant la propriété centrale d'être représentables par des treillis, il est très simple de combiner par composition plusieurs d'entre eux pour obtenir un système plus complexe présentant les mêmes caractéristiques. En effet, deux relations d'ordre partiel sur deux espaces induisent naturellement une relation d'ordre partiel sur l'espace produit.

De cette manière, il est trivial de faire cohabiter au sein d'un même système des types de nature totalement distinctes, le type d'un objet étant obtenu comme le tuple des types de cet objet dans chacun des systèmes considérés.

Par exemple, la constance d'un objet se combine facilement avec sa nature d'entier pour produire le type d'entier constant. Ou encore la « reclassification » dont il est fait mention en section 2.3.3 peut se concevoir simplement comme l'exploitation d'un type composé de deux types : l'un étant le type structurel, l'autre étant l'état.

2.4 Organisation des travaux

Il existe de nombreuses déclinaisons de la notion de typage, dont les finalités diffèrent. La notion de typage familière au programmeur est celle qui consiste à typer le code source d'une application, généralement de façon à lever toute ambiguïté sur les intentions du programmeur. Le typage d'un code intermédiaire au contraire est une démarche de sécurité, qui vise à s'assurer que le code est en mesure de s'exécuter correctement car se trouvant dans les conditions adéquates pour ce faire.

Dans un premier temps, nous nous intéresserons au paradigme de programmation orientée objet à travers la conception d'un système de types totalement objet pour un langage intermédiaire. Nous verrons en particulier comment concilier une forte extensibilité au sein de ce modèle avec une grande efficacité du code exécuté par la plate-forme. Par la suite, nous nous concentrerons davantage sur l'utilisation de types d'effet à des fins d'optimisation de gestion de ressources et/ou de sûreté de fonctionnement.

Chapitre 3

Exploitation d'un système de types

Les conditions d'utilisation d'une plate-forme déterminent dans une large mesure la forme des systèmes de types destinés à sous-tendre les logiciels amenés à s'y exécuter. En effet, les différentes possibilités présentées précédemment ont un impact réel sur les possibilités offertes par le système, tant en termes de sécurité que de performance ou d'expressivité. Bien souvent, ces trois notions sont largement antagonistes et une attention particulière apportée à l'une tendra à avoir des conséquences négatives sur les deux autres. La conception d'un système de types relève donc avant tout du compromis entre plusieurs préoccupations, en accord avec les objectifs affichés du système qui l'utilisera.

3.1 Impact des conditions d'utilisation

Notre plate-forme de prédilection dans le cadre de cette étude est la carte à puce, dont les contraintes physiques imposent un souci d'optimisation supérieur à ce que l'on peut accepter pour des plates-formes plus puissantes. Par ailleurs, l'utilisation typique de ces cartes se trouve dans des domaines pour lesquelles la sécurité et la confidentialité des opérations est critique puisque touchant à des données très privées (domaine bancaire, cryptographie, monde de la santé). C'est donc tout naturellement qu'un système voué à fonctionner sur une carte à puce présentera des exigences fortes vis à vis de ces deux contraintes.

Par ailleurs, bien que la mise à jour de ces systèmes ne soit pas physiquement simple (ils se trouvent souvent isolés, voire non-alimentés), une importance de plus en plus grande est accordée au fait de pouvoir les faire évoluer, notamment dans les environnements pour lesquels ils jouent le rôle d'interface entre un utilisateur et une plate-forme « anonyme », afin que le premier puisse transporter ses informations et ses applications d'une borne d'accès à une autre de façon quasiment transparente et toujours sécurisée.

Le monde de la téléphonie mobile est un exemple typique d'une telle évolution des solutions à base de cartes puisque de nouveaux logiciels voient le jour sans cesse, requérant le stockage de données et leur conversion pour s'adapter aux nouvelles applications s'exécutant

sur les mobiles. Il est donc particulièrement souhaitable, en plus des exigences formulées plus haut, qu'un système destiné à la carte à puce soit en mesure de tolérer l'adjonction de nouveaux composants chargés dynamiquement, et de garantir un fonctionnement compatible avec l'existant. Il sera donc nécessaire de disposer d'un système permettant de définir et de charger tardivement des blocs de code dont la fonction sera de modifier ou d'étendre l'ensemble des possibilités de la plate-forme.

Une dernière spécificité de ces plates-formes est la très faible interaction possible avec leur propriétaire ou utilisateur. À cause de cette limitation, il est généralement considéré comme relevant de la bonne pratique de détecter toutes les erreurs possibles au plus tôt, et si possible avant que le code potentiellement erroné n'ait commencé son exécution. Le choix d'un langage typé pour le développement des applications semble être une facilité dont il serait regrettable de ne pas bénéficier, dans la mesure où la propriété de typage correct représente un indice de correction du programme généralement très simple à calculer, même si il est évidemment insuffisant dans l'absolu. Les types représentent des contrats d'utilisation de portions de codes, ce qui s'applique parfaitement à la politique de conception par composants dont nous avons exprimé la nécessité.

Ainsi donc, les habitudes de programmation en vigueur sur ces plates-formes et surtout un gain certain de performances (les vérifications de typage correct ayant lieu une fois pour toutes) nous amènent vers une modèle fortement dominé par le typage statique.

Des langages statiquement typés existent déjà pour des systèmes contraints, tels que Javacard[Che00], qui présentent la caractéristique d'offrir un code exécutable (bytecode) possédant un niveau d'abstraction comparable au langage source à partir duquel ils sont généralement compilés. Ceci a pour conséquence de rendre difficile l'utilisation d'un langage différent pour écrire les logiciels nécessaires, puisqu'il est assez fastidieux de transposer des concepts quelconques dans un paradigme *a priori* sans relation.

Dans la suite de ce chapitre, nous présenterons au contraire un système de types (et un langage associé) très générique et compact, pouvant servir de cible pour la compilation de divers langages de haut niveau [GHSR06b, GHSR05]. De ce fait, le programmeur bénéficie d'une certaine indépendance vis à vis du langage source, pour peu qu'il existe un compilateur associé.

3.2 Conception du système de types

Comme annoncé précédemment, tout système destiné à s'exécuter sur un support de type « carte à puce » se voit attribuer une politique sécuritaire visant à préserver les données manipulées. Dans cette étude, nous nous intéresserons plus particulièrement aux garanties apportées par des propriétés de typage.

Le terme de « fiabilité » sera employé ici comme regroupant les notions d'intégrité et de confidentialité des données, qui représentent l'ensemble minimal de propriétés que l'on souhaite garantir à l'application chargée dans le contexte des systèmes embarqués sécurisés. La plupart du temps, ces propriétés reposent simplement sur un typage correct de l'en-

semble du code présent dans le système (mise en correspondance des données effectivement manipulées avec les annotations de types), et sur quelques règles élémentaires généralement exprimées informellement sous la forme suivante :

- il est interdit de transtyper un objet (au sens large du terme : « représentant d'un type ») : tout changement de type doit être le fait d'un processus vérifiable, aucun recours à la coercition ;
- il est interdit de forger un pointeur, et accessoirement d'accéder à l'adresse en mémoire d'un objet : il n'est pas permis d'accéder à l'information « non-typée ».

Ces règles garantissent que l'utilisation faite de chacune des données est conforme aux interfaces installées, et qu'elle est dans une certaine mesure observable sans qu'il soit besoin de procéder à l'exécution réelle du programme. En effet, cela rend possible le suivi des données depuis leur création jusqu'à leur utilisation, même si les analyses pour obtenir un suivi fin sont bien souvent coûteuses.

Un système de types respectant ces règles devient incontournable puisqu'aucune donnée n'est en mesure de s'affranchir par elle-même d'un type. Par conséquent, la vérification de la cohérence d'un programme vis-à-vis d'un tel système de types permet de garantir que le programme présente au moins une sécurité minimale et qu'il est possible de l'intégrer dans un ensemble déjà cohérent de programmes (toujours au sens de la correction des types) sans qu'aucune interface d'utilisation ne soit violée. De la sorte, on conserve un contrôle suffisamment fin sur l'utilisation présente et future des composants installés dans le système embarqué, ce qui permet d'apporter des garanties sur le système global par le biais de propriétés locales aux différents composants.

3.2.1 Du choix d'un langage intermédiaire

Depuis l'apparition de Java, de nombreux travaux de recherche s'intéressent à l'utilisation de langages intermédiaires, notamment dans le domaine des systèmes embarqués. Ceux-ci exhibent notamment la capacité d'exprimer des concepts évolués tels que le typage, qui sont généralement perdus en cas d'écriture en assembleur « machine ». Parallèlement, ils sont généralement suffisamment primaires pour être exécutables directement (en mode interprété) ou compilés de façon simple en code natif si besoin. La classe des langages intermédiaires comprend aussi bien des éléments à visée généraliste, largement utilisés dans les logiciels « grand public » (par exemple, le jeu de bytecode Java [LY96], ou encore CLI [cli] pour la plateforme .NET), mais également des làngages plus spécifiques au monde de l'embarqué moderne à travers Javacard[Che00] et Multos[Mao] par exemple, destinés tous deux à l'écriture de code mobile.

La possibilité, au sein des systèmes embarqués, d'accéder à des propriétés de haut niveau est extrêmement bénéfique dans la mesure où la plupart des politiques de sécurité qui seront exprimées seront en liaison avec ces propriétés. Pour respecter ce critère, essentiellement deux approches sont possibles. À la manière de Java, il est possible de refléter aussi directement que possible les concepts exposés dans le langage source dans le langage intermédiaire. Ainsi, la sémantique des diverses opérations du bytecode Java est-elle calquée

sur les éléments visibles du langage (voir notamment les différents appels de méthodes). Une autre approche, orthogonale, est de réduire au maximum la complexité du langage intermédiaire, de sorte à ce qu'il soit capable de refléter bien plus qu'un seul langage source. Puisque nous désirons être en mesure de supporter l'utilisation de plusieurs langages objets source et leurs concepts associés (toujours pour contraindre le moins possible le champ d'action du programmeur), c'est cette deuxième option que nous avons choisie. Par conséquent, le système de type du langage intermédiaire ne saurait être spécialisé à une certaine catégories d'objets.

Pour terminer sur la pertinence du choix d'un langage intermédiaire, il est à noter que la compacité d'une telle forme est primordiale dans le contexte qui nous importe. En effet, l'expression d'un programme en code natif est bien plus volumineuse, car reposant sur des mécanismes de plus « bas niveau ». Adopter un langage intermédiaire, c'est donc aussi s'affranchir de la complexité et du coût de transfert d'une application vers une plate-forme typiquement lente et pour laquelle des communications volumineuses sont nécessairement handicapantes. Cependant, ces gains en volume sont bien souvent compensés par un coût non négligeable dû au support d'exécution nécessaire à l'utilisation des programmes ainsi chargés : une machine virtuelle est généralement mise en place, qui est chargée d'effectuer la traduction du langage intermédiaire à destination du matériel. Puisque nous visons en particulier la performance, nous préférons éviter l'emploi systématique d'une machine virtuelle, et avoir à la place la possibilité de compiler tardivement le langage intermédiaire sous une forme native équivalente. Le bénéfice d'une telle approche est évident en ce qui concerne les couches primitives du cœur du système : les fonctionnalités exposées par le noyau sont utilisées de façon intensive (à travers les diverses API fournies), et doivent être aussi efficaces que possible. En revanche, en ce qui concerne le code purement applicatif, il peut être intéressant de conserver une vue « haut niveau » des appels de fonction, et de ce fait échanger un peu de performance (en des emplacements non critiques) contre une plus grande concision. En résumé, nous voulons conserver la possibilité de doser l'équilibre entre espace et temps de la manière la plus fine possible, préservant l'efficacité là où elle est le plus requise (ce qui correspond aux couches les plus basses du système, et donc celles dans lesquelles une expressivité maximale n'est généralement pas nécessaire), et économisant la mémoire là où cela est possible (ce qui correspond au contraire aux couches hautes).

3.2.2 Extensibilité et sécurité

Le langage choisi se doit de présenter un certain nombre de propriétés qui permettent de répondre de façon adéquate à la problématique envisagée, tant en termes d'extensibilité que de sécurité.

En premier lieu, il doit être compact. En effet, dans une optique d'extensibilité maximale, la plupart des mécanismes à disposition du programmeur seront amenés à évoluer en fonction des choix faits. Il est donc primordial de se limiter à un petit nombre d'éléments suffisamment génériques pour que toute fonctionnalité désirée puisse s'exprimer en tant que spécialisation d'un ou plusieurs éléments de base.

La manière la plus simple possible d'accéder à cette forme de simplicité extrême est de ne garder que les deux briques fondamentales des langages de programmation, à savoir le flot de contrôle (à travers une ou plusieurs instructions de saut) et l'appel de fonctions (à travers le couple formé par les instructions d'invocation et de retour à l'appelant).

D'autre part, le langage doit être typé statiquement, de sorte à simplifier, voire même simplement à rendre possible, le rejet lors de leur chargement des programmes ne permettant pas d'assurer un respect total des interfaces d'utilisation des composants préexistants. Comme nous l'avons dit précédemment, une connaissance partielle de la nature des objets manipulés par un programme est une donnée fondamentale dans l'applicabilité de la plupart des vérifications de propriétés.

Enfin, la vérification embarquée des propriétés doit être rendue la plus aisée possible : l'acte de vérification doit être effectué au niveau du bytecode pour tout langage « source » afin d'éviter la duplication. Par ailleurs, un système embarqué doit être autonome en termes de sécurité. À savoir, il doit être capable de garantir les propriétés souhaitées sans devoir faire confiance au « monde extérieur », ce qui comprend notamment le fournisseur de l'application. On s'orientera donc vers des techniques de vérification par preuve au chargement, quitte à fournir des indications suffisantes pour la simplifier, mais qui ne sont pas nécessaires dans l'absolu. Par la suite, nous considérerons une seule et unique base de confiance reconnue par le système embarqué, constituée de deux éléments :

- lui-même, à savoir le code constituant son état initial (au moment de la production) ;
- les extensions chargées, après vérification de leur bon comportement.

Dans cette optique, nous proposons pour le langage intermédiaire choisi une approche à base de types statiques et explicites. Notons que l'aspect explicite de ce typage est davantage une commodité qu'une nécessité, néanmoins un tel typage présente l'avantage d'être plus immédiatement vérifiable, et donc d'alléger le processus de chargement de code, ce qui est une préoccupation importante pour les systèmes concernés par cette étude.

3.2.3 Extensibilité et objets

Avant d'entrer plus avant dans les détails de conception du système de types, il convient de clarifier quelles sont les propriétés recherchées. Dans la mesure où le langage intermédiaire cible devra permettre de supporter un large panel de concepts objets (toujours dans l'optique d'une généricité par rapport aux langages source), le système de types devra le refléter en fournissant à l'utilisateur une interface extensible. Ainsi, la notion d'héritage sera-t-elle centrale, puisqu'elle permet à l'utilisateur de créer ses propres types à partir de l'existant. Parallèlement, il n'est pas souhaitable que le langage intermédiaire soit trop proche d'un langage source particulier, sous peine de se spécialiser à l'« interprétation » de ce seul langage. Dans cette optique, le système de types lui-même doit posséder une notion d'extensibilité afin de toujours laisser la possibilité d'introduire de nouveaux concepts *a posteriori* sans remettre en cause l'architecture globale du système. Par exemple, l'introduction d'une notion d'héritage multiple « à la C++ » devrait dans l'idéal pouvoir se faire sans pour autant briser l'ensemble des propriétés obtenues après analyse du système

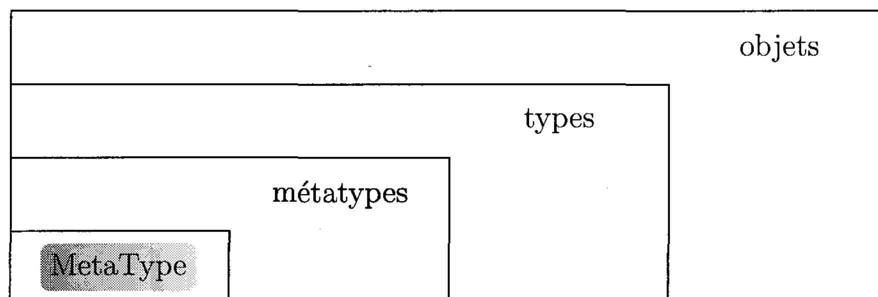


FIG. 3.1 – Génération des niveaux d'abstraction

lorsqu'il ne comportait qu'un héritage simple « à la Java ». Cet exemple d'extension fera d'ailleurs l'objet d'une analyse plus poussée en section 3.6.2

Ceci met en exergue un avantage majeur du recours à un langage intermédiaire : toute vérification ayant lieu sur ce code compilé, le langage source utilisé importe peu, pour peu que les concepts souhaités soient exprimables dans le langage intermédiaire. De la sorte, il est relativement aisé de fournir une infrastructure commune pour les opérations de vérification.

En résumé, le système de types que nous souhaitons concevoir devra si possible être en mesure de supporter tout concept objet que l'on pourra trouver dans un langage de haut niveau (et ce, sans remettre en cause la stabilité générale lorsqu'un nouveau concept est ajouté). Pour ce faire, l'unification de tous ces concepts au sein d'un système de types (et donc d'une hiérarchie) unique apparaît comme un vecteur d'extension commode. Le système de types sera donc lui-même fondé sur une conception objet.

Afin de réaliser concrètement le système de types suivant le modèle présenté, il apparaît intéressant de considérer les éléments du système de types comme des objets eux-mêmes, instances de ce que nous appellerons par la suite des « métatypes ». Un métatype est donc défini comme un type dont les objets possèdent une notion d'instantiation, ou encore comme un générateur de types. De la sorte, on obtient un modèle unifié pour la représentation de toutes les entités ayant un rôle à jouer dans le programme.

Les métatypes permettent de classer naturellement les différents types disponibles, selon leur générateur. Les familles formées présentent des propriétés communes en regard de divers concepts liés à la programmation objet. Leur rôle est de fournir une abstraction réaliste des différents mécanismes entrant en jeu lors de la conception d'un langage particulier. Ceci est grandement facilité par la compacité du langage intermédiaire évoqué en section 3.2.1. En effet, en restreignant au maximum le langage, on est naturellement amené à limiter le nombre de constructions intégrées correspondant à des mécanismes de haut niveau, et à exprimer différemment les mécanismes d'appel, d'héritage, *etc.* Ceci nous amène à envisager la mise en place d'un protocole métaobjet exhibant des caractéristiques semblables à celles des protocoles mis en places pour des langages de haut ou très haut niveau tels que Lisp [KdRB91] ou C++ [Chi95].

La figure 3.1 représente la répartition des objets du système. Toute structure du système

étant matérialisée par un objet, il est naturel que les types eux-mêmes le soient (cette notion est retrouvée dans de nombreux langages objets, de Java à Python). De même les métatypes, et enfin l'objet unique (et central) `MetaType` sont tous des objets.

Inversement, chaque région est nécessaire et suffisante pour générer la région qui l'englobe. Ainsi, tous les métatypes sont générés par `MetaType`, ceux-ci génèrent les types, qui génèrent à leur tour le reste des objets. Notons que le point de départ du processus est `MetaType`, qui se génère (ou plutôt se définit) lui-même.

Finalement, on observe deux préoccupations complémentaires : celle de pouvoir tout manipuler en tant qu'objet, et celle de pouvoir tout définir sans recourir à une notion extérieure au système.

3.2.4 Intégration des métatypes

Une façon relativement naturelle d'introduire les métatypes est de les matérialiser par des instances de « métaclasses », à la manière de Lisp par exemple [BK88, GLS84]. Les métaclasses sont des classes ordinaires, si ce n'est que leurs instances sont elles-mêmes des classes. L'avantage de ce procédé est la possibilité d'intégrer ce niveau d'abstraction supplémentaire au sein du système de types existant. Ainsi, la représentation globale de l'ensemble des objets manipulés par le système reste unifiée et cohérente. Les différences affichées par les différents types amenés à évoluer au sein du système seront exprimées par les différents métatypes qui les définissent. Par exemple, la distinction entre les objets de type *valeur* ou *référence* sera exprimée dans les métatypes concernés : l'existence de champs pour une référence, la façon d'invoquer une méthode sur un objet de l'une et l'autre sorte, les capacités de polymorphismes seront toutes déterminées par les métatypes.

L'objectif principal de l'utilisation des métatypes est de fournir une base pour la description des mécanismes en relation avec la notion de type (en majeure partie, il s'agit de mécanismes objets, mais il n'y a pas de restriction *a priori*). Ces mécanismes sont apparents lorsque l'on considère le comportement d'éléments d'un certain type. Parmi d'autres exemples on pourra citer l'existence du concept de liaison dynamique, qui permet traditionnellement de matérialiser le concept objet de surcharge de fonction dans les sous-classes. Une alternative, très employée également, consiste à considérer l'objet comme étant d'un type que l'on a pu calculer (statiquement) et d'invoquer la méthode correspondante.

Un extrait de structuration des métatypes (boîtes rectangulaires) est présenté sur la figure 3.2. Il s'agit de la structure correspondante au cœur d'un système existant : CAMILLE. C'est pourquoi certains éléments (comme `Unboxed`) sont inhabituels : ils correspondent à des mécanismes généralement strictement internes même si rien ne s'oppose *a priori* à leur utilisation à un niveau élevé.

Dans cet extrait coexistent des types objets « primitifs » (les instances de `Value`) et des types objets « structurés » (le reste). Parmi ces types structurés, une distinction est faite entre les instances de `UnBoxed` [Gud93] qui sont des types dont les instances ne sont

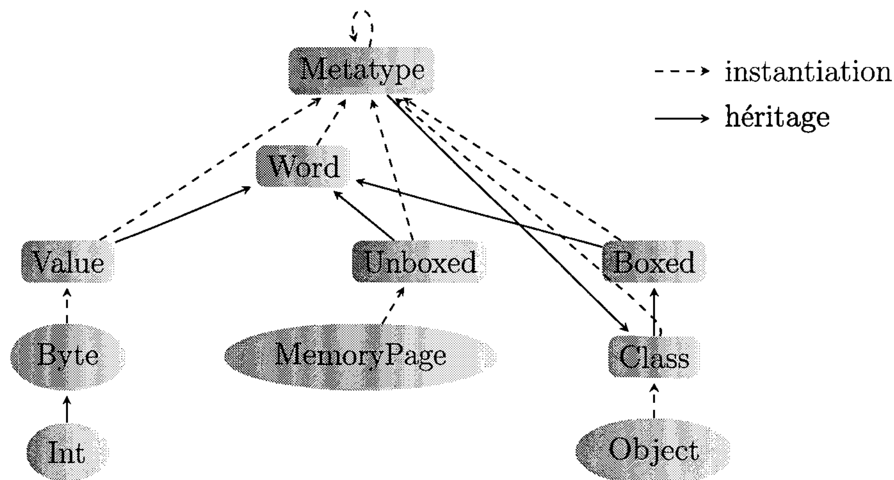


FIG. 3.2 – Classification des objets

accessibles qu'à travers un type statique[1], et les instances de `Boxed`, qui donnent à leurs instances la connaissance de leur propre type (type dynamique donc).

La classe `Word` représente le dénominateur commun entre ces différentes sortes d'objets, le « top » des métatypes au sens de la relation d'héritage. Pour ne pas complexifier inutilement le schéma, les classes parentes de `Word` n'y sont pas représentées mais feront l'objet d'une remarque séparée.

Le classe `Value` est le métatype des objets qui sont des valeurs immédiates (numériques), et non des références (à savoir des pointeurs vers une zone mémoire structurée). Comme dit précédemment au sujet de la classe `Object`, `Class` spécialise le comportement de `Boxed` en ajoutant à la notion de type dynamique celle d'héritage simple (ce qui influe notamment sur la fonction de recherche de fonction applicable dans des circonstances données).

Enfin, la classe `MetaType` est celle dont tous les métatypes sont des instances, et dont le rôle est notamment de définir la sémantique de la relation d'héritage pour ces instances.

De la sorte, on obtient un système cohérent et correctement architecturé supportant différents sous-systèmes de types sans interférence. Le système objet sélectionné pour la hiérarchie des métatypes est un système à héritage simple, et donc une branche de `Object` (qui est donc la classe parente de `Word`). Ce choix est motivé par la combinaison des faits suivants :

- il est nécessaire de faire participer les métaclasses au graphe d'héritage global, pour la simple raison qu'elles sont elles-mêmes des classes ;
- il est nécessaires que ces classes aient les moyens d'exprimer différemment des propriétés de même nature (appels de méthodes par exemple, ou encore accès à des

[1] le type de l'objet n'est alors connu que depuis l'extérieur, l'objet lui-même n'ayant aucune connaissance de son type ni aucun moyen de le découvrir.

champs), ce qui correspond assez bien aux possibilités de surcharge offertes par `Class` ;
- la branche des objets à héritage simple est de toute manière nécessaire à toute modélisation non triviale d'un système à objets ;
- l'héritage simple est suffisant pour représenter les interactions entre les quelques classes que sont les métaclasses.

Concernant ce dernier point, il serait possible d'imaginer un jeu de métaclasses tel qu'il puisse tirer avantage d'un mode d'héritage plus complexe (tel que de l'héritage multiple). Néanmoins, les systèmes considérés n'ont pas vocation à héberger simultanément une grande diversité de métatypes ; aussi les avantages d'une expressivité accrue seraient-ils contrebalancés par la lourdeur des mécanismes à mettre en place au sein du système. Au besoin, une duplication de comportements dans plusieurs métaclasses restera acceptable pour compenser les « lacunes » du modèle d'héritage choisi.

Les concepts exprimés à travers les métatypes sont relativement peu nombreux, eu égard à l'extrême simplicité exigée du langage intermédiaire. En fait, toute la sémantique de l'appel de méthode (qui se trouve être la seule opération non triviale dans le jeu de bytecodes) est adaptable en fonction du métatype considéré.

Les métatypes décrivent en particulier :
- comment les instances de classes sont structurées (sont-elles des valeurs immédiates ou des références) ;
- quelle signification apporter au fait qu'une classe hérite d'une autre (surcharge de méthode notamment, ou simplement existence de méthode) ;
- comment sont traités les arguments d'une invocation de méthode sur un objet (l'appel peut être résolu statiquement ou dynamiquement).

À titre d'exemple, on peut mentionner que l'existence d'un appel virtuel pour les méthodes est avérée uniquement pour les instances de la métaclasse `Boxed` (à moins qu'une telle fonctionnalité s'avère nécessaire ailleurs dans la hiérarchie globale). Parallèlement, pour les instances de la métaclasse `UnBoxed`, seule est disponible la résolution des appels de méthode suivant le type statique de l'objet recevant l'appel (ce qui est cohérent avec le fait que l'objet lui-même n'est pas conscient de son type).

3.2.5 Exemple de projection

Certains systèmes bien connus et employés dans le domaine de l'embarqué, tels que ceux de Java ou même de son dérivé Javacard, peuvent se projeter assez aisément sur le système décrit dans la section précédente.

En effet, pour ces deux langages, les données exprimables dans un programme sont soit d'un type dit « primitif » (`char`, `int`,...), soit d'un type « object » (descendant de `Object`, dans une structure arborescente guidée par un héritage simple).

La hiérarchie nécessaire pour représenter les éléments d'un programme Java ou Javacard est donc similaire à un sous-ensemble de la hiérarchie présentée pour l'état actuel de

CAMILLE. La branche issue du métatype UnBoxed est inutile pour représenter les objets dont dispose le programmeur, par ailleurs le métatype Class, avec la notion d'héritage simple qu'il apporte à ses instances, est adapté pour servir de métatype aux types « objets ». Même dans l'hypothèse où les besoins exprimés dans les classes de Java ou Javacard ne recouperaient que partiellement ceux satisfaits par les instances de Class, il serait tout à fait possible, au choix, de sous-typer Class afin de spécialiser les comportements objets, ou encore de sous-typer Boxed afin d'en redéfinir une partie.

Concernant les objets quelque peu particuliers que sont les tableaux de Java, il est là encore possible de les greffer dans le modèle de CAMILLE en définissant des instances de Class caractérisées par une taille, une classe associée (qui devra être une classe parente de l'ensemble des éléments du tableau) et une liste non bornée de champs (contenus dans une liste de pages mémoire), chacun correspondant à un élément.

3.2.6 Détails de conception

Revenons sur les propriétés définies dans les métatypes : il s'agit d'un ensemble restreint et très abstrait de points caractéristiques d'un certain comportement objet. Intuitivement, les métatypes définissent par eux-mêmes la quasi-totalité des utilisations possibles d'un objet, en concrétisant un petit nombre de schémas basiques.

Structure des métatypes

Pour que le code intermédiaire considéré reste indépendant du code source utilisé, il est intéressant que l'ensemble des caractéristiques des divers langages sources (en fait de leurs systèmes de types) puissent être exprimées le plus facilement possible dans ce langage cible.

Ces caractéristiques incluent (entre autres) la définition des notions suivantes : l'héritage, l'appel de méthodes, l'existence de champs. Pour conserver un maximum de flexibilité dans l'expression de ces concepts, une solution simple est de les définir par le biais d'une méthode qu'il soit possible de surcharger et de réutiliser.

En ce qui concerne les relations d'héritage entre les différentes classes du systèmes, les métatypes introduisent un certain nombre de méthodes qui visent à générer ou interroger ces relations :

- `setParent()` crée une relation d'héritage entre la classe courante et une autre classe ;
- `isAKindOf()` teste le fait que la classe courante est bien une sous-classe ;
- `unify()` calcule un supremum dans le treillis de classes ;
- `lookupMethod()` calcule la méthode qui doit être appelée lors de l'écriture d'une invocation.

Toutes ces méthodes peuvent bien sûr être définies différemment suivant les métaclasses, ce qui génère une relation d'héritage différente pour chaque cas. Nous reviendrons sur certaines ultérieurement.

Par ailleurs, comme nous le verrons par la suite, les métatypes interviennent également dans le processus de génération de code, si bien que d'autres opérations essentielles y sont définies, telles que :

- `genCall()`, qui génère un appel à la méthode calculée par `lookupMethod()` (c'est ici que l'on pourra générer des appels tenant compte du type dynamique ou du type statique par exemple) ;
- `genPutfield()`, uniquement dans les sous-classes de `Boxed`, pour générer des accès à des champs ;
- ...

Enfin, les métaclasses sont également des classes, ainsi que des objets, aussi est-il possible de leur appliquer les méthodes réservées à cet usage.

Héritage

La seule définition de l'héritage qui soit commune à l'ensemble des classes du système (en tant qu'instances de la classe `Word`) est simple : une instance d'une sous-classe peut être utilisée en lieu et place d'une instance d'une super-classe (et inclue donc néanmoins la notion de sous-typage). Ce choix se justifie par le fait que sans cette propriété la réutilisation de code par factorisation se rapproche de l'encapsulation, ce qui ferait de l'héritage une simple facilité syntaxique.

Dans les faits, cette relation d'héritage entre deux classes est matérialisée par la valeur de retour d'une méthode `isAKindOf()`, dont le comportement est déterminé dans `MetaType`. Ceci nous donne une généralité suffisante pour que l'ensemble des classes puisse être intégré dans une « hiérarchie » unique, là où des systèmes tels que Java font le choix de discriminer objets « réels » et « types primitifs ».

Cependant, la hiérarchie en question est plus complexe qu'un simple arbre, ou même qu'un graphe orienté. En effet, dans la mesure où les métatypes se permettent de définir eux-mêmes la plupart des aspects du typage qui sont bien souvent du domaine du système de types lui-même, la notion d'héritage qui sous-tend traditionnellement le graphe des classes est ici variable suivant le positionnement dans le schéma proposé.

Comme on peut le constater sur la figure 3.2, les traits pleins (matérialisant les relations d'héritage) ne suffisent pas à décrire un chemin entre deux nœuds quelconques. Le système de types présenté est donc, relativement à la relation d'héritage, divisé en plusieurs composantes connexes. Cette séparation rend possible la variation annoncée quant à la sémantique exacte associée à la notion d'héritage.

Par voie de conséquence, la relation d'héritage n'est pas suffisante pour unifier l'ensemble des types (comme c'est le cas dans de nombreux autres langages, tels que Java, où des hiérarchies de types totalement distinctes cohabitent). L'unification et la mise en cohérence des types est assurée par la conjonction des relations d'héritage et d'instantiation, cette dernière permettant de rassembler les hiérarchies distinctes par l'intermédiaire de leurs générateurs.

Correction par construction

De fait, il existe une corrélation forte entre la notion d'héritage (en termes de classes) et celle d'instance (d'une métaclasse). Une propriété fondamentale du graphe généré par ces deux relations est qu'il est strictement interdit pour une classe `B` d'hériter d'une classe `A` si la métaclasse de `A` n'est pas une super-classe de la métaclasse de `B`. Ceci, afin de garantir une séparation nette entre les types de natures profondément incompatibles.

L'exemple le plus immédiat de la nécessité de cette limitation est la non interchangeabilité entre des « valeurs » et des « références » (les premières ne pouvant être que dupliquées, et les secondes étant des points d'accès vers des données structurées uniques). Pouvoir construire une classe ayant un comportement d'objet mais se trouvant dans la branche d'héritage des entiers reviendrait à pouvoir transtyper ou à pouvoir forger des pointeurs (suivant le sens qu'il conviendrait de donner à une telle opération).

De même, si l'on imagine l'ajout de classes supportant l'héritage multiple, la propriété garantit que ces classes ne pourront en aucun cas être utilisées comme des classes à héritage simple, lesquelles existent par ailleurs également dans le système. Cette propriété de séparation ne sert qu'à garantir la cohérence du système.

L'isolation stricte des classes qui présentent des comportements objets différents garantit que les propriétés prouvées sur une branche du graphe ne sont pas remises en cause par l'ajout de branches parallèles. En revanche, la séparation elle-même doit être prise en charge par les métatypes, puisqu'ils sont seuls à même de définir ce qu'est la notion d'héritage pour leurs instances. Ceci impose, pour conserver la sécurité globale du système de types, d'interdire l'extension au niveau des métaclasses par un utilisateur non privilégié : en effet, il est nécessaire de garantir au préalable qu'un métatype fournit toutes les garanties nécessaires à sa bonne intégration au sein du système.

Néanmoins, une certaine notion d'extensibilité existe au niveau du modèle, et se calque bien sur la notion d'extensibilité telle qu'elle est perçue par l'utilisateur. En déportant certaines notions habituellement propres aux systèmes de types dans des types particuliers, nous avons introduit une grande flexibilité dans l'ensemble. Mais cette flexibilité, vecteur d'expressivité, a un coût en termes de sécurité, aussi est-il nécessaire d'en contrôler sévèrement l'accès en le réservant à la période précédant le déploiement, pendant laquelle le noyau est en cours de construction.

Notons néanmoins que ces restrictions d'accès ne sont nullement un cas particulier dans le système de types. En effet, il est très simple d'écrire la classe `MetaType` de sorte qu'elle garantisse elle-même qu'il sera impossible d'en créer une instance en dehors de la phase de déploiement du système. Encore une fois, l'expressivité du système de types suffit à garantir par construction sa propre sécurité.

3.3 Processus de compilation embarquée

Le système de types précédemment défini pour le langage intermédiaire est à la fois puissant et extensible. Au vu des motivations exposées, il est naturel de s'orienter vers une solution à base d'objets (le langage intermédiaire sera plus naturellement à même de refléter les concepts présents dans les divers langages sources). Cependant, il est souhaitable de s'affranchir des inconvénients traditionnellement liés à ce genre de système. Certains langages « totalement objet » tels Smalltalk [GR83] payent le coût d'utilisation des objets à chaque action, même lorsque cela se justifie difficilement en considérant les instructions réellement utiles d'un point de vue purement fonctionnel.

Ainsi, il apparaît toujours regrettable de ne pas simplement générer une instruction machine `ADD` (exécutée directement par le processeur) lorsque l'opération à effectuer est une addition de deux entiers. D'une manière générale, les concepts objets sont gourmands en temps de calcul, notamment en ce qui concerne les calculs basiques, car reposant totalement sur une abstraction bien éloignée du processeur. Cette surcharge est bien moins évidente dans les couches hautes du système, où les concepts objets prennent tout leur sens car il n'existe pas d'équivalent bas-niveau trivial. Les conséquences de ce fait sont particulièrement critiques dans le cas des systèmes embarqués, où les contraintes élevées sur la disponibilité de ressources rendent ce genre de situations plus que dommageables.

Certains langages tels Java tentent de s'affranchir du problème en maintenant une double hiérarchie pour certains types basiques. À la hiérarchie des types entiers primitifs (`int`, `float`, *etc.*) correspond une hiérarchie de types entiers objets (`Integer`, `Float`, *etc.*), qui réalisent l'encapsulation des types primitifs afin de les rendre « compatibles » avec l'ensemble des fonctions qui manipulent exclusivement des objets. Bien évidemment, le coût en espace et en temps de l'utilisation de tels objets est bien plus important que celui de leurs équivalents primitifs, outre le fait que l'élégance d'une telle approche est discutable.

Cet état de fait correspond à ce qui est présenté comme un paradoxe par [KdRB91] : l'expression à très haut niveau des fonctionnalités d'un programme devrait permettre une meilleure capture du problème réel auquel elles devraient apporter une solution, et donc accroître l'efficacité de la solution. En dépit de cela, les processus de compilation usuels se montrent généralement incapables de traduire les algorithmes de haut niveau sans les découper en morceaux plus petits, plus génériques, et en définitive plus simples à compiler. Il est possible de voir cette manière de procéder comme étant un peu trop anticipée, du fait qu'elle cherche à isoler les composantes d'un programme de leur contexte, ce qui mène nécessairement à une perte d'efficacité.

