Une machine virtuelle en héritage multiple basée sur le hachage parfait

Julien Pagès

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **LIRMM**

Spécialité: **Informatique**

Présentée par **Julien Pagès**

Une machine virtuelle en héritage multiple basée sur le hachage parfait

Soutenue le 14 décembre 2016 devant le jury composé de

M. Roland Ducournau	Professeur	LIRMM, Université de Montpellier	Directeur
M. Philippe Clauss	Professeur	INRIA, Université de Strasbourg	Rapporteur
M. Gaël Thomas	Professeur	Télécom SudParis	Rapporteur
M. Christophe Paul	Directeur de recherche	LIRMM, CNRS	Examinateur
M. Jean Privat	Professeur	Université du Québec à Montréal	Examinateur
M. Manuel Serrano	Directeur de recherche	INRIA, Sophia Antipolis	Examinateur
M. Floréal Morandat	Maître de conférences	LaBRI, ENSEIRB-MATMECA	Invité

Université Montpellier

LIRMM

Remerciements

Durant toute cette thèse, de nombreuses personnes ont compté et m'ont aidé à mener à bien ce travail.

Tout d'abord, je tiens déjà à remercier Roland, mon directeur pour son incroyable connaissance et sa présence tout au long de la thèse. Merci de m'avoir ammené où j'en suis aujourd'hui en plus de m'avoir appris à faire de la recherche sérieuse et poussée. Un jour peut-être j'en saurais autant que lui.

Merci à tous les membres du jury de m'avoir fait l'honneur d'y participer : Philippe Clauss et Gaël Thomas les rapporteurs et Christophe Paul, Jean Privat, Manuel Serrano les examinateurs, merci également à Floréal Morandat d'avoir être présent le jour de la soutenance.

Les ex-doctorants et ingénieurs du LIRMM : Adel mon co-bureau avec qui nous avons tant échangé. Manu l'homme organisé aux 2 publications par mois. Guillerme l'homme en kilt au coeur tendre. Vincent et ses histoires incroyables. Nous avons refait le monde tous les midis ensemble en plus de se serrer les coudes pendant la thèse.

Nicolas et Laurie pour avoir réponse à tout et nous avoir tiré de tous les mauvais pas possibles (et avec le sourire !). Merci aussi à tous les autres du LIRMM que je ne peux pas citer ici pour tous les moments partagés durant ces trois années qui ont très largement dépassés l'informatique.

L'équipe MaREL : Clémentine, Marianne, Hinde, Chouki, Christophe, Djamel pour les échanges, les enseignements en commun et autres séminaires.

L'équipe de Nit à l'UQAM : Jean, Alex x 2, Lucas pour leur accueil au cours de nos divers voyages à Montréal.

Merci à ma seconde équipe : ICPS/CAMUS pour m'avoir exceptionnellement bien accueilli à Strasbourg pour mon ATER et pour tout ce qu'ils m'ont appris en seulement quelques mois.

Enfin, merci à tous mes amis qui m'ont permis de tenir durant la thèse et pour tous les bons moments avec eux : Baptiste, Clément, Geoffrey, Julien x 2, Kevin, Maël, Paul, Tristan, Vivien et tous les pouers-mouers de slack en général pour des après-midis studieuses au quotidien.

Merci à ma famille d'avoir cru en moi et de m'avoir soutenu durant toutes ces années. Enfin, merci à Pauline pour sa présence au quotidien.

Table des matières

Table des figures

Chapitre 1

Introduction générale

Contents

1.1 Introduction

Cette thèse est centrée sur les langages à objet en héritage multiple en typage statique exécutés avec une machine virtuelle. Les langages à objet sont depuis quelques décennies devenus un paradigme de programmation très utilisé. Ces langages ont l'avantage d'offrir une bonne capacité d'abstraction et aident au découpage de l'application et à la modularité grâce aux concepts centraux de classe et d'objet.