3.3.1 Rôle de la compilation tardive

Pour remédier à ce problème, il est souhaitable d'être en mesure d'installer les divers composants constitutifs d'un programme de telle manière que les performances à l'exécution soient aussi bonnes que possible, et en particulier qu'elles soient comparables à celles des

programmes équivalents écrits à l'aide de paradigmes de programmation réputés plus efficaces (pour une situation donnée). La phase de chargement du code intermédiaire nous offre une opportunité de résoudre en partie les appels objets qui se révéleraient inutilement coûteux à l'exécution.

L'obtention des performances souhaitées pour un système embarqué de petite taille passe par l'adoption d'un mécanisme plus complexe qu'une simple interprétation de bytecode. À la place, nous considérons un processus de compilation embarquée chargée de transformer le langage intermédiaire (encore considéré comme un code de haut niveau, notamment du fait qu'il est fortement typé) en sa forme exécutable par la plateforme. L'objectif principal est de compenser l'inadéquation entre la forme simple du code intermédiaire et l'efficacité du code machine, notamment en ce qui concerne les couches les plus primitives (et donc les plus critiques). Par exemple, l'ensemble des opérations arithmétiques est extrêmement sollicité lors de l'écriture de programmes usuels, il est donc important que des choses aussi simples que celles-ci soient exécutées avec la plus grande efficacité.

Cependant, la génération de code natif n'est pas l'unique finalité de la compilation embarquée. Il est tout à fait possible (et dans certains cas probablement souhaitable) d'effectuer des transformations d'une autre nature, par exemple en générant une représentation du programme dans une autre forme intermédiaire (telle que du bytecode Java). À titre d'illustration, dans l'optique d'un redéploiement ultérieur d'une application locale vers un système différent, il peut être intéressant de générer directement du code exploitable par ce dernier et ainsi de faciliter la transition.

De même que l'introduction des métatypes apportait une réponse aux nécessités de flexibilité accrue dans l'expression des concepts de nature objet et d'unification de types très différents au sein d'une même structure, de même la compilation tardive et embarquée laisse un grand degré de liberté à chaque classe afin qu'elle puisse déterminer de quelle manière son code doit être converti en programme. Il s'agit d'un mécanisme destiné à apporter flexibilité et unification dans le processus de déploiement de l'application là où les métatypes intervenaient dans le processus de conception.

3.3.2 Flexibilité du mécanisme de liaison

La phase de chargement d'un code dans le système est l'étape adéquate pour effectuer les dernières transformations qui aboutiront à son état final. Nous utilisons un mécanisme de liaison au chargement introduit par Gilles Grimaud [Gri00], qui permet aux différents éléments mis en jeu par le programme de s'adapter et de s'exprimer dans la forme adaptée à la situation, afin de générer le programme exécutable. Plusieurs possibilités d'adaptation sont possibles :

- une compilation directe du code intermédiaire en code natif, par sollicitation d'un composant particulier du système appelé « générateur de code » ;
- une transformation par invocation d'une ou plusieurs méthodes ;
- une génération de code virtuel.

Partant de cette base à la fois flexible et permettant de générer un code efficace, nous proposons d'enrichir et d'intégrer le mécanisme de liaison existant. Le but de cette opération est d'une part d'obtenir la sécurité de celui-ci, à savoir une garantie de typage sur le code réellement exécuté, dès lors que le code chargé est lui-même correctement typé. D'autre part, nous proposons d'utiliser le potentiel de ce mécanisme dans le but de permettre la concrétisation des mécanismes objets décrits par les métatypes présentés précédemment.

Méthode de liaison

De la même façon que les mécanismes objets ont été exprimés dans la section 3.2.6 sous la forme d'appels de méthodes, la liaison elle-même est laissée partiellement sous la responsabilité des classes et est réalisée par l'exécution d'une méthode particulière présente dans chacune d'entre elles : `bind()` est appelée par le chargeur de code lui-même, afin de réaliser la liaison du code en cours de chargement. Il est à noter que cette méthode diffère peu des autres, et est écrite en langage intermédiaire. De ce fait, elle est elle-même amenée à subir le processus de chargement, et donc à voir son code lié par le même mécanisme à d'autres composants préexistants, ce qui implique les méthodes `bind()` des composants en question. Lors de son invocation, la méthode `bind()` reçoit une description de la méthode à lier (ce qui limite le nombre de telles méthodes à une par classe), ainsi que le contexte de liaison, décrivant notamment sous forme symbolique l'ensemble des contextes d'exécution de la méthode correspondante.

Le contexte de liaison d'une méthode m_1 est donc une information calculée au chargement d'un code (méthode m_2) faisant appel à m_1, et qui contient principalement :
- une description de la méthode m_2 : classe d'appartenance, identifiant de méthode;
- une description des registres virtuels correspondant à la convention d'appel de m_1 : arguments et valeur de retour de cette méthode;
- les types correspondant à ces registres.

Cette information est suffisante pour procéder à la compilation de m_1 dans le cadre précis de l'appel effectué par m_2.

La méthode `bind()` se distingue néanmoins des autres par le fait qu'elle ne doit pas être invoquée directement, ne serait-ce que parce que le système de types serait incapable d'effectuer la moindre vérification sur les données symboliques qu'elle reçoit en paramètres (d'un certain point de vue, on peut considérer que la méthode `bind()` forge des pointeurs vers les objets que la méthode liée est censée utiliser). Pour le programmeur, une méthode `callBind()` est disponible, qui enrobe l'appel à `bind()` dans une couche de vérifications destinée notamment à s'assurer que la méthode liée est valide pour l'objet `this`, ou encore que la classe dont est originaire le `callBind()` est autorisée à effectuer cette opération. Un point crucial concernant les méthodes `bind()` est le fait que le code de ces méthodes est exécuté à *chaque chargement* (et non pas exécution) d'une instruction qui fait appel à une fonction de leur classe parente. Par exemple, `A.bind()` est invoquée pour chaque réalisation d'un appel de méthode sur un objet de la classe `A`. De la sorte, il est possible que la méthode `bind()` se comporte différemment suivant le contexte de chargement dans lequel

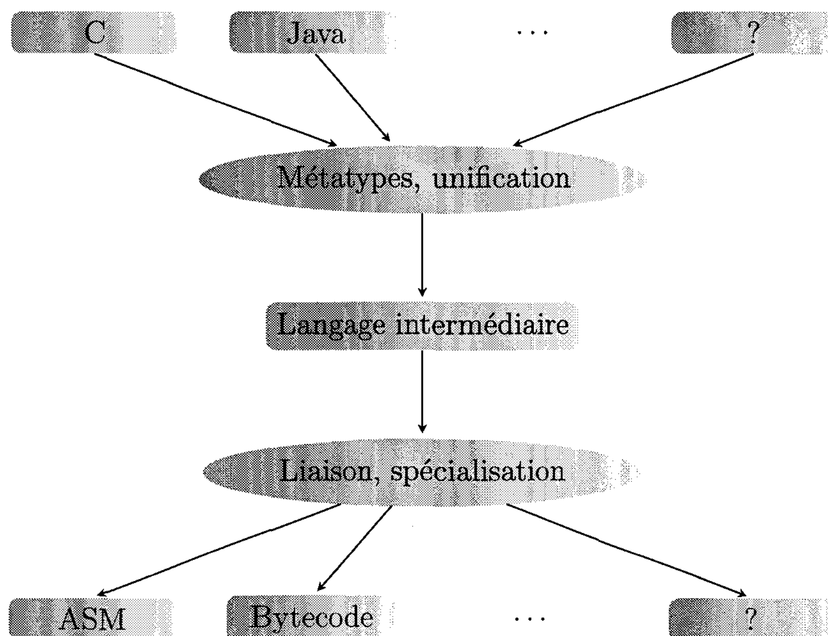


FIG. 3.3 – Transformation des systèmes de types et du code.

elle est appelée, au lieu de décrire de façon absolue la façon dont la classe est compilée. Ceci permet notamment d'effectuer des contrôles plus fins lors du chargement de code : par exemple il est possible de rejeter la liaison si la classe appelante ne fait pas partie d'un ensemble déterminé, ce qui est une façon d'exprimer des propriétés beaucoup plus fines que la simple opposition entre partie publique et partie privée, classique dans les langages tels que Java.

En résumé, la compilation d'un code de haut niveau en code intermédiaire générique permet l'unification des systèmes de types des langages source, tandis que le mécanisme de liaison permet la spécialisation du code intermédiaire afin de l'adapter au mieux à son environnement d'exécution. La figure 3.3 reflète le schéma de transition du code source à l'exécutable.

Accès maximal à la plate-forme

Dans une perspective d'optimisation de programme, ce mécanisme de liaison ouvre des pistes intéressantes. Grâce à sa grande souplesse, il donne au code toute latitude pour se déployer sous la forme qu'il jugera la mieux adaptée. Bien évidemment, cette forme pourra varier suivant la situation, ce qui est rendu possible par le contrôle très fin du contexte de liaison.

En ceci, ce modèle présente une approche totalement distincte de celle suivie par Java notamment (ou encore .NET et Multos). Ce dernier offre une interface de programma-

tion raisonnablement complète et un environnement possédant des propriétés de sécurité intéressantes. Mais lorsque les limites de l'expressivité de Java sont atteintes, un mécanisme annexe (les méthodes natives) permet de les repousser au prix des propriétés en question : si le code des méthodes natives est erroné, la stabilité du système et sa cohérence sont remises en question. De plus, le mécanisme de méthodes natives présente également des limites (notamment en ce qui concerne l'intégration) lorsque ce ne sont pas simplement des méthodes mais des types qui devraient faire appel à des mécanismes natifs.

Un exemple d'un tel manque pourrait être l'inexistence d'un type `byte` non-signé en Java. La simulation de ces types à l'aide d'objets aussi bien que le calcul des opérations désirées à l'aide de méthodes natives sont possibles mais demeurent peu satisfaisants. Dans le premier cas, il sera nécessaire d'encapsuler la donnée utile dans une structure manipulée au travers de références. Dans le deuxième cas, le coût des conventions d'appel s'appliqueront alors qu'il existe au niveau de la machine les instructions suffisantes pour effectuer les manipulations adéquates.

À l'inverse, la méthode de liaison présentée ici prend la forme d'un compilateur invoqué tardivement. Ainsi, il n'existe qu'une façon d'étendre le système, et aucun aspect des plates-formes d'exécution ne lui est définitivement inaccessible.

Cette phase de compilation peut en particulier exploiter certaines particularités du système, telles que des instructions spécifiques à un processeur donné, ou encore des accès à un matériel optionnel.

Ceci correspond à une préoccupation fondamentale des exo-noyaux, qui est de présenter une interface aussi mince que possible entre le système et le matériel sous-jacent [EK95]. En même temps, le haut niveau d'abstraction requis par les langages orientés objet n'entre pas en conflit avec cette préoccupation, comme la forte utilisation du C++ l'a déjà démontré.

Extension de la projection Javacard

Un exemple immédiat d'application des méthodes de liaison peut être observé dans le prolongement de la simulation de systèmes de types Javacard, telle que détaillée en section 3.2.5. Ainsi par exemple, il est possible d'envisager d'étendre un tel système en lui adjoignant une notion de types flottants (`float`, `double`,...), ce qui peut se faire sans nécessiter de modification dans la hiérarchie des métatypes existants.

Selon la complexité des opérations à effectuer sur les objets de ces types, il est intéressant de faire varier la nature du code chargé de réaliser ces opérations. Ainsi, de même qu'un compilateur `C` génère une opération simple pour l'inversion de signe d'un nombre flottant, la concrétisation de la méthode correspondante pour les objets flottants pourrait simplement être la génération d'une opération binaire (généralement une simple opération machine) sur la représentation du flottant. Au contraire, si l'opération est complexe (division par exemple), et nécessite la génération d'un grand nombre d'instructions élémentaires, on préférera sans doute générer une procédure pour la concrétiser, et éviter ainsi la duplication d'un code volumineux.

Comme nous le verrons plus tard, ces deux stratégies sont rendues viables par le fait que la génération des opérations binaires est légale pour les types concernés, et plus généralement pour l'ensemble des types instances du métatype `Value`.

La conséquence de l'applicabilité de ces techniques est la capacité de conserver un comportement optimal, alors même que le système global a été conçu sans les types. De ce fait, nous obtenons ici un compromis idéal entre les types « primitifs » de Java, intéressants pour leur faible occupation spatiale et l'efficacité des opérations qui s'y rapportent, et les types « objets » correspondants, pouvant être chargés ou étendus dynamiquement.

Exemple d'utilisation non-triviale

Si la compilation en code machine est une option intéressante apportée par le mécanisme de liaison adopté, il convient de noter que ce n'est pas la seule. Une autre utilisation pourrait être la combinaison de méthodes de liaison existantes de sorte à générer des opérations composites optimisées sans pour autant devoir réécrire quoi que ce soit.

On peut mentionner la possibilité d'appliquer les concepts de programmation par aspects aux programmes développés. L'écriture d'une méthode `bind()` adaptée permet de spécialiser la sémantique d'une méthode donnée. Ainsi, il est parfaitement envisageable de déclencher certaines actions avant et après chaque accès à la mémoire, et ce de façon totalement transparente pour le code utilisant les accesseurs fournis. De fait, le mécanisme de liaison peut encore être vu comme un tisseur d'aspects simple, mais néanmoins suffisamment expressif pour être utilisé dans des cas non triviaux.

À titre d'exemple, l'écriture d'invariants de classes à l'aide d'une méthode `bind()` est relativement simple, ainsi qu'il est montré en figure 3.4 (le code est exprimé en pseudo-C pour raison de simplicité). La première étape est d'écrire une méthode chargée de la vérification de l'invariant. Ceci fait, la méthode `bind()` aura le comportement suivant : puisqu'il s'agit bien d'un invariant de classe, pour chaque appel à une méthode de cette classe (à savoir, pour chaque compilation du code de cette méthode, ou encore pour chaque invocation de la méthode `bind()`), l'appel réel à la méthode (ou plus exactement, le code reflétant l'exécution de cette méthode) sera suivi par la vérification de l'invariant.

La figure 3.5 présente une transformation possible de code, induite par la méthode de liaison écrite pour cet invariant de classe. Si l'on considère une méthode quelconque `m()` amenée à invoquer une méthode couverte par l'invariant de classe, le processus est le suivant : lorsque l'instruction d'invocation est chargée, la méthode de liaison associée à `f()` (`bind()`) est exécutée. Ceci provoque la génération d'un code cible ayant globalement la forme présentée sur cette figure (la forme exacte dépend bien sûr des possibilités du langage machine cible). De manière informelle, la sémantique du programme final est celle que l'on aurait obtenue par des appels successifs à `f()` et `m()` dans un langage source ne faisant pas intervenir de `bind()`.

Il est intéressant de noter que l'assertion concernant l'invariant de classe est exécutée *après* chaque invocation de méthode, et non à la *fin* de chaque invocation (à savoir juste avant le `return`), ce qui est la manière usuelle (et moins expressive) de simuler les post-

```
void check_invariant() { ... }

void f() { ... }

void bind(ctxt) {
  switch(ctxt.METHOD) {
    ...
    case F:
      invoke(F);
  }

  invoke(CHECK_INVARIANT);
}
```

FIG. 3.4 – Invariant de classe.

conditions à l'aide d'un `assert()` classique. Par extension, il est envisageable de construire un environnement complet de pré- et post-conditions à l'aide des méthodes de liaison. L'intégration complète d'un tel environnement demanderait quelques efforts de conception sur les interfaces, notamment pour éviter d'être obligé de déporter les assertions dans des fonctions séparées (ce qui n'est pas très satisfaisant vis-à-vis de la transparence souhaitée pour ce genre de mécanismes).

Grâce au mécanisme de liaison présenté, le programmeur a accès à la définition même d'une liaison (ce qui est généralement considéré comme un « appel de fonction »), et peut le rendre aussi flexible que voulu. La plupart des possibilités offertes ont déjà été exploitées ponctuellement au sein d'autres plates-formes, comme par exemple le JIT (Just In Time compiler) de Java[OSM+00], l'interception de méthodes dans AspectJ[KHH+01], ou encore l'introspection de pile pour l'examen des appelants[WBDF97]. Tous ces mécanismes peuvent être unifiés par le biais du mécanisme de liaison présenté ici, et ainsi rendus indépendants des langages source.

3.3.3 Intégration dans le processus de chargement

Au cours de son chargement par le système, une classe passe par différents états, qui l'amènent à évoluer progressivement vers une entité utilisable. Une représentation simplifiée de ces états est donnée en figure 3.6. Notamment, pour parvenir à gérer convenablement les appels à d'autres méthodes de la même classe (le cas classique étant celui de l'accesseur), il est nécessaire d'avoir recours à une phase de déclaration de la classe, avant de donner explicitement le code de ses méthodes.

De même que toute action dans ce système, la phase déclarative revient en fait à

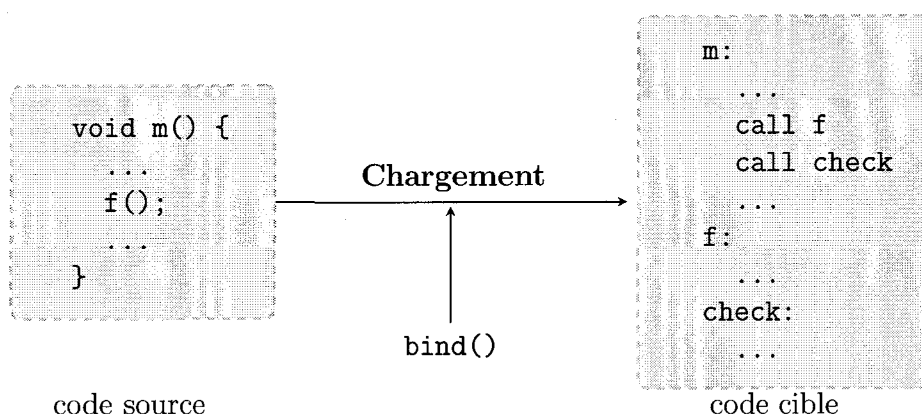


FIG. 3.5 – Déroulement du processus de liaison

l'exécution d'un code, qui correspond à la création de l'objet classe. C'est également à ce moment qu'est fournie l'implémentation complète de la seule méthode `bind()`. Cette légère particularité est rendue nécessaire par le fait que la méthode `bind()` joue le rôle de déclarateur pour les méthodes de la classe, puisque c'est son exécution qui permet d'effectuer le lien entre l'invocation d'une méthode de sa classe et le code de celle-ci.

Ceci met en exergue le fait que la séparation en phases de déclaration puis d'implantation est assez arbitraire, dans le sens où les mécanismes mis en jeu lors de ces deux phases sont strictement identiques et que les phases elles-mêmes ne sont pas distinguables du point de vue du système qui procède au chargement : toutes deux sont réalisées par exécution de code. Seul l'état des objets qui représentent les classes indique à quelle phase de chargement ces dernières se trouvent.

Bien sûr, le chargement est en relation forte avec les opérations de liaison. Lorsqu'une classe est chargée, son code est lié aux composants qu'il utilise, par l'exécution des méthodes `bind()` correspondant aux composants. Ceci concerne également la méthode `bind()` de la classe en cours de chargement. Une fois le chargement terminé (la classe est dans l'état « chargé », conformément au schéma présenté en figure 3.6), le nouveau composant est prêt à être utilisé, c'est à dire à être à son tour lié depuis des classes qui seront chargées ultérieurement.

Notons que le processus de chargement est en définitive extrêmement simple. Il s'agit essentiellement d'une boucle de lecture d'instructions qui, pour chacune d'elle, délègue tout acte de compilation non trivial (invocations de méthodes) à la méthode de liaison adaptée pour générer du code. Une première partie de ces instructions correspond à la mise en place de la structure d'une classe (phase de déclaration), puis le reste des instructions associe un code exécutable à cette structure (phase d'implémentation).

FIG. 3.6 – États de chargement.

3.4 Liaison et sécurité

Les méthodes de liaison étant pleinement responsables de la façon dont elles génèrent le code qui se trouve dans leur sphère d'influence, il est évident qu'elles sont un point central dans la sécurité du système dans sa globalité. Une faille à ce niveau pourrait conduire à un accès a priori illimité au système compromis. Comme mentionné précédemment, le mécanisme de liaison joue le rôle d'un compilateur, invoqué tardivement, et il est nécessaire qu'il soit fiable.

Par conséquent, un nombre important de vérifications doivent être opérées afin d'assurer par exemple la consistance du système de types à travers sa transformation en code natif. Cette préoccupation s'inscrit dans la continuité de travaux portant sur les compilateurs certifiés[Ler06, BDL06]. Concrètement, ces vérifications se traduisent par l'exécution d'une série de tests à la fin de chaque étape de compilation, afin de s'assurer que l'exécution du processus de chargement n'a pas dévié dans une direction illégale. La difficulté provient de l'impossibilité de faire confiance au programmeur d'une application pour effectuer ces vérifications. Un programme malicieux ou simplement mal écrit pourrait remettre en question la stabilité du système de types. La notion de certificat d'origine, ou autre variante, n'est pas non plus satisfaisante, car on souhaite toujours que le système demeure le plus indépendant possible de toute entité extérieure.

3.4.1 Attaques envisageables

Au cas où les vérifications ne seraient pas faites correctement, il serait par exemple possible de contourner le typage, et de manipuler des pointeurs explicitement. Ainsi, l'écriture de méthodes telles que `int cast(Object)` et `bind()`, exposées en figure 3.7, serait triviale. L'effet serait la compilation de la méthode `cast()` en opération neutre qui ne ferait que copier sans aucun contrôle son argument dans la variable de retour d'un type incompatible. La copie de registre étant une opération trop « bas niveau » (dépendant du générateur de code) pour être typée, le transtypage serait possible.

Afin de contrôler l'accès au générateur de code, il est nécessaire de disposer d'une couche intermédiaire, dont le rôle est de garantir la cohérence des opérations de compilation vis-à-vis des contraintes de types visibles dans le code, et donc de s'assurer que les méthodes, après leur transformation, continuent d'être équivalentes à une méthode typée de même signature. Pour chaque type, le métatype associé est un emplacement très adapté pour placer ces contrôles.

```
int cast(Object) {
  return 0;
}

void bind(ctxt) {
  switch (ctxt.METHOD) {
    ...

    case CAST:
    copy(ctxt.ARG1, ctxt.RET);
    ...
  }
}
```

FIG. 3.7 – Attaque basique.

Outre leur rôle d'unificateur pour l'ensemble des types du système, les métatypes ont donc également pour fonction de fournir les règles nécessaires à l'exécution correcte du code intermédiaire généré en regard du code source considéré. En effet, puisque les métatypes ont la responsabilité de la traduction en code effectif des mécanismes objets tels que les appels de méthodes, eux seuls déterminent la sémantique à donner à un bloc d'instructions. En particulier, il sera possible grâce à eux de compiler nativement le code intermédiaire de la façon la plus efficace possible, afin d'obtenir un programme exécutable performant dans ses sections critiques.

Un autre exemple pourrait être l'écriture d'un code tentant d'appliquer une méthode virtuelle sur un objet dont la classe serait une instance de `Value` (un type « valeur immédiate » donc), ce qui est manifestement une violation des contraintes imposées par les types et pourrait mener à un trou de sécurité important (manipulation indirecte d'objets par forgeage de pointeurs, pour peu que la représentation binaire soit compatible, ce qui est le cas). Dans un tel cas, le métatype doit être en mesure de rejeter le processus de compilation : plus précisément, il doit être en mesure de contrôler que l'objet cible du message, `this`, est bien habilité à recevoir le message en question, et donc d'effectuer un contrôle de type. Dans l'exemple donné en figure 3.8, une compilation de la méthode virtuelle `A_METHOD` de la classe `A_CLASS` est tentée. Ce code doit manifestement être rejeté car l'objet `this` est de type entier, et par conséquent incompatile avec la classe `A_CLASS`.

3.4.2 Schéma de liaison

La figure 3.9 expose plus en détails le schéma de liaison à travers un exemple simple : le cas d'une méthode `m()` d'une classe `C` qui désire additionner deux éléments de type `byte`

```
void cast(int i) {
  return;
}

void bind(ctxt) {
  switch (ctxt.METHOD) {
    ...
    case CAST:
    callBind(A_CLASS, A_METHOD, ctxt);
    ...
  }
}
```

FIG. 3.8 – Attaque indirecte.

(type entier, matérialisé par la classe `Byte`).

Typage

La première étape de réalisation de cette liaison est la création d'un contexte de compilation (`ctxt`) contenant les informations relatives au contexte dans lequel l'appel à la méthode + est réalisé. Ensuite, ce contexte est confronté à la signature associée à la méthode afin de vérifier qu'elle est appelée dans des conditions acceptables. Si tel est le cas, la méthode `Add` de la classe `Byte` sera liée dans le contexte en question. Cette étape correspond donc à la phase de contrôle de la cohérence des types annoncés par la signature de méthode avec les types des objets effectivement amenés à être manipulés par elle. Ce contrôle est effectué directement par le chargeur de code, à la manière de la plupart des systèmes acceptant du code venant de l'extérieur (tels que Java).

Une fois cette étape franchie, le code source peut être considéré comme bien typé, et les actions ultérieures ne sont plus du ressort du chargeur de code, qui délègue la responsabilité de la liaison aux classes elles-mêmes.

Droits d'accès

La seconde vérification à effectuer concerne la légalité de la liaison, à savoir si la classe exécutant la liaison (`Byte` en l'occurrence) permet la liaison dans le contexte `ctxt`. Les raisons de refus de liaison de la part de `Byte` peuvent être variées et être des conséquences de tout calcul sur les informations contenues dans `ctxt`.

La demande d'approbation de la liaison par la classe cible s'effectue par l'intermédiaire de sa métaclasse, qui interroge l'ensemble des super-classes de la classe concernée sur la pertinence de la liaison souhaitée. Cette requête se traduit par des appels à une méthode

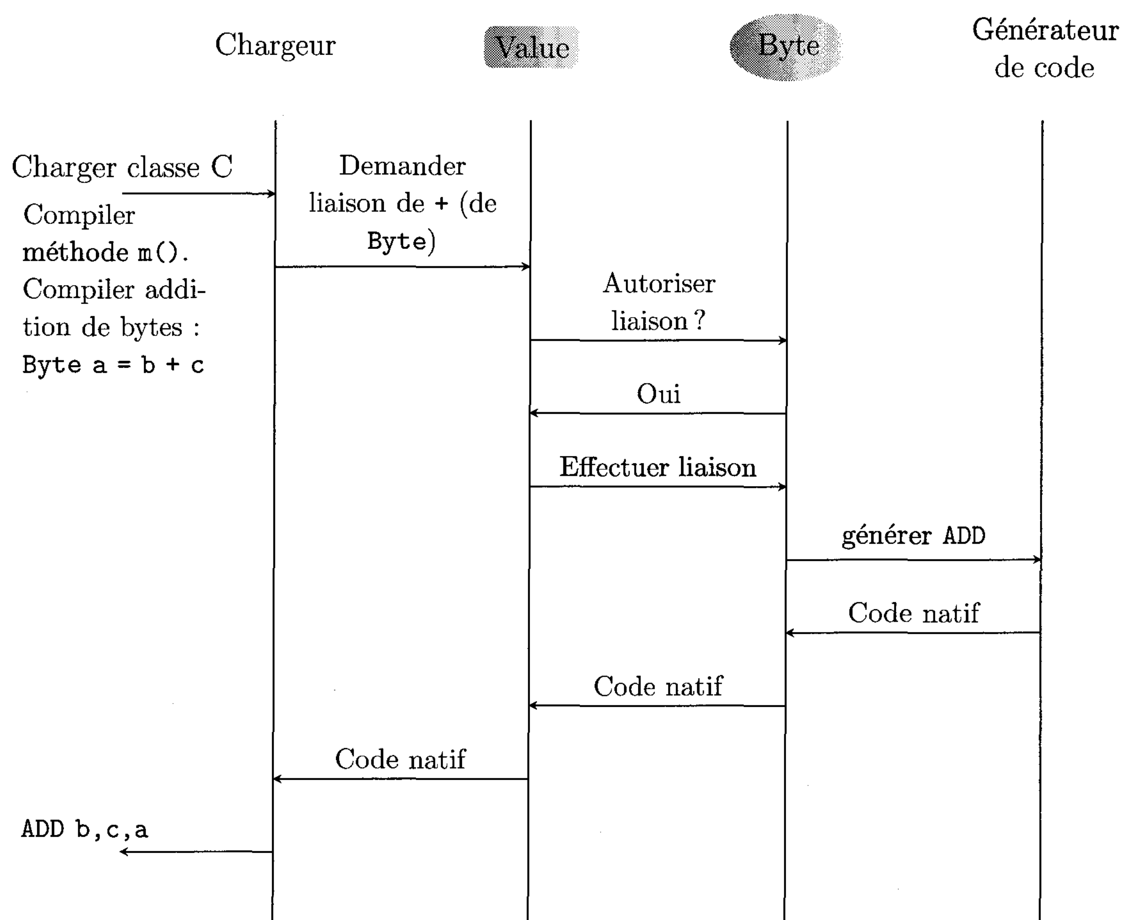


FIG. 3.9 – Schéma de liaison.

particulière présente dans chaque classe, `allowBind()`, qui a pour seule mission de répondre par l'affirmative ou la négative à la question de savoir si le contexte de liaison autorise l'accomplissement de celle-ci.

On peut voir cette méthode comme l'équivalent d'un mécanisme classique dans les systèmes à objets : les droits d'accès. En Java, suivant qu'une méthode est publique ou privée (entre autres) il est possible ou non à certaines méthodes de l'invoquer. Ici, le principe est généralisé, dans le sens où il n'y a pas de granularité fixe pour les permissions comme en Java. Au contraire, il est possible d'accorder des permissions très précises pour certaines classes, voire même pour certaines méthodes : toutes les informations disponibles dans le contexte de liaison sont susceptibles d'être utilisées pour effectuer la discrimination entre les liaisons autorisées et les liaisons interdites.

Néanmoins, comme dans la plupart des systèmes objets, il convient, pour une méthode surchargée, de se conformer au comportement établi par la version de la classe parente, de sorte qu'elle ne rende pas possible une chose qui a été interdite auparavant. En conséquence,

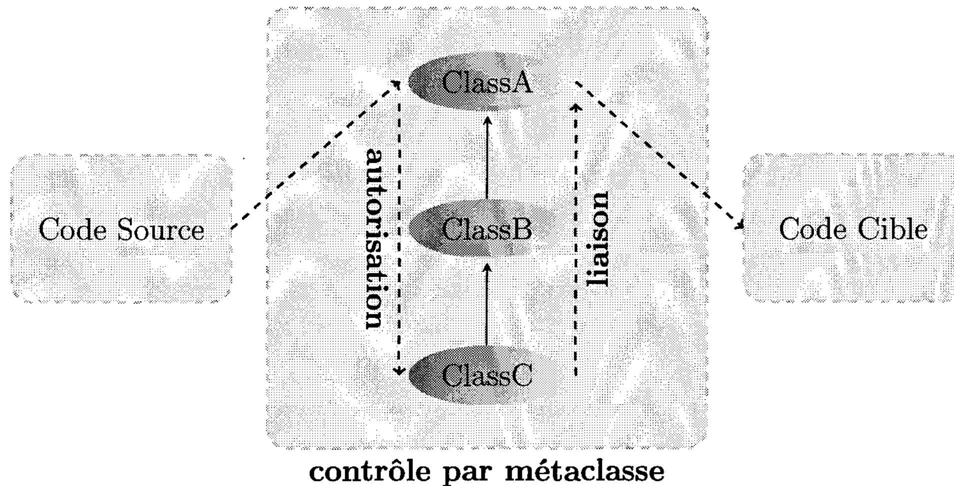


FIG. 3.10 – Autorisation et liaison effective.

l'élargissement des droits d'accès dans une sous-classe est interdit. Ceci est matérialisé par le fait que la méthode `allowBind()` est appelée non seulement dans la classe concernée par la liaison, mais aussi dans toutes ses super-classes, comme il a été précisé précédemment. Les appels à ces méthodes sont en fait effectués depuis la classe la plus élevée (au sens de la hiérarchie de types) possédant la méthode considérée jusqu'à la classe qui reçoit la demande de liaison : si une classe parmi les éléments de cette hiérarchie refuse la liaison dans le contexte donné, alors celle-ci est rejetée.

Si la liaison est acceptée, alors le processus se poursuit. Notons que pour éviter d'imposer l'écriture d'une méthode `bind()` à chaque niveau de la hiérarchie des types, il est légal d'autoriser une liaison sans pour autant fournir une implémentation de la liaison. Dans ce cas, la réalisation (ou non) de la liaison est laissée à la discrétion de la super-classe. Ce mécanisme correspond à l'existence d'une méthode héritée non surchargée : la classe fille possède effectivement la méthode, mais son comportement est entièrement déterminé par la classe parente.

Au contraire des appels à `allowBind()` qui circulent des classes parentes vers les classes filles, les appels à `bind()` remontent la hiérarchie depuis la classe visée par la liaison : la première classe qui définit une liaison pour la méthode concernée est chargée de la liaison. La figure 3.10 présente l'agencement de ces différents appels au sein du mécanisme de liaison lorsqu'un appel à une méthode définie à partir de `ClassA` doit être concrétisé sur un objet de `ClassC`. Toutes les actions de coordination entre les différentes phases de négociation de la liaison (appels successifs aux classes parentes ou filles) sont effectuées par le biais de la métaclasse commune aux différentes classes impliquées, laquelle existe nécessairement puisque ces classes sont reliées par relation d'héritage. Les métaclasses sont donc de fait placées dans la base de confiance du système. Notons que le chargeur de code ne joue aucun rôle à ce niveau de la chaîne de chargement du code.

Liaison effective

A priori, l'addition d'entiers n'a aucune raison d'être rendue illégale ; aussi suppose-t-on que la vérification est effectuée et la permission accordée. Par ailleurs, il semble probable que la liaison concrète de cette opération soit réalisée à l'aide d'une simple instruction machine, de sorte que la méthode `bind()` de la classe `Byte` définit elle-même son processus de liaison pour la méthode `+`. La fin du processus conduit à la génération d'une simple instruction assembleur `ADD` entre deux registres issus du contexte d'appel de la méthode, laquelle remonte jusqu'au chargeur pour écriture dans la mémoire de code.

Pour revenir à l'exemple d'attaque présenté précédemment en figure 3.8, le premier jeu de tests (préalables à la liaison proprement dite) ne déclenche pas d'erreur pour la simple raison que la méthode `cast()` ne fait rien. Le problème se situe au moment où le `callBind()` invoqué par l'attaquant tente de matérialiser un appel à une méthode virtuelle qui ne dispose d'aucun support légal dans le contexte courant, puisque s'appliquant sur un `int`. La détection de ce problème a donc lieu dans la métaclasse correspondant à la méthode réellement invoquée (la métaclasse de `A_CLASS`), lorsque `ctxt` est mis en correspondance avec la signature de la méthode, ce qui met en évidence le fait que le premier paramètre est d'un type incorrect et incompatible pour l'invocation considérée. De cette manière, la liaison est (heureusement) rejetée, ce qui provoque l'échec de l'installation du composant fautif.

3.4.3 Simulation de politiques de sécurité

Outre les considérations détaillées précédemment, qui se rapportent à la garantie de fiabilité du typage des applications, il est également possible de tirer avantage de la structure du schéma de liaison en vue de simuler un certain nombre de politiques de sécurité annexes.

Par exemple, une spécificité des systèmes Javacard est l'utilisation d'un mécanisme de pare-feu chargé de vérifier le bien-fondé des appels de méthodes : tout passage de message est susceptible de faire l'objet d'un contrôle qui ne se limite pas au typage, mais peut par exemple servir à garantir la non-interférence entre deux applications.

Ce genre de mécanisme pourrait être implanté par la génération, avant le code des méthodes, d'un code chargé de faire cette même vérification. La génération d'un tel code pourrait être tissée avant l'appel aux méthodes de liaison, de la même façon que nous avons présenté le tissage de vérification d'invariants en section 3.3.2.

Dans le même esprit, bien qu'à un niveau différent, Java définit la notion de « chargeur de classes »[2]. Ces chargeurs étant essentiellement des conteneurs de classes, ils servent à définir les zones de visibilité de classes chargées, ceci à des fins d'isolation essentiellement. La structure associée de « gestionnaire de sécurité »[3], quant à elle, permet de définir des

[2]class loader.

[3]security manager.

droits à l'exécution.

Le couple formé par `allowBind()` et `Bind()` (autorisation de liaison, liaison effective) peut encore une fois remplir le même office, avec la différence que l'exécution de ces étapes a lieu au chargement et non à l'exécution. De ce fait, le système de liaison intégré dans la structure des métatypes est à la fois plus expressif, et potentiellement moins coûteux.

Un exemple de droit d'accès évolué pourrait être matérialisé par la nécessité pour un composant de répondre à une question d'ordre cryptographique afin d'être autorisé à en utiliser un second. Typiquement ce type d'authentification ne nécessite pas d'être rejoué plusieurs fois, aussi le déporter à l'exécution est inutile. En revanche il trouverait parfaitement sa place durant la période de chargement. Une fois la liaison établie, plus rien ne la distingue des autres, et le composant est authentifié de façon permanente.

Cependant, si dans d'autres circonstances il s'avère nécessaire de recourir à une politique d'accès dépendant de l'exécution, il reste possible de générer un code de vérification dans la liaison des méthodes, si bien que le mécanisme à l'exécution reste simulable.

Notons néanmoins dans Java la possibilité d'intervenir au niveau de `findClass()`, ce qui devrait fournir un moyen d'intervenir en période de chargement même si aucun support n'est fourni dans ce sens. Une solution basée sur cette approche pourrait probablement donner lieu à des comportement similaires à ceux définis dans notre cas au chargement, mais serait très certainement complexe, et génèrerait probablement une duplication d'efforts avec la fonction de création de classe proprement dite, sur laquelle il n'est pas possible d'intervenir.

3.5 Liaison et métatypes

Les notions de métatypes et de processus de liaison se trouvent donc être très liées, par la volonté d'une part de disposer d'un modèle unique au niveau de la conception (unification des types) et d'autre part de disposer d'un modèle unique au niveau du déploiement (unification des processus en tant que simple exécution de code). L'intégration du mécanisme de liaison au cœur des métatypes a une impact sur les propriétés du système à la fois sur le plan de la sécurité du déploiement et sur celui de l'optimisation des performances.

3.5.1 Facteur de sécurité

L'une des motivations fortes de l'utilisation des métatypes est de leur faire jouer un rôle de point de passage obligatoire pour le processus de compilation, de sorte que les vérifications qui doivent être faites puissent être centralisées aisément et effectuées dans un contexte suffisamment contrôlé et fiable pour que le code généré soit conforme aux attentes du concepteur de l'application comme à celles de l'utilisateur, en terme de sécurité et d'isolation. Par ailleurs, la centralisation permet de réduire à la fois la taille du code de contrôle et les risques engendrés par une réplication abusive. De ce fait, le nombre de failles et d'opportunités de négligence est grandement réduit.

La règle générale en ce qui concerne les opérations de vérification de type, ou de transtypage (nécessaires en interne), est qu'elles doivent être effectuées dans le corps d'une méthode appartenant à un métatype (et donc dans la base de confiance). Cette politique de sécurité assure notamment l'impossibilité (car il s'agit d'un comportement illégal) pour un utilisateur quelconque d'avoir recours à une opération de transtypage. Néanmoins, une telle opération peut être sûre et utile lorsqu'elle est encapsulée dans un appel système qui s'assure qu'elle est employée avec toutes les précautions nécessaires : elle peut ainsi faire partie d'un bloc d'instructions qui séparément enfreignent les règles de typage, mais qui, lorsqu'on les considère dans leur ensemble, le respectent globalement. Bien que n'étant pas autorisé à exécuter directement une action de transtypage, l'utilisateur pourra donc bien sûr faire certains appels au système, correctement typés, qui à leur tour effectueront en sous-main les appels risqués. Toute la différence réside dans la connaissance globale de la situation dans laquelle les entorses au système de type ont lieu. Notons que ces appels système doivent être la dernière étape avant que le code ne soit converti en son équivalent en langage machine et ne perde définitivement de ce fait toute notion de ce qu'est un type.

Cette restriction concernant les possibilités de transtypage pose naturellement problème lorsque le langage source à partir duquel un programme est compilé possède lui-même une notion de transtypage assez libre (comme le C par exemple). Dans un tel cas, la compilation vers un langage intermédiaire au typage strict n'est pas toujours possible. Néanmoins, certaines opérations de transtypage peuvent être traduites lorsque leur effet est contrôlable : par exemple, une instruction C contenant une conversion d'un type `int` vers un type `byte` peut être facilement compilée vers un code bien typé, ce qui n'est pas le cas d'une conversion d'un type `int` vers un type `char *`. Certains programmes « corrects » (au sens défini par le langage source) peuvent donc se révéler impossibles à compiler vers le langage intermédiaire et donc être rejetés comme incorrects car ne respectant pas les contraintes, plus fortes, imposées vis-à-vis du code généré. Ou inversement, on peut considérer que seul un sous-ensemble convenablement typé des programmes exprimables dans ces langages peut être traduit dans le langage intermédiaire.

Concernant les vérifications explicites de typage, elles peuvent bien sûr être autorisées, mais se doivent d'échouer ou d'être redondantes, et par là même d'être inutiles. De sorte qu'elles pourraient en fait aussi bien être interdites, sans que cela ne restreigne en aucune façon la sécurité du système. Encore une fois, cette politique n'est pas extérieure au système de type lui-même, puisqu'elle est très simplement implémentée en usant du mécanisme de liaison : il s'agit juste de vérifier le type d'un objet, ce qui est une information parfaitement connue, et de façon fiable, lors de la phase de liaison.

Un avantage supplémentaire d'une dernière vérification de type avant la génération finale de code machine est qu'elle procure un mécanisme de conversion de code aisément vérifiable du point de vue du typage : en effet, les seules hypothèses nécessaires sur une plate-forme donnée deviennent alors le bon comportement de l'ensemble des primitives de génération de code (les méthodes du générateur), qui sont généralement très simples et ne nécessitent pour la plupart aucun effort de preuve. En s'appuyant sur ces comportements, l'évolution du typage des objets dans le processus de chargement est correcte par

construction.

3.5.2 Facteur d'optimisation

Bien que cela n'ait pas été une préoccupation majeure à la conception du système de types, les opportunités d'optimisation offertes par l'adoption des métatypes sont vraiment très intéressantes : tout d'abord à cause de la factorisation d'une grande partie du code, qui a réduit la taille globale du système de façon non négligeable, mais surtout par le fait qu'en introduisant un processus de compilation tardif au chargement, il est possible de réaliser des optimisations basées sur des analyses qui ne pourraient être menées de manière totalement statique (à savoir en dehors de la plate-forme de déploiement et exécution). Par exemple, si l'on veut s'assurer qu'une méthode donnée n'est jamais appelée que depuis une certaine classe (un équivalent des classes « friend » du C++), cette vérification ne peut en aucun cas avoir lieu de manière purement statique puisque les systèmes considérés sont ouverts. Par ailleurs, une vérification purement dynamique serait extrêmement coûteuse. En revanche, en période de chargement, il est trivial d'injecter une telle vérification dans la méthode `allowBind` concernée, de sorte que toutes les invocations soient garanties de respecter cette contrainte par construction.

On peut voir l'étape de chargement de code comme une période durant laquelle l'univers des applications est localement fermé : il est donc possible d'effectuer des analyses semi-statiques du code qui s'appuient sur le contexte d'utilisation de la méthode courante. Cette technique preśente deux avantages majeurs : d'une part, elle permet d'éviter le coût d'une vérification à l'exécution pour chaque appel, et d'autre part, elle réduit la nécessité de recourir à une programmation défensive elle aussi coûteuse, puisque les appels illégaux dans le corps d'une méthode aboutiront à son rejet.

3.6 Résultats expérimentaux et possibilités d'extensions

3.6.1 Camille et Façade

L'implémentation originale de l'architecture précédemment décrite a été réalisée pour le système CAMILLE[Gri], qui est un système d'exploitation dont le domaine d'application est l'ensemble des plates-formes de petites tailles comme les cartes à puce. CAMILLE est basé sur un exo-noyau fournissant un nombre aussi réduit que possible de services supposés fiables, et un mécanisme d'extension permettant :

- de charger de nouvelles applications (ce en quoi il diffère des systèmes historiques pour carte, dont la philosophie était plutôt d'effectuer une tâche immuable) ;
- de charger de nouveaux composants pour le système, de façon à étendre ses possibilités, voire à corriger l'existant.

Bien sûr, ces différentes possibilités tiennent davantage de la vue de l'esprit que de la vraie multiplicité, puisque le système est conçu de telle sorte que ces différentes motivations relèvent des mêmes mécanismes.

Par défaut, CAMILLE charge un code intermédiaire écrit dans le langage FAÇADE, qui répond aux attentes énoncées précédemment. Il s'agit d'un langage totalement orienté objet, dont les métatypes font désormais partie. L'avantage de l'utilisation de FAÇADE a été montré dans [RCG00] : à partir de ce code, CAMILLE est en mesure d'effectuer une vérification de type sur le code chargé en utilisant un mécanisme proche du PCC [Nec97] permettant une vérification linéaire [RR98]. Ceci rend le système embarqué capable de s'assurer, sans assistance extérieure, du bon typage des applications qu'il charge.

FAÇADE est un langage totalement orienté objet très simple, dont la propriété de typage correct peut être très facilement vérifiée. Le langage lui-même est composé en tout et pour tout de cinq instructions : `return`, `jump`, `jumpif`, `jumplist` and `invoke` et utilise un jeu de registres virtuels en tant que variables de programme. FAÇADE présente donc des caractéristiques proches des langages assembleurs très bas niveau.

La correction du typage d'un programme est garantie par preuve lors du chargement de celui-ci [RCG00] et prélude à la traduction de FAÇADE vers un code exécutable (généralement du code natif à destination du processeur de la plate-forme considérée). Grâce à ces propriétés, le système CAMILLE fournit des garanties de portabilité, sécurité et extensibilité.

Le langage intermédiaire FAÇADE correspond bien aux attentes exprimées en section 3.2.1. Il est extrêmement compact, puisqu'il ne comporte en fait qu'une notion de fonction (`invoke`), les instructions de retour de fonction et de flot de contrôle jouant un rôle plus basique en ce qui concerne les concepts exprimés.

La notion de classe en elle-même est des plus minimales, puisqu'elle correspond globalement à un bloc de code associé à un espace de noms. Ce bloc de code contient lui-même des instructions pour définir ce qu'est la classe, mais il n'y a pas de contrainte structurelle à ce niveau. Ceci laisse toute latitude pour intégrer les différents modèles objet classiques dans ce schéma simple.

De ce fait, l'intégration des métatypes et de l'ensemble des mécanismes précédemment décrits dans l'architecture existante n'a pas posé de problème particulier. Ceux-ci ont apporté une extensibilité accrue au système, tout en conservant un souci fort de sécurité, par le contrôle strict des fonctions d'extensibilité accessibles à toutes les applications. Comme préconisé en section 3.2.6, l'ajout de métatypes supplémentaires est inaccessible à une application non-privilégiée, de sorte que toute classe créée durant la vie du système l'est sous l'une des deux conditions suivantes :

- une autorité légitime accepte la classe comme pouvant interagir sainement avec le système en place ;
- la classe hérite d'une classe existante en respectant les contrôles de sécurité mis en place.

La mise en place de la structure de métatypes dans CAMILLE a provoqué une légère augmentation de la taille du système (classes supplémentaires), contrebalancée par un gain dû à la factorisation de l'ensemble des politiques de sécurité en place, non seulement en termes de volume de code mais également en termes de fiabilité puisque les points de contrôle sont moins nombreux. Globalement, on estime à 10% du total le volume de code matérialisant les métatypes (dans un système d'une taille approximative de 120 Ko).

3.6.2 Proposition d'héritage multiple

La possibilité d'introduction d'un héritage multiple a été rapidement évoquée auparavant, mais sans entrer dans les détails de la démarche à suivre : comment ajouter relativement simplement cette notion dans un système ne comportant *a priori* que de l'héritage simple, et ce bien évidemment, en conservant les propriétés existantes du système ? Dans cette section, nous allons nous intéresser aux différentes étapes d'intégration d'une telle extension.

L'ajout d'une nouvelle notion d'héritage n'est évidemment pas neutre vis-à-vis de la sécurité du système car elle redéfinit partiellement la relation entre classe et représentant. De ce fait, une erreur à ce niveau peut mettre en péril le système complet, en fournissant un moyen de violer une règle de sécurité. C'est pour cette raison que l'ajout d'une nouvelle sorte de classes d'objets ne peut être laissé à la portée de l'utilisateur.

La matérialisation d'une sous-hiérarchie de classes à héritage multiple se ferait en plusieurs étapes. La première, primordiale, est l'introduction d'un nouveau métatype qui servira de type à toutes les classes concernées, et qui définira donc la possibilité pour un objet d'avoir plusieurs parents. Par ailleurs, ce métatype sera probablement (sans qu'il y ait d'obligation) placé sur le même plan que `Class` et donc héritant de `Boxed` (afin de bénéficier d'un type dynamique). Cette situation est présentée en figure 3.11.

La description dans le métatype des comportements des objets implique notamment la description de la relation qui existe entre un objet et *ses* classes de rattachement, ce qui est une nouveauté par rapport aux autres types d'objets déjà présents. La méthode `isAKindOf` devra donc être réécrite afin d'avoir un comportement adapté à l'héritage multiple.

En parallèle, l'existence d'une possibilité d'héritage multiple implique des adaptations de structure au niveau de l'objet. Dans un modèle à héritage simple, l'héritage implique juste d'*ajouter* des entrées (champs, méthodes) à un objet, ce qui n'invalide jamais la description fournie par les classes parentes. Dans un modèle à héritage multiple, les mécanismes sont plus complexes, puisqu'ils requièrent le rassemblement de différentes descriptions.

Tout mécanisme objet est concrétisé par l'appel d'une méthode lors de la construction d'une classe (comme présenté en section 3.2.6), et le fait d'hériter d'une autre classe ne fait pas exception. Il existe donc une méthode `setParent()`, chargée de matérialiser l'héritage au niveau structurel, qui doit donc également être réécrite en conséquence. Pour un héritage simple, cette méthode doit être appelée exactement une fois, tandis que pour un héritage multiple, un nombre arbitraire d'invocations doit être autorisé, pourvu que toutes les classes

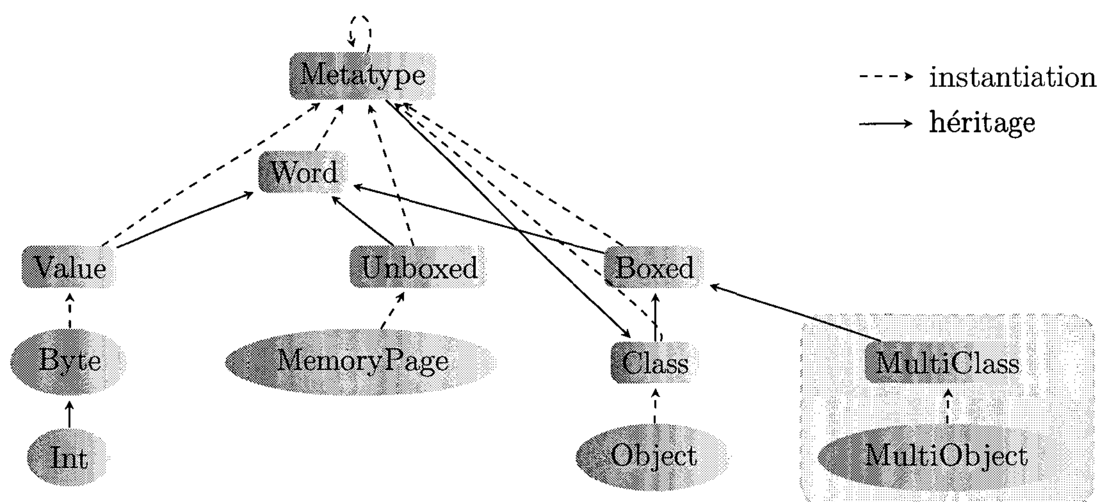


FIG. 3.11 – Hiérarchie des métatypes

déclarées parentes supportent elles-mêmes un héritage multiple. Au passage, on notera qu'il existe une vérification primordiale concernant l'héritage : une classe ne peut en aucun cas déclarer hériter d'une classe d'un métatype incompatible, cela n'ayant aucune signification.

L'un des effets des invocations multiples à `setParent()` est donc le rassemblement des méthodes et des champs des classes considérés. L'une des possibilités pour gérer cet effet est de fournir au sein de la classe une table d'indirections vers chaque classe parente, de façon à rendre possible l'association d'une méthode ou d'un champ du type *statique* d'un objet avec la méthode ou le champ correspondant dans l'implémentation.

De là, le problème de la recherche des informations à travers la hiérarchie se pose. En effet, il est nécessaire d'envisager des conflits de descripteurs entre diverses branches amenées à fusionner. L'acte même d'invocation de méthode doit donc être adapté à cette nouvelle situation. Il est possible d'imaginer l'implémentation de tous les mécanismes de résolution d'appel de méthode existant dans divers langages (priorité selon l'ordre d'héritage par exemple). Remarquons que le choix fait à ce niveau pourrait être remis en question dans une branche encore différente d'objets à héritage multiple, mais qui seraient alors incompatibles.

D'autres problèmes se situent dans des parties moins triviales du processus de chargement. En effet, l'une des particularités du langage intermédiaire considéré (FAÇADE) est qu'il utilise des variables *temporaires*, dont le type n'est pas déclaré mais inféré. Dans un cas simple comme celui de la figure 3.13, où `tmp` est une variable temporaire *a priori* non typée, le phénomène suivant se produit. Si les objets manipulés sont à héritage simple, le type de `tmp` à la sortie du bloc serait inféré comme étant le plus petit parent commun à `A` et `B`. En revanche, si l'héritage est multiple, le cas est légèrement plus compliqué puisque le plus petit parent commun n'a aucune raison d'être unique. Par conséquent tout choix

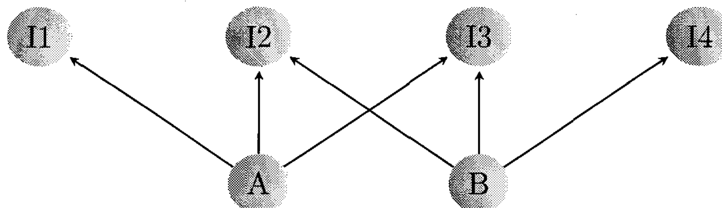


FIG. 3.12 – Héritage problématique

arbitraire à ce niveau empêcherait de considérer l'interface complète exposée par `tmp`. Par exemple, dans le cas exposé en figure 3.12, le type de `tmp` devrait être `I2` *et* `I3`.

Une fois encore, ce problème trouve une solution élégante dans les métatypes. En effet, lorsqu'une unification entre deux types est demandée, une méthode de leur métatype est appelée (si le métatype n'est pas le même, le code est incorrect), qui retourne une instance (un type donc). Dans le cas de l'héritage simple, cette instance est un type existant (le plus petit parent commun), tandis que dans le cas de l'héritage multiple, il n'existe pas nécessairement d'instance du métatype correspondant à l'interface souhaitée.

Dans l'exemple, les seuls types disponibles pour accepter les interfaces `I2` et `I3` sont `A` et `B`, mais toutes deux ont des interfaces supplémentaires incompatibles avec une utilisation correcte de l'objet considéré.

Cependant, cette instance peut aussi bien être générée à la demande, puisqu'elle est pleinement spécifiée : dans le cas qui nous occupe, il suffit de générer à la demande une classe héritant uniquement de `I2` et `I3`. Ce type « virtuel », puisque ne correspondant à aucune nécessité concrète du programme, peut d'ailleurs être éliminé à la fin du processus de chargement si besoin est, car jamais un objet de ce type ne sera construit : le type « virtuel » ne sert qu'à la vérification. Il est intéressant de remarquer que dans la solution envisagée pour cet exemple on a recours à une simulation de typage par interface dans un langage n'autorisant qu'un typage par classe.

La demande d'unification de deux types (si elle est possible) étant effectuée par le biais de l'appel à une méthode `unify`, la simple surcharge de cette méthode dans le métatype `MultiClass` suffit à résoudre ce problème. Notons néanmoins que dans le cas d'un langage requérant un typage explicite de l'ensemble des variables manipulées, le mécanisme d'unification n'aurait pas lieu d'être, puisqu'il représente simplement l'opération de base d'un système d'inférence.

En résumé, on peut conclure de cette expérience deux choses : l'ajout de concepts objets différents de l'existant est très accessible, mais les conséquences sont parfois non-triviales. Sous réserve que le métatype soit correctement implanté, il s'intègre harmonieusement dans le système de types existant, sans causer d'interférences. Néanmoins, il est nécessaire que la possibilité d'enrichir le système d'un nouveau métatype (et donc d'une nouvelle classe de types dont les propriétés diffèrent radicalement de celles des types existants et ne peuvent être contrôlées par le système en place) ne soit pas accessible au même titre que l'extension

```
if(condition) {
  tmp = f(); // returns a A
} else {
  tmp = g(); // returns a B
}
```

FIG. 3.13 – Unification des types

« normale » du système par ajout de classe héritée.

3.7 Consistance du mécanisme de liaison

3.7.1 Liaison

Une méthode est *liée* à chaque fois qu'elle est appelée (via un bytecode `invoke`) par une nouvelle méthode que l'on charge dans le système. L'effet induit vise à matérialiser par une opération de compilation (génération de code natif par exemple) ce que sera l'« invocation » de la méthode. Cet effet peut prendre des formes très diverses : ainsi, dans le cas le plus « simple », la liaison d'une méthode se traduira par l'application d'une convention d'appel habituelle (saut au début de la fonction appelée, au niveau de la gestion des paramètres, etc.). Mais on pourra également envisager des constructions plus subtiles telles qu'une expansion en ligne (*inlining*) du corps de la fonction (d'où une réplication du code utile, contre le gain de la convention d'appel), ou encore telles que le tissage de certains aspects autour de la fonction appelée.

Il est important de noter encore une fois que le bytecode ne reflète pas directement la structure du programme qui sera exécuté en dernier lieu. Au lieu de cela, il fournit au chargeur de code les instructions nécessaires à la composition du programme, à partir d'éléments préexistants (déjà chargés). Le bytecode sera donc interprété par le chargeur de code comme une série d'opérations de génération de code.

À titre récapitulatif, et pour mieux cerner les propriétés à démontrer, les descriptions d'actions suivantes sont équivalentes :

- compiler le code correspondant à un appel à une méthode f ;
- lier la méthode f dans le contexte de liaison approprié ;
- exécuter une méthode $Bind^f$ (par la sémantique des invocations de méthode) ;
- exécuter le résultat de la liaison d'une séquence d'invocations (le jeu de bytecode de FAÇADE ne contient rien d'autre que des invocations, hormis les sauts et instructions de retour de fonction) ;
- exécuter le résultat d'une séquence de résultats d'exécutions de $Bind^*$ (la méthode de liaison a elle-même été liée lors de son chargement, ce qui introduit une récursivité

```
void Bind(method f) {
  ctxt = ctxt(f);
  check(pre(f),ctxt);
  exec(Bind_f,ctxt);
  check(post(f),ctxt);
}
```

FIG. 3.14 – Mécanisme de chargement

dans la construction du système) ;
- exécuter le résultat d'une séquence de résultats d'exécutions de $Bind^{*}_{kernel}$ (toutes les méthodes de liaison sont soit écrites en FAÇADE, soit en code natif pour certaines méthodes du noyau).

Remarque : dans le schéma présenté, on suppose que lors du chargement d'un composant, toute structure dont ce dernier pourrait avoir besoin est déjà chargée et prête à l'emploi. Cet état est garanti par le chargeur de code lui-même, qui ne saurait exécuter une méthode (de liaison en l'occurrence) non chargée. Le corollaire de cette propriété est qu'il est impossible d'introduire des dépendances circulaires ou même récursives dans le mécanisme de chargement, et ce faisant de briser le déroulement purement linéaire du processus de chargement. Notons par ailleurs que cette limitation n'empêche nullement la génération d'un code qui puisse concrétiser des appels de méthodes récursifs.

Dans la réalité, le mécanisme est un peu plus complexe, puisque le chargement d'une classe se déroule en plusieurs étapes, comme montré précédemment en figure 3.6. Toutefois, la partie qui nous intéresse (la méthode `bind()`) étant associée à la phase de déclaration de la classe, l'approximation reste valide.

Le schéma de base du mécanisme de liaison d'une méthode est présenté dans la figure 3.14.

Le contexte `ctxt` représente les paramètres effectifs de `f`, qui sont ainsi rendus accessibles (sous leur forme « statique » et non pas en tant que valeurs dynamiques) au mécanisme de liaison. Libre à celui-ci de générer à partir de ces données le code d'un contrôle dynamique de valeur par exemple.

`pre(f)` et `post(f)` sont les pré- et post-conditions associées à `f` : il s'agit de prédicats relatifs à la signature (au sens du typage) de `f`. `Bind_f` est la méthode de liaison effectivement invoquée lorsqu'un appel à la méthode `f` est présent dans le code d'une méthode en cours de chargement. Ainsi, l'« invocation » de `f` correspondra, lors de l'exécution du programme, à l'exécution du code en résultat de cette méthode de liaison.

Remarque : les seuls contrôles qui ont lieu lors de la liaison d'une méthode `f` dans un certain contexte sont `pre(f)` et `post(f)`. Les autres vérifications, faites « récursivement » lors de l'appel en cascade des méthodes de liaison jusqu'à arriver au noyau, ont été appelées bien avant le démarrage du chargement courant (et ont réussi), lors du chargement des

composants concernés.

3.7.2 Preuve de cohérence vis-à-vis du système de types

Étant donné que le code généré pour un programme n'est pas en correspondance directe avec le code source (intermédiaire) qui le décrit, il est naturel de s'interroger sur la validité des transformations exécutées vis à vis du système de types. En particulier, il serait inacceptable qu'un code *a priori* bien typé puisse déclencher la génération d'un code sémantiquement équivalent à un programme mal typé.

La garantie de cette propriété de bon typage est bien entendu fortement dépendante des points de contrôle insérés dans le processus de chargement, et notamment des vérifications effectuées par les métatypes au cours du processus.

Définitions préalables

Définition 1 (Listes). *On notera $[h|T]$ la liste composée de l'élément h suivi de la liste T. La liste vide sera notée $[]$.*

Définition 2 (Typage). *Un état des types est une partie du produit de l'ensemble des registres virtuels V disponibles et de l'ensemble des types possibles T.*

Pour τ un état de types, on notera $Dom(\tau)$ le domaine de définition de τ, à savoir la projection de τ sur V (tout registre n'est pas nécessairement affecté à tout instant).

Définition 3 (Instructions). *L'ensemble des instructions disponibles pour composer un programme est noté I.*

Définition 4 (Résultat d'exécution). *L'ensemble des résultats possibles d'une exécution vis à vis du typage est $R = 2^{(V \times T)} \cup \{Failure\}$, constitué de l'ensemble des états de types possibles, ainsi que de l'état particulier $Failure$ indiquant un échec dans la chaîne de liaison.*

Définition 5 (Exécution). *On note $Exec : I \to R \to R$ la fonction d'exécution d'une instruction.*

Suffisance des contrôles

Soit $s = [h|T]$ une liste arbitraire de $Bind^*_{kernel}$ représentant la suite des instructions permettant la génération du code d'une fonction f et τ un état des types quelconque (l'état des registres avant l'exécution de f). Soit $\tau' = \tau \cup \{(ret, \alpha)\}$, avec $ret \notin Dom(\tau)$ et α le type correspondant au type de retour annoncé pour f.

Le prédicat $Exec(s, \tau) \supset \tau'$ est vérifiable avec les points de contrôle installés dans le processus de chargement, ce qui signifie que $\tau' \not\subset Exec(s, \tau) \Rightarrow Exec(s, \tau) = Failure$.

$$Exec([], \tau) = \tau \tag{3.1}$$
$$\forall s, Exec(s, Failure) = Failure \tag{3.2}$$
$$pre(h) \not\subset \tau \Rightarrow \forall T, Exec([h|T], \tau) = Failure \tag{3.3}$$
$$post(h) \not\subset Exec([h], \tau) \Rightarrow Exec(s, \tau) = Failure \tag{3.4}$$
$$pre(h) \subset \tau \Rightarrow Exec([s|T], \tau) = Exec(T, Exec([h], \tau)) \tag{3.5}$$

TAB. 3.1 – Règles d'exécution.

La vérification de ce prédicat implique que l'ensemble des types fixés avant exécution de la liste d'instructions s est conservé intact par l'exécution, et que le type de retour de la fonction est également respecté. De ce fait, les transformations effectuées lors du chargement de la méthode m sont valides vis-à-vis de son typage.

Invoke

Montrons que le prédicat $Exec(s, \tau) \supset \tau$ est vérifiable : la restriction de l'état final (après exécution) au domaine de l'état initial (avant exécution) est l'état initial.

Pour tout $\tau \subset V \times T$, les équations présentées en table 3.1 sont vérifiées.

L'équation (3.1) indique qu'une action vide n'a aucune incidence sur le typage courant des registres. L'équation (3.2) correspond à la règle de propagation d'erreur dans l'exécution. Les équations (3.3) et (3.4) fixent les cas dans lesquels un échec est généré. Enfin, l'équation (3.5) correspond au cas régulier, dans lequel les conditions sont respectées.

Ainsi, $Exec([x_1, \dots x_{n-1}, x_n], \tau) \neq Failure$ implique que

$$Exec([x_1, \dots x_n], \tau) = Exec([x_n], Exec([x_{n-1}], \dots (Exec([x_1], \tau) \dots)$$

Comme τ est préservé au cours de chaque étape, il est préservé par la composition de toutes. □

Return

Supposons que (ret, α) n'appartienne pas à $Exec(s, \tau)$.

Étant donné que s représente la séquence de liaison de f, le mécanisme de liaison (présenté dans la figure 3.14) vérifie que l'état final des types des objets du système est compatible avec la signature de f (qui retourne un type α).

Par conséquent cette vérification soulève une erreur, et le prédicat est vérifiable. □

Existence des contrôles

Pour conclure que le mécanisme de chargement conserve bien la sémantique des types (et donc les résultats d'une inférence de type éventuelle), il reste à établir que les contrôles utilisés pour justifier la cohérence du système sont effectivement présents dans le mécanisme de chargement.

Or, ces contrôles sont précisément les points qui ont été centralisés dans les métatypes. Pour rappel, dans la figure 3.9, l'accord du métatype est sollicité avant toute opération de liaison effective, ce qui se traduit par l'application de `pre(f)`, à savoir le contrôle du contexte d'appel. Réciproquement, le code produit transite en premier lieu par le métatype, qui contrôle alors le contexte de retour.

Le passage impératif par les métatypes garantit donc (sous réserve que ces derniers soient corrects) que le mécanisme de transformation de types respecte la sémantique de ces derniers. Le code généré ne peut donc pas avoir le comportement d'un programme dont le typage serait différent du code source duquel il a été extrait. Notons encore une fois que cela n'engage bien sûr pas la sémantique fonctionnelle du programme.

3.8 Conclusion

Dans ce chapitre nous avons présenté une architecture destinée à un système d'exploitation ouvert et flexible. Nous avons pu voir que les notions d'efficacité et de flexibilité ne s'opposaient pas nécessairement, à travers l'utilisation d'un mécanisme de liaison au chargement qui permet à la fois la génération de code efficace (en particulier natif) et l'expression de propriétés suffisamment fines pour permettre la concrétisation de procédés proche du tissage d'aspects dans le code (possibilité entrevue à travers l'exemple des invariants). Par ailleurs, l'introduction des métatypes nous a permis de fiabiliser le processus aussi bien que d'introduire une flexibilité importante au sein même de l'architecture, puisque ceux-ci permettent notamment l'introduction de nouveaux concepts objets avec une facilité proche du simple acte d'héritage.

Chapitre 4

Analyse d'alias

Nous avons détaillé précédemment comment tirer partie d'un système de types en amont de la conception de logiciel : jusqu'à maintenant l'approche suivie était essentiellement constructive car les types examinés avaient pour finalité de classifier les objets à leur création, pour ensuite suivre leur évolution.

Dans ce chapitre et les suivants, nous nous intéressons d'avantage à une approche déductive ou analytique du typage, puisque les types considérés relèvent plus de l'information calculée par rapport à un comportement observé (généralement abstraitement), même si il reste *a priori* envisageable d'avoir recours à d'autres mécanismes que l'inférence. Il s'agit donc d'un typage en aval de la conception logicielle, dont la finalité est d'apporter une garantie globale sur le comportement du programme.

Dans tous les cas, l'assurance, préalable à toute exécution, qu'un code ne viole aucune condition de typage ne peut être obtenue que par une analyse plus ou moins fine du code.

Ce chapitre vise à établir les bases nécessaire à l'implantation la plus simple possible de plusieurs analyses, dont une analyse d'échappement. À ce titre, il est important de bien tenir compte des choix (notamment d'approximation) faits ici afin de pouvoir évaluer leur impact dans les analyses futures. De fait, l'analyse d'alias présentée ici nous permettra ultérieurement de déduire de façon quasi immédiate les propriétés qui nous intéressent.

4.1 Analyse de code et interprétation

Des langages tels que Java tentent de rendre la plus simple possible l'accession à cette garantie en aidant par quelques contraintes (ne réduisant pas l'expressivité) au niveau du code à exécuter. Ainsi, une condition nécessaire pour qu'un programme Java soit accepté est qu'il soit possible d'associer une hauteur de pile constante (relativement au contexte de la méthode courante) à tout point d'un programme. Il est clair que cette contrainte simplifie nettement la vérification de la correction du typage de la pile d'exécution.

Suivant la préoccupation motivant l'analyse de code, il est possible de la mener de

plusieurs façons. Certaines analyses qui requièrent un niveau maximal de précision peuvent exiger l'intervention d'un opérateur humain pour être menées à bien : c'est le cas par exemple de certaines preuves de programmes concernant des propriétés indécidables. Des outils de preuve de théorèmes, tels que ceux de Boyer et Moore [BM90, BM88], ou encore Coq [BBC+97], bien que possédant un certain nombre de processus automatiques pour résoudre des situations usuelles, se reposent beaucoup sur l'interaction avec un opérateur pour parvenir au résultat souhaité.

À l'inverse, de nombreux problèmes sont décidables, et peuvent donc être traités automatiquement. Pour ceux-ci, il existe principalement deux méthodologies applicables :

- d'une part, le « model checking » [CWA+96], consistant à déterminer un modèle formel de l'application pour lequel la propriété soit simple à prouver, et à tester l'adéquation de l'implantation avec ce modèle ;
- d'autre part, la technique d'« interprétation abstraite » [CC77], consistant à demeurer au niveau du programme lui-même, en associant à chacune de ses sous-parties une sémantique propre à exprimer l'action effectuée relativement à la propriété considérée.

C'est essentiellement sur cette dernière technique que se baseront les travaux présentés dans le reste de ce document. En effet, elle est très adaptée pour s'assurer de la correction des programmes par rapport à des considérations non-fonctionnelles, telles que le typage et autres propriétés similaires. De fait, on s'intéressera davantage à des notions proches de la forme syntaxique du code qu'à des notions touchant à la correction des résultats produits par son exécution.

Dans ce chapitre, nous présentons une analyse à portée générique baptisée « analyse d'alias ». Dans la littérature, ce genre d'analyse est plutôt appelée « analyse de pointeurs »[1] [WL02, LH99, EGH94], car sa finalité est de détecter les liens de références entre différents objets. Néanmoins, l'usage que nous comptons en faire, ainsi que diverses approximations choisies pour cette analyse particulière rend légitime l'appellation choisie.

4.2 Présentation de l'analyse

La plupart des analyses menées statiquement sur le code, qu'elles soient à but d'optimisation, de sécurité, ou même simplement de statistiques, reposent sur la notion de type, qui permet de décrire la structure des objets ainsi que l'ensemble des opérations qu'il est possible de leur faire subir. Cette notion est tellement centrale qu'elle caractérise bien souvent la propriété minimale de sécurité pour un système : les programmes qui y évoluent doivent être correctement typés pour être autorisés à s'exécuter. Il s'agit en effet d'une propriété importante, garantissant une certaine prévisibilité de l'exécution du programme.

Néanmoins, dans de nombreux cas, la seule connaissance des types, pour nécessaire qu'elle soit, n'est pas du tout suffisante pour mener à bien l'analyse. Aussi bien pour de l'analyse d'échappement que pour de l'analyse de flot, ou encore de la spécialisation de code,

[1] « points-to analysis » en anglais

une connaissance plus approfondie du profil d'exécution est nécessaire. Cette information supplémentaire se résume souvent (sous une forme ou une autre) à la capacité de « suivre » statiquement le cours de la vie d'un objet (en fait l'ensemble des cours possibles). C'est cette information que nous recherchons à l'aide d'une analyse d'alias.

Le résultat de cette analyse nous permettra d'associer à chaque structure représentant un objet dans un programme (variable, retour de fonction) une vue de l'ensemble des objets susceptibles d'être représentés. De la sorte, il sera possible de disposer d'une information distincte des types, concernant le contenu des variables en tout point d'un programme. Comme nous le verrons ultérieurement, cette information pourra être utilisée, au même titre que l'information de typage, par d'autres analyses s'intéressant à des propriétés nécessitant plus d'informations que les types.

Cette analyse est conçue dans un double but de légèreté, et de vérifiabilité. L'analyse en elle-même est assez semblable à ce qui est proposé dans des travaux tels que [WR99] ou [EGH94]. Néanmoins, les choix spécifiques faits dans le contexte d'application jouent un rôle important à la fois dans son applicabilité aux systèmes de petite taille, et dans la possibilité d'étendre cette analyse à des propriétés plus évoluées par la suite.

4.3 Moteur d'alias

Dans cette section, nous décrivons la base de notre analyse d'alias, dont la conception reflète l'ensemble des préoccupations qui sous-tendent ce travail. En particulier, les caractéristiques des plates-formes visées influent énormément sur les choix de conception. Par exemple, en ce qui concerne la plate-forme Java, le « code mobile », qui est une notion particulièrement importante pour les petits systèmes embarqués, est exprimé sous la forme de code intermédiaire (bytecode). Sur d'autres plates-formes, des notions équivalentes existent, et font généralement appel à leur propre forme de code intermédiaire, suffisamment bas-niveau pour être traduit efficacement. De ce fait, l'analyse présentée ici doit œuvrer sur cette forme bas-niveau du code, de façon à être la plus proche possible de la plate-forme d'exécution et ainsi lui permettre de tirer partie le plus simplement possible des résultats de l'analyse.

Dans la suite, nous nous intéresserons essentiellement à Java, qui demeure la plate-forme canonique pour la famille de systèmes qui nous occupe, et dont les fonctionnalités essentielles sont quasiment incontournables dans le contexte embarqué et ont donc presque toujours un équivalent plus ou moins immédiat sur toutes les plates-formes.