Une classe représente la structure et permet de spécifier les types de données stockés (les attributs) ainsi que les opérations (les méthodes). Les objets sont instance d'une classe et stockent leurs propres données. Dans ces langages, trois mécanismes de base permettent d'interagir avec les objets :

— L'appel de méthode à travers le mécanisme de liaison tardive

— L'accès aux attributs de l'objet

— Le test de sous-typage

Le premier mécanisme permet d'*envoyer un message* à un objet en appelant une opération particulière. Cette opération aura généralement un effet sur cet objet, ou un autre objet, en modifiant ses données, donc la valeur de ses attributs. Ces derniers sont lus ou modifiés à travers le mécanisme d'accès aux attributs. Enfin, le test de sous-typage permet de tester le type dynamique d'un objet, pour savoir si il est instance d'une classe donnée.

Le typage statique apporte un avantage pour le programmeur en offrant des annotations de types directement visibles dans le code. De plus, il apporte naturellement une sécurité dans le typage en permettant de vérifier un grand nombre d'éléments lors de

la compilation plutôt qu'à l'exécution. Par exemple, un environnement de développement comme Eclipse bénéficie du typage statique afin de détecter les erreurs le plus tôt possible.

Dans les années 1990, un langage est devenu très populaire : Java. Ce langage a été créé par l'entreprise Sun et a été le premier langage à objet très largement diffusé qui fonctionnait avec un système d'exécution particulier : une machine virtuelle. Une machine virtuelle permet d'exécuter du code, elle se différencie des autres types de systèmes d'exécution par une très grande dynamicité. Java et sa machine virtuelle permettent de charger dynamiquement du code pendant l'exécution, à travers l'introspection par exemple. Les machines virtuelles permettent aussi d'effectuer de multiples vérifications du code pendant l'exécution et offrent donc une sécurité accrue au niveau du typage ainsi que du point de vue du système. Le langage Java a été suivi par C♯ ainsi que d'autres langages ultérieurement. De nos jours, les machines virtuelles pour des langages à objet sont courantes, mais il existe aussi des machines virtuelles pour des langages qui ne sont pas forcément à objet.

L'héritage multiple quant à lui, est généralement présent dans les langages à objet en typage statique. En Java et C♯, une forme dégradée est présente à travers le sous-typage multiple, avec des interfaces. Ada offre également depuis sa version 2005 des interfaces tandis que C++ est en héritage multiple depuis sa création. L'héritage multiple offre en effet une facilité de modélisation et de réutilisabilité puisqu'une classe peut réutiliser les propriétés de deux autres classes, conçues séparément, en les combinant.

Cette thèse est centrée sur l'étude des machines virtuelles pour des langages ressemblant à Java, mais en héritage multiple. L'héritage multiple paraît naturel d'un point de vue modélisation, mais la spécification précise et son implémentation ne sont pas des problèmes triviaux. Les aspects sémantiques sont maintenant considérés comme résolus [Ducournau and Privat, 2011].

Cependant, une critique récurrente faite à l'héritage multiple est son inefficacité supposée. On sait aujourd'hui compiler de manière optimale des langages en héritage multiple, mais uniquement dans un compilateur global et statique, à travers la coloration [Pugh and Weddell, 1990, Vitek et al., 1997, Sallenave and Ducournau, 2012]. Les machines virtuelles modernes pour des langages à objet fonctionnent en monde ouvert, les classes du programme sont alors découvertes au fil de l'exécution. Dans ce contexte, il reste à évaluer le surcoût de l'héritage multiple.