4.3.1 Alias et pointeurs

Dans la suite de ce document, si les alias représentent effectivement la notion qui nous intéresse, la façon de les exprimer est assez indirecte. Les alias sont calculés à partir des liens qui se créent entre les différents objets du système. En effet, la connaissance de ces liens permet de savoir à tout moment quel objet est accessible à travers l'utilisation d'une

```
Object f(Stack s) {
  if (condition) {
    s.push(new Object());
  }
  ...
  Object o = s.pop();
  return o;
}
```

FIG. 4.1 – Suivi de liens

variable.

Dans l'exemple donné en figure 4.1, c'est le suivi de lien qui nous permet de détecter que la valeur de retour de la fonction est un objet lié au paramètre s, créé ou non dans le corps de la méthode.

4.3.2 Définitions préliminaires

Dans la suite, O_m dénote l'ensemble de tous les objets qui peuvent être utilisés dans une méthode (Java par exemple) m. Notre but, dans une analyse d'alias, et de calculer avec la plus grande précision possible l'image de O_m par la relation suivante.

Définition 6 (Relation de pointage). *Soit* $\hookrightarrow\ :\ O_m \rightarrow O_m$ *la relation de pointage. La relation* $o \hookrightarrow f$ *se lit "o pointe vers f" et signifie que f est un objet contenu dans un champ de o.*

Note : Afin de simplifier le modèle, mais sans perte de généralité, un tableau est assimilé à un objet comportant un nombre potentiellement infini de champs, ce qui nous permet de dire qu'un tableau pointe vers chacun de ses éléments.

Définition 7 (Atteignabilité). *La clôture transitive de* $\hookrightarrow$ *est notée* $\hookrightarrow^*$*, et la relation* $o \hookrightarrow^* f$ *se lit "f est atteignable depuis o".*

4.3.3 Abstraction

Pour se placer dans le domaine d'application des analyses statiques (abstraites), il est nécessaire de disposer d'une abstraction de la relation $\hookrightarrow$ qui s'applique sur des objets abstraits à la place d'objets concrets (les objets de la phase d'exécution). En effet, il est par définition impossible de décider dans le cas général quels objets concrets seront utilisés par simple analyse du code. Par la suite, sauf exception, il sera fait uniquement référence aux objets abstraits, qui sont les seuls à pouvoir être manipulés à notre niveau.

L'ensemble des objets abstraits est constitué des éléments suivants :

Définition 8 (Objets abstraits pour une méthode). *Soit m une méthode, l'ensemble des objets abstraits pour m est $\breve{O}_m = Para_m \cup \{Ret_m\} \cup \{Except_m\} \cup Alloc_m \cup Func_m \cup \{Static\}$ où :*

- *$Para_m$ représente l'ensemble $\{p_0 = this, p_1, \ldots\}$ des paramètres de m (restreint aux paramètres objets, par opposition aux valeurs immédiates) ;*
- *Ret_m la valeur de retour de la méthode (si il s'agit d'un objet, une fois encore) ;*
- *$Except_m$ l'ensemble des exceptions pouvant être pouvant être levées ;*
- *$Alloc_m$ l'ensemble des sites d'allocation présents dans la méthode ;*
- *$Func_m$ l'ensemble des appels de fonctions retournant un objet dans le corps de la méthode ;*
- *et $Static$ une abstraction globale de l'ensemble des objets accessibles statiquement dans le système.*

Les éléments ainsi isolés fournissent une couverture totale de l'ensemble des objets pouvant intervenir dans l'exécution du code d'une méthode. Notons que l'ensemble des objets accessibles depuis un champ statique quelconque est projeté sur un seul objet abstrait, puisqu'il est impossible de déterminer statiquement le résultat d'un accès en lecture ou en écriture (ne serait-ce qu'à cause des problèmes d'exécution concurrente). De même, l'ensemble des exceptions susceptibles de traverser une méthode est projeté sur un unique représentant, pour des raisons évoquées ultérieurement. Enfin, il est admis que la valeur de retour d'une fonction est unique pour correspondre au modèle utilisé par Java. En cas de valeurs de retour multiple, l'analyse présentée pourrait être adaptée sans que cela pose de souci majeur.

Parmi ces éléments, un sous-ensemble important est $Alloc_m$, qui représente tous les objets pouvant être créés au cours de l'exécution de la méthode m. Chaque site d'allocation est associé à un unique objet abstrait (en Java, il s'agit de toutes les occurrences des bytecodes `new` et assimilés). Par conséquent on notera r_N l'objet abstrait correspondant aux objets créés par l'exécution du $N^{\text{ième}}$ bytecode, qui se trouve être une allocation. De même, pour $Func_m$, qui représente l'ensemble des objets apportés par un appel de fonction, on notera f_N l'objet abstrait correspondant aux objets créés par l'exécution du $N^{\text{ième}}$ bytecode, qui se trouve être un appel de fonction.

Définition 9 (Abstraction des objets). *Soient $o \in O_m$ et $\breve{o} \in \breve{O}_m$. On note $o \propto \breve{o}$ si $\breve{o}$ est une abstraction de o.*

Les relations décrites dans les définitions 6 et 7 ne peuvent être calculées statiquement, pour la simple raison qu'elles concernent des objets réels, et qu'une simple analyse du code ne suffit généralement pas à retrouver la trace de ces objets suffisamment précisément à travers le flot de contrôle. En particulier la coexistence potentielle de plusieurs objets en tant que contenu d'une même variable est un obstacle insurmontable dans le cas général. En effet, les analyses menées uniquement sur le code tendent à identifier « syntaxiquement » des instructions dont l'exécution a lieu à des moments distincts (et donc dans des contextes différents), comme c'est le cas pour les boucles. À l'inverse, les instructions peuvent également subir l'effet de factorisation de code, qui intervient lorsque deux

branches d'exécution différentes se rejoignent, comme les instructions qui suivent un bloc `if (...) { ... } else { ... }`. Aussi est-il nécessaire de se tourner vers une abstraction de ces relations.

Définition 10 (Abstraction de la relation de pointage). *L'abstraction de la relation de pointage est en fait son quotient par le domaine des objets abstraits. Elle est définie comme suit :* $\breve{o} \overset{\alpha}{\hookrightarrow} \breve{f} \iff \exists o, f \ : \ o \propto \breve{o} \wedge f \propto \breve{f} \wedge o \hookrightarrow f$.

Définition 11 (Abstraction de l'atteignabilité). *Comme précédemment,* $\overset{\alpha}{\hookrightarrow}^*$ *dénote la clôture transitive de* $\overset{\alpha}{\hookrightarrow}$.

Une approximation fiable et calculable statiquement de $\overset{\alpha}{\hookrightarrow}^*$ peut être trouvée : en maintenant au cours de notre analyse un tableau de liens à jour, nous disposons d'un moyen de décrire (à un instant précis) les relations pouvant exister entre les divers objets abstraits manipulés (au sens de la relation $\overset{\alpha}{\hookrightarrow}^*$). L'ampleur de l'approximation est dépendante de la façon de suivre le flot de contrôle et d'unifier les références.

Définition 12 (Tableau de liens). *Un tableau de liens est une partie de* $\breve{O}_m \times \breve{O}_m$.

On note $Link_m$ *l'ensemble des tableaux de liens pour une méthode* m *(*$Link_m$ *est l'ensemble des parties de* $\breve{O}_m \times \breve{O}_m$*).*

Définition 13 (Correction du tableau de liens). *Un tableau de liens* A *est dit correct par rapport à son contexte si* $\forall f, t \in \breve{O}_m \ : \ f \overset{\alpha}{\hookrightarrow}^* t \Rightarrow (f, t) \in A$.

Remarquons qu'il s'agit là d'une simple implication entre la description stockée dans le tableau et la relation existant entre les objets abstraits. Autrement dit, on considérera comme correct tout tableau dont l'ensemble des éléments décrit une relation impliquant $\overset{\alpha}{\hookrightarrow}^*$.

On pourra également définir un tableau de liens optimal $\bar{A}$ tel que $\forall f, t \in \breve{O}_m \ : \ f \overset{\alpha}{\hookrightarrow}^* t \iff (f, t) \in \bar{A}$. L'ensemble des tableaux de liens corrects est alors $\{A \in 2^{\breve{O}_m \times \breve{O}_m} \mid A \supset \bar{A}\}$, à savoir l'ensemble des tableaux de liens contenant $\bar{A}$.

L'attention accordée à la notion de correction des tableaux de liens trouve sa justification dans le fait qu'il est rare d'être en mesure de calculer une solution optimale, et qu'il faut se contenter de solutions approchées, obtenues par le biais de simplifications ou d'unifications dans les algorithmes les calculant. Or ces solutions approchées ne doivent pas provoquer des comportements indésirables comme des optimisations injustifiées.

Dans le cas qui nous intéresse, le résultat critique, et qui se doit d'être exact, n'est pas qu'un objet abstrait se trouve dans une variable, mais au contraire ne s'y trouve pas. Le contenu des variables, dans une analyse abstraite est nécessairement imprécis, par la simple action du flot de contrôle qui « range » différents objets dans les mêmes emplacements. En revanche, si l'analyse abstraite n'associe pas un objet abstrait à une variable, alors il est certain qu'aucun objet concret rattaché à l'objet abstrait ne pourra, en aucune circonstance, jouer le rôle de contenu de la variable dans le code exécuté.

4.3.4 Algorithme

De nombreuses analyses d'alias (ou des analyses se basant sur celles-ci) ont recours à une approche descendante[2] : en commençant l'analyse sur le point d'entrée du programme (la méthode `main`), celui-ci est globalement « déplié » en un graphe d'appels pouvant fournir une vue précise des alias générés [EGH94, WL95].

Si cette méthode est très certainement celle qui donne les résultats les plus proches du comportement réel de l'application (car le code est toujours analysé en tenant compte de son contexte d'appel), elle mène néanmoins à une analyse très lourde, du fait du volume de code ainsi déplié. De plus, cette méthode suppose la connaissance *a priori* du graphe d'appels complet pour l'analyse d'une application (monde fermé). Ce dernier point entre manifestement en contradiction avec le souci affiché de pouvoir charger à l'exécution du code supplémentaire et garder ainsi une plate-forme ouverte (notamment par l'intermédiaire de la compilation séparée de Java).

Afin de répondre à ces deux problèmes, il est possible d'adopter la démarche ascendante[3], qui consiste à considérer la méthode comme l'unité de plus petite granularité au sein de l'algorithme général. Dans cette optique, la méthode est considérée en dehors de tout contexte d'appel, et les informations conservées sont « absolues ». En recourant à cette démarche, il est possible d'obtenir une analyse considérablement plus légère, bien que moins précise. Considérant les restrictions importantes dont souffrent les systèmes embarqués qui auront la charge d'une part de récupérer les résultats de l'analyse, d'autre part de les vérifier et les appliquer, la légèreté du processus est un facteur vital pour notre analyse, qui sera donc compositionnelle et vérifiable. En particulier, cela permet une analyse compatible avec l'ouverture du système au chargement dynamique, à la manière de [WR99].

L'unité centrale de l'analyse étant la méthode, cette analyse est nécessairement composée de deux volets majeurs : une analyse intra-méthode qui correspond à l'interprétation du code de la méthode, et une analyse inter-méthodes dont le rôle est de garantir que les comportements détectés dans une méthode sont bien pris en compte dans l'ensemble des méthodes qui l'utilisent. La zone de communication entre ces deux analyses est constituée des différentes façons d'appeler une méthode (les bytecodes de la forme `invoke`). Ces deux mécanismes étant très liés, il est difficile de les présenter l'un après l'autre ; aussi, certaines zones d'ombre subsistent dans un premier temps quant aux choix effectués, ou quant à la manière d'appliquer les résultats d'une analyse sur l'autre. Néanmoins, à la fin de la présentation des différents volets de l'analyse d'alias, le procédé sera décrit complètement.

Analyse inter-méthodes

Les informations relatives à chaque méthode seront regroupées au sein d'une structure persistante appelée signature, dont le rôle est de refléter le comportement de la méthode relativement à la création et à la propagation des alias, indépendamment de tout contexte

[2] « top-down »
[3] « bottom-up »

d'appel.

L'utilisation du terme « signature » vient du rapprochement immédiat qu'il est possible de faire avec la notion classique de signature de types, qui sert le même but : disposer d'une vue abstraite de la méthode suffisante pour effectuer un calcul sans avoir nécessairement besoin de son code.

Ainsi il n'est pas nécessaire de ré-analyser le corps de la méthode une fois la signature définitivement établie, celle-ci contenant des informations suffisamment génériques pour être applicables à tous les contextes d'appel. L'analyse d'alias elle-même est réalisée au moyen d'une interprétation abstraite [CC77] de chacune des méthodes, et sa correction est naturellement dépendante de la correction des signatures dont elle dépend : celles-ci sont les signatures des méthodes invoquées dans la méthode analysée, sur lesquelles l'analyse s'est reposée.

La signature est une transformation devant être appliquée à un vecteur d'arguments pour obtenir de façon immédiate l'ensemble des liens créés entre ceux-ci par la méthode appelée. La signature représente l'information manquante pour pouvoir associer un unique pas de sémantique à l'action d'invocation d'une méthode. Elle fournit un raccourci immédiat (et correct) pour l'ensemble des instructions élémentaires réellement exécutées du fait de l'invocation. En effet l'invocation n'a aucune action par elle-même sur les tableaux de liens conservés au niveau de la méthode analysée. Seul le code correspondant à la méthode invoquée à une influence.

La signature est donc le résultat de la composition des sémantiques de chacune des instructions de la méthode à laquelle elle se rapporte. Bien entendu, d'autres signatures peuvent jouer un rôle dans le calcul de la méthode courante, celle-ci pouvant invoquer à son tour d'autres méthodes.

Définition 14 (Liens visibles et signatures). *L'ensemble des objets abstraits visibles pour une méthode m est défini comme* $Vis_m = Param_m \cup \{Ret_m\} \cup \{Except_m\} \cup Static$.

L'ensemble des liens visibles $VLink_m$ *est le résultat de la projection de* $Link_m$ *sur* $Vis_m \times Vis_m$.

Une signature pour une méthode m est un $VLink_m$ *particulier qui représente les liens visibles (ou toujours une sur-approximation) créés par l'exécution complète du corps de la méthode m : elle est égale à la projection de* $Link_m^r$*, l'ensemble des liens créés par l'exécution de m (et donc l'union des liens créés aux différents points de retour).*

Définition 15 (Application d'une signature). *On note* $s(l)$ *l'application d'une signature s sur un tableau de liens l. Le résultat de cette opération est un tableau de liens contenant l et l'ensemble des liens créés par s.*

$$s(l) = \{(\alpha, \beta) \mid \exists \gamma \ t.q. \ ((\alpha, \gamma), (\gamma, \beta)) \in (s \cup l)^2\}$$

Les objets visibles d'une méthode sont en fait l'ensemble des objets pouvant intervenir dans le contexte d'une méthode et qui restent accessibles à l'extérieur de celle-ci. Sont donc exclus les objets dont la visibilité est limitée au corps de la méthode, à savoir les allocations

locales et les objets récupérés par appel de fonction. Le fait que ces objets puissent sortir de la méthode par liaison depuis un objet visible est implicite pour le calcul de la signature, puisque celle-ci ne s'intéresse qu'aux liens créés entre les objets existants avant l'appel de la méthode.

L'ensemble des signatures possibles pour une méthode m forment un treillis, défini à partir de l'ordre sur les tableaux de liens :

Définition 16 (Ordre des tableaux de liens). *Soient A_1, A_2 deux tableaux de liens pour une même méthode, alors* $A_1 \leq A_2 \iff \forall(\alpha,\beta) \in A_1, (\alpha,\beta) \in A_2$

Définition 17 (Ordre des signatures). *Soient $s_1, s_2 \in Sign$ deux signatures pour la même méthode.* $s_1 \leq s_2 \iff \forall\ v_1, v_2 \in Vis_m, \forall\ l, (v_1, v_2) \in s_1(l) \Rightarrow (v_1, v_2) \in s_2(l)$.

Note : $\top_m$ et $\bot_m$ sont deux signatures particulières telles que : $\forall\ v_1, v_2 \in Vis_m, \forall\ l$: $(v_1, v_2) \in \top_m(l)$ *et* $(v_1, v_2) \notin \bot_m(l)$.

$\top_m$ est la signature la plus pessimiste dans le sens où elle considère le pire des cas possibles (et donc le cas trivialement correct), et $\bot_m$ la signature la plus optimiste, celle qui reflète une action totalement neutre (du point de vue de l'analyse, la méthode correspondante n'a aucun impact).

En raison de l'existence potentielle de méthodes mutuellement récursives, il n'est pas possible de calculer au préalable toutes les signatures des méthodes intervenant dans le code de celle que l'on souhaite analyser. La conséquence en est que le résultat de l'analyse d'une méthode ne peut être considéré comme fiable directement. Au contraire, pour des méthodes mutuellement récursives, la découverte des liens créés dans chacune d'entre elles ne peut que se faire progressivement, à mesure que l'on perçoit mieux l'impact de chacun des appels et qu'on le répercute sur ce qui en dépend.

Par conséquent, il est nécessaire de calculer un point fixe pour chaque groupe de méthodes interdépendantes (ou par simplification, de classes interdépendantes). Pour chaque groupe, des analyses répétées provoqueront la génération de signatures croissantes (à mesure que l'on prend en compte des comportements plus proches de la réalité), jusqu'à stabilisation. Le point de départ de l'analyse consiste donc à attribuer à une méthode inconnue la signature la plus optimiste, et à revoir à la hausse l'ensemble des liens qu'elle génère à chaque fois que son corps est analysé.

En résumé, la signature associée à chaque méthode est tout d'abord *a priori* fausse, puis les analyses successives finissent par détecter l'ensemble des liens réellement créés, faisant évoluer la signature vers un état correct et donc stable (lorsque les signatures de toutes les méthodes concernées sont correctes, plus aucun lien supplémentaire ne peut être détecté).

Analyse intra-méthode

L'algorithme d'analyse du corps d'une méthode est itératif : le tableau de liens est vide au départ de la méthode analysée. En effet, hors de tout contexte d'appel, les éventuels

`putfield` (`f` est un champ valide de `o`) :

$$\frac{\{P, n, [S|o, x], L\} \qquad P[n] = putfield\ \#f}{\{P, n+1, S, L\}\ s.t.\ L[n+1] = closure(L[n]\{r_1 \overset{\alpha}{\hookrightarrow} r_2 : r_1 \in \bar{o} \cup \{y \mid o \overset{\alpha}{\hookrightarrow}^* y\}, r_2 \in \bar{x}\})}$$

`getfield` (`f` est un champ valide de `o`) :

$$\frac{\{P, n, [S|o], L\} \qquad P[n] = getfield\ \#f}{\{P, n+1, [S|x], L\}\ s.t.\ x = \{y \mid o \overset{\alpha}{\hookrightarrow}^* y\}}$$

`invokeVirtual` (`f()` est une fonction de $i+1$ arguments, `this` inclus) :

$$\frac{\{P, n, [S|o, p_0, \ldots, p_i], L\} \qquad P[n] = invokevirtual\ \#f \qquad Sig = exactSign(f)}{\{P, n+1, S, L\}\ s.t.\ L[n+1] = closure(L[n]\{\pi_{(p_0,\ldots,p_i)}(L) \star Sig\})}$$

`goto` (en admettant que le numéro du bytecode est utilisé à la place de son offset) :

$$\frac{\{P, n, S, L\} \qquad P[n] = goto\ \#l \qquad \alpha = L[l]}{\{P, l, S, L\}\ s.t.\ L[l] = \alpha \cup L[n]}$$

TAB. 4.1 – Extrait de sémantique pour le bytecode Java.

liens préexistants entre des arguments ne peuvent être pris en considération. C'est pourquoi le tableau de liens représente à tout moment l'ensemble des liens « créés » et non pas les liens existants.

À partir de cette situation initiale, le bytecode est analysé en suivant le flot de contrôle (présenté en section 4.4.1) et en appliquant les signatures disponibles à chaque fois qu'une invocation de méthode est rencontrée. Comme mentionné dans la définition 14, la signature de la méthode correspond à l'ensemble des liens visibles qui sont créés lors de l'exécution de la méthode invoquée.

Tout bytecode Java est associé à une sémantique qui exprime la transformation à appliquer à l'état courant du tableau de liens. La plupart des bytecodes ont une action nulle sur ce tableau, puisqu'ils ne modifient pas l'état des relations entre les objets dans le système. La table 4.1 donne une sémantique pour les actions principales, qui fait usage de plusieurs opérateurs et notations, détaillés ci-après :

- $P[n]$ représente la $n^{\text{ième}}$ instruction du programme : par simplification, nous assimilons le numéro d'une instruction et son offset ;
- $L[n]$ représente l'état du tableau de liens à la $n^{\text{ième}}$ instruction du programme ;
- $[S|x_n, \ldots, x_0]$ représente la pile dont les $n+1$ premiers éléments sont $x_0, \ldots, x_n$ (x_0 est le sommet de pile) et dont le reste est S ;
- $closure(L)$ fait en sorte que le tableau de liens L demeure transitivement clos et symétrique ;

- $\pi_{\vec{p}}(L)$ dénote la projection de L sur le vecteur des arguments effectifs ;
- $\cup$ effectue l'union close des liens, à savoir $\alpha \cup \beta = closure(\{x | x \in \alpha \text{ } or \text{ } x \in \beta\}$;
- $I \star S$ est l'opérateur de composition, qui assure la propagation des liens de la signature S depuis la situation initiale I, qui représente l'état courant du tableau de liens pour la méthode analysée ;
- $\bar{v}$ représente le contenu abstrait de la variable v.

Indépendamment de l'algorithme de suivi du flot de contrôle (présenté en section 4.4.1), pour peu que ce dernier soit correct et couvre la totalité des chemins possibles pour l'exécution, l'algorithme atteint nécessairement un point fixe pour l'état du tableau de liens en chaque point du programme.

Ce point fixe existe puisqu'un pas de l'algorithme consiste en l'application d'une transformation monotone sur un espace fini (l'ajout des liens créés par l'instruction courante s'appliquant sur l'ensemble fini de tous les liens possibles pour la méthode).

Le tableau de liens composé à partir des éléments de ce point fixe qui correspondent aux différents points de retour possibles de la méthode est noté $Link^r_m$. Il s'agit du plus petit élément supérieur à tous les tableaux de liens (selon l'ordre établi par la définition 16) calculés sur ces points de retour. La projection de ce tableau sur l'ensemble des liens visibles de m donne l'effet qu'a la méthode m vue de l'extérieur, c'est à dire sa signature.

Ainsi, le typage d'une méthode est obtenu par unification de l'ensemble des tableaux de liens aux divers points de retour, puis projection sur l'espace des références ayant une existence visible dans l'appelant. De ce fait, toute signature incluant au moins celle précédemment calculée est un type correct pour la méthode.

4.4 Choix et approximations

La plupart des choix faits dans la conception et l'implantation de l'algorithme de calcul des alias sont liés à un petit nombre de préoccupations constantes : conserver un temps de calcul aussi court que possible, et rendre le processus de vérification des données générées par l'algorithme le moins coûteux possible, autant en temps qu'en espace. Tout ceci, en laissant un degré élevé de liberté au programmeur (de façon à pouvoir accepter du code quelconque, y compris non spécifiquement développé avec les préoccupations ici exposées) et en conservant un degré de précision suffisamment satisfaisant.

4.4.1 Flot de contrôle

L'un des premiers problèmes de l'analyse du code est de choisir la façon dont seront pris en compte les différents branchement présents dans tout code non trivial. Ceux-ci peuvent être simplement ignorés dans l'optique d'une analyse insensible au flot de contrôle[4]. L'avantage d'une telle approche est que l'analyse s'en trouve singulièrement allégée et

[4]« flow insensitive » dans la littérature anglaise.

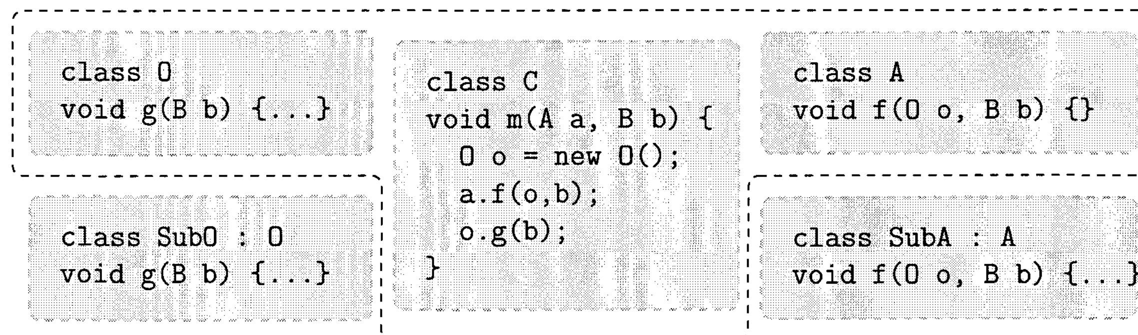


FIG. 4.2 – Chargement dynamique et héritage.

simplifiée [BCCH95, SH97].

Néanmoins, choisir cette approche revient nécessairement à unifier des situations potentiellement disjointes (ce qui est déjà le cas par le simple fait que l'analyse est statique, mais à un degré inférieur).

Une approche contraire, également très développée, consiste à suivre assez précisément les valeurs pouvant être manipulées par un programme, de sorte à pouvoir éventuellement couper des branches du flot de contrôle, et ainsi gagner en précision au niveau de l'analyse. Certains travaux [BJP, Cou06] utilisent notamment une notion d'intervalles pour délimiter les valeurs possibles pour une certaine variable au cours de son existence.

L'inconvénient de ce choix se situe dans la complexité ajoutée au niveau de l'analyseur, qui doit effectuer des calculs passablement lourds (notamment en espace). Cette complexité se reporte naturellement à la vérification puisque, à moins de donner la trace complète des calculs effectués, les étapes de la vérification impliquent des calculs similaires à ceux effectués lors du calcul initial.

Le choix fait dans le cadre de ce travail tente de correspondre à un juste milieu entre les deux approches, en suivant inconditionnellement le flot de contrôle : toute opération de saut est suivie (selon toutes les possibilités). De la sorte, on obtient évidemment une analyse plus précise que celles ne tenant aucun compte du flot de contrôle, et qui par ailleurs ne demande aucun calcul annexe et se focalise uniquement sur la propagation des liens.

4.4.2 Problème de l'héritage

Le souci d'opérer dans un contexte ouvert, où le chargement dynamique de code doit être possible, pose un problème lorsqu'il est combiné à la notion d'héritage dans toute sa généralité. En effet, une analyse compositionnelle repose sur la correction des résultats obtenus (et appliqués) pour chacun des blocs de base. L'héritage, lorsqu'il concerne une classe inconnue *a priori*, remet en question tous les comportements potentiellement calculés des parents (par l'intermédiaire de leurs signatures associées), et donc l'analyse complète.

En effet, dans les langages considérés, une méthode peut être surchargée de n'importe quelle façon, tout ce qui importe est exprimé dans le typage de la fonction.

Le problème est d'ailleurs plus vaste, puisque aucune sémantique associée à une méthode ne peut être conservée de façon sûre à travers les appels à des surcharges. Or il est impossible, dans un cadre où l'on ne contrôle pas l'ensemble des classes présentes, de déterminer dans le cas général quel est le code qui est effectivement appelé. La figure 4.2 montre un exemple dans lequel les classes entourées sont suffisantes pour que la compilation de la classe principale `C` soit possible. Néanmoins, rien n'empêche le système de charger ultérieurement des sous-classes de `O` et `A`, qui apportent leurs propres implantations de certaines méthodes.

Ceci est également la raison pour laquelle de nombreuses analyses se tournent vers un modèle clos, où le graphe d'appel permet d'aplanir la plupart des difficultés liées à la surcharge. Cependant, dans le contexte dans lequel nous évoluons, le chargement dynamique et incrémental est l'un des bénéfices principaux de l'utilisation quasi généralisée des machines virtuelles et des langages orientés objet. Il est donc nécessaire de trouver une autre solution.

4.4.3 Types exacts

La notion de type exact a été utilisée précédemment dans la littérature (notamment par Beers, Stork et Franz [BSF04]) dans le but de compenser, là où cela est possible, l'imprécision engendrée par les appels de méthode, en raison précisément des liaisons dynamiques qui obligent les analyses à se montrer pessimistes. L'idée est de détecter, parmi l'ensemble des objets, ceux dont on peut déterminer statiquement le type au lieu de se reposer sur le type déclaré des variables qui les contiennent.

Clairement, le cas des `invokeSpecial` de Java (qui sert notamment à invoquer les méthodes privées et celles de la classe parente) est un cas dans lequel l'objet sur lequel s'applique la méthode est associé à un type exact. En revanche, pour des invocations par `invokeVirtual`, la question est beaucoup moins simple à traiter. Néanmoins, dans certains cas, un type exact peut encore être déterminé. Par exemple dans la figure 4.2, après l'instruction `O o = new O();` l'objet `o` est nécessairement une instance de la classe `O` et rien d'autre. Il est possible de conserver la certitude de cette exactitude tant que l'objet abstrait ne se trouve pas confondu à d'autres au sein d'une variable. Tant que tel est le cas, il est possible de résoudre statiquement tous les appels de méthodes dont il fera l'objet, même si un `invokeVirtual` est généré par le compilateur Java.

L'analyse peut tirer avantage de la connaissance de types exacts pour se montrer la plus précise possible, en appliquant la signature calculée pour une méthode précise, sans qu'il soit nécessaire de se poser la question de l'héritage.

En revanche, si aucun type exact ne peut être calculé, il est nécessaire de se contenter d'une précision moindre, et la signature à appliquer doit être compatible (au sens de l'inclusion des tableaux de liens générés) avec l'ensemble des méthodes qui peuvent être appelées. Il est donc nécessaire de disposer d'une signature plus tolérante qui n'engage

`invokeVirtual` :

$$\frac{\{P, n, [S|o, p_n, \ldots, p_0], L\} \qquad P[n] = invokevirtual\ \#f \qquad Sig = sign(o,f)}{\{P, n+1, S, L\}\ s.t.\ L[n+1] = closure(L[n]\{\pi_{\vec{p}}(L) \star Sig\})}$$

avec $sign(o,f) = exactType(o)?exactSign(f) : approxSign(f)$

TAB. 4.2 – Sémantique du bytecode `invokeVirtual`.

pas les méthodes susceptibles d'être effectivement appelées à adopter un comportement incompatible avec leur propre code.

Définition 18 (Type exact). *Un objet (et donc un objet abstrait) est associé à un type exact s'il est possible de décider statiquement quel est le constructeur qui a été utilisé pour créer l'instance. Cette propriété est décrite par le prédicat suivant :* $exactType : \breve{O} \longrightarrow bool$.

Les types exacts étant relativement simples à déterminer, nous les faisons intervenir dans l'analyse sous la forme suivante : certaines méthodes pourront avoir en fait deux signatures, la signature exacte représentant le comportement calculé pour le code de la méthode, et l'autre dite « approchée » qui est une approximation de la signature exacte et présente la propriété d'être compatible avec l'ensemble des signatures de méthodes pouvant être effectivement appelées en lieu et place de la méthode considérée. Cette dernière signature est obtenue par unification des signatures calculées sur l'ensemble de la branche d'héritage subordonnée à la classe d'appartenance de la méthode. Il s'agit donc d'un supremum pour la relation d'ordre définie précédemment.

La signature exacte d'une méthode est donc utilisée lorsque l'objet sur lequel elle s'applique peut être associé à un type exact. Dans les autres cas, il n'y a pas d'autre choix que de se référer à la signature approchée pour obtenir un comportement que l'on sait correct, à défaut d'être précis.

Ceci nous permet de préciser la sémantique donnée pour le bytecode `invokeVirtual` en table 4.1. La table 4.2 fait apparaître que, selon la possibilité ou non d'attribuer un type exact à l'objet recevant l'appel à une méthode, c'est la signature exacte ou la signature approchée qui sera appliquée.

Dans l'exemple de la figure 4.2, l'objet `o` construit dans la méthode `m()` de la classe `C` possède un type exact au moment de l'invocation `o.g(b)`. De ce fait, l'existence de la classe `SubO` ne change rien dans ce cas, et c'est bien la signature exact de `O.g()` qui est appliquée. *A contrario*, l'objet `a` n'a pas de type exact (pour la simple raison qu'il s'agit d'un paramètre, et donc que sa construction sort du cadre de la méthode en cours d'analyse), de sorte qu'il est tout à fait possible que `SubA` joue le rôle de `A` à l'exécution, pour peu que l'on appelle la méthode avec un `SubA` plutôt qu'un `A`. Il est donc nécessaire d'analyser `a.f(o,b)` en utilisant la signature approchée correspondante.

4.4.4 Dictionnaire et calcul des signatures

La structure qui permet de faire le lien entre les analyses intra-procédurale et inter-procédurale est le dictionnaire, chargé du stockage des signatures associées aux diverses méthodes intervenant dans le système.

Définition 19 (Dictionnaire). *Soit Func l'ensemble des méthodes d'un programme, et Sign l'ensemble de toutes les signatures possibles. Un dictionnaire est une application* d : $Func \longrightarrow (Sign \times Sign)$.

Pour un dictionnaire, les signatures exactes et approchées sont accédées à travers les projections *exactSign* et *approxSign*, appliquées à la fonction considérée.

Comme précisé précédemment, si la signature exacte d'une méthode est immédiate (puisque correspondant à un code existant), le calcul de la signature approchée est au contraire complexe, car cette dernière doit permettre une compatibilité avec l'ensemble des sous-classes (potentiellement inconnues).

Il est clair qu'à moins d'associer à toute méthode susceptible d'être surchargée la signature la plus pessimiste possible, cette propriété contraint l'extensibilité dynamique du système. En effet, dans le cadre des seules contraintes imposées par le langage Java, rien ne saurait imposer la vérification d'une quelconque propriété sur les liens créés par une méthode.

Néanmoins, cette limitation de l'extensibilité n'est pas nécessairement négative. Les contraintes apportées sur la création de liens peuvent être vues comme une sorte de contrat, passé entre le concepteur du code original et les concepteurs d'extensions. La seule différence avec un contrat « classique » repose dans le fait que le code « fautif » sera rejeté au chargement du code au lieu de déclencher l'échec d'un contrôle à l'exécution[5]. Par ailleurs, le concepteur initial dispose toujours de la possibilité d'associer volontairement des signatures très permissives aux méthodes pour lesquelles il envisage une possibilité de surcharge très libre. En faisant le choix de contraindre par endroits l'extensibilité du système, nous nous rapprochons donc de notions de sécurité, toujours centrales dans les systèmes embarqués considérés. Au final, l'extensibilité du système n'est pas remise en cause pour peu qu'elle soit conforme aux spécifications des interfaces initiales.

Concernant le calcul effectif des signatures approchées, il est possible de considérer un ensemble donné de classes disponibles, et de faire en sorte que la signature approchée de chaque méthode permette impérativement le chargement de toutes ces classes. L'algorithme est très simple, et consiste à incorporer chaque signature exacte calculée à l'ensemble des signatures approchées concernées. Si une méthode doit fournir un point d'extension plus permissif encore, il est toujours possible de lui associer au préalable une signature plus pessimiste. En procédant de la sorte, la signature approchée ne sera jamais modifiée par l'unification avec l'une des signatures exactes calculées.

Enfin, pour les méthodes abstraites ou d'interfaces ne disposant pas d'implantation dans l'ensemble de classes choisi, ce procédé permet d'éviter qu'il soit totalement impossible de

[5]par exemple à travers l'utilisation d'une assertion.

```
class ArrayList {
  private Vector v = new Vector();

  public void add(Object element) {
    v.add(element);
  }
}
```

FIG. 4.3 – Exemple d'implantation.

les surcharger, puisqu'un corps de méthode vide ne peut exhiber aucune création de lien et que la signature exacte qui en découle n'est compatible qu'avec elle-même.

4.4.5 Objets et champs

Comme mentionné précédemment, l'un des points critiques dans le suivi d'alias d'objets est le niveau de précision conservé par rapport à l'utilisation des champs d'objets.

Il est tout à fait possible d'être parfaitement précis, en conservant le fait que tel objet se trouve entre autres dans le champ `x` du champ `y` du champ `z` d'un objet `o`, dans la lignée des travaux de Bruno Blanchet [Bla99], qui fournissent une représentation aussi complète que possible du graphe de références entre les objets.

Les inconvénients d'une telle approche ne sont néanmoins pas négligeables, si l'on considère le cadre dans lequel nous évoluons. Tout d'abord, nous voulons fournir des informations qui soient vérifiables par de petits systèmes embarqués, et le type de description précédemment cité ne saurait atteindre un niveau de concision suffisant, et ce manque de concision engendrerait des traitements particulièrement lourds.

De plus, la conservation d'informations aussi précises aurait des répercussions fortes sur l'extensibilité du système. En effet, nous avons déjà mentionné les problèmes posés par l'héritage, qui impose que les méthodes héritées aient un comportement compatible avec celui de leurs parentes, dans le cas où il est impossible de distinguer statiquement quel sera le code effectivement exécuté. Ces problèmes grandissent évidemment à mesure que les comportements décrits sont précis : plus de précision dans les assertions implique naturellement moins de flexibilité. Dans le cas qui nous occupe, on atteindrait un niveau de précision tel que les méthodes se verraient imposer bien plus qu'une simple sémantique générale décrivant de façon approximative les liens entre les diverses données manipulées.

Pour prendre un exemple, il paraît raisonnable et suffisamment générique d'associer à une méthode `add()` d'une collection quelconque la signature caractérisant un lien créé entre `this` et l'argument (ce qui est contraignant, mais néanmoins conforme à l'idée que l'on se fait d'une collection « normale »). En revanche, si une collection quelconque, `ArrayList` par exemple (*cf.* figure 4.3 pour une implantation purement fictive), fait apparaître que le

lien spécifique, en ce qui la concerne, est matérialisé par un lien vers un `Vector`, lui-même étant lié à un `Object[]` sous-jacent, alors il ne semble plus du tout pertinent d'obliger toutes ses sous-classes à adopter une structure identique.

Donner ces détails contraindrait donc partiellement l'écriture des méthodes jusque dans leurs mécanismes internes, ce qui est totalement contraire aux préoccupations de la programmation orientée objet, qui vise à l'établissement d'interfaces communes pour des mécanismes sous-jacents quelconques.

Afin d'éviter tout cela, nous avons décidé d'unifier dans l'abstraction tous les champs d'un objet : à chaque accès à un champ, l'analyse considère que tout l'objet est concerné. De façon similaire, lorsqu'on accède à un objet, tout champ est potentiellement mis en cause. Il s'agit là d'une approximation de taille, mais qui se révèle à l'usage particulièrement intéressante, aussi bien en termes d'efficacité que de simplicité.

4.4.6 Exceptions

Les exceptions sont un mécanisme qu'il est très important de prendre en compte, puisqu'elles influent directement sur le flot de contrôle. Malheureusement, leur manipulation peut se révéler assez complexe y compris dans les situations les plus simples, notamment dans le cas de plates-formes Java, sur lequel nous reposerons le reste de cette section.

```
class SomeException extends Exception {}

class MyException extends SomeException {}

class C {
  void m() {
    try {
      ...
      throw new MyException();
      ...
    } catch (SomeException e) {
      ...
  . }
  }
}
```

FIG. 4.4 – Exemple de gestion d'exception.

L'exemple de code donné en figure 4.4 peut sembler trivial pour un analyste humain : l'exception levée à l'exécution sera récupérée quelques lignes de code plus loin, car `MyException` est une sorte de `SomeException`. Malheureusement, ce raisonnement est

```
class MyException extends Exception {}
```

FIG. 4.5 – Code « malicieux ».

faux, ou du moins peut le devenir, suivant les conditions d'exécution.

En effet, si la compilation de ce code produit un ensemble d'éléments dont l'exécution conjointe produirait le résultat attendu, le fait que la compilation soit séparée, et le code chargé sans contrainte particulière de provenance, permet très simplement de faire s'exécuter de concert des éléments n'ayant que peu de liens entre eux. Ainsi, le code légèrement différent présenté en figure 4.5 ne pose pas de problème pour une exécution avec les éléments de la première version qui n'ont pas été altérés. En revanche, il n'y a plus de lien d'héritage entre les deux classes d'exception, ce qui induit un flot de contrôle radicalement différent.

Plus généralement, il est impossible dans le cadre d'une analyse statique de se fier à la hiérarchie des classes Java sauf à considérer un mécanisme externe (type authentification) chargé de vérifier la cohérence des conditions d'exécution vis à vis des conditions de compilation, ce qui se combine mal avec la volonté de conserver une analyse compositionnelle (reflétant le mode de fonctionnement de Java). Le cas des exceptions (et de `istanceof`) s'applique lors d'une analyse considérant le flot de contrôle, et fait que l'on doit considérer celles-ci comme empruntant tous les chemins envisageables, quittant la méthode ou passant par un gestionnaire d'exceptions actif.

Les seuls cas dans lesquels le chemin d'exécution suivi à la suite d'une levée d'exception est statiquement calculable sont les suivants :

- le type exact (*cf.* section 4.4.3) de l'exception est connu dans le gestionnaire d'exception (et correspond exactement au type considéré par ce gestionnaire), ce qui dans notre cas revient à dire que l'exception a été générée dans la méthode courante, et qu'il s'agit donc de l'équivalent d'un `goto` ;
- le gestionnaire prend en charge toutes les exceptions et erreurs (à travers la capture de l'interface `Throwable`), ce qui est considéré à juste titre comme une mauvaise idée, puisque cela implique de capturer également les erreurs de la machine virtuelle et d'aboutir à des situations totalement instables et difficilement maîtrisables.

Afin de prendre en compte ces différentes considérations, nous avons jugé qu'il était plus simple de traiter les exceptions comme créant des liens entre tous les objets se trouvant à sa disposition, et en particulier comme mettant à disposition de tout le système (par l'intermédiaire d'un pseudo champ statique) tous ses constituants. Cela revient à considérer d'une certaine manière que pour une méthode m, les éléments $Except_m$ et $Static$ tels qu'introduits dans la définition 8 sont inconditionnellement liés.

4.5 Exemple

Un exemple détaillé de l'exécution de l'analyse est présenté en figure 4.7 sur la base du code très simple donné en figure 4.6. Ce code consiste en la construction d'une liste chaînée, et son bytecode est assez aisément compréhensible.

Comme précisé précédemment lors de la présentation de l'analyse intra-méthode (section 4.3.4), la situation initiale pour l'algorithme est de considérer l'action du premier bytecode sur un tableau de liens vide, puisqu'il est impossible de présumer de l'état des relations entre les objets réellement concernés par l'invocation de la méthode.

À chaque instruction, les instructions immédiatement suivantes sont empilées, de sorte qu'aucun chemin d'exécution possible n'est oublié. Afin de respecter le flot de contrôle, chaque instruction est examinée dans le contexte de sortie de l'instruction précédente, même si son analyse n'intervient pas immédiatement à la suite (il est possible d'explorer une branche complète, puis une autre, et de fait c'est ce qui se produit).

Remarquons que le déroulement de l'algorithme, et notamment les choix faits lors des branchements conditionnels, montrent qu'il est généralement préférable de traiter en priorité les labels les plus proches du début du code. Ceci n'est pas une vérité absolue bien évidemment, mais les habitudes de programmation et de compilation en font un facteur non négligeable pour la performance générale des algorithmes d'analyse abstraite.

Ainsi, lorsque l'analyse prend en considération le bytecode 7 (`ifle 28`), qui est une instruction de branchement conditionnelle, les deux instructions suivantes (28 et 10) sont empilées, puis la première instruction disponible (10) est dépilée pour analyse. Par souci de simplification, nous n'avons donc fait apparaître que les mises en pile n'étant pas consommées immédiatement.

Notons par ailleurs qu'il est inutile d'empiler plusieurs fois la même instruction, pour peu que lors de son analyse le contexte dans lequel elle sera évaluée soit suffisamment détaillé pour inclure toutes les situations dans lesquelles une analyse a été demandée. De la sorte on évite d'analyser plus souvent que nécessaire des bytecodes cibles de branchement.

```
public static C chain(C head, int n) {
  C p = head;
  while(n-- > 0) {
    C x = new c();
    p.link = x;
    p = x;
  }
  return p;
}
```

FIG. 4.6 – Exemple d'analyse d'alias.

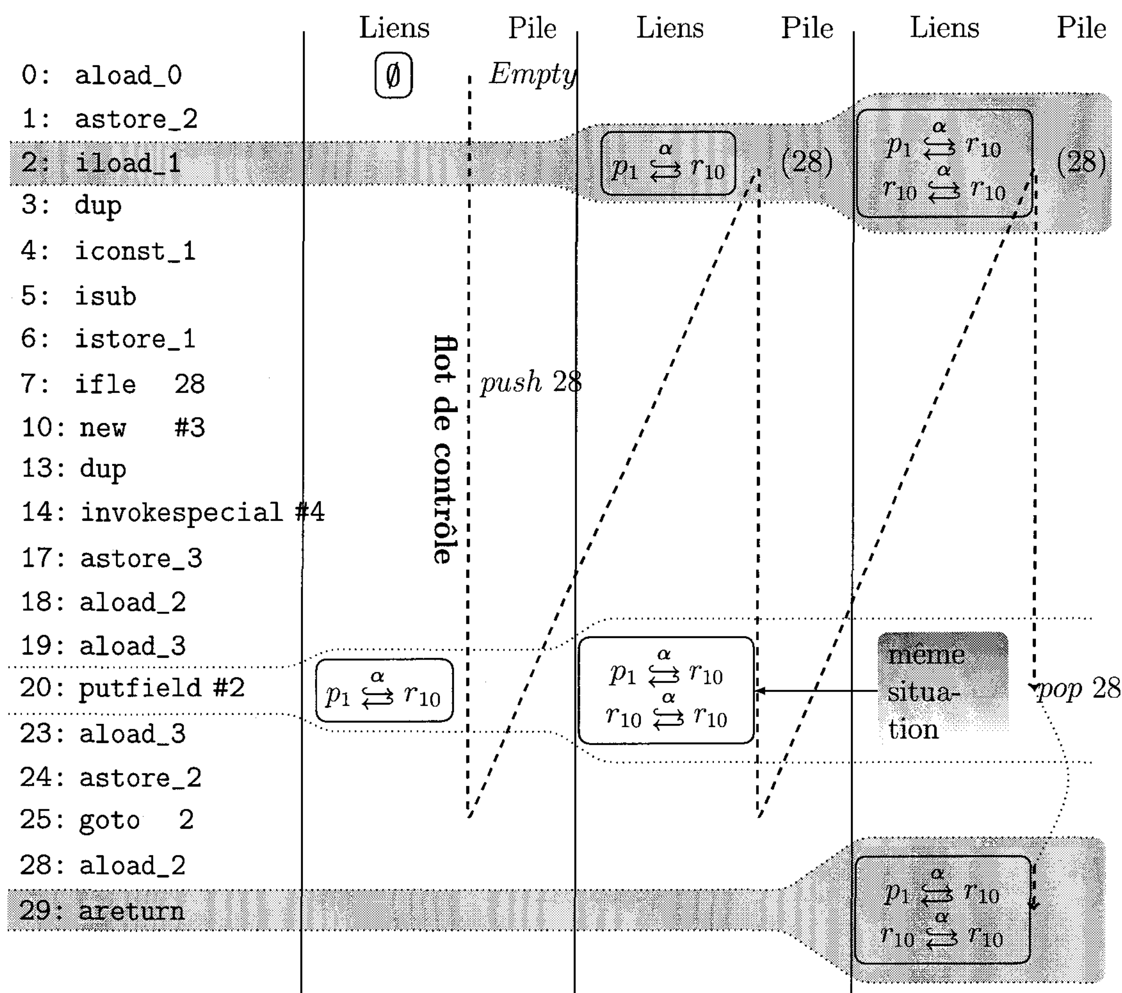


FIG. 4.7 – Exécution de l'algorithme.

Dans notre exemple, la seule véritable création de lien se produit lors de l'exécution du bytecode `putfield`, ce qui correspond sans surprise à la liaison d'un nouveau maillon dans la liste chaînée (`p.link = x`).

De plus, il est inutile de poursuivre l'analyse dans une branche si il s'avère que le code analysé l'a déjà été dans des circonstances identiques. En effet, les instructions suivantes ne peuvent dans ce cas rien apporter qui ne soit déjà connu. Lorsque ce cas se produit, une nouvelle instruction peut-être dépilée et analysée dans son propre contexte. C'est ce qui se passe lors de la dernière analyse du bytecode 20, son action n'apportant rien de nouveau par rapport à sa dernière exécution. L'analyse reprend alors le bytecode 28, empilé tout au début et n'ayant pas encore été rencontré. En revanche, conformément à la remarque suivante, il serait incorrect de l'analyser avec le tableau de liens vide qui lui était associé au moment de sa mise en pile. Au lieu de cela, c'est le dernier contexte de sortie de son

prédécesseur qui est considéré, à savoir le tableau de lien complet tel que présenté en résultat.

Naturellement, l'analyse se termine complètement lorsque la pile des instructions restant à analyser est vide, puisqu'alors toute instruction a été analysée dans un état abstrait regroupant l'ensemble des états atteignables en ce point du programme (ou alors l'instruction appartient à du code pouvant être statiquement détecté comme mort, ce qui n'est pas problématique).

Le tableau de lien, calculé à partir de l'ensemble des tableaux de liens aux points de retour de la fonction, est donc de la forme suivante : $\{p_1 \overset{\alpha}{\rightleftarrows} r_{10}, r_{10} \overset{\alpha}{\rightleftarrows} r_{10}, Ret \overset{\alpha}{\rightleftarrows} r_{10}, Ret \overset{\alpha}{\rightleftarrows} p_1\}$ Ceci met en évidence le fait que la méthode lie entre eux son premier argument et sa valeur de retour, ainsi qu'un (ou plusieurs) objets internes. La signature de la méthode, projection de ce tableau, fait abstraction des objets internes pour ne concerner que l'observable depuis l'extérieur : $\{Ret \overset{\alpha}{\rightleftarrows} p_1\}$. Tout appel de cette méthode aura donc pour conséquence de relier la valeur de retour au premier paramètre.

4.6 Vérification

Une fois calculées, les informations générées par l'analyse doivent être mise à disposition de la plate-forme d'exécution, afin que celle-ci puisse en tirer partie. Par ailleurs, la plate-forme ne saurait se fier aveuglément aux résultats qui lui sont fournis, et doit donc procéder à une vérification de la conformité des résultats annoncés avec le code sur lequel ils doivent être appliqués.

De ce fait, deux éléments sont finalement transmis : les résultats de l'analyse d'une part, et d'autre part une quantité d'information suffisante pour que le système cible soit en mesure de garantir leur correction. Afin d'obtenir ce résultat, nous nous basons sur une technique de vérification de bytecode [RR98]. Cette méthode peut être employée grâce au fait qu'il est aisé de traduire les résultats de l'analyse d'alias sous la forme d'informations de typage, et donc de se placer dans le domaine d'applicabilité de l'algorithme.

Le principe directeur de la vérification est d'effectuer, dans le système embarqué, une analyse très similaire à celle qui a produit les résultats à tester, tout en réduisant sa complexité en la guidant par la conservation d'indications de preuve en certains points des programmes (ces indications étant extraites du déroulement de l'analyse complète). Ainsi, le fichier `class` contient un certain nombre d'informations additionnelles :

- les signatures de l'ensemble des méthodes de la classe ;
- les signatures des méthodes d'autres classes invoquées par cette classe (qui sont donc nécessaires au processus de vérification) ;
- pour chaque méthode, l'état $L[i]$ des variables locales pour chaque bytecode i qui est la cible d'un saut.

Ce dernier point nous permet d'effectuer la vérification en temps linéaire par rapport à la taille du code chargé, et avec un espace mémoire constant. L'algorithme est le suivant : pour

chaque instruction (le respect strict du flot de contrôle n'est pas nécessaire), la sémantique associée à cette instruction est appliquée à l'état d'entrée pour produire l'état de sortie. L'état d'entrée est obtenu par le biais d'une indication de preuve si l'instruction est la cible d'un saut, ou par l'état de sortie de l'instruction précédente sinon. Pour chaque instruction de saut, il est nécessaire de vérifier que l'état de sortie (qui est égal à l'état d'entrée, les sauts étant neutres vis-à-vis de la création de liens) soit inclus dans l'état fourni en annotation de preuve pour les instructions suivantes.

Nous fournissons également les signatures de méthodes : ces dernières pourraient simplement être calculées par l'algorithme de vérification, mais nous autorisons l'utilisateur à signer manuellement les méthodes à l'aide de signatures plus larges (incluant la signature calculée), de sorte qu'il est nécessaire de fournir cette information supplémentaire. Enfin, nous fournissons les signatures des méthodes extérieures invoquées dans la méthode à vérifier, de sorte que les problèmes liés à l'ordre de chargement soient résolus.

Une fois la méthode vérifiée, les annotations de preuve peuvent être supprimées, et les signatures enregistrées dans le dictionnaire interne pour utilisation future. Pour chacune de ces signatures (celles de la méthode vérifiée et celles des méthodes utilisées) :
- si la signature est déjà présente dans le dictionnaire, la signature chargée soit être inférieure ;
- si elle n'est pas présente, elle doit néanmoins être compatible avec les signatures des méthodes surchargées par héritage ;

Dans l'éventualité où la méthode ne peut pas être vérifiée ou si l'une de ces conditions n'est pas respectée, le chargement de la méthode échoue.

Pour revenir à l'exemple de la Figure 4.6, les informations mises en valeur (grisées) dans la figure 4.7 sont suffisantes puisque les bytecodes 2 et 28 sont les seuls à être une destination possible pour une instruction de saut. On se convainc aisément en rejouant l'algorithme avec ces postulats que le système est en mesure de vérifier la correction de la signature en analysant chaque bytecode exactement une fois. Notons tout de même que le fait que l'analyse finisse par associer le même tableau de liens à l'ensemble des cibles de saut relève du hasard, ou plus précisément du fait qu'il y a essentiellement un seul segment de code dans lequel le programme boucle.

4.7 Implantation

4.7.1 Généricité de l'implantation

L'analyse est effectuée à l'aide d'un moteur très générique, bâti autour d'un simple suivi de flot de contrôle. Des annotations, qui peuvent être greffées pour répondre à des besoins spécifiques, se chargent d'apporter une sémantique à chacune des instructions. Cette conception permet d'ajouter facilement des mécanismes adaptés à des analyses statiques quelque peu différentes, comme nous le verrons dans les chapitres suivants.

Pour le problème qui nous intéresse ici, à savoir le calcul des liaisons existant entre les

objets, une annotation spécifique se charge d'associer à chaque bytecode la sémantique que nous avons brièvement présentée dans les tables 4.1 et 4.2.

4.7.2 Performances

Notre analyse est en définitive relativement précise, ne serait-ce que par le fait que le flot de contrôle est véritablement pris en considération, mais le prix à payer en termes de calculs est bien trop élevé pour envisager de la transposer directement dans les systèmes embarqués visés.

À titre d'illustration, le point fixe de l'analyse d'une API Java basique (celle de JITS, comportant environ 550 classes), est obtenu au bout de quatre itérations sur l'ensemble des classes du projet, et du point de vue d'une méthode, chaque bytecode est analysé en moyenne 2,5 fois (pour chaque itération de l'analyse donc). Globalement, l'ensemble du code est analysé dix fois avant que l'algorithme ne converge, ce qui est bien trop coûteux pour nos objectifs. Notons également que la « linéarité » constatée n'est que moyenne et expérimentale : il est tout à fait possible de construire un exemple défavorable dont la complexité d'analyse sera exponentielle par rapport à la taille du code.

Pour donner une idée de la complexité du code mis en œuvre, l'interpréteur abstrait complet est écrit en Java, ce qui représente environ 2300 lignes de code, et fait appel à une version légèrement étendue de la bibliothèque BCEL[6] pour la manipulation explicite du bytecode. Les composants essentiels sont d'une part l'analyseur générique (650 lignes) et les classes de gestion des signatures (450 lignes). En dehors de ce cœur, l'annotation centrale qui maintient la trace des alias de références, comporte environ 350 lignes supplémentaires.

En ce qui concerne les performances, la taille de l'analyseur complet, STAN[7], est de plus ou moins 160 ko de bytecode, et une machine récente (processeur Pentium® 4 cadencé à 2GHz) produit le résultat de l'analyse d'une API Java en quelques minutes, ce qui est parfaitement raisonnable pour un résultat pré-calculé, mais totalement inacceptable relativement à la capacité des systèmes cibles. Notons néanmoins que l'analyseur n'a, pour le moment, pas fait l'objet d'une phase d'optimisation très poussée, de sorte qu'il est vraisemblable que les performances s'améliorent grandement avec le temps. En tout état de cause, la simple complexité intrinsèque de l'algorithme de calcul (10 analyses en moyenne par bytecode) justifie la séparation en deux phases et l'intérêt porté à une vérification en une passe.

4.7.3 Discussion sur la vérification

Afin de pouvoir tirer partie des résultats de l'analyse, il apparaît comme nécessaire de recourir à une division fiable entre l'exécution de l'analyse et l'exploitation de ses résultats. Cet objectif peut être atteint à l'aide de mécanismes proches du « proof-carrying

[6] http ://jakarta.apache.org/bcel/
[7] http ://www.lifl.fr/ ghindici/STAN/

code » [Nec97], et plus précisément de la vérification légère de bytecode [RR98], qui présente l'avantage de garantir un processus de vérification linéaire par rapport à la taille du code (chaque bytecode est en fait analysé exactement une fois).

L'algorithme original de vérification est conçu pour être appliqué au calcul de types (en effet la vérification du typage correct d'une application est bien souvent le premier souci d'un système sécurisé), si bien qu'il nous faut nous placer à notre tour dans des conditions similaires pour l'appliquer sans crainte de laisser une faille dans le processus. Pour cela il suffit de considérer un treillis (qui est la forme canonique des systèmes de types utilisés), ce qui est aisément faisable en utilisant la définition de l'ordre sur les signatures de méthodes manipulées.

Intuitivement, une signature est correcte si le comportement qui y est décrit est similaire ou plus pessimiste que le comportement réel : ainsi dans le contexte de l'analyse d'alias, présumer qu'une variable est un alias vers plus d'objets que strictement nécessaire est une simple perte de précision et non une erreur (l'inverse au contraire est inacceptable). En particulier, rien n'empêche une méthode d'afficher délibérément un comportement plus pessimiste que ce que l'analyse est en mesure de prouver. De telles manipulations peuvent se révéler particulièrement payantes lorsqu'il s'agit de méthodes abstraites : comme elles n'ont aucun code associé, la signature « exacte » (qui n'a de fait aucun lieu d'exister, toute invocation devant être reliée à une méthode concrète) ne produit aucun lien, comportement qui serait manifestement un problème si l'on désire tolérer des sous-classes chargées dynamiquement. Il est donc possible d'associer à de telles méthodes une signature « raisonnable » (ou même tout à fait pessimiste) pour raisons d'extensibilité.

L'avantage majeur de cette méthode de vérification est qu'elle permet une vérification linéaire lorsque le code chargé par le système embarqué est annoté à l'aide d'indications de preuve aux points du code constituant la cible d'un saut quelconque. Cette méthode a par ailleurs été exploitée par le passé avec succès dans le contexte de CAMILLE [GLV99], pour la vérification du typage de son code intermédiaire. Si le code (ou la preuve, évidemment) est modifié par un attaquant de telle manière que la propriété soit violée, alors le processus de vérification ne peut en aucun cas aboutir, et le code fautif peut être rejeté simplement. Notons bien que la preuve fournie ne garantit pas la non modification du binaire, mais uniquement le bon respect de la propriété annoncée (laquelle est très générale par rapport au code).

L'obtention de la linéarité de la vérification nous permet de ne conserver que cette complexité (cette fois acceptable) pour la partie embarquée de l'analyse. Les expériences menées sur le coût additionnel engendré montrent une augmentation de 1,65% de la taille de code pour le stockage du dictionnaire de signatures de l'API (ce qui devrait être assez représentatif du coût moyen pour toute application), ce à quoi il faut ajouter en moyenne 45 octets d'indications de preuve par méthode (qui peuvent être supprimées une fois la méthode définitivement chargée), ce qui représente, selon les méthodes, une augmentation de 5% à 10% du volume des fichiers classes à l'entrée du chargeur de code. Ces chiffres proviennent de l'implantation courante de l'algorithme, qui prend également en compte des notions de flux d'information pour des besoins spécifiques [Ghi05].

Pour conclure sur l'implantation de l'analyseur présenté ici, le moteur est constitué d'un interpréteur abstrait aussi générique que possible, auquel il est possible d'adjoindre des greffons concernant le calcul d'informations obtenues à partir du traitement du flot de contrôle. De cette façon, l'analyseur, ou du moins sa partie embarquée peut être intégré au vérifieur de types qui se trouve nécessairement dans un système embarqué sécurisé, et peut également être étendu pour mener à bien des analyses différentes, sans qu'il soit nécessaire de tout réécrire. En regroupant ces mécanismes à finalités différentes, il est possible de réduire le coût du processus de chargement (à la fois en taille de code, et en temps de calcul).

4.8 Conclusion

L'analyse d'alias présentée dans ce chapitre est divisée en deux partie (avant déploiement, et au chargement). De ce fait il est possible, comme nous l'avons vu, de déporter la partie coûteuse du traitement au niveau du producteur de code, tandis que le consommateur peut se contenter d'effectuer le travail beaucoup moins lourd de vérification du code qu'il reçoit.

De plus, le fait de concevoir le résultat de cette analyse comme une information de typage permet de tirer avantage de techniques bien connues de vérification dans ce domaine, ce qui simplifie grandement l'obtention de la garantie de correction et permet d'envisager la factorisation d'une grande partie du code effectif.

En procédant de la sorte, il est possible d'obtenir une analyse d'alias suffisamment légère pour être utilisée dans le contexte des systèmes embarqués, et suffisamment précise pour être utilisée comme base pour d'autres analyses. Ce sont ces dernières que nous allons présenter dans les chapitres suivants.

Chapitre 5

Analyse d'échappement

Dans le chapitre précédent 4, nous avons présenté une analyse d'alias destinée à servir de base pour le calcul d'analyses d'un plus grand intérêt pour les systèmes embarqués. En particulier, dans ce chapitre nous nous intéresserons à l'extraction immédiate de résultats d'analyse d'échappement à partir des résultats de l'analyse d'alias. Cette analyse sera donc également compositionnelle et la vérification de sa correction globalement équivalente à la vérification de l'analyse d'alias.

5.1 Présentation

L'analyse d'échappement[1] est une analyse qui fut développée, dans un premier temps, dans le contexte des langages fonctionnels [PG92], qui plus tard a été adaptée aux langages orientés objet. Son but est de détecter statiquement, dans la mesure du possible, à quel point du programme un objet alloué devient inaccessible et donc inutile. Une fois ceci établi, il devient possible, par des moyens divers, de supprimer les structures désormais inutiles sans avoir besoin de mécanismes auxiliaires tels que les ramasse-miettes[2]. Ceux-ci pourraient également détecter que les objets en question ne sont plus atteignables dans le programme, mais au prix d'opérations supplémentaires de test sur tout ou partie de la mémoire disponible, ce qui peut se révéler très coûteux. Obtenir statiquement une telle information permet de parvenir à une meilleure gestion de la mémoire.

Dans le cas de Java, langage dans lequel la mémoire est gérée automatiquement, les notions de vie ou mort d'objets peuvent se définir très simplement relativement à l'action d'un ramasse-miettes. On attend en effet du passage de ce dernier un tri (et une suppression) des objets morts pour ne laisser en place que les objets vivants (les seuls encore utiles pour la suite de l'exécution). Ainsi donc, l'analyse sera assurée correcte si elle ne marque comme pouvant être éliminés en un point du programme que des objets qui seraient effectivement supprimés en cas de passage d'un ramasse-miettes en ce point. De tels objets,

[1]« escape analysis » en anglais.
[2]« garbage collectors » en anglais.

lorsqu'ils sont identifiés par l'analyse d'échappement, sont appelés « capturés ».

Définition 20 (Objets capturés). *Un objet o est dit capturé dans le domaine d'une méthode m si il est créé dans m et qu'après l'exécution de m il n'existe plus de référence pointant vers o.*

Définition 21 (Définition opérationnelle). *Soit m une méthode. L'ensemble $\mathcal{C}_m$ des objets capturés dans m est défini par :*

$$\mathcal{C}_m = \{o \mid o \in Alloc_m, \forall (o', o) \in Link^r_m, o' \in Alloc_m \cup Func_m\}.^3$$

Un avantage souvent avancé du recours à l'analyse d'échappement est le fait que les objets s'en trouvent mieux placés en mémoire. En effet, par nature, l'analyse d'échappement capture les objets temporaires (ou de travail), qui sont généralement utilisés intensément mais de façon très localisée : c'est à dire bien souvent dans une seule méthode. Par ailleurs, pour des raisons de gestion homogène de la mémoire, il est fréquent de déporter lors de leur création ces objets dans une zone mémoire spécifique (une « région ») dont l'organisation sera adaptée à leur destruction. La combinaison de ces deux facteurs a pour conséquence de regrouper naturellement les objets de travail dans une même zone mémoire. Or, la proximité spatiale de données les rend plus susceptibles de se trouver simultanément dans la mémoire cache, ce qui est très intéressant lorsque les objets sont également proches en termes d'accès rapprochés (en temps) par le programme. L'une des implantations communément utilisées de ces régions est liée étroitement à la pile d'exécution du programme : dans la mesure où la vie des objets capturés est généralement bornée par le temps d'exécution d'une fonction, il est logique (et très pratique car peu coûteux) de détruire la région utilisée pour leur allocation à la fin de l'exécution de la fonction. Certaines implantations se contentent d'ailleurs de placer les objets ainsi détectés dans la zone des variables locales de la fonction, de sorte qu'aucune manipulation supplémentaire n'est nécessaire.

Cependant, cette exploitation de l'analyse d'échappement est d'un intérêt secondaire dans notre contexte de travail, et d'autres propriétés sont plus intéressantes en ce qui nous concerne. Tout d'abord, il faut noter que bien des petits systèmes embarqués (qui restent notre cible de prédilection) ne comportent simplement pas de mémoire cache, ce qui rend la question caduque. Mais surtout, il existe une conséquence de la mise en place de l'exploitation d'une analyse d'échappement qui est d'un bien plus grand intérêt dans notre contexte. Typiquement, les systèmes que nous visons possèdent de très petites quantités de mémoire, ce qui fait que les passages du ramasse-miettes sont très rapprochés dans le temps et finissent par représenter une part importante du temps total d'exécution. De plus, de manière très générale, des passages de ramasse-miettes à fréquence élevée sont vecteurs d'inefficacité, puisque la probabilité pour un objet de survivre à un passage augmente avec cette fréquence. Ceci mène donc à des analyses de la mémoire répétées pour un gain faible. Une analyse d'échappement a typiquement pour effet de réduire le nombre de passages du ramasse-miettes, puisqu'elle tend à favoriser la destruction des objets au plus tôt. Comme

[3] $o' \in Alloc_m \cup Func_m$ est suffisant, la relation étant symétrique et close par transitivité.

présenté dans [GHSR06a], la conséquence est donc de réduire considérablement le temps passé à réorganiser la mémoire, et de réserver l'utilisation du ramasse-miettes aux cas dans lesquels il apporte réellement un bénéfice quant à l'occupation de la mémoire.

Notons enfin que la suppression de synchronisations inutiles grâce à l'exploitation de l'analyse d'échappement sur Java [BH99] est également pertinent dans notre contexte : bien que les applications déployées sur les systèmes qui nous intéressent ne cherchent pas à tirer partie de parallélisme, de nombreuses synchronisations défensives sont présentes dans le code. Il est donc bénéfique de s'affranchir de ces dernières partout où cela est possible.

5.2 Analyses d'alias et d'échappement

L'objectif de cette section est de présenter une analyse d'échappement basée sur l'analyse d'alias présentée au chapitre 4. La finalité de l'analyse d'échappement étant de calculer un ensemble d'objets susceptibles d'être détruits en un certain point d'un programme, elle est bien entendu fortement liée à la capacité de garantir qu'aucune référence vers un objet n'a été conservée depuis un objet « vivant », à savoir un objet qui ne pourrait être détruit par le passage d'un ramasse-miettes à cet instant. Cette notion de référence conservée est précisément ce qui est calculé par l'analyse d'alias présentée précédemment.

Puisque le but est d'altérer la stratégie d'allocation des objets en mémoire, seuls les objets de type « référence » nous intéressent, par opposition aux types « primitifs » qui n'ont pas de zone de stockage[4]. En Java, cela revient à s'intéresser exclusivement aux instances de `Object`. Comme cela a été fait dans de précédents travaux autour de l'analyse d'échappement (tels que [BSF04, Bla99]), notre approche se base sur l'analyse de pointeurs (ou d'alias) présentée au chapitre 4. Selon les conditions d'applicabilité de l'analyse, et les limites des plates-formes concernées, l'analyse d'alias varie dans ses détails : ainsi par exemple dans le cas de [BSF04], l'analyse d'alias est axée sur les variables plus que sur leur contenu individuel, et elle unifie tous les contenus possibles des variables, sans notion de temps. La conséquence est une analyse légère et facilement vérifiable, mais assez imprécise puisqu'elle considère comme échappés tous les objets susceptibles de se trouver à un instant de leur vie dans la même variable qu'un objet échappant (et ce même si ce n'est pas à ce moment que l'objet échappe).

D'un autre côté, les choix faits dans [Bla99] garantissent un maximum de précision, puisque tous les chemins vers les différents attributs sont conservés à tout instant. Mais le revers de la médaille est une analyse extrêmement lourde et complexe, dont les résultats sont à eux seuls trop volumineux pour envisager une utilisation dans un contexte contraint, et d'autant plus lorsque le contexte d'exécution se veut ouvert au chargement dynamique de composants.

[4]les types référence décrivent la structure pointée par la donnée considérée, tandis que les types valeurs décrivent la donnée elle-même.

5.2.1 Exemple de résultat attendu

La figure 5.1 présente un exemple, très simple, de ce que l'on attend d'une analyse d'échappement, et illustre le lien qu'a cette dernière avec l'analyse d'alias présentée précédemment. La méthode `m()` crée un alias (un nouveau point d'accès) pour l'une des deux références passées en arguments, dès lors que son résultat est stocké pour un usage ultérieur. Cependant, on ne peut en général pas déterminer statiquement laquelle des deux références est concernée, faute de pouvoir évaluer `condition`.

L'analyse d'alias conclut donc naturellement que les deux arguments sont retournés, ce qui est la plus petite approximation correcte calculable statiquement. Du point de vue de l'appelant de la méthode `m()`, les deux objets créés sont associés à la valeur de retour `o`.

```
Object m(Object a, Object b) {
  return condition ? a : b;
}

Object f() {
  Object o = m(new Object(), new Object());
  return o;
}

Object g() {
  Object o = m(new Object(), new Object());
  return null;
}
```

FIG. 5.1 – Alias.

La connaissance du contenu potentiel de chaque variable (au sens large de variable, on ne parle pas ici des variables locales d'un programme, mais plutôt de tout ce qui peut être nommé) à un instant donné engendre de façon évidente une surestimation de l'ensemble des objets vivants du système, ou par complémentarité, une sous-estimation de l'ensemble des objets morts. Rappelons que les objets morts sont ceux qui disparaîtraient si un passage de ramasse-miettes devait être effectué à la position courante du programme.

Dans la méthode `f()` de la figure 5.1, la variable `o` contient en fait une seule référence (et l'autre est morte) mais il n'est pas possible de le décider statiquement, aussi l'analyse devra-t-elle marquer les deux références comme vivantes à la sortie de `f()` (puisqu'elles sont retournées à l'appelant de `f()`), et laisser le ramasse-miettes œuvrer.

En revanche, en ce qui concerne la méthode `g()` de la figure 5.1, peu importe quel objet se trouve réellement dans la variable `o` puisque l'objet en question ne survit pas à l'appel de méthode. Les deux objets créés pour servir de paramètres à `m()` sont des objets temporaires, purement locaux à la fonction `g()`. Ceux-ci devront être marqués comme

capturés par l'analyse d'échappement, et pourront être détruits dès la fin de l'exécution de la méthode `g()`.

L'analyse d'alias est le cœur de l'analyse d'échappement proposée ici, car elle fournit des résultats permettant de déterminer l'échappement ou non d'un objet par simple lecture de ceux-ci. Bien évidemment, la vérification est elle aussi très simple à effectuer, une fois les alias validés.

L'analyse d'échappement étant construite à partir de l'analyse d'alias (et n'étant en fait qu'une interprétation de ses résultats), elle hérite de ses propriétés majeures :
- le code analysé, contrairement à beaucoup d'autres analyses d'échappement, est le bytecode Java plutôt que le code source ;
- l'analyse proposée couvre intégralement les fonctionnalités de Java ;
- l'analyse est incrémentale (avec une granularité au niveau des méthodes) et permet le chargement dynamique de nouvelles classes ;
- l'embarquement et la vérification au chargement des classes ne pose pas de problème.

En outre, contrairement à d'autres travaux voisins [Gal05], l'analyse d'échappement que nous présentons ici tient compte du flot de contrôle, ce qui permet une précision accrue des résultats en contrepartie d'une taille plus importante des signatures et annotations de preuve à destination du vérifieur.

5.2.2 L'analyse d'échappement

Donnons tout d'abord une définition plus opérationnelle de ce qu'est précisément l'analyse d'échappement dans notre contexte.

Selon la définition usuelle, on dit qu'un objet est capturé lorsqu'il est créé dans une méthode et peut-être détruit (ou collecté, par référence au terme anglais de « garbage collector ») à la fin de cette méthode. C'est à dire qu'il est garanti que l'objet disparaîtrait si un ramasse-miettes complet (détruisant tous les objets inaccessibles) était déclenché à la fin de la méthode. Grâce à l'analyse d'échappement, on sait donc que l'objet ne peut en aucun cas être encore utile (ou utilisé) après ce point.

Partant d'une analyse d'alias, il est plus simple de définir les objets survivants (ou échappés) que les objets capturés. Ceux-ci sont définis de la manière suivante : est considéré comme échappé tout objet pour lequel il est possible de trouver un lien (statiquement parlant, donc un lien potentiel) pointant vers lui qui ait pour conséquence de le faire survivre à la méthode courante. Cette définition est une approximation un peu large dans la mesure où il est tout à fait envisageable qu'un objet conserve un lien vers un autre sans pour autant être amené à l'utiliser dans le futur. Néanmoins, dans notre contexte d'application, qui inclut la notion d'ouverture au monde extérieur, et notamment par le chargement dynamique de code, il n'est pas possible *a priori* de détecter et de contrôler les utilisations futures d'un tel objet. En effet, il n'existe aucune connaissance de l'ensemble des méthodes qui appellent la méthode analysée.

Définition 22. *Un objet est dit échappé d'une méthode s'il est créé dans cette méthode,*

et potentiellement rendu accessible depuis l'extérieur de cette méthode.

En prenant l'analyse d'alias pour base, l'ensemble de conditions suivant décrit les différentes alternatives amenant à la conclusion qu'un objet est échappé.