Aujourd'hui, il n'existe pas à notre connaissance de machine virtuelle pour un langage à objet en héritage multiple et en typage statique. Nous nous intéresserons aux optimisations dans les machines virtuelles qui sont impératives pour assurer de bonnes performances puisque le chargement des classes représente un surcoût à l'exécution par rapport aux compilateurs. La problématique des optimisations est donc centrale. De plus, le chargement dynamique implique que l'ensemble des classes connues du programme change au fil de l'exécution, il faut donc se baser sur la connaissance courante du programme pour prendre les décisions d'optimisations. Ces optimisations devront alors tenter de limiter l'utilisation de l'héritage multiple pour réduire le plus possible son surcoût. Le chargement dynamique peut amener à effectuer des optimisations spé-

culatives : elles sont correctes lors de leur réalisation mais peuvent devenir incorrectes plus tard. Dans ce cas, il faut alors défaire ces optimisations. Ces mécanismes sont relativement complexes et ne sont pas toujours décrits précisément dans la littérature scientifique.

Il y a deux objectifs majeurs dans le cadre de cette thèse :

1. Spécifier et réaliser un prototype d'une machine virtuelle pour un langage à objet en héritage multiple et en typage statique
2. Étudier les systèmes d'optimisations dans une telle machine virtuelle

1.2 Plan de la thèse

Le manuscrit est divisé en trois grandes parties précédées de la présente introduction au chapitre 1. La première partie contient essentiellement l'état de l'art des langages à objet, des machines virtuelles et des expériences réalisées lors des tests de performances de machines virtuelles. La deuxième partie contient une description détaillée de la façon dont est réalisée notre machine virtuelle. Les solutions choisies y sont décrites et discutées. Enfin, la troisième partie contient les principales contributions, le modèle et la représentation intermédiaire utilisés dans notre machine virtuelle, les extensions apportées à la préexistence, une technique pour limiter les réparations de codes suite à des optimisations agressives, ainsi que toute la partie contenant les expérimentations.

Partie I

Le chapitre 2 présente un rapide historique des langages à objet. Le cadre de la thèse est ensuite posé, notamment quel type de langage nous considérons dans cette étude. Suit, une présentation des machines virtuelles en général, puis surtout les machines virtuelles pour des langages à objet. Ce chapitre se termine par un état de l'art des techniques d'implémentations des mécanismes objet.

Le chapitre 3 présente les systèmes d'optimisation utilisés dans les machines virtuelles. La problématique est posée avant de définir la notion de protocole de compilation/recompilation. Ensuite, nous étudions quelques techniques d'optimisation pour voir par la suite des protocoles complets dans quelques machines virtuelles Java.

Le chapitre 4 présente l'état de l'art des techniques utilisées pour effectuer des tests de performances dans les compilateurs et machines virtuelles. Nous présentons la problématique et les difficultés rencontrées lors de telles expériences. Puis nous présentons des techniques utilisées dans la littérature en les discutant avant de proposer nos propres pistes en définissant ce qu'il faudrait faire selon nous.

Partie II

Le chapitre 5 présente un état des techniques possibles pour développer la machine virtuelle Nit. Ensuite nous décrivons les solutions retenues en les discutant. Le chapitre 6 contient la description de l'architecture de notre machine virtuelle. Il comprend des

détails sur la façon dont a été réalisé le développement ainsi que sur la manière dont sont implémentés les aspects importants de la machine virtuelle.

Partie III

Le chapitre 7 rappelle la présentation du méta-modèle de Nit, avant de présenter les extensions apportées. Nous décrivons en particulier comment est structurée la représentation intermédiaire dans la machine virtuelle. Le chapitre 8 revient sur le concept de préexistence. Ensuite, nous présentons les extensions réalisées dans le cadre de la thèse qui élargissent son usage et sa définition. Le chapitre 9 contient toutes les expérimentations. Des expériences ont été effectuées pour mesurer l'effet des extensions de la préexistence ainsi que l'efficacité des protocoles de compilation/recompilation. Les données récoltées pendant les expériences sont présentées et discutées.

Conclusion

La conclusion résume le travail, nous réexaminons notre problématique initiale : comment s'approcher de l'efficacité d'un compilateur en monde fermé dans une machine virtuelle pour un langage en héritage multiple. Nous présentons ensuite des pistes pour étendre et continuer la recherche dans ce domaine

Première partie

État de l'art

Chapitre 2

Langages à objet et machines virtuelles

Contents

2.1 Introduction

Le paradigme des langages orientés objets est apparu au fil du temps. Aujourd'hui il est reconnu que les langages à objet présentent des avantages pour programmer de grosses applications grâce à la structure qu'ils apportent.