Règle 1. *Un objet sera considéré comme échappé si :*
- *il est référencé depuis un champ statique, ou un paramètre, ce qui le rend naturellement accessible depuis l'extérieur de la méthode (les exceptions font l'objet d'un cas détaillé un peu plus loin) ;*
- *ou il est retourné par la méthode ;*
- *ou il est référencé depuis un objet qui échappe.*

Remarque : les approximations héritées de l'analyse d'alias exposée précédemment ont pour conséquence directe le fait qu'un objet sera considéré comme échappé dès que l'un de ses champs le sera.

Par complémentarité, un objet sera considéré comme capturé s'il ne se trouve dans aucune des situations énumérées dans les règles 1. Notons que ces règles peuvent déboucher sur une fausse détection d'objets échappés, mais en aucun cas sur une fausse détection d'objets capturés, ce qui est la condition de correction de l'algorithme. En effet, les résultats sont utilisés pour optimiser l'allocation des objets capturés, et donc changer légèrement la sémantique du programme en détruisant ces objets plus rapidement. Il va de soi qu'un objet détruit prématurément de façon incorrecte aurait un impact désastreux sur le système qui chercherait encore à le joindre.

Toutes les informations nécessaires à l'analyse d'échappement sont calculées par l'analyse d'alias sous-jacente : afin de détecter les objets échappés d'une méthode, il est suffisant d'analyser les tableaux de liens générés par l'analyse d'alias pour cette méthode, et d'appliquer les règles 1. Aucune analyse inter-procédurale supplémentaire n'est requise, et la vérification est également très simple, puisque les quelques calculs supplémentaires requis pour obtenir l'analyse d'échappement à partir de l'analyse d'alias sont très légers et peuvent donc être effectués dans leur intégralité par le système embarqué sans pour autant infliger de pénalité trop sévère aux performances de ce dernier.

Allocations en boucle d'objets capturés

Certains travaux préexistants [GS00, WR99] font des choix d'implantation qui leur permettent d'allouer certains objets dans la zone des variables locales. En terme de performances, cette stratégie est très pratique car elle dispense de toute gestion ultérieure des objets placés là : en effet, ceux-ci seront naturellement détruits au retour de la fonction, sans qu'il ne soit besoin d'autre chose que du mécanisme de la convention d'appel de Java. Néanmoins dans cette stratégie, l'allocation des objets en pile dans une boucle pose problème, car la zone des variables locales ne pourrait plus être bornée statiquement, ce qui va à l'encontre des contraintes des environnements d'exécution Java (Marmot [FKR+00] par exemple travaille à taille de fenêtre d'exécution fixe), et est extrêmement préjudiciable

pour toutes les vérifications (notamment de typage) qui ont lieu au sein du système. La solution évidente, utilisée dans les travaux de Whaley et Rinard pour leur implantation basée sur Jalapeño [AAC+99], est de désactiver l'allocation en pile pour les créations d'objets ayant lieu dans une boucle, mais cela est peu satisfaisant puisque le cas est fréquent et le manque à gagner proportionnel à la taille (à l'exécution) de la boucle.

Dans notre contexte, la solution adoptée consiste en la gestion d'une pile séparée pour les allocations d'objets capturés, afin de s'affranchir des contraintes évoquées en termes de taille des données. Cette pile est gérée de façon très similaire à la pile d'exécution, et en particulier ses étages sont supprimés de façon synchrone. À chaque étage de la pile d'exécution est associé un étage de la pile d'allocation, créé et détruit simultanément Cette modification de l'environnement d'exécution est légèrement plus lourde que la précédente solution, mais reste néanmoins très simple à mettre en œuvre.

Exceptions

La gestion des exceptions est un problème qui se pose de manière récurrente dans l'analyse d'échappement [CRL98]. Comme il a été exposé précédemment en section 4.4.6, il est impossible de déterminer statiquement le flot de contrôle engendré par la présence des exceptions et des codes de gestion associés, puisqu'on ne sait pas dans le cas général si une exception est gérée ou non. De ce fait, il est plus sûr de considérer tout objet attaché à une exception comme échappé.

Cependant, bien qu'étant absolument sûre, cette stratégie présente le défaut de dégrader considérablement les performances de l'analyse, à cause du grand nombre d'exceptions potentiellement levées au cours de l'exécution du programme, notamment les exceptions « runtime ». Par ailleurs, la survenue d'exceptions devrait être, comme leur nom l'indique, exceptionnelle. Il est donc d'autant plus regrettable de renoncer à des allocations optimisées en sachant qu'elles seraient légitimes la plupart du temps, même si une exception est susceptible d'être déclenchée en de rares occasions.

Pour ces raisons, nous choisissons de ne pas considérer les objets liés à des exceptions comme échappés. Pour compenser les erreurs potentiellement engendrées par cette stratégie *a priori* erronée, nous introduisons un mécanisme à l'exécution qui sera chargé de déplacer les objets alloués incorrectement en pile vers le tas mémoire lorsqu'ils échappent par exception. Bien évidemment, ce mécanisme est relativement coûteux, si bien qu'il n'est rentable que dans le cas où la nécessité de recourir à lui est suffisamment rare, ce qui devrait être le cas dans un mode de programmation où l'exception n'est pas utilisée pour autre chose que la gestion des cas exceptionnels nécessitant la traversée rapide de la pile d'appels.

L'utilisation d'exceptions à seules fins de gestion de flot de contrôle (ce qui est en général considéré comme une très mauvaise idée à cause du coût des traitements des exceptions) pourrait certainement conduire à une perte d'efficacité spectaculaire.

Notons néanmoins que les exceptions elles-mêmes ne sont pas allouées en pile, car nous considérons que le fait de les créer suppose une utilisation non locale à la fonction courante, de sorte qu'il serait généralement nécessaire de les déplacer immédiatement. En revanche,

pour l'ensemble des autres objets (non exceptions), le fait d'être attaché à une exception n'est pas suffisant pour le faire échapper, et il est donc possible de l'allouer en pile. La volonté sous-jacente est d'éviter que des objets pouvant être attachés à une exception à l'exécution (mais l'étant rarement en pratique) se retrouvent systématiquement dans le tas mémoire.

5.2.3 Comparaison à d'autres analyses

Allocations cachées

Le fait d'analyser le bytecode Java nous permet de détecter naturellement toutes les allocations d'objets, sans avoir besoin de recourir à des mécanismes auxiliaires pour les allocations cachées au niveau du code source. En particulier, le code exposé en figure 5.2 contient une allocation d'un nouvel objet, qui n'est nullement apparente à la seule vue du code Java (absence d'instruction `new`). De même, les allocations statiques de tableaux ont une syntaxe telle que les allocations sous-jacentes sont peu détectables au niveau du code source.

```
class C {
  String m(String s){
    return "Hello" + s;
  }
}
```

FIG. 5.2 – Allocation cachée.

En revanche, en regardant le bytecode de la méthode `m()` reproduit en figure 5.3, il apparaît qu'un objet de type `StringBuffer` est créé[5], qui est utilisé pour la concaténation des chaînes de caractères.

Certains travaux [BSF04] qui analysent le code source choisissent de substituer à la forme « additive » de la concaténation l'expression équivalente correspondant exactement au bytecode généré, comme présenté en figure 5.4. Néanmoins, cette approche introduit une dépendance par rapport au compilateur employé, puisque l'on présume de son comportement dans un cas particulier, chose qui n'est pas garantie pour les versions futures des compilateurs. Cette approche n'est donc pas très satisfaisante.

D'autres travaux comme [BSF04] proposent des solutions pour gérer ces cas, en décidant arbitrairement que l'objet `StringBuffer` n'échappe pas et peut être alloué en pile, ce qui est exact. Néanmoins, un problème de cohérence de l'analyse se pose sur ce cas particulier,

[5]Ceci n'est vrai que dans les versions de Java inférieures à 1.5. Néanmoins le principe demeure le même dans les versions ultérieures, bien que les classes mises en jeu soient différentes.

car si le code était substitué de la façon présentée en figure 5.4, l'analyse de[BSF04] tirerait probablement la conclusion que l'objet StringBuffer est potentiellement échappé.

En effet, dans de nombreuses implantations de l'API Java, il est fait usage de stratégies « create-on-write » (création à l'écriture, ou création retardée) là où cela est possible, pour déporter la création des objets le plus tard possible, au cas où il ne serait finalement pas nécessaire de le faire, ce qui assure bien évidemment des performances accrues. En particulier, la méthode toString() fait souvent usage de cette technique, en créant simplement un objet String qui ne fait que partager le tableau de caractères compris dans le StringBuffer. Le tableau ne sera copié que si le StringBuffer doit subir une altération.

Pour des raisons propres à la gestion des champs d'un objet, ce partage d'information entre les deux objets (et la survie de l'objet String) déclencherait un artefact de l'analyse de [BSF04] dénonçant un échappement de l'objet StringBuffer, ce qui se révèle être trop pessimiste. Ainsi donc, en l'absence d'une gestion particulière de ce cas précis, l'analyse présentée ne pourrait détecter le non échappement des StringBuffer. De même, l'analyse présentée ici rencontre ce problème à cause des approximations présentées pour l'analyse d'alias. Le cas se présentant fréquemment, il serait d'autant plus dommage de renoncer au bénéfice potentiel.

```
java.lang.String m(java.lang.String);
Code:
0:   new     #2; //class StringBuffer
3:   dup
4:   invokespecial   #3; //StringBuffer();
7:   ldc     #4; //String Hello
9:   invokevirtual   #5; //StringBuffer.append(String);
12:  aload_1
13:  invokevirtual   #5; //StringBuffer.append(String);
16:  invokevirtual   #6; //StringBuffer.toString();
19:  areturn
```

FIG. 5.3 – Bytecode correspondant.

Bien sûr, avec une implantation naïve de StringBuffer.toString(), l'analyse conclurait naturellement au non échappement de l'objet StringBuffer. Afin de pouvoir se comparer de façon honnête à [BSF04], nous faisons le même choix qu'eux, à savoir marquer arbitrairement ces allocations de StringBuffer comme capturées et donc à effectuer en pile. Encore une fois, ceci n'est pas faux, même si aucune des deux analyses n'est capable de le détecter d'elle même. Là où une analyse sur le code source doit de toute façon considérer un cas particulier, une analyse sur le bytecode n'a pas d'autre choix que de forcer la signature de la méthode StringBuffer.toString() afin de lui faire refléter le comportement qu'elle aurait si elle était implantée naïvement sans optimisation « create-on-write ». Ainsi,

```
class C {
  String m(String s){
    return new StringBuffer().append("Hello")
                             .append(s).toString();
  }
}
```

FIG. 5.4 – Expansion du code.

on supprime le lien entre le `StringBuffer` et la `String` résultante, en faisant comme si il n'y avait aucun partage de données entre les deux.

Cette adaptation, quoique paraissant peu légitime au premier abord, est essentielle pour permettre de se comparer dans de bonnes conditions à des résultats existants. Il convient néanmoins de noter qu'elle est avant toute chose correcte vis-à-vis de l'analyse d'échappement : le fait que l'analyse décrite ici ne soit pas en mesure de détecter le comportement n'est pas pertinent en la matière. Mais en outre, il ne paraît finalement pas déraisonnable de fournir des résultats plus précis que ce que l'on est capable d'inférer, dans des cas suffisamment courants pour justifier la plus grande précision possible. Ceci permet de compenser en partie les approximations qui sont faites pour gagner du temps de calcul ou de l'espace. Enfin, comme nous le voyons dans la section suivante, l'inadéquation de l'analyse avec les situations de cette espèce se justifie par le conflit entre une optimisation manuelle et une optimisation automatique.

Interférences avec d'autres optimisations

Un point notable dans l'exemple précédent est qu'il arrive que certaines optimisations jouent en défaveur de l'analyse d'échappement. Nous avons pu observer le cas des stratégies de création retardée, mais il en va de même par exemple pour l'utilisation d'objets pré-alloués référencés par des champs statiques en lieu et place d'objets temporaires fréquemment utilisés.

De telles optimisations ont été introduites principalement pour compenser l'obligation de s'en remettre au ramasse-miettes pour détruire les références (et les choix d'implantation des fonctions concernées pourraient être totalement différents dans des langages laissant la responsabilité de la mémoire au programmeur). De ce point de vue, ces optimisations tentent précisément de compenser l'absence `a priori` d'optimisations automatiques telles que l'exploitation de l'analyse d'échappement. En définitive, il n'est pas tellement surprenant que ces optimisations rendent l'analyse d'échappement plus difficile et moins efficace, puisqu'elles tentent de contourner l'absence d'allocation statique, et ont donc tendance à produire des effets similaires, à savoir faire sortir les objets du champ d'action de l'allocation dynamique, et donc du champ des mécanismes et optimisations conçus pour ce modèle. Il apparaît donc que l'analyse d'échappement et les diverses optimisations évoquées

procèdent d'une démarche orthogonale, et il est logique que leur utilisation conjointe soit rarement possible.

Par ailleurs, il ne semble pas vraiment dérangeant que l'analyse ait un impact relativement faible sur un code d'API déjà optimisé. Après tout, le fait qu'il soit optimisé diminue déjà de beaucoup le bénéfice qu'il y aurait à le traiter différemment. À l'opposé, l'utilisateur n'est pas censé penser aux limitations de son environnement d'exécution lorsqu'il écrit une application. De ce fait, il est la vraie cible d'optimisations *a posteriori* telles que l'analyse d'échappement, dont son code bénéficiera pleinement.

Chargement dynamique

Le fait que notre approche permette le chargement dynamique de classes est essentiel. Par conception, Java est dédié au code mobile, à la liaison dynamique de code, à la réflexivité. De nombreuses analyses d'échappement existantes fonctionnent uniquement dans un monde clos, où tout le code d'une application est disponible *a priori*, et donc analysable en tant qu'ensemble fixe. Ceci implique que chaque méthode peut être analysée précisément dans tous ses contextes d'appel, déterminés à l'aide d'un graphe d'appels, ce qui permet une précision bien plus grande, notamment au niveau de la gestion des champs des objets. C'est le cas de l'analyse proposée en [Bla99].

L'analyse que nous proposons est bien plus proche de celle présentée en [BSF04], en raison de son souci de conserver une possibilité d'ouverture pour l'environnement d'exécution. Certaines différences sont néanmoins présentes :

1. nous travaillons sur les fichiers de bytecode, ce qui supprime tout écart entre les données à analyser et les données à vérifier ;
2. les paramètres sont suivis de manière plus précise, de sorte que passer un objet en paramètre effectif d'une fonction ne provoque pas son échappement systématique, même si la méthode appliquée n'est pas parfaitement déterminée ;
3. nous proposons par ailleurs un environnement d'exécution qui est capable de tirer parti des résultats fournis par l'analyse.

5.2.4 Implantation

Le fait que l'analyse d'échappement repose sur une analyse d'alias, comme nous l'avons vu, a bien évidemment un impact fort sur les résultats obtenus, la première héritant tout naturellement des choix faits pour la seconde. Par exemple, le fait d'avoir renoncé au suivi des champs des objets dans l'analyse d'alias (*cf.* section 4.4.5) signifie que l'analyse d'échappement ne peut en aucun cas distinguer l'échappement d'un objet de l'échappement d'un de ses champs, ceux-ci étant identifiés à un niveau inférieur. Nous verrons dans la section dédiée aux résultats expérimentaux 5.2.5 que cette imprécision a en fait un impact assez mineur sur les résultats obtenus.

L'implantation détaillée ici de l'analyse d'échappement est une interprétation directe des résultats obtenus par analyse d'alias, et plus précisément une lecture de la structure de tableau de liens calculée par cette analyse. Le code additionnel requis pour l'exploitation de ces résultats comporte moins de 300 lignes de code Java. Après l'analyse d'échappement, les résultats obtenus sont utilisés pour générer une information à destination du chargeur de code de la machine virtuelle qui décrit les positions des bytecodes correspondant à des allocations pouvant être effectuées en pile. La modification subie par le fichier consiste simplement en l'ajout d'attributs (tels que définis par la spécification de Java), ce qui permet donc de conserver un fichier tout à fait standard. Une machine virtuelle ignorant comment traiter ces informations peut simplement les ignorer sans poser de problème de cohérence. Enfin, les informations fournies sont très aisément vérifiables une fois que les informations relevant de l'analyse d'alias ont été validées.

Comme plate-forme d'expérimentation, le choix s'est porté sur JITS[6], car il s'agit d'une plate-forme expérimentale possédant déjà une infrastructure facilement adaptable aux besoins de l'analyse d'échappement, notamment grâce à la disponibilité de mécanismes d'allocation en pile. La machine virtuelle de JITS a donc été modifiée afin de supporter de nouvelles méthodes d'allocations pour exploiter les résultats de l'analyse d'échappement.

Pour simplifier les choses, les modifications apportées reviennent à fournir à la machine virtuelle les moyens d'interpréter quatre nouveaux bytecodes : `newstack`, `newarraystack`, `anewarraystack` et `multianewarraystack` qui ont des sémantiques proches de celles de `new`, `newarray`, `anewarray` et `multianewarray` respectivement, à l'exception notable du fait qu'ils allouent les objets demandés en pile.

Bien entendu, ces bytecodes n'ont aucune existence à l'extérieur de la machine virtuelle, et notamment ils ne sauraient être considérés comme valides dans un fichier de code intermédiaire. Aussi, il est tout à fait possible (et de fait, c'est le cas) de s'en affranchir en dehors du mode d'exécution « interprété ». Néanmoins, la vision sous forme de bytecodes additionnels semble la plus adaptée pour visualiser simplement l'enchaînement des étapes du déploiement du logiciel.

Ainsi donc, avant chargement, le fichier de code intermédiaire est tout à fait standard, et ne comporte que des allocations déclenchées par les bytecodes de la famille de `new`. Le chargeur de code, s'il le peut, prend en considération les annotations d'analyse d'échappement associées au fichier, les vérifie avec le code, puis en cas de succès substitue à un certain nombre de bytecodes « `new` » leur équivalent « `newstack` ». L'interprétation ultérieure de ce code déclenchera donc des allocations en pile aux endroits indiqués, sans que les nouvelles fonctionnalités soient accessibles à l'utilisateur.

Le choix d'implanter ces fonctionnalités spécifiques sous la forme « haut niveau » de bytecodes se justifie essentiellement de deux façons :

- la spécification de la machine virtuelle standard laisse un certain nombre d'emplacements disponibles pour de nouvelles instructions sans altération du codage existant, qui sont précisément prévus à cet effet. Bien que les bytecodes spécifiques ajoutés

[6]http ://www.lifl.fr/RD2P/JITS

n'aient pas d'existence à l'extérieur de la machine virtuelle, cela permet de se greffer aisément (sans modification majeure) dans la boucle interne de gestion des bytecodes ;
- l'interprétation de nouveaux bytecodes est plus efficace à l'exécution que la modification (avec un ajout dynamique) des bytecodes d'allocations existants.

La pile d'exécution de JITS a été légèrement modifiée afin de supporter une allocation en pile fortement liée aux fenêtres d'exécution de la pile Java. De ce fait le contrôle des objets ainsi alloués (et qui sortent donc de la sphère d'influence du ramasse-miettes) est particulièrement simple puisque directement rattaché au mécanisme de convention d'appel de fonction. À chaque retour de fonction, et donc à chaque opération de restauration de contexte dans la pile d'exécution, les objets se trouvant avoir été alloués en pile dans le contexte de la méthode en cours de terminaison sont supprimés, la fin de la méthode étant une borne fiable de leur vie maximale. Ainsi, bien que les piles soient séparées pour des raisons annexes exposées précédemment, la suppression d'un étage de la pile (ou « frame ») se fait de manière synchrone entre les deux structures. Ceci rend l'implantation des bytecodes `newstack` et assimilés triviale : il s'agit simplement d'empiler l'objet souhaité dans la fenêtre courante de la pile dédiée, sachant que la création et la suppression de cette fenêtre sera prise en charge de manière inconditionnelle par les actions d'appel et de retour de la fonction englobante. Les objets marqués comme non échappés ne survivant pas à l'exécution de la méthode, l'objectif est atteint.

5.2.5 Résultats expérimentaux

Pour illustrer du mieux possible les résultats obtenus par l'analyse d'échappement décrite en section 5.2.2, il a été choisi de se référer aux jeux de tests couramment utilisés dans la littérature pour tester des analyses similaires, notamment [BSF04] et [WR99]. Par ailleurs, d'autres exemples ont été sélectionnés, moins usuels mais peut-être plus représentatifs des situations où l'analyse d'échappement a un rôle fort à jouer en tant que processus optimisant.

En particulier, nous avons ajouté l'analyse d'un exemple particulièrement adapté à l'analyse d'échappement : Dhrystone [Wei84]. Ce programme utilise de manière intensive es tableaux d'entiers, qui sont passés en paramètre aux méthodes de traitement, mais n'échappe pas à leur méthode de création, ce qui est détecté grâce à la précision des signatures de méthodes.

Enfin, il convient de noter que l'intégralité de l'API Java implantée dans JITS (suffisante pour exécuter les jeux de tests) a également été analysée.

Les travaux venant de [BSF04] définissent une analyse d'échappement destinée aux environnements ouverts et constituent donc notre référence majeure en matière de comparaison. Inversement, les travaux issus de [WR99] concernent des environnements fermés, nous fournissant une borne supérieure pour l'évaluation de la qualité de nos résultats.

Certains jeux de test, en provenance de la suite JavaGrande ont été exclus des résultats présentés ici pour cause de manques dans la plate-forme JITS, encore expérimentale, qui

ont empêché de mener à bien l'analyse et/ou l'exécution de ces programmes. Néanmoins, le panel de tests effectués est suffisant pour évaluer l'algorithme proposé.

La colonne « statique » de la table 5.1 décrit les résultats obtenus par l'analyse des programmes de test, présentant pour chacune des analyses le ratio de sites d'allocation détectés comme pouvant créer des objets en pile.

Précisons tout d'abord que le nombre total de sites d'allocation détecté varie suivant les analyses. Ceci s'explique par le fait que l'analyse présentée ici est appliquée au code intermédiaire alors que les deux autres analyses [BSF04, WR99] agissent sur le code source Java. De ce fait, un certain nombre d'allocations cachées dans le code source sont apparentes pour notre analyse. En l'occurrence ce phénomène semble jouer en notre faveur, puisqu'il apparaît que bon nombre de sites d'allocation cachés sont en définitive capturés.

Pour chaque jeu de test, les résultats obtenus sont meilleurs que ceux en provenance de [BSF04]. Ceci s'explique en partie par le nombre supérieure d'allocations détectées, qui semblent être souvent concernées par une allocation en pile, mais surtout par le fait que nous proposons une approximation moins pessimiste vis-à-vis des objets pour lesquels aucun type exact ne peut être déterminé.

Les colonnes « dynamiques » de la table 5.1, qui présentent les résultats à l'exécution des jeux de test sont également particulièrement intéressantes puisqu'elles fournissent des informations sur l'impact réel de l'analyse d'échappement sur l'exécution : les résultats purement statiques n'apportent eux que peu d'information, puisque traitant à égalité des instructions répétées en boucle et du code non-exécuté.

Bien sûr, les résultats présentés pourraient varier suivant les entrées données aux programmes (ceux-ci font grand usage de boucles, aussi un nombre différent d'itérations aurait une influence sur les ratios constatés). Néanmoins, ces résultats demeurent représentatifs de l'impact réel de l'analyse d'échappement.

Les résultats « dynamiques » sont divisés en deux parties, car il est impossible de considérer sur un même plan le code du test lui-même et le code des API, utilisé indirectement : le premier est constant, alors que le second est naturellement dépendant de l'implantation utilisée pour la machine virtuelle. Les résultats sont donc toujours présentés en deux sections : la partie propre au test, et la partie propre à la machine virtuelle sous-jacente. Dans ces deux groupes, les données intéressantes sont l'ensemble des objets alloués et le pourcentage d'objets qui le sont en pile.

En ce qui concerne le code propre des tests, on peut constater que les résultats varient grandement d'un test à un autre. Incidemment, ces résultats montrent sans ambiguïté le peu de pertinence des chiffres statiques. Un ratio relativement peu élevé de sites d'allocations détectés, comme celui d'Euler et de Search, ou encore de Raytrace et Raytracer peuvent déboucher sur des bénéfices très importants. À l'inverse, des jeux de test comme Check semblent offrir des perspectives intéressantes statiquement et se révèlent en définitive une illustration assez médiocre de l'intérêt de l'analyse.

Ceci met en valeur la démarche suivie dans cette étude : définir conjointement l'analyse statique à effectuer sur le code et adapter un support d'exécution afin de pouvoir mesurer

l'impact réel des résultats obtenus.

Les résultats correspondant au code de l'API utilisée (celle de JITS) sont assez faibles : ceci peut être expliqué en grande partie par le fait que l'API est majoritairement composée de code optimisé qui ne se marrie pas bien avec l'analyse d'échappement (notamment, le code d'API évite au maximum l'utilisation massive d'objets temporaires lorsque cela est possible, ce qui élimine la cible de prédilection de l'analyse).

5.3 Conclusion

Nous avons mis en évidence dans ce chapitre la possibilité de tirer parti d'une analyse d'échappement dans des environnements très contraints. Cette analyse d'échappement constitue un facteur d'optimisation non négligeable, en contribuant à rendre l'exécution des programmes plus légère, à travers une gestion moins coûteuse de la mémoire. Ceci est bien sûr d'autant plus important que les systèmes de petite taille sont très sensibles aux coûts supplémentaires (tels que ceux des ramasse-miettes). Enfin, la mise en place de la structure nécessaire à l'exploitation des résultats de l'analyse est elle-même d'un coût raisonnable puisqu'elle se base essentiellement sur des mécanismes de vérification de types pour lesquels un support existe, qui peut être adapté simplement.

Tests		S.A. = Allocations en pile, T.A. = Allocations totales										
Source	Nom	Statique					Dynamique (tests)			Dynamique (API)		
		[BSF04]	[WR99]	S.A.	T.A.	%	S.A.	T.A.	%	S.A.	T.A.	%
	Dhrystone			9	13	69%	20004	20011	100%	247	3902	6%
J. G.	euler	26%	28%	20	47	43%	6427102	6531342	98%	2590	34027	8%
	fft	63%	62%	12	14	86%	4	6	67%	184	3115	6%
	heapsort	40%	60%	4	6	67%	2	4	50%	140	1664	8%
	lufact	38%	37%	6	11	55%	2	9	22%	171	4067	4%
	raytracer	11%	19%	33	56	59%	426746	427224	100%	863	11408	8%
	series	80%	80%	10	12	83%	5	9	56%	144	1951	7%
	crypt	27%	36%	9	16	56%	30	37	81%	162	4277	4%
	moldyn	29%	28%	3	8	38%	3	10	30%	2689	44213	6%
	search	42%	44%	15	28	54%	7321078	7321091	100%	306	7694	4%
	sor	40%	40%	4	6	67%	3	5	60%	139	1752	7%
	sparsematmult	25%	25%	4	12	33%	3	11	27%	137	1954	7%
spec	check			103	122	84%	21	275	8%	1098	16538	7%
	compress			13	28	46%	20	73	27%	557	8624	6%
	raytrace	10%	42%	58	134	43%	5533239	6277696	88%	29092	133816	22%

TAB. 5.1 – Résultats obtenus sur les jeux de test.

Chapitre 6

Extensions de l'analyse d'échappement

Nous abordons dans ce chapitre la mise en place de mécanismes permettant l'extension de l'analyse d'échappement afin d'être en meilleure adéquation avec les méthodologies objet appliquées dans le domaine du génie logiciel et dont nous voudrions faire bénéficier les systèmes contraints. Ce chapitre revient en détail sur les motivations et les résultats publiés dans [GHSR07].

6.1 Analyse d'échappement et agrégation

L'analyse d'échappement présentée en Section 5.2 permet de détecter les objets dont la durée de vie est bornée par la fin de l'exécution de la méthode dans laquelle ils ont été créés. Ces objets trouvent naturellement leur place dans un modèle de gestion de mémoire différent du ramasse-miettes, et peu coûteux s'il est combiné aux mécanismes usuels de conventions d'appel. Cette technique donne de bons résultats à l'exécution, notamment du fait que les langages concernés incitent fortement à la création de structures de travail temporaires sans trop se soucier de la mémoire utilisée. Néanmoins, dans certaines conditions, malheureusement fréquentes, cette analyse peut s'avérer décevante en laissant de côté des cas qui, intuitivement, pourraient sans doute être optimisés de la même façon. L'exemple le plus immédiat est celui de l'agrégation d'objets (ou encapsulation), dans lequel des objets de travail sont alloués directement par un constructeur, afin de répondre à un besoin interne à la classe utilisée.

6.1.1 Allocations dans les constructeurs

Les exemples d'utilisation de l'agrégation sont bien sûr extrêmement nombreux dans l'API Java standard. Dans l'exemple de la figure 5.2, nous avons pu constater que l'analyse d'échappement est en mesure de détecter l'objet `StringBuffer` alloué pour réaliser

la concaténation souhaitée dans la méthode m. Or, si l'on regarde de quoi est fait un objet StringBuffer (*cf.* figure 6.1), il est certainement regrettable que la structure interne contenant les caractères effectifs ne soit pas prise en compte et ne puisse être considérée autrement que comme échappée.

```
public final class StringBuffer implements
Serializable, CharSequence {
  private static final long serialVersionUID = ...
  int count;
  char[] value;
  boolean shared;
  private static final int DEFAULT_CAPACITY = 16;
  public StringBuffer()
  {
    this(DEFAULT_CAPACITY);
  }
  public StringBuffer(int capacity)
  {
    value = new char[capacity];
  }
  ...
}
```

FIG. 6.1 – Exemple d'API : StringBuffer.

La figure nous montre clairement que l'un des champ de la classe StringBuffer est un tableau de caractères, alloué par le constructeur. Ce tableau étant lié (et pour cause) à l'objet this dans la méthode constructeur, il échappe à l'analyse, suivant la définition 22 donnée.

Cependant, il semble naturel d'envisager que si l'objet StringBuffer est capturé, et donc alloué en pile, il y a de fortes chances que le tableau de caractères n'ait plus grande utilité une fois le StringBuffer détruit. En tout état de cause, la raison d'un échappement éventuel de ce tableau ne saurait être directement le lien existant avec le StringBuffer, attendu que celui-ci va disparaître de façon certaine. Pour résumer, il est possible que les sous-objets alloués par un objet lui-même destiné à être alloué en pile puissent suivre le même chemin, si il n'existe pas d'autre moyen de les atteindre que de passer par ledit objet.

Plus généralement, l'API Java fournit un grand nombre de classes « conteneurs », à savoir des classes dont le rôle est d'agréger un certain nombre (potentiellement indéterminé) d'objets en fournissant des points d'accès limités pour raison de transparence vis à vis de l'utilisateur. Dans la liste de ces conteneurs on retrouve notamment toutes les classes de type Collection, mais également les flux d'entrée/sortie Stream ou encore les abstractions du système de fichiers. Typiquement, les structures internes de ces classes (tableaux

d'objets, tampons de données...) n'ont aucune raison d'être exposés à l'utilisateur, qui ne s'intéresse qu'à un petit nombre de points d'entrée génériques. Il est donc très vraisemblable que ces structures disparaissent en même temps que l'objet hôte. Une fois encore, il semble rentable de se pencher sur ces sous-objets lorsqu'un objet de la classe hôte est alloué en pile.

```
String foo(int arg1, char arg2) {
  StringBuffer sb = new StringBuffer();
  sb.append(arg1).append(arg2);
  return "res" + sb;
}
```

FIG. 6.2 – Exemple d'API : utilisation de StringBuffer.

Dans l'exemple présenté en figure 6.2, l'objet de classe `StringBuffer sb` doit être alloué en pile, et rien ne devrait s'opposer à ce que le tableau de `char` sous-jacent l'y suive, car il disparaîtra en même temps. D'un autre côté, le `StringBuffer` caché (qui se trouve dans la concaténation finale) doit également être alloué en pile, mais le tableau de `char` correspondant, lui, échappe dans le cas où l'API implante une stratégie « create-on-write » (comme présentée en section 5.2.3) pour la méthode `toString()`. On voit donc que le devenir des sous-objets dépend énormément de l'utilisation qui en est faite, et leur capacité à être alloués en pile est conditionnée par l'existence (et l'usage) de méthodes leur permettant de « sortir » de l'objet hôte.

Les figures 6.3 et 6.4 représentent les situations typiques dans lesquelles un objet peut ou ne peut pas suivre son conteneur en pile. La transformation n'est valide que si le sous-objet devient inaccessible lors de la disparition de son conteneur. Dans un tel cas, il est visible que le passage d'un ramasse-miettes à la destruction du conteneur provoquerait de toute façon la mort du sous-objet, ce qui garantit que la transformation est fonctionnellement neutre. Dans le cas de la figure 6.4, allouer l'objet `B` en pile reviendrait à prendre le risque d'invalider la référence possédée par `C` lors de la destruction des objets en sommet de pile, ce qui ne peut être autorisé.

Nous nous proposons d'étendre l'algorithme d'analyse d'échappement ainsi que le support d'exécution qui en tire partie, afin de nous permettre de déterminer à l'exécution (et pour un coût quasiment nul) si la situation est propice à l'allocation en pile des sous-objets.

6.1.2 Solution

L'objectif premier est d'étendre le mécanisme de base de l'analyse d'échappement afin de détecter les allocations (dans les constructeurs) qui sont susceptibles d'être détournées vers la pile si les conditions s'y prêtent. Par ailleurs, il sera nécessaire de proposer un calcul permettant d'évaluer les conditions en question (à l'exécution). La combinaison de

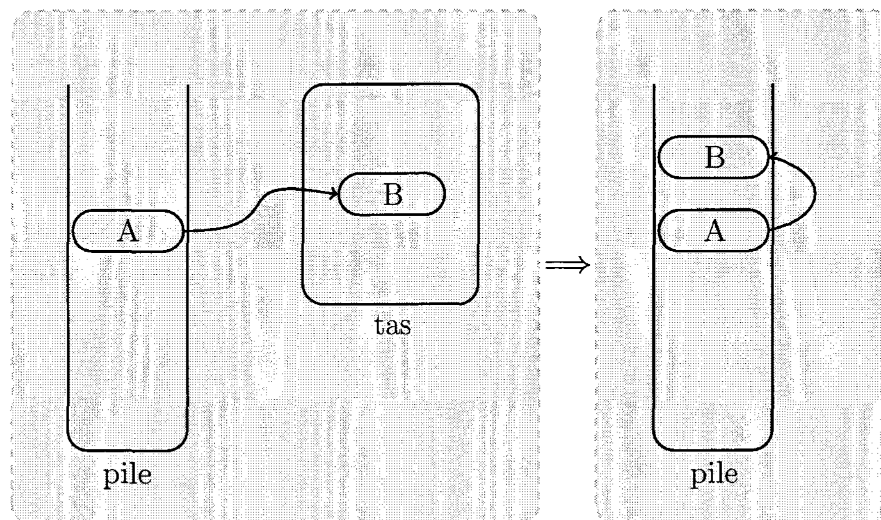


FIG. 6.3 – Possibilité d'allocation en pile

ces deux ajouts permettra donc d'allouer en pile des objets qui échappent à leur méthode de création mais dont on peut néanmoins borner l'existence lors de leur création sous réserve de disposer du contexte d'exécution courant.

Objectif 1. *Les points cruciaux sont les suivants :*

1. *L'analyse d'un constructeur permet de déterminer si les objets créés dans ce constructeur et conservés dans un ou plusieurs champ pourraient être alloués en pile dans l'hypothèse où l'objet conteneur serait lui-même alloué en pile. Ceci revient à dire qu'un certain nombre d'allocations sont statiquement détectables comme pouvant être déportées en pile sous conditions.*

2. *La condition d'allocation est déterminée par la méthode appelante (celle qui crée l'objet), de sorte qu'il est nécessaire à l'exécution d'avoir un transfert d'information de l'appelant vers l'appelé.*

Remarque : étant donné qu'il est nécessaire de s'assurer qu'aucune autre méthode ne peut faire échapper l'un des objets considérés, l'analyse doit prendre en compte des informations en provenance de plusieurs méthodes, dont certaines ne sont pas appelées directement dans la branche du graphe d'appel contenant l'allocation.

Ainsi, dans l'exemple de code présenté en figure 6.5 la construction de l'objet `Container` ne provoque pas l'échappement du champ `field`, mais une utilisation ultérieure de l'accesseur `getField()` peut remettre en question ce résultat.

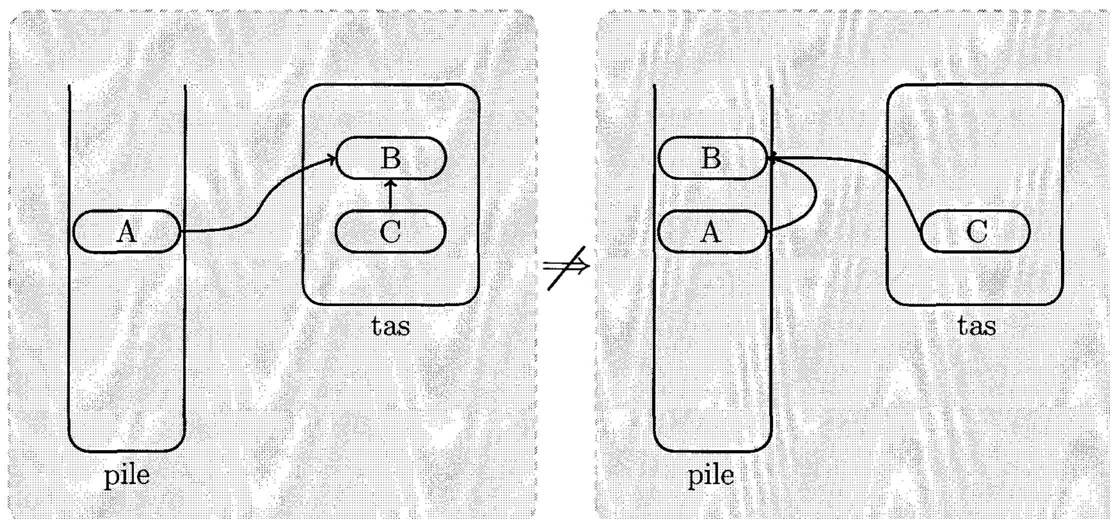


FIG. 6.4 – Impossibilité d'allocation en pile

Objets membres

Le premier objectif à atteindre est donc l'identification des allocations dans des constructeurs, qui ont une chance de pouvoir être détournées vers la pile. Pour cela nous introduisons une petite extension de la notion d'échappement, et considérons les objets n'échappant pas, dans le constructeur, à l'objet qui a demandé leur création. C'est à dire qu'il existe bien un lien persistant vers ces objets, qui survit à l'appel du constructeur, mais ce lien ne peut exister qu'avec l'objet construit (ou éventuellement avec un autre sous-objet n'échappant pas non plus au constructeur).

Règle 2. *Dans un constructeur, tout objet nouvellement créé qui est lié à l'objet initialisé par ce constructeur (`this`), et qui n'est lié qu'à des objets se trouvant dans la même situation que lui, peut être alloué en pile si l'objet initialisé s'y trouve.*

Un objet membre échappant par définition au constructeur dans lequel il est créé, les objets concernés par la règle 2 sont totalement disjoints des objets capturés par analyse d'échappement standard, de sorte que ces derniers sont toujours alloués en pile suivant les règles exposées en section 5.2.2.

Une fois encore, une simple lecture des résultats de l'analyse d'alias (ainsi que de l'analyse d'échappement) permet d'identifier les objets respectant la règle 2 et de les marquer comme candidats pour une allocation en pile conditionnelle. En effet, il s'agit simplement de s'assurer que le seul objet persistant permettant d'atteindre les objets en question n'est autre que l'objet initialisé par l'appel au constructeur. Le marquage de ces objets permet d'atteindre l'objectif 1.

Définition 23. *Soit m un constructeur. L'ensemble des sites d'allocations pour lesquelles*

```
public class Container {
  private Object field;

  public Container() {
    field = new Object();
  }

  public Object getField() {
    return field;
  }
}

public static Object keeper;

public static void main(String[] args) {
  Container c = new Container();
  ...
  keeper = c.getField();
  ...
}
```

FIG. 6.5 – Exemple d'un conteneur.

l'allocation en pile sous conditions est envisageable est défini par : $\mathcal{A}_m = \{o \mid o \in (Alloc_m \setminus \mathcal{C}_m), \forall(o', o) \in Link^r_m : o' \in \{this\} \cup Alloc_m \cup Func_m\}$.

La seconde partie de l'objectif 1 semble à première vue quelque peu contradictoire avec notre volonté de conserver une analyse statique, puisqu'elle sous-entend qu'une information venant de la méthode appelante, et disponible uniquement à l'exécution, est nécessaire pour l'exploiter. Néanmoins, la détection des objets concernés est purement statique, et il est bon de noter que tous les objets marqués pourraient être alloués en pile si les conditions étaient favorables (ce qui ne veut pas dire que cela arrivera).

Par ailleurs, l'information nécessaire pour évaluer la condition déterminant le placement en pile ou non des objets marqués est extrêmement simple, puisqu'il s'agit d'une information booléenne caractérisant le fait que le constructeur en cours d'exécution est appliqué sur un objet alloué en pile ou non.

Règle 3. *Quand un objet est marqué comme capturé par l'analyse d'échappement, son constructeur doit être appelé de telle sorte que l'information d'allocation en pile soit accessible dans la nouvelle fenêtre d'exécution. Ceci permet de déporter les allocations conditionnelles en pile par lecture de cette information.*

L'implantation de ce mécanisme est assez trivial, notamment en ce qui concerne la méthode de communication de l'appelant vers l'appelé. L'analyse d'échappement fournissant des références vers les allocations qui peuvent être effectuées en pile, il suffit de rattacher les invocations de constructeurs aux allocations correspondantes (ce qui est simple pendant la phase d'analyse), et de marquer les appels de constructeurs qui servent uniquement à initialiser des objets capturés par analyse d'échappement. Ensuite, de même que les bytecodes d'allocation sont modifiés pour s'exécuter ailleurs, il est tout à fait envisageable de modifier les bytecodes d'invocation (`invokespecial`) pour incorporer l'information selon laquelle l'objet construit a été mis en pile. Ces invocations sont donc transformées afin de déclencher la mise en pile des objets membres qui peuvent l'être.

Propriété 1. *Soit m un constructeur. Les bytecodes* `invokespecial` *utilisés uniquement pour construire des objets appartenant à C_m doivent être marqués comme déclenchant les allocations en pile conditionnelles.*

Cascade de constructeurs

Cette extension de l'analyse n'est cependant pas encore satisfaisante : en effet il n'est pas suffisant de mettre en pile les objets rattachés uniquement à des objets en pile, il faut également considérer l'exécution de leurs propres constructeurs. Dans l'exemple exposé en figure 6.6, l'analyse donne pour le moment les résultats suivants :

- le champ `s` de `C` n'échappe pas à `this` lors de l'exécution du constructeur ;
- l'allocation de `s` (offset 5) est marquée comme allocation en pile conditionnelle ;
- il n'existe en fait aucune possibilité pour `s` d'être référencé par un autre objet ;
- si un objet de type `C` est alloué en pile, alors `s` sera également alloué en pile.

En revanche, il est, en l'état actuel des choses, impossible de décider comment doit être géré l'`invokespecial` de l'offset 9. En effet, on ne peut décider statiquement que cette invocation doit elle aussi permettre de déclencher des allocations en pile, car cela dépend du fait que l'objet de classe `C` soit, ou non, alloué en pile lui-même.

Le même problème se produit en fait, de façon moins visible, avec l'`invokespecial` de l'offset 1. Ceci montre la nécessité de pouvoir « transmettre » l'information de mise en pile de l'objet de base, afin de pouvoir gérer une profondeur d'appel quelconque. Il est donc besoin d'un nouveau vecteur de propagation de l'information, et d'une règle associée :

Règle 4. *L'information déterminant la possibilité d'allouer en pile les sous-objets doit être transmise aux constructeurs de ces sous-objets ainsi qu'au constructeur de la super-classe.*

L'implantation de la règle 4 implique que l'analyse marque comme devant propager l'information de mise en pile les bytecodes `invokespecial` qui correspondent aux sites d'allocations détectés comme pouvant aller conditionnellement en pile, ainsi que les appels aux constructeurs de super-classe. Il est donc nécessaire de connaître les associations entre sites d'allocations et constructeurs, mais cette information est déjà disponible puisqu'elle est également nécessaire à l'implantation de la règle 3. Le seul ajout est donc le marquage

```
class C {
  private StringBuffer s;
  public C (){ s = new StringBuffer(); }
}
--- Bytecode of class C
class C extends java.lang.Object{
  public C();
  Code:
  0:   aload_0
  1:   invokespecial    #1; //Object();
  4:   aload_0
  5:   new              #2; //class StringBuffer
  8:   dup
  9:   invokespecial    #3; //StringBuffer();
  12:  putfield         #4; //StringBuffer s;
  15:  return
}
```

FIG. 6.6 – Constructeurs imbriqués.

d'un nouveau type de bytecodes d'invocation, capables de transmettre l'information sans la modifier.

Propriété 2. *Soit m un constructeur. Pour tout $o \in \mathcal{A}_m$, les constructeurs de o (depuis la classe de o jusqu'au sommet de la hiérarchie) doivent être marqués comme propageant l'information de mise en pile.*

Justification

Une zone d'ombre demeure néanmoins sur la méthode employée pour déterminer qu'un sous-objet marqué comme pouvant être mis en pile sous condition ne puisse en aucun cas échapper par ailleurs. Ceci correspond à la remarque faite brièvement lors de l'exposition de l'objectif 1. On a vu notamment que certaines stratégies telles que le « create-on-write » peuvent être amenées à exporter des objets relevant de la mécanique interne d'une classe. La détection statique des situations favorables à la mise en pile des sous-objets est donc un problème crucial.

Une première idée pourrait être de considérer l'ensemble des méthodes disponibles dans la classe et de s'assurer qu'aucune n'est en mesure de « faire sortir » un sous-objet donné. Ceci est tout à fait faisable, et a constitué (sans le mettre particulièrement en évidence) un des arguments lors de l'analyse de l'exemple fourni en figure 6.6. Fort logiquement, si il n'existe statiquement aucun moyen de faire sortir une information, il en sera de même à

l'exécution.

Cependant, cette solution n'est pas vraiment satisfaisante, car bien évidemment le fait de disposer de méthodes provoquant l'échappement d'un objet ne signifie pas qu'elles soient utilisées dans les cas qui nous concernent. Il serait donc avantageux de disposer d'une vue suffisamment précise de l'utilisation d'un objet pour pouvoir tirer le meilleur parti des annotations ajoutées.

Or, dans ce cas précis l'information est en fait disponible à l'exécution, et de façon immédiate : dans les cas qui nous intéressent, l'objet qui construit les sous-objets a été alloué en pile. Or, de par l'approximation faite sur les champs dans l'analyse d'alias, un champ survivant à la méthode qui crée l'objet provoque la survie (dans l'analyse) de l'objet lui-même. Par contraposée, le simple fait que l'objet soit détecté comme capturé implique que tous ses champs ont une durée de vie n'excédant pas la sienne, et donc que l'on est bien dans la situation où un objet est créé (en pile) et où ses champs peuvent l'y suivre. Ainsi il n'y a en fait pas besoin d'une analyse supplémentaire de la classe, ni même du contexte d'appel, pour savoir avec certitude que les champs marqués peuvent être alloués en pile. L'approximation, alors un peu large, présentée en section 4.4.5 permet en fait de bénéficier sans coût additionnel d'une analyse allant plus loin qu'une analyse d'échappement classique.

6.1.3 Adaptation du support d'exécution

Cette extension de l'analyse d'échappement est encore une fois implantée comme une interprétation relativement simple des résultats fournis par l'analyse d'alias. Les informations obtenues sont toujours purement statiques, mais leur utilisation demande un peu d'information en provenance de la plate-forme d'exécution, au contraire de l'analyse d'échappement pure.

Comment empiler

À l'issue de l'analyse du code, une nouvelle annotation est ajoutée dans les fichiers `.class`, qui rend les résultats de l'analyse accessibles au chargeur de code pour vérification. Une fois ces résultats validés, ils auront pour conséquence l'adaptation d'un certain nombre d'instructions par adjonction d'un comportement adapté en ce qui concerne l'allocation en pile.

Par ailleurs, le moteur d'exécution lui-même a dû subir une légère modification, afin que la propagation de la condition d'allocation soit possible. Dans ce but, c'est la structure de fenêtre d'exécution qui a été modifiée, pour embarquer une information booléenne déterminant le comportement des bytecodes spécialisés exécutés dans cette fenêtre. La valeur s'interprète comme suit :

- `vrai` quand les allocations marquées comme possibles en pile sous condition peuvent être effectivement déportées.

– faux dans les autres cas.

Pour poursuivre dans la même voie que pour l'analyse d'échappement (*cf* section 5.2.4), on peut voir l'exploitation des résultats de l'extension de l'analyse d'échappement présentée comme la création de quelques bytecodes spécifiques :

- newif, newarrayif, anewarrayif, et multianewarrayif remplacent respectivement new, newarray, anewarray, et multianewarray lorsque ces derniers sont détectés comme des sites d'allocation en pile sous condition (à savoir que les allocations seront effectuées en pile si et seulement si la donnée ajoutée à la fenêtre d'exécution courante est vrai. Dans le cas contraire, l'allocation est faite suivant la méthode par défaut).
- invokespecialTrue remplace invokespecial quand l'invocation est marquée comme associée à une (ou plusieurs) allocation en pile. La sémantique est la même que pour le bytecode invokespecial, si ce n'est que la valeur booléenne de la fenêtre d'exécution nouvellement créée est initialisée à vrai, alors que le comportement inverse est associé à invokespecial.
- invokespecialCopy remplace invokespecial quand l'invocation est marquée comme devant transmettre l'information d'autorisation de mise en pile telle qu'elle est dans la fenêtre d'exécution courante. L'effet est donc de recopier la valeur booléenne de la fenêtre courante dans la fenêtre nouvellement créée.

Notons encore une fois que l'expression sous forme de bytecodes ici présentées n'est qu'une façon simple de représenter l'existence de comportements différents de ceux des bytecodes standards. Dans la mesure où ces « bytecodes » n'ont aucune existence en dehors de la machine virtuelle, il est parfaitement légitime de les concrétiser directement (en code machine) sans passer par une transformation de bytecode préalable.

Comment dépiler

Pour compléter le descriptif de l'évolution de la pile d'allocation, il est nécessaire de revenir sur la manière de dépiler. Cette opération était très simple pour l'analyse d'échappement classique, puisqu'elle était systématiquement entreprise à chaque retour de fonction, chaque objet en pile étant lié à la fenêtre d'exécution courante.

Avec l'extension présentée dans cette section, les choses sont un peu moins simples. En effet, les objets qui se trouvent sur la pile à l'issue de cette analyse étendue ne sont *a priori* pas destinés à disparaître avec la méthode qui les a créés. Au contraire, nous avons tout fait pour « récupérer » des objets alloués lors d'appels imbriqués, qui ont une durée de vie bornée certes, mais pas par l'exécution de la méthode qui les a créés.

Néanmoins, c'est toujours au point de retour que l'état de la pile évolue. Il y a en fait deux cas possibles lorsque l'exécution en arrive à un retour de fonction :

- soit l'étage actuel de la pile d'allocation ne contient que des objets capturés par l'analyse d'échappement pure et les sous-objets qui les ont suivis, auquel cas il est légitime de simplement tout éliminer ;
- soit il contient des sous-objets alloués en pile pour suivre un conteneur mais pas le conteneur lui-même, auquel cas ils doivent survivre au retour de fonction jusqu'à se

retrouver au même niveau que le conteneur. Il est donc nécessaire de fusionner l'étage courant avec le précédant.

Dans les deux cas, la hauteur de pile diminue de 1, ce qui nous laisse une pile d'allocation de hauteur invariablement identique à celle de la pile d'exécution.

Pour effectuer cette opération, il est également nécessaire de conserver une information relative à la fenêtre d'exécution considérée (et qui ne sera utilisée que pour la détruire) : cette information consiste en fait en l'occurrence ou non d'une allocation en pile sous condition vérifiée lors de l'exécution de la méthode correspondante. Si toutes les allocations faites en pile ne concernaient que des objets capturés par analyse d'échappement pure, alors le premier cas est vérifié et l'étage entier de la pile peut être éliminé.

Cette opération, quelque peu complexe, est rendue possible par le choix d'une pile séparée pour les allocations. En effet, dans l'hypothèse d'une pile unique pour l'exécution et les allocations, il serait délicat de recourir à un mécanisme similaire car il aurait pour conséquence de « bloquer » les variables locales de toutes les fonctions appelées au-delà des limites raisonnables.

Remarquons au passage que cette opération est sujette à optimisations. En effet, même dans le cas où une fusion d'étages de pile est nécessaire, il est possible de supprimer sans coût supplémentaire tous les objets capturés par analyse d'échappement pure qui se trouvent en sommet de pile (en supprimer d'autres demanderait une réorganisation de la pile et donc une opération de maintenance sur les références déplacées).

Par conséquent, l'ordre dans lequel sont effectuées les allocations est loin d'être neutre : un ordre « intelligent » diminuera typiquement la hauteur de la pile d'allocation, et autorisera donc l'exécution effective de davantage de programmes. Bien sûr l'existence de boucles n'autorise pas la suppression systématique de toutes les références possibles au plus tôt, mais au sein de code linéaire, réordonner les allocations (mais pas les constructions) serait très vraisemblablement une optimisation intéressante.

6.1.4 Exemple détaillé

Un exemple simple, mais représentatif des cas devant entrer dans le cadre de l'analyse détaillée précédemment est présenté en figure 6.7. Il s'agit d'une implantation vraiment minimaliste d'une structure `LIFO`, soit une pile. L'implantation choisie fait intervenir un `Vector` en tant que structure interne pour stocker les éléments de la pile.

Il y a exactement trois sites d'allocation présents dans ce code, et par ailleurs on constate simplement que chacun d'eux est exécuté exactement une fois au cours du déroulement du programme (si la méthode `main` est invoquée).

L'objet de type `LIFO` peut de façon triviale être alloué en pile, puisque n'échappant pas à la méthode `main` dans laquelle il est créé. Ceci implique que l'objet de type `Integer` peut également être alloué en pile, puisque n'étant attaché qu'à un objet lui-même alloué en pile (*cf.* section 5.1). Ceci est déduit aisément de la signature d'alias calculée pour la méthode `push()`, qui affiche ce seul lien entre les objets.

```
class LIFO {
  private Vector v = new Vector();

  public void push(Object o) {
    v.add(o);
  }

  public Object pop() {
    return v.remove(v.size() - 1);
  }

  public static void main(String[] args) {
    LIFO p = new LIFO();
    p.push(new Integer(5));
  }
}
```

FIG. 6.7 – Exemple d'agrégation.

Enfin, en ce qui concerne l'objet de type `Vector`, il entre parfaitement dans le cadre de l'extension d'analyse d'échappement présentée précédemment, en tant qu'objet strictement à usage interne. De ce fait, son seul lien avec l'objet `LIFO` fait de lui un candidat à l'allocation en pile dès lors que l'objet `LIFO` est alloué en pile, ce qui est le cas dans la situation qui nous occupe. Au final, dans cet exemple simple, toutes les allocations ayant lieu à l'exécution peuvent être placées en pile, alors qu'une analyse d'échappement pure n'aurait pas pu aboutir à un autre résultat que l'échappement de l'objet `Vector`.

Notons néanmoins que si l'objet `LIFO` avait été alloué dans un espace statique, le changement au niveau du contexte d'appel de push aurait suffi à empêcher le `Vector` de se retrouver en pile, ce qui met bien en lumière l'importance de l'information supplémentaire nécessaire accessible uniquement à l'exécution.

6.1.5 Apports de l'analyse

L'extension présentée dans cette section permet donc d'exprimer une propriété qui participe des mêmes préoccupations que l'analyse d'échappement, mais qui réclame une légère évaluation de la situation à l'exécution. Il est important de réaliser que toute l'analyse et tous les comportements possibles sont déterminés statiquement, mais que seul l'enchaînement de ces comportements (qui influe sur le comportement global) est indéterminé avant exécution.

Cet aspect légèrement dynamique permet en fait de retrouver une partie de ce qui a

été perdu lors du renoncement à un monde fermé : la gestion fine des allocations entre des méthodes différentes.

6.2 Analyse d'échappement et fabriques d'objets

Dans la lignée des objets construits par les cascades de constructeurs, il est naturel de s'intéresser à une autre famille d'objets, dont le procédé de création est très proche, si ce n'est au niveau de la méthode, du moins au niveau de l'intention : ceux qui sont construits par le biais d'une fabrique[1]. En effet, ce schéma de construction classique en génie logiciel, et qui présente bien des avantages d'un point de vue objet, souffre du même « problème » que le schéma de « conteneur » traité précédemment en laissant très naturellement échapper systématiquement tous les objets qu'il crée, quelles que soient les circonstances d'utilisation. Encore une fois, l'intérêt porté à la seule méthode de création d'un objet apparaît comme un facteur limitant, lorsqu'un problème est découpé suffisamment finement pour que l'effet d'une méthode soit uniquement une création d'objet.

6.2.1 Utilisation de fabriques

La première étape vers la gestion du problème des fabriques est bien évidemment de détecter celles-ci automatiquement. Une première approche, triviale, pourrait être de présupposer le respect d'une convention concernant les noms de ces structures. Mais cette méthode manque clairement d'élégance, en se reposant sur la bonne volonté du programmeur plutôt que sur une pure inférence, ce qui est un problème majeur dans un environnement *a priori* ouvert, et pouvant donc recevoir du code de provenance arbitraire.

Au lieu de cela, nous proposons de poser une caractérisation simple de ce qui sera dans la suite appelé fabrique, même si de fait elle englobe davantage que ce qui se qualifierait comme fabrique d'un point de vue de pur génie logiciel.

Définition 24 (Fabriques). *Est considérée comme « fabrique » toute méthode retournant nécessairement un objet qui ne peut être atteint que par le biais de cette valeur de retour.*

Les objets en question sont appelés « neufs »[2] pour respecter la terminologie habituelle [GS00].

L'effet d'un appel à une fabrique est très similaire à ce que l'on obtient de l'exécution d'un bytecode « new » : un nouvel objet est mis à disposition. À titre d'exemple, considérons le cas présenté en figure 6.8.

Des objets de classe `C1` peuvent être créés par l'intermédiaire de la classe `Factory_C1`, et plus précisément par l'appel à la méthode `create_C1()`. C'est ce qui se produit dans la méthode `m()` de la classe `Test`. De façon tout à fait intuitive et informelle, il est clair au

[1] « factory » dans la littérature anglaise.

[2] ou « fresh ».

```
class C1 {
  private Object o;
  C1 (){ o = new Object(); }
}

class Factory_C1{
  private  static Factory_C1 f;

  private Factory_C1 (){
    f = this ;
  }

  public static Factory_C1 getInstance (){
    if (f == null) return new Factory_C1();
    return f;
  }

  public C1 create_C1 (){
    return new C1();
  }
}

class Test{
  void m () {
    Factory_C1 f = Factory_C1.getInstance();
    C1 o = f.create_C1();
  }
}
```

FIG. 6.8 – Exemple d'utilisation de fabrique.

vu du code que l'objet ainsi créé ne peut en aucun cas survivre à la méthode `m()`, et que si sa création était due à un bytecode `new` là où se trouve l'appel à la fabrique, une analyse d'échappement détecterait sans la moindre ambiguïté que l'objet peut être alloué en pile.

Malheureusement, dans l'état actuel de l'analyse proposée, il n'existe aucune possibilité d'aboutir à la même conclusion par le raisonnement systématique présenté. Tel est le problème que nous nous proposons de résoudre.

6.2.2 Solution

Il convient de remarquer au préalable que la possibilité de mise en pile de tel ou tel objet construit par une fabrique dépendra encore une fois nécessairement du contexte d'utilisation de l'objet. Comme nous l'avons fait remarquer, un appel à une fabrique ne diffère guère de l'exécution d'un bytecode `new`, aussi prétendre mettre en pile tout objet provenant d'une fabrique quelconque reviendrait à prétendre contrôler la capacité d'échappement de tout objet créé par un `new`, ce qui est absurde.

Ainsi donc, de même que dans le cas du conteneur, il est nécessaire de procéder en deux étapes, en sélectionnant tout d'abord un certain nombre d'objets potentiellement éligibles à l'allocation en pile, puis en calculant les conditions sous lesquelles ils pourront être effectivement déportés en pile.

Objectif 2. *1. Il est nécessaire de détecter automatiquement les fabriques (selon la définition 24).*

2. Les objets alloués dans les fabriques pourront être alloués conditionnellement en pile si l'appelant ne les laisse pas échapper.

3. La méthode appelante doit détecter les appels aux fabriques et déterminer si les objets créés doivent finalement aller en pile ou non.

La définition 24, donnée pour les fabriques, nous permet relativement simplement de détecter les méthodes qui seront considérées comme telles, par simple lecture des résultats de l'analyse d'alias, qui reste centrale dans ces travaux. Il n'est donc aucun besoin de code additionnel pour gérer cette détection.

Règle 5. *Toute méthode retournant un objet qui échappe (au sens donné par la définition 22) par la seule valeur de retour est considérée comme une fabrique.*

Cette définition est relativement imprécise dans la mesure où elle peut tout à fait inclure des méthodes effectuant de nombreux calculs (ce qui n'est *a priori* pas le rôle d'une fabrique telle que conçue par la communauté du génie logiciel), ou encore une méthode qui n'effectue qu'un appel délégué à une fabrique, comme présenté dans la figure 6.9 : la classe `Test2` n'est donc pas elle-même une vraie fabrique, bien que l'objet retourné satisfasse les bonnes conditions.

Néanmoins, il n'est absolument pas erroné de chercher à allouer les objets ainsi créés en pile, et il est au contraire plutôt positif de répondre à un problème en traitant de façon correcte un sur-ensemble des éléments souhaités. Comme dit précédemment, ce sur-ensemble présente par ailleurs l'avantage d'être aisément identifiable par analyse d'alias : la discrimination se fait sur l'objet retourné, qui doit n'être lié à rien d'autre qu'à la valeur de retour. Ceci résout donc le premier objectif, à savoir l'identification automatique des méthodes concernées.

Définition 25. *Soit m une méthode retournant un objet. L'ensemble des sites d'allocation conditionnelle pour m est défini par :* $\mathcal{A}_m = \{o \mid o \in (Alloc_m \setminus \mathcal{C}_m), \forall(o', o) \in Link^r_m : o' \in \{Ret_m\} \cup Alloc_m \cup Func_m\}$.

```
class Test2 {
  C1 m () {
    Factory_C1 f = Factory_C1.getInstance();
    C1 o = f.create_C1();
    ... // some computation that does not
    ... // make o escape
    return o;
  }
}
```

FIG. 6.9 – Récupération d'objets en dehors des fabriques.

Le second objectif à atteindre, la détection statique des objets susceptibles d'être déportés en pile si les conditions s'y prêtent, met bien sûr en jeu le mécanisme d'allocation en pile conditionnelle, déjà présenté pour son intérêt dans la gestion des cascades de constructeurs et du schéma de « conteneur ». En définitive, la seule vraie différence entre les fabriques et les constructeurs réside dans la position de l'objet par rapport à la méthode appelée : là où un constructeur agit sur un objet spécifique « this », une fabrique retourne un objet qui ne faisait pas partie de son contexte d'appel.

Règle 6. *Dans une méthode (hors constructeur), tout nouvel objet qui se retrouve attaché uniquement à la valeur de retour peut être alloué à l'aide d'un opérateur d'allocation en pile conditionnelle.*

Notons que cette règle n'entre pas en conflit avec les règles précédemment établies pour l'analyse des constructeurs (par définition), non plus qu'avec les règles régissant l'analyse d'échappement régulière : en effet les objets considérés ici ne sauraient entrer dans le domaine des objets capturés puisque la valeur de retour à laquelle il est fait référence est une raison suffisante pour qu'ils échappent.

Toutefois, il s'agit d'un échappement contrôlé, qui ne se produit qu'à destination de la méthode appelant la fabrique. Ceci nous amène au dernier point, concernant l'analyse du contexte d'appel d'une fabrique afin d'en tirer les informations nécessaires pour déterminer la mise en pile finale (ou l'allocation « normale ») de l'objet construit. Il s'agit donc de fournir les structures qui permettront à la plate-forme d'exécution de choisir entre les deux comportements possibles de la mise en pile conditionnelle. Deux cas se présentent :

Déclenchement de la mise en pile

Règle 7. *Tout appel à une méthode retournant un objet n'échappant pas dans la méthode appelante*[3] *doit provoquer l'allocation en pile (dans la méthode appelée) de tous les objets*

[3]le calcul est le même que pour l'analyse d'échappement normale, si ce n'est que l'appel à la fabrique est considéré à l'égal d'un `new`.

soumis à l'allocation en pile conditionnelle.

Ceci s'exprime sous la forme de la propriété suivante :

Propriété 3. *L'ensemble des allocations par fabriques capturées dans une méthode* m *est défini par :* $\mathcal{CF}_m = \{f_N \in Func_m \mid \forall(o, f_N) \in Link^r_m, o \in Alloc_m \cup Func_m\}$*, et les invocations correspondantes (au* $N^{ième}$ *bytecode) doivent déclencher les allocations en pile conditionnelles.*

Propagation de la mise en pile

Règle 8. *Tout appel à une méthode retournant un objet se trouvant dans une des conditions suivantes dans la méthode appelante doit être effectué de sorte que la méthode appelée se comporte vis à vis des allocations en pile conditionnelles comme la méthode appelante (il y a donc transfert de l'information qui détermine le comportement).*
- *soit l'objet est attaché uniquement à des objets alloués eux-même conditionnellement en pile ;*
- *soit l'objet est attaché à un objet construit par un constructeur ;*
- *soit l'objet est attaché à la valeur de retour de l'appelant.*

Cette dernière règle est l'expression de la dernière propriété :

Propriété 4. *Soit* m *une méthode. Pour tout objet* $f_N \in Func_m \setminus \mathcal{CF}_m$ *tel que* $\forall(o, f_N) \in Link^r_m, o \in \{\alpha\} \cup \mathcal{A}_m \cup Alloc_m \cup Func_m$*, l'invocation correspondante de la fabrique au* N^e*bytecode est marquée comme* devant propager l'ordre d'allocation en pile. *Comme précédemment,* $\alpha =$ `this` *si* m *est une méthode* `<init>`*, et* Ret_m *sinon.*

L'implantation des procédés décrits par les règles 7 and 8 passe évidemment par la détection des invocations de fabriques. Ces invocations peuvent être marquées de deux façons différentes :
- si le cas correspond à la règle 7, l'invocation doit comporter l'information nécessaire à la mise en place effective de l'allocation en pile pour toutes les créations d'objets conditionnellement en pile dans la méthode appelée ;
- si le cas correspond à la règle 8, la méthode appelée propagera les instructions appliquées dans la méthode appelante en ce qui concerne les allocations en pile conditionnelles.

Bien évidemment, une fois encore les décisions sont prises relativement aux résultats produits par l'analyse d'alias. L'information pour chaque méthode est obtenue statiquement, seule la combinaison des informations doit être faite dynamiquement pour permettre la gestion de tous les enchaînements de méthodes possibles. En particulier, la méthode appelante donne l'instruction à la méthode appelée sans avoir d'*a priori* sur le fait que l'objet retourné est effectivement neuf ou pas. Cette instruction doit être comprise comme « mettre tout ce qui est possible en pile, car ce qui y sera n'échappera pas de mon fait ». Si la méthode appelée n'a rien à allouer en pile, alors celle-ci restera vide.

Remarquons également que la solution apportée au problème des fabriques est totalement indépendante et ne remet nullement en cause les méthodes exposées précédemment pour tenir compte des constructeurs ou simplement des objets échappés au sens classique du terme. Les constructeurs et les fabriques peuvent très bien être imbriqués, et interagir sans que l'analyse en souffre, et les objets capturés pourront toujours être alloués en pile. À titre d'exemple, considérons le code présenté en figure 6.10. L'analyse complète donne les résultats suivants pour le constructeur de la classe `Student` :

- le `StringBuffer` utilisé pour le calcul de `name = "name : "+ s ;` peut être alloué en pile ;
- le constructeur et la méthode `append` de `StringBuffer` sont appelés de telle sorte qu'ils doivent déclencher la mise en pile des allocations conditionnelles qui en sont dépendantes ;
- le `Vector` est alloué conditionnellement en pile ;
- les appels au constructeur de la superclasse de `Student`, au constructeur de `Vector`, à la méthode `toString` de `StringBuffer`, et à la méthode `create_Address` de `Factory_Address` sont effectués de sorte qu'ils transmettent l'instruction qu'ils ont reçue quant à la mise en pile des allocations conditionnelles.

Le devenir de tous ces objets dépend massivement de l'objet de classe `Student` qui les utilise. En effet, deux scénarios sont envisageables :

- si l'objet de classe `Student` échappe pour une raison quelconque, le `StringBuffer` sera néanmoins alloué en pile, et le tableau de caractères sous-jacent l'y suivra ;
- si l'objet de classe `Student` est capturé, alors les objets précédemment mentionnés se trouveront toujours en pile, mais en plus le `Vector` (et son tableau sous-jacent), l'objet `Address` et même l'objet `String` créé par la concaténation pourront également être alloués en pile.

6.2.3 Transformation du ficher `.class`

Les informations obtenues par l'analyse étendue aux fabriques nécessitent l'adjonction d'annotations spécifiques pour pouvoir être prises en compte par la plate-forme d'exécution. De nouveaux bytecodes ont été ajoutés pour refléter ces informations, qui demeurent néanmoins proches de celles nécessaires à la bonne gestion des constructeurs. En particulier, le bytecode d'allocation conditionnelle en pile est le même que celui précédemment introduit (ce qui implique que l'information booléenne utilisée pour déterminer le comportement à l'exécution est également la même). Plus globalement, toute la gestion de la fenêtre d'exécution vis à vis des allocations en pile demeure inchangée. En revanche, puisqu'on ne s'intéresse plus uniquement aux constructeurs, plusieurs variantes des bytecodes d'invocation doivent être introduites.

- `invokevirtualTrue`, `invokeinterfaceTrue`, et `invokestaticTrue` ont respectivement la même sémantique que `invokevirtual`, `invokeinterface`, et `invokestatic`, à l'exception du fait qu'ils positionnent la valeur booléenne dans la nouvelle fenêtre de sorte qu'elle déclenche les allocations conditionnelles (les bytecodes « normaux »

```
class Student {
  private String name;
  private Address a;
  private Vector book_list;
  public Student (String s, int num){
    name = "name: "+ s ;
    a = Factory_Address.getInstance().create_Address(num);
    book_list = new Vector();
  }
}

class Address {
  private int number;
  public Address (int n){ number = n;}
}

class Factory_Address{
  private  static Factory_Address f;
  private Factory_Address () {f = this ;}
  public static Factory_Address getInstance (){
    if (f == null) return new Factory_Address();
    return f;
  }
  public Address create_Address (int n){
    return new Address(n);
  }
}
```

FIG. 6.10 – Exemple.

ayant le comportement inverse) ;

- `invokevirtualFlag`, `invokeinterfaceFlag`, et `invokestaticFlag` ont respectivement la même sémantique que `invokevirtual`, `invokeinterface`, et `invokestatic`, si ce n'est qu'ils transfèrent la valeur booléenne à l'identique dans la nouvelle fenêtre d'exécution.

6.2.4 Résultats expérimentaux

Malheureusement les jeux de test utilisés traditionnellement pour l'évaluation de l'analyse d'échappement, et que nous avons nous-mêmes utilisés en section 5.2.5 ne s'avèrent pas utiles pour illustrer l'intérêt de prendre en compte les schémas de développement tels que

```
public class MazeGame {
  public Maze CreateMaze () {
    Maze aMaze = MakeMaze();

    Room r1 = MakeRoom(1);
    Room r2 = MakeRoom(2);
    Door theDoor = MakeDoor(r1, r2);

    aMaze.AddRoom(r1);
    aMaze.AddRoom(r2);

    r1.SetSide(Room.North, MakeWall());
    r1.SetSide(Room.East, theDoor);
    r1.SetSide(Room.South, MakeWall());
    r1.SetSide(Room.West, MakeWall());
    r2.SetSide(Room.North, MakeWall());
    r2.SetSide(Room.East, MakeWall());
    r2.SetSide(Room.South, MakeWall());
    r2.SetSide(Room.West, theDoor);

    return aMaze;
  }

  // factory methods:
  public Maze MakeMaze() { return new Maze(); }
  public Room MakeRoom(int n) { return new Room(n); }
  public Wall MakeWall() { return new Wall(); }
  public Door MakeDoor(Room r1, Room r2) {
    return new Door(r1, r2);
  }
}

public class Main {
  public static void main(String[] args) {
    MazeGame mg = new MazeGame();
    mg.CreateMaze();
  }
}
```

FIG. 6.11 – Labyrinthes.

les fabriques, puisque ces schémas n'y sont de fait pas utilisés. Cela se comprend par ailleurs aisément puisque par définition, l'utilisation de tels schémas rend l'analyse d'échappement inopérante.

Par conséquent, au lieu d'analyser en vain ces jeux de tests, il apparaît plus intéressant de voir ce que donne l'analyse dans un cas canonique d'utilisation des schémas visés. Le livre "Design Patterns" [GHJV95] présente le schéma de fabrique à l'aide d'un exemple simple écrit en C++, mais trivialement transposable en Java. Cet exemple sert à créer des labyrinthes à partir de plusieurs éléments de base (pièces, murs, portes...) créés à la demande par des fabriques. En particulier, la méthode `CreateMaze` de la classe principale (`MazeGame`, *cf.* figure 6.11) est le point de départ de la création d'un labyrinthe, et exécute plusieurs appels à des fabriques.

Une analyse d'échappement standard marquerait immanquablement tous les objets créés de la sorte comme échappés. Notre analyse, au contraire, leur laisse une chance de se laisser capturer si leur contexte de création s'y prête, et les marque donc comme pouvant aller en pile sous conditions. Dans l'exemple exposé ici, l'ensemble des constituants du labyrinthe se retrouve de fait en pile car l'objet de plus haut niveau est trivialement capturé. Le même exemple, étudié selon une analyse d'échappement inter-procédurale classique, détecterait un échappement pour chacun des objets créés par fabrique, et empêcherait donc toute allocation en pile.

Notons néanmoins que dans le cas général il ne saurait y avoir aucune garantie qu'un objet marqué comme pouvant aller en pile sous conditions voit effectivement son allocation déportée. Mais laisser une chance au moteur d'exécution de se rendre compte au dernier moment que l'objet est en fait capturable ne peut être qu'un bénéfice dans la mesure où cela ne saurait en aucun cas dégrader les résultats d'une analyse d'échappement.

6.3 Conclusion

Ceci clôt le volet de ces travaux dédié à l'extension de l'analyse d'échappement à une analyse statique permettant d'annoter un programme avec des informations qui ne peuvent se révéler utiles que très tardivement, à savoir durant l'exécution. Cette extension permet de donner une dernière chance au gestionnaire mémoire d'allouer en pile certains objets qui, suivant le contexte dans lequel ils sont créés, peuvent se révéler être des objets de travail au même titre que les objets capturés par une analyse d'échappement classique. Ce faisant, on peut espérer retrouver des performances proches de ce qui est obtenu en monde fermé, alors même qu'aucune présupposition de ce genre n'est formulée, et que le chargement dynamique de classe est *a priori* autorisé.