Par langage à objet, on parle généralement de langages à classes (par opposition aux langages à prototype comme Javascript). Le concept de base de l'objet est qu'un objet est instance d'une classe. La classe représente l'ensemble des opérations (les méthodes) que peut effectuer l'objet ainsi qu'une description de ses données (les attributs). L'objet encapsule ses données, qui sont accessibles au monde extérieur à travers les méthodes.

Un programme a besoin d'un système d'exécution pour être exécuté et fournir le résultat attendu. On peut classer en trois catégories les systèmes d'exécution : les interpréteurs, les compilateurs et les machines virtuelles. Une machine virtuelle peut être vue comme un système intermédiaire entre l'interpréteur et le compilateur. Ce chapitre contient deux parties, la première partie présente les langages à objet ainsi que l'historique de leur apparition. La seconde partie présente les différents systèmes d'exécution, en particulier les machines virtuelles, ainsi que leurs différentes problématiques.

2.2 Les langages à objet

2.2.1 Historique

Les paradigmes de programmation sont nombreux et sont apparus au fil de l'histoire des langages de programmation. Chaque paradigme s'avère bien sûr plus ou moins efficaces pour modéliser un problème particulier. Mais les styles de programmations sont aussi fluctuants au cours du temps. La programmation fonctionnelle par exemple, a connu ses heures de gloire du temps de Lisp et Scheme puis Smalltalk, ces langages sont ensuite devenus moins populaires avec l'avènement d'autres langages, en particulier orientés objet.

Aujourd'hui la programmation fonctionnelle semble revenir au goût du jour en étant une caractéristique présente dans plusieurs nouveaux langages (Rust, Javascript, Scala).

L'origine des langages orientés objet remonte aux années 1960. Simula est le premier langage orienté objet, sa première version est disponible en 1967. Ce langage permet de programmer avec des classes, le polymorphisme et l'héritage sont également présents.

Smalltalk [Ingalls, 1978, Goldberg and Robson, 1983] est créé ensuite dans les années 1970 et parfait le concept de programmation orientée objets. Les années 1980 voient l'objet se populariser avec plusieurs nouveaux langages : Objective-C (1983), C++ (1983), Eiffel (1984). Parallèlement Lisp s'enrichit d'une couche objet avec CLISP (Common Lisp Object System), ce langage permet également la méta-programmation.

Les années 1990 sont ensuite marquées par l'arrivée du langage Java en 1995, qui sera très largement diffusé et utilisé. C♯ est sorti quelques années après, en 2001 et présente des caractéristiques très similaires à Java.

2.2.2 Caractéristiques

Les langages à objet doivent leur popularité aux facilités offertes pour modéliser le monde réel. Le modèle objet introduit en outre de nombreux concepts (classes, héritage, encapsulation...) qui facilitent la réutilisation et la structuration des applications. De manière générale, les langages à objet augmentent la réutilisabilité d'une application car un programme avec des classes est facilement extensible. De plus, le découpage de l'application en classes augmente la modularité, les concepts du programme sont isolés et contenus dans la classe, qui est l'unité de code de ces langages. La compréhension globale de l'application est ainsi facilitée par rapport à un langage non-objet.

Le paradigme objet est rapidement explicable par avec une métaphore du monde réel, cet élément a sans doute joué dans l'adoption de ce paradigme.

Exemple : On peut décrire de manière abstraite une table et donc en définir la classe comme possédant un plateau, un certain nombre de pieds etc. Un objet représentera une instance, et donc dans ce cas, une table particulière. Cette table aura une couleur, une forme et une hauteur qui lui sont propres. La modélisation est donc partagée en deux : la partie abstraite est contenue dans la classe, la partie concrète est dans les objets instances des classes.