Chapitre 7

Incursion dans le monde fonctionnel

Dans les chapitres précédents, nous avons vu comment les systèmes de types peuvent être conçus et utilisés de manière à répondre de façon homogène à un certain nombre de questions relatives à des comportements non-fonctionnels des applications. Ces comportements recouvrent les différents aspects de l'existence des objets, depuis l'adaptation de leur mode de création jusqu'à la vérification du bien-fondé de leur utilisation finale, mais toujours au travers d'abstractions permettant l'établissement des propriétés à travers une analyse statique du code.

Néanmoins, la garantie de correction des programmes met en jeu également, et surtout, des propriétés fonctionnelles : le résultat doit être bon ou acceptable, suivant les critères considérés. Or, dans le cas général, ces propriétés sont indécidables et nombre d'entre elles ne pourraient être démontrées qu'au prix d'un effort de preuve considérable et qui ne saurait être automatique.

7.1 Du non-fonctionnel au fonctionnel

Nous nous intéressons dans ce chapitre à des propriétés de programmes, qui bien qu'elles soient fonctionnelles appartiennent néanmoins à un fragment dans lequel une notion de typage, quoique insuffisante pour apporter la garantie, aide grandement à leur établissement.

Dans la mesure où nous désirons fournir, autant que faire se peut, un procédé automatique pour établir les propriétés de bon fonctionnement des systèmes (et une vérification automatique selon un algorithme similaire mais dégradé), l'idée de test exhaustif apparaît naturellement comme une possibilité pour démontrer toute propriété, fonctionnelle ou non. Le schéma de déploiement d'une application se découpe alors en trois phases : réception du code, test de bon fonctionnement, et mise en œuvre.

Cette méthode présente cependant deux inconvénients : d'une part, elle est dans le cas général peu praticable du fait de la taille des espaces de données concernés, et d'autre part, elle implique une notion de contrôle absolu de l'ensemble des paramètres du programme, ce qui n'est pas nécessairement acquis. Néanmoins, les travaux d'Engler [Eng99] ont mis en

évidence le fait qu'un nombre non négligeable de fonctionnalités centrales d'un systèmes s'appliquent dans un espace de données restreint.

Il est tout à fait possible d'envisager qu'une fonction « triche » lors de son test afin de se conformer aux résultats attendus, puis adopte un comportement malicieux une fois le programme déployé. En effet, la volonté de garantir au maximum les propriétés recherchées au chargement des applications impose que les tests soient effectués dans des conditions bien particulières, et ce *a priori* une seule fois.

Dans l'hypothèse où il serait néanmoins possible d'explorer exhaustivement l'espace des données possibles pour une fonction, le typage peut aider à résoudre le problème de reproductibilité fiable des tests effectués (à savoir, neutraliser la capacité des extensions à « tricher »).

7.2 Cas d'étude

7.2.1 Problématique

La plupart des travaux menés sur CAMILLE et FAÇADE concernaient l'isolation des diverses extensions apportées au système, afin que leur fonctionnement ne soit pas modifié illégalement par un code malicieux. Cela revient à s'assurer qu'il n'est besoin de faire confiance à personne car toutes les zones d'influences sont suffisamment séparées.

Néanmoins, il existe une classe de problèmes dont la résolution exige une forme de collaboration entre différentes extensions, si bien que l'isolation n'est plus suffisante pour garantir la sécurité. En effet, il est envisageable que plusieurs extensions d'un système aient besoin de s'accorder pour se partager l'utilisation d'une ressource par exemple.

Les programmeurs de systèmes d'exploitation ont besoin de fournir des moyens de vérifier les liaisons entre composants. Dans la sphère des systèmes d'exploitation et plates-formes d'exécution à visée générique, la vérification fonctionnelle des composants est connue comme étant un problème indécidable en général. Même dans un scénario dans lequel les domaines d'exécution sont bornés, les ressources requises pour effectuer la vérification augmentent très rapidement avec le champ des données d'entrée possibles et avec la taille des composants.

Le cas d'étude pour le reste de cette section consiste en le chargement dynamique d'extensions temps « réel » dans CAMILLE, suivant la problématique introduite par Damien Deville [Dev04]. Dans l'univers de la carte à puce, il existe un nombre important de traitements périodiques comportant une notion d'échéance à respecter, tels que la génération de clés cryptographiques pour la maintenance d'un service de téléphonie mobile.

Suivant les recommandations de la littérature propre aux exo-noyaux [Eng99, chapter 3], la stratégie d'allocation de temps est par défaut strictement équitable : chaque extension de CAMILLE bénéficie de la même fraction du temps de processeur disponible. L'inconvénient de cette pratique est que les besoins des différentes applications sont rarement équivalents,

et un ensemble de composants qui pourrait être ordonné dans un temps imparti, quitte à briser l'équilibre entre les extensions, ne pourra pas nécessairement l'être avec la contrainte supplémentaire d'équité.

Il est donc nécessaire de mettre en place une structure permettant aux extensions les moins demandeuses de collaborer avec d'autres pour exploiter au maximum les ressources disponibles et ne pas refuser à une extension d'exploiter ce qu'une autre laisse disponible au nom de l'équité.

Chaque extension désireuse de collaborer (et donc de participer à une redistribution partielle des ressources, voire d'en profiter) doit proposer un plan d'ordonnancement (dans les faits, une fonction `plan()`) au minimum convenable pour son propre ensemble de tâches périodiques. Ce plan est chargé d'élire une tâche qui sera exécutée lors du prochain quantum de temps disponible. Toutes ces extensions sont regroupées sous le contrôle d'un superviseur et une fonction `plan()` doit être élue parmi celles proposées afin de réaliser l'ordonnancement de l'ensemble des tâches des extensions.

Cette élection soulève bien évidemment deux problèmes majeurs. d'une part rien ne garantit que l'une des fonctions sera capable d'ordonnancer l'ensemble, auquel cas la collaboration est impossible sans plan d'ordonnancement supplémentaire. D'autre part, il existe un problème manifeste de confiance entre les extensions : toutes (sauf une) devront faire confiance à l'« honnêteté » d'un code dont elles ignorent tout, ce qui brise l'isolation entre extensions et la sécurité qu'elle procure.

Pour résoudre ce second problème, nous proposons un algorithme simple consistant à tester toutes les fonctions sur un groupe de tâches tel qu'une réussite sur cet ensemble garantisse un bon fonctionnement général (aucune tâche ne dépasse sa date limite).

7.2.2 Concrétisation du test exhaustif

Un ensemble de tâches périodiques est lui-même un système périodique dont la période (appelée hyper-période) est égale au plus petit commun multiple des périodes propres des tâches considérées. Dans l'hypothèse où la fonction `plan()` est une fonction pure et déterministe, il est suffisant d'effectuer un test exhaustif sur son hyper-période pour obtenir la garantie de la faisabilité de l'ordonnancement.

Cependant, dans la pratique, de telles fonctions sont optimisées de sorte à permettre un mécanisme d'ordonnancement le plus léger possible. En particulier, il est courant d'écrire ces fonctions en exploitant au maximum les calculs déjà effectués lors d'invocations précédentes et donc d'accéder à la mémoire. La conséquence est que bien souvent ces fonctions ne sont pas exemptes d'effets de bord et sont amenées à manipuler un état interne.

Ce problème peut être mis en relation avec celui énoncé par Engler dans [Eng99, chapter 4], lorsqu'il manipule la fonction `owns()` dans le but de sécuriser les accès aux méta-données. Plutôt que d'interdire les états internes, la méthode adéquate est de rendre ceux-ci visibles par le vérifieur. Cette approche fonctionne dans le cadre de fonctions déterministes,

qui sont alors qualifiées de « fonctions déterministes contrôlées ».

Dans le cas où la fonction est déterministe, il est suffisant de connaître précisément l'état interne utilisé (à savoir l'ensemble des zones de la mémoire qui le composent) pour être en mesure de neutraliser toute utilisation malicieuse que le programme pourrait en faire, sans pour autant porter préjudice aux optimisations nécessitant cet état interne.

En effet, la propriété souhaitée est que l'algorithme se comporte toujours de façon identique sur une hyper-période, dans des conditions identiques (mêmes tâches). Aussi, réinitialiser l'état interne potentiellement malicieux à la fin de chaque hyper-période est-il suffisant pour que l'on ait la garantie de la reproduction à l'identique, et périodique, de la politique d'ordonnancement telle qu'elle a été testée. La fonction `plan()` n'est alors pas en mesure d'être partiale vis-à-vis des tâches qu'elle doit ordonnancer.

Si la propriété de déterminisme est simple à établir dans le cas de fonctions pures, il n'en va pas de même pour les fonctions à effets de bord. Cependant, au même titre que la détection d'état interne, il s'agit d'une propriété non-fonctionnelle, qui peut être déterminée par analyse du code.

7.2.3 Utilisation des modes d'accès

Dans la mesure où nous nous intéressons aux effets de bord produits par les fonctions, la notion de mode d'accès à une entité joue naturellement un rôle primordial : le fait qu'une valeur soit lue ou écrite en mémoire nous renseigne sur la source possible de l'état interne que nous voulons contrôler.

Le fait qu'il soit possible ou non de contrôler suffisamment l'état interne pour assurer la fiabilité des tests détermine si il est possible d'accepter la fonction étudiée. Par exemple, l'accès à une ressource aléatoire inaltérable comme la date, ou une modification de l'état global du système (accès en écriture à un champ statique, entre autres) doivent être prohibés. La propriété exposée est, comme l'ensemble des propriétés de typage étudiées précédemment, vérifiée au chargement de l'application.

En résumé, une fonction valide pour l'ordonnancement doit respecter les contraintes suivantes : il lui est possible de tirer parti d'un état interne, mais uniquement sous la condition que ce dernier soit contrôlable depuis l'extérieur, ce qui implique que ni l'utilisation d'informations non déterministes, ni celle d'un état global du système ne sont admises.

Le test exhaustif de la fonction d'ordonnancement est effectué à chaque chargement d'une extension désirant collaborer avec les extensions existantes. Dès qu'une fonction remplit les conditions requises et passe avec succès le test, l'exo-noyau l'installe et commence à l'utiliser.

Notons que l'utilisation de la fonction au lieu de la simple interprétation de la trace obtenue lors du test (qui aurait nécessairement le même résultat, et dont le déterminisme serait acquis) se justifie par les faibles ressources dont disposent les plates-formes ciblées : la mémoire disponible ne rend généralement pas possible la conservation d'une donnée aussi

volumineuse que la trace d'exécution, au contraire du code qui la génère.

À la fin de chaque hyper-période, la zone mémoire contenant l'état interne éventuellement utilisé par la fonction d'ordonnancement est réinitialisée, de façon à ce que la fonction ne soit pas capable d'adapter son comportement à la phase de déploiement.

7.2.4 Détails de conception

Définition 1 (Modes d'accès). *Soit* $E = \{R\} \cup A$, *où* $A \subset \mathbb{N}$ *est un ensemble fini (de cardinal égal au nombre maximal d'arguments pour une fonction) et* R *un élément particulier représentant la valeur de retour de cette fonction. Un élément du treillis suivant est associé à chaque argument de la fonction :*

- $\perp$ *: état par défaut, l'argument est utilisé de manière sure ;*
- $\top$ *: l'argument est utilisé de manière à obtenir un état interne (par exemple un de ses champs a été accédé en écriture) ;*
- $Link_\alpha$ *pour* $\alpha \subseteq E$ *: une référence à l'argument a été conservée durant l'appel à cette fonction (par exemple, si l'argument a été attaché à la valeur de retour de la fonction, alors* $R \in \alpha$*) ; Toute modification future d'un élément de* α *a pour conséquence la modification de l'objet qui sert d'argument.*

L'ordre partiel entre les éléments de ce treillis est $\subseteq$. *Dans le reste de ce chapitre, nous noterons* $\mathcal{M}$ *l'ensemble des éléments de ce treillis.*

De même que pour l'analyse d'alias présentée en chapitre 4, le calcul des modes d'accès est réalisé à l'aide d'une interprétation abstraite [CC77] du code. La signature d'une méthode à n éléments relativement à ce mode de typage est un élément de $\mathcal{M}^n$.

Lors d'une analyse de code Java, les méthodes natives devaient être signées à la main, puisque ne pouvant être analysées. CAMILLE exporte également un certain nombre de fonctionnalités primitives depuis le noyau, qui échappent aussi à l'analyse car elles ne sont pas exprimées en langage FAÇADE : actuellement il y a approximativement 120 fonctions pour lesquelles l'analyse manuelle est nécessaire. Ces fonctions couvrent essentiellement l'ensemble des opérations arithmétiques basiques et les primitives d'accès à la mémoire, ce qui rend leur signature généralement simple à déterminer.

L'algorithme complet (exprimé en pseudo-C++ en figure 7.1) procède de la même méthode que celle exposée au chapitre 4 : il vise à l'obtention d'un point fixe sur l'ensemble des méthodes disponibles quant à la signature de celles-ci pour les modes d'accès des objets manipulés.

7.3 Résultats expérimentaux

Le processus de chargement de CAMILLE a été modifié de façon à vérifier les modes d'accès associés aux méthodes, de la même façon que les autres types utilisés.

Nous avons implanté différentes stratégies d'ordonnancement dans CAMILLE, notamment les classiques *Round Robin*, *Earliest Deadline First* ou encore *Least Laxity First*, comme détaillé dans [DHSR05].

À titre d'exemple, le prototype C d'une fonction fournissant un *Round Robin* est `int planRR( Task tl[], int nb_task, int curr_slice)` et la signature associée pour ce qui concerne les modes d'accès caractérise une fonction pure : $(\bot, \bot, \bot, \bot, \bot)$ (les deux premiers arguments sont les arguments implicites représentant respectivement l'ensemble des données statiques et `this`). De ce fait, cette politique d'ordonnancement ne nécessite aucun contrôle supplémentaire à l'exécution.

Un deuxième exemple intéressant est celui de la politique standard *EDF*, dont le prototype C est `int planEDF( Task tl[], int nb_task, int curr_slice, byte cmem[])`. Sa signature relative aux modes d'accès est $(\bot, \bot, \bot, \bot, \bot, \top)$, ce qui révèle l'utilisation potentielle (et en l'occurrence avérée) d'un état interne par l'intermédiaire du dernier argument. Dans les faits, cette politique est optimisée en stockant la progression des tâches dans cette zone mémoire. Néanmoins, cet état interne peut être contrôlé (par réinitialisation après chaque hyper-période), ce qui permet d'utiliser la politique à des fins de collaboration sans risquer de compromettre les propriétés temps-réel du système.

7.4 Conclusion

Nous avons pu illustrer, à travers l'exemple des stratégies d'ordonnancement pour le temps réel, une manière différente d'exploiter les résultats d'une analyse statique. Celle-ci accorde une place encore plus importante que les précédentes à l'exécution effective du code, puisque ce n'est qu'à travers cette exécution que la propriété recherchée (ici le respect des délais de chaque tâche) peut être garantie. Notons enfin que le coût de l'analyse sous-jacente peut être amorti par l'utilisation de ses résultats dans le but de déterminer des propriétés similaires sur d'autres fonctions centrales des systèmes (la fonction `own()` d'Engler en serait un bon exemple).

```
method::Sign() { // sign is the signature to be computed
  stack.push(entry_point);
  while(! stack.empty()) {
    line = stack.pop();
    switch(instructions[line].type()) {
      case Jump(target):
        // << is a injection operator:
        // any dependency at line is copied at target.
        if(deps[target] << deps[line])
          stack.pushIfNotIn(target);
      case JumpIf(target):
        if(deps[target] << deps[line])
          stack.pushIfNotIn(target);
        if(deps[line+1] << deps[line])
          stack.pushIfNotIn(line+1);
      case JumpList(targets):
        foreach i in targets
          if(deps[i] << deps[line])
            stack.pushIfNotIn(i);
      case Invoke(m):
        if(deps[line+1] << m.applyOn(deps[line]))
          stack.pushIfNotIn(line+1);
      case Return(ret): // args is the arguments' array
        foreach i in args do {
          // |= is an accumulative bitwise "or"
          if(deps[line][i].contains(ret))
            sign[i] |= IO_R;
          foreach j in args do
            if(deps[line][i].contains(args[j]))
              sign[i] |= IO_j;
        }
    }
  }
}

main() {
  // cat is the set of methods to be considered, the "catalog"
  until(! cat.modified()) {
    foreach m in cat do
      m.signature = m.Sign();
  }
}
```

FIG. 7.1 – Algorithme des calculs de signatures.

```
bool verify(method,sign) {
  bool res = true;
  dependencies_vector dep = initDeps();
  foreach instr in method.body() do {
    if(instr.hasLabel()) {
      // apply() computes the action of instr over dep
      // includes() check if computed deps
      // are compatible with proof informations.
      res |= (instr.proof.includes(dep.apply(instr)));
      dep = instr.proof;
    }
    else
      dep = dep.apply(instr);
    }
  return res;
}
```

FIG. 7.2 – Algorithme de vérification.

Chapitre 8

Conclusion

Au cours de cette étude, nous avons eu l'opportunité d'explorer en profondeur les différentes possibilités pour déduire un maximum d'informations d'un code, dans le but de rendre son exécution meilleure :

- par des méthodes d'optimisation, présentées ici à travers l'exemple critique (du moins dans le domaine de l'embarqué) de la gestion de la mémoire,
- en s'assurant de la fiabilité de l'ensemble, notamment en ce qui concerne l'utilisation de composants ou leur coopération.

L'approche suivie tout au long de cette thèse a été de toujours apporter les garanties ou résultats au plus tôt, afin qu'un code ne puisse être installé que si il correspond aux exigences exprimées (propriétés de sécurité) ou induites (bon comportement vis-à-vis d'une optimisation effectuée) par la plate-forme.

De ce fait, l'expression des résultats d'analyse, ainsi que leur vérification, peut se transposer aisément dans le domaine classique des systèmes de types. Ceci permet de tirer profit des propriétés connues de ces systèmes, et notamment de se reposer sur un découpage en deux phases du processus de calcul :

- d'une part, un calcul à l'extérieur du système embarqué, qui est chargé de déterminer précisément les types mis en jeu,
- et d'autre part la vérification à l'intérieur de ce système de la cohérence de types déclarés en certains points du programme afin de s'affranchir de la complexité des algorithmes mis en jeu à l'extérieur.

Cette stratégie nous permet donc d'encoder sous forme de types un certain nombre de propriétés (ou de résultats impliquant trivialement ces propriétés), ceci sans qu'il soit possible de mentir au système sur le typage.

En d'autres termes, les analyses présentées dans ce document sont exploitées simplement comme un typage fort (tel que défini en section 2.2.1), généralement à l'aide d'une inférence de types directement dérivée de l'algorithme de calcul extérieur.

8.1 Contributions

Plus précisément, les contributions présentées dans cette thèse concernent en particulier deux aspects complémentaires de l'approche des systèmes de types.

8.1.1 Design de systèmes de types

En premier lieu, nous avons détaillé la façon dont nous avons conçu un système de types destiné à l'embarqué (sans pour autant que cela l'empêche d'être adapté à des plates-formes plus traditionnelles). Le système de types actuel de CAMILLE présente un certain nombre de propriétés intéressantes et peu communes dans le domaine des matériels contraints.

La conception de ce système autour des relations complexes qui existent entre fiabilité, extensibilité et performance a donné lieu à la recherche d'un compromis acceptable entre ces trois axes. Compromis qui, de façon assez surprenante, ne se traduit globalement pas par un renoncement à un quelconque gain. Bien évidemment, ni l'extensibilité ni la performance ne sont systématiquement maximales, mais il demeure qu'en définitive, le système de types de CAMILLE est suffisamment riche pour permettre davantage que ce que proposent des systèmes établis comme Javacard.

8.1.2 Conception d'extensions

En second lieu, nous avons cherché à enrichir les systèmes de types (ce qui pourrait être fait par le biais du design présenté précédemment) grâce à des notions quelque peu marginales, dans le sens où elles ne correspondent pas véritablement aux préoccupations classiques exprimées à travers les types.

Cette démarche prend sa source dans le constat basique qu'un système embarqué, disposant de ressources contraintes, devrait être en mesure de réutiliser au maximum le code de certains composants pour des tâches similaires, sous réserves d'adaptations mineures. Le composant visé ici est le tout premier de la chaîne de déploiement des systèmes embarqués, le chargeur de code, qui joue le rôle de contrôleur pour les applications entrantes : avant même son installation, toute application doit apporter une certaine garantie de « bon comportement » vis-à-vis de la globalité du système. Notre but ici est de permettre l'élargissement de la définition de ces « bons comportements » sans pour autànt faire prendre une ampleur inconsidérée au chargeur.

Optimisation

La première partie de cette recherche d'élargissement a consisté en l'expression d'analyses classiques (telles que l'analyse d'échappement) en termes de typage, simplement en la présentant comme la lecture directe d'une information résultant purement d'une interprétation abstraite (l'analyse d'alias). Nous avons cherché à rendre cette analyse la plus

adaptée possible à l'environnement dans lequel elle est utilisée, en adoptant un compromis entre la précision des résultats offerte par la connaissance totale du système et le flou engendré par la possibilité de charger dynamiquement du code (possibilité à laquelle il était impossible de renoncer).

Par ailleurs, l'étude approfondie de ce facteur d'optimisation de la gestion mémoire nous a permis de mettre en avant une extension de cette analyse d'échappement permettant de profiter à moindre coût des bénéfices de l'allocation en région séparée lors de l'utilisation de paradigmes de programmation particuliers. Les schémas de conception objet classiques de fabriques ou d'agrégation, qui rendent en pratique inutile l'analyse d'échappement traditionnelle, ont fait l'objet d'un traitement particulier afin d'être réintégrés dans le domaine d'action de cette analyse étendue.

Fiabilité

Dans un registre différent, nous nous sommes intéressés à la possibilité d'augmenter la notion de fiabilité grâce à la vérification de types. Bien sûr, suivant les propriétés exprimées par les types la fiabilité du système augmente par le simple fait que l'information disponible à propos des objets manipulés augmente.

Néanmoins, une vraie notion de fiabilité fait intervenir d'autres propriétés que celles qui sont calculables par interprétation abstraite : les aspects strictement fonctionnels de l'exécution du code échappent naturellement à cette vision schématique.

Dans le cas présent, il nous a paru intéressant de montrer que la frontière entre aspects fonctionnels et non-fonctionnels est parfois assez diffuse, et qu'au prix d'un effort supplémentaire de test, la garantie d'une propriété non-fonctionnelle peut aider à la garantie d'une propriété fonctionnelle. Le cas présenté dans ces travaux a mis en lumière que le calcul combiné de deux propriétés non-fonctionnelles (le déterminisme et les modes d'accès) pouvait déboucher sur la garantie d'honnêteté (ou plus précisément la garantie d'incapacité de tricher) d'un ordonnanceur temps-réel.

8.1.3 Bilan

L'ensemble de ces travaux a donné lieu à la mise en place de deux réalisations concrètes.

D'une part le système de types élaboré pour CAMILLE est effectivement en usage, et des travaux à la fois universitaires et industriels sont en cours sur cette plate-forme afin d'en tirer un bénéfice maximal et, peut-être, la déployer à une échelle moins confidentielle.

D'autre part, l'intérêt porté aux différentes analyses abstraites de code a donné lieu à la conception et à l'utilisation d'une plate-forme générique visant à supporter ces différentes analyses en exhibant précisément leurs points communs et leurs spécificités. Cette plateforme, `STAN`, permet l'enrichissement d'une interprétation abstraite basique par l'adjonction d'une sémantique spécifique et dont l'expressivité n'est *a priori* pas contrainte. Notons que cette plate-forme est à destination première de Java, mais qu'une adaptation à un autre

langage serait tout à fait envisageable.

8.2 Perspectives

À l'issue de cette thèse, un certain nombre de problématiques intéressantes mériteraient un approfondissement certain.

En premier lieu, il serait intéressant d'explorer plus avant les capacités d'expression des systèmes de types basés sur la conception par métatypes présentée ici. En particulier, il serait bon d'intégrer véritablement des systèmes de types différents à la structure en place : l'exemple présenté d'extension par héritage multiple, bien que faisant la démonstration d'un certain nombre de mécanismes incontournables, reste un exemple assez classique et pourtant déjà relativement incomplet. La difficulté d'ajouter ou de modifier des concepts objets devrait être évaluée afin de pouvoir tenter de la réduire par le biais de mécanismes de très haut niveau. En effet il est plus que probable que les abstractions de modèles objets introduites dans les métatypes (et donc l'interface présentée par les métaclasses) soient insuffisantes pour exprimer simplement l'ensemble des variations imaginables autour du paradigme de programmation objet. Par conséquent, se livrer à une étude systématique des variations en question afin d'en extraire les concepts fondateurs aiderait grandement à rendre plus souple la définition des métatypes, et donc augmenterait la flexibilité du système.

Dans le même ordre d'idées, il serait particulièrement intéressant d'étudier la possibilité de laisser une extension quelconque créer sa propre hiérarchie d'objets, intégrée au tout. En l'état actuel des choses, l'utilisation de ce mécanisme est restreinte à un mode « de confiance » (déploiement, utilisateur privilégié...), mais il est probablement envisageable de la rendre possible dans un domaine plus large, au prix de contrôles stricts. Les conditions de base sont donc volontairement défensives, faute de savoir exactement quels sont les mécanismes à protéger, et comment le faire. Il est évident que pouvoir permettre, dans une certaine mesure, à du code « utilisateur » de charger sa propre hiérarchie de types, serait d'un grand intérêt.

Concernant l'extension des systèmes de types à des propriétés arbitraires statiquement calculables, une première démarche pourrait être d'élargir l'inventaire des propriétés exprimables de la même façon que celles présentées dans ces travaux. En effet, les diverses propriétés considérées ici sont presque toutes en relation, et organisées suivant une structure hiérarchique : elles sont toutes basées sur l'interprétation relativement immédiate des résultats fournis par une autre, jusqu'à ce que la simple interprétation abstraite du programme avec une sémantique de contenu (de variables) suffise à conclure.

Il serait aussi très vraisemblablement riche d'enseignement d'explorer plus en détails les interactions et relations qui existent entre les domaines non-fonctionnel et fonctionnel. Bien que le dernier ne puisse évidemment pas s'exprimer dans toute sa généralité dans le premier, il apparaît que le non-fonctionnel joue un rôle important en tant que base solide pour l'élaboration de preuves à destination du fonctionnel. L'exemple, étudié ici, du com-

posant d'ordonnancement se prêtait particulièrement bien à l'utilisation quasi immédiate d'informations obtenues par analyse statique pour la simple raison qu'il était envisageable de tester intégralement ce composant. Bien évidemment, tous les composants système ne peuvent être soumis au même traitement, mais d'autres composants, et peut-être d'autres façons de franchir la frontière entre ces deux mondes, restent certainement à déterminer. On peut penser notamment à des composants de gestion mémoire garantissant une forme d'équité au niveau de la répartition des emplacements utilisés, afin d'obtenir une meilleure durée de vie du système : de tels composants semblent présenter suffisamment de similarités avec les cas d'études présentés ici pour constituer une bonne approche initiale.

Enfin, la mise en relation des deux problématiques abordées dans cette thèse pourrait donner lieu à des développements intéressants. En effet, la première partie s'intéressant au typage structurel (et explicite) des objets, et la deuxième faisant appel à du typage inféré car décrivant un effet subi par un objet, ces considérations apparaissent comme assez orthogonales.

Néanmoins, l'expression de propriétés telles que celle calculée par analyse d'échappement à travers la structure de types organisée par des métatypes pourrait être particulièrement intéressante, et ceci d'autant plus si cette piste est rapprochée du point évoqué précédemment concernant l'extension de la possibilité d'ajouter des branches de types d'objets à un système en fonctionnement. Être capable de faire cela offrirait un grand avantage face à des solutions telles que celles employées pour les plates-formes Java actuelles, qui nécessitent la présence d'un mécanisme *ad hoc* dans la machine virtuelle pour interpréter les résultats d'analyse, fournis sous la forme d'annotations annexes au code : ici au contraire, ce serait directement au vérifieur de typage que l'opération de vérification des résultats d'analyse seraient confiée, le code quant à lui restant parfaitement standard, bien qu'utilisant un système de types particulier.

Une piste dans cette direction pourrait être de sous-classer les métatypes `Boxed` et `UnBoxed` présentés au chapitre 3 (les types non-référence n'ayant pas d'intérêt dans le cadre de l'analyse d'échappement) pour ajouter l'existence d'un état dans leurs instances, qui corresponde au tableau de liens prévus pour l'objet considéré. Étant donné que dans le modèle objet présenté, tout est matérialisé par un appel de méthode, la création de liens ne pourrait également se faire que par ce biais (il n'existe pas de `putfield` à la Java). La signature de ces méthodes serait directement donnée par les types des paramètres (eux-mêmes descendant de `Boxed` ou `UnBoxed`), et il suffirait d'introduire dans la primitive d'autorisation de liaison un contrôle de cohérence entre les liens créés par l'appel de méthode et les liens autorisés pour les objets qui subissent une modification.

Un détail supplémentaire, qui n'ajoute pas grand chose au fond, est qu'il faudrait également surcharger les opérations de génération de code correspondant à la création d'un lien : en effet, il ne serait pas acceptable de contourner les systèmes de types étendus par génération de code non protégé. Ces nouvelles opérations devraient donc également se conformer aux contraintes de vérification de type et vérifier que l'argument pour lequel un lien est créé est bien autorisé à le recevoir.

Il serait donc très instructif de mener à bien une étude complémentaire afin d'évaluer

la pertinence de l'unification des deux modèles de typage étudiés dans ce document.

Bibliographie

[AAC+99] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in java. In *OOPSLA '99 : Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, New York, NY, USA, 1999. ACM Press.

[BBC+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, 1997.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-Insensitive Interprocedural Alias Analysis in the Pressence of Pointers. In David Gelertner, Alexandru Nicolau, and David Padua, editors, *Lecture Notes in Computer Science, 892*. Springer-Verlag, 1995.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer-Verlag, 2006.

[BH99] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in java. Technical report, Santa Barbara, CA, USA, 1999.

[BJP] Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. submitted to Elsevier Science.

[BK88] Daniel G. Bobrow and Gregor Kiczales. The common lisp object system metaobject kernel : a status report. In *LFP '88 : Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 309–315. ACM Press, 1988.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages. application to java. In *14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1999.

[BM88] Robert S. Boyer and J. Strother Moore. *A computational logic handbook.* Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[BM90] Robert S. Boyer and J. Strother Moore. A theorem prover for a computational logic. In M. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449, pages 1–15. Springer-Verlag, 1990.

[BSF04] Matthew Q. Beers, Christian H. Stork, and Michael Franz. Efficiently verifiable escape analysis. In M. Odersky, editor, *ECOOP 2004, LNCS 3086*, pages 75–95, Oslo, 2004. Springer-Verlag.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, California, 1977.

[Che00] Zhiqun Chen. *Java Card™ Technology for Smart Cards : Architecture and Programmer's Guide.* The Java™ Series. Addison Wesley, 2000.

[Chi95] Shigeru Chiba. A metaobject protocol for c++. In *OOPSLA '95 : Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.

[CJPS05] David Cachera, Thomas P. Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In *FM*, pages 91–106, 2005.

[cli] Cli. http ://www.ecma-international.org/publications/standards/Ecma-335.htm.

[Cou06] Alexandre Courbot. *Spécialisation tardive de systèmes Java embarqués pour petits objets portables et sécurisés.* PhD thesis, Univ. Lille 1, France, september 2006.

[CRL98] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Complexity of concrete type-inference in the presence of exceptions. In *ESOP '98 : Proceedings of the 7th European Symposium on Programming*, pages 57–74, London, UK, 1998. Springer-Verlag.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods : state of the art and future directions. *ACM Computing Surveys*, 28(4) :626–643, 1996.

[Dah68] Ole-Johan Dahl. *SIMULA 67 common base language, (Norwegian Computing Center. Publication).* 1968.

[Dah92] Ole-Johan Dahl. *Verifiable Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1992.

[DDDCG01] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In *ECOOP*, pages 130–149, 2001.

[Dev04] Damien Deville. *CamilleRT : un Système d'Exploitation Temps Réel Extensible pour Carte à Microprocesseur.* PhD thesis, Université de Lille1, Décembre 2004.

[DGGJ03] Damien Deville, Antoine Galland, Gilles Grimaud, and Sébastien Jean. Smart Card operating systems : Past, Present and Future. In *5th NORDU/USENIX Conference*, pages 14–28, Västerås, Sweden, February 2003.

[DHSR05] Damien Deville, Yann Hodique, and Isabelle Simplot-Ryl. Safe collaboration in extensible operating systems : A study on real time extensions. *International Journal of Computers and Applications*, 1 :20–26, january 2005.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94 : Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[EK95] Dawson R. Engler and Frans Kaashoek. Exterminate All Operating System Abstractions. In *the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, USA, 1995.

[Eng99] Dawson R. Engler. *The Exokernel operating system architecture.* PhD thesis, Massachusetts Institute of Technology (MIT), 1999.

[Fas01] Jean-Philippe Fassino. *THINK : vers une architecture de systèmes flexibles.* PhD thesis, École Nationale Supérieure des Télécommunications, December 2001.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot : an optimizing compiler for java. *Softw. Pract. Exper.*, 30(3) :199–232, 2000.

[FPR98] Bertil Folliot, Ian Piumarta, and Fabio Riccardi. A dynamically configurable, multi-language execution platform. In *EW 8 : Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 175–181, New York, NY, USA, 1998. ACM Press.

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think : A software framework for component-based operating system kernels. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX Association.

[Gal05] Antoine Galland. *Contrôle des ressources dans les cartes à microprocesseur.* PhD thesis, Pierre and Marie Curie University, 2005.

[Ghi05] Dorina Ghindici. Information flow analysis. Application to Java bytecode. Master's thesis, Univ. Lille 1, France, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHSR05] Gilles Grimaud, Yann Hodique, and Isabelle Simplot-Ryl. Secure extensible type system for efficient embedded operating system by using metatypes. In *SaNSO 2005*, volume 2, pages 83–87, july 2005.

[GHSR06a] Gilles Grimaud, Yann Hodique, and Isabelle Simplot-Ryl. Can small and open embedded systems benefit from escape analysis ? In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, 2006.

[GHSR06b] Gilles Grimaud, Yann Hodique, and Isabelle Simplot-Ryl. On the use of metatypes for safe embedded operating system extension. *International Journal of Parallel, Emargent and Distributed Systems (IJPEDS)*, 22 :1–13, 2006.

[GHSR07] Gilles Grimaud, Yann Hodique, and Isabelle Simplot-Ryl. A verifiable lightweight escape analysis supporting creational design patterns. In *The 2007 IEEE International Symposium on Ubisafe Computing (UbiSafe-07)*, 2007. to appear.

[GLS84] Jr. Guy L. Steele. *Common LISP : the language.* Digital Press, 1984.

[GLV99] Gilles Grimaud, Jean-Louis Lanet, and Jean-Jacques Vandewalle. FACADE : A typed intermediate language dedicated to smart cards. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering – ESEC/FSE'99*, volume 1687, pages 476–493. Springer-Verlag, Berlin Germany, 1999.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Gri] Gilles Grimaud. Camille. http ://www.lifl.fr/RD2P/CAMILLE.

[Gri00] Gilles Grimaud. *CAMILLE : un Système d'Exploitation Ouvert pour Carte à Microprocesseur.* PhD thesis, Univ. Lille 1, France, December 2000.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction (CC'2000)*, volume 1781. Springer-Verlag, 2000.

[Gud93] David Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, Tucson, Arizona, 1993.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The art of metaobject protocol.* MIT Press, Cambridge, MA, USA, 1991.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[LDGV07] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system, 1996–2007.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[LH99] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *ESEC/FSE-7 : Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 199–215, London, UK, 1999. Springer-Verlag.

[Luc87] John M. Lucassen. Types and effects : Towards the integration of functional and imperative programming. Technical Report TR-408, August 1987.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[Mao] Maosco. Multos web site. http ://www.multos.com.

[MMMP90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Mølier-Pedersen. Strong typing of object-oriented languages revisited. In *OOPSLA/ECOOP '90 : Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 140–150, New York, NY, USA, 1990. ACM Press.

[Nec97] George C. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, January 1997.

[OSM+00] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. Openjit : An open-ended, reflective jit compiler framework for java. In *ECOOP '00 : Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 362–387, London, UK, 2000. Springer-Verlag.

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 116–127, New York, NY, 1992. ACM Press.

[RCG00] Antoine Requet, Ludovic Casset, and G. Grimaud. Application of the B formal method to the proof of a type verification algorithm. In *Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE 2000)*, pages 115–124, 2000.

[RR98] Eva Rose and Kristoffer H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

[RS02] Christophe Rippert and Jean-Bernard Stefani. Building secure embedded kernels with the think architecture. In *Proc. of the workshop on Engineering Context-aware Object-Oriented Systems and Environments - OOPSLA*, Seattle, USA, 2002. IEEE Press.

PPN : 115260285

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL '97 : Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1997. ACM Press.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for java. In *Proc. of the 16th symposium on Operating Systems Principles*. ACM Press, 1997.

[Wei84] Reinhold P. Weicker. Dhrystone : A synthetic systems programming benchmark. *Commun. ACM*, 27(10) :1013–1030, 1984.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[WL99] Pierre Weis and Xavier Leroy. *Le langage Caml.* Dunod, 1999.

[WL02] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *SAS '02 : Proceedings of the 9th International Symposium on Static Analysis*, pages 180–195, London, UK, 2002. Springer-Verlag.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10) :187–206, 1999.