L'héritage est aussi une notion facilement explicable, puisque comme noté dans [Ducournau, 2016], le modèle de classification des espèces naturelles peut être représenté par de l'héritage et des objets de façon intuitive.

2.3 Définitions et cadre de la thèse

2.3.1 Langages à classe et prototype

Parmi les langages orientés objet, plusieurs variantes existent. Tout d'abord, il faut distinguer les langages à classe des langages à prototype.

Définition 1. Langage à prototype Les classes n'existent pas dans ce système. La réutilisation est effectuée en dérivant un objet d'un autre objet existant.

Définition 2. Langage à classe Les classes sont les entités structurantes du langage. Elles contiennent la définition abstraite de leurs instances, avec les données (attributs) et les opérations (méthodes). Les objets sont ensuite créés à partir des classes et ils possèdent et encapsulent leurs propres données.

La très grande majorité des langages à objet sont basés sur des classes. Parmi les langages à prototype, Javascript est le plus connu aujourd'hui, le premier d'entre eux qui a été largement diffusé étant Self [Ungar and Smith, 1987a], d'autres langages ont existé, par exemple Garnet. Dans le cadre de ce projet, nous travaillerons exclusivement sur les langages à base de classes, qui sont majoritaires parmi les langages à objet.

2.3.2 Typage dans les langages à objet

Le typage statique est un système dans lequel les types sont explicitement écrits dans le programme, la vérification des types est ensuite effectuée à la compilation. Le typage dynamique déporte la problématique de vérification des types à l'exécution. Les programmeurs n'ont pas à les préciser dans le programme, mais cela ne signifie pas que les types n'existent pas ou ne sont pas vérifiés.

Parmi les langages à objet, ces deux formes de typage existent avec toutefois une plus grande présence du typage statique. Généralement, un langage ne fait pas cohabiter ces deux systèmes de types. Par contre, un langage typé statiquement peut avoir des vérifications de types à effectuer à l'exécution à travers les casts ou les tests de sous-typage. Dans les langages orientés objet en typage dynamique, on peut citer :

— Python (et une machine virtuelle python [Rigo and Pedroni, 2006])

— CLOS [Bobrow et al., 1988]

— Ruby [Flanagan and Matsumoto, 2008]

Les langages à objet en typage statique sont eux plus largement utilisés :

— Java [Gosling, 2000]

— C♯ [Hejlsberg et al., 2003]

— C++ [Stroustrup, 1995]

— OCamL [Leroy et al., 2014]

— Eiffel [Meyer, 2002]

— Rust [Matsakis and Klock II, 2014]

— Ada [Barnes, 2006]

Nous allons nous concentrer sur les langages en typage statique, comme Java, C♯ et C++, qui n'est pas complètement inféré comme l'est le système de types de OCamL. Dans les langages considérés, un type est une classe.

2.3.3 Héritage et langages à objet

L'héritage est une caractéristique importante des langages à objet. Tous les langages n'offrent cependant pas les mêmes possibilités vis à vis de l'héritage.

Dans les langages à objet l'héritage est de deux sortes.

Définition 3. Héritage simple Une classe peut avoir une seule superclasse directe.

Définition 4. Héritage multiple Une classe peut avoir plusieurs superclasses directes.

Le modèle des interfaces à la Java peut être vu comme une forme dégradée d'héritage multiple. Les interfaces ne possédant pas de code dans les méthodes ni d'attributs, l'héritage n'est pas réellement multiple. Cette caractéristique est appelée sous-typage multiple. Les langages Java, C♯ et Ada sont en sous-typage multiple.

Définition 5. Sous-typage simple Le sous-typage simple signifie qu'un type (une classe) peut avoir un unique super-type direct.

Définition 6. Sous-typage multiple Une classe peut avoir une seule super-classe directe mais elle peut implémenter plusieurs interfaces. On parle de sous-typage multiple car les méthodes des interfaces ne possèdent pas de code et les interfaces servent essentiellement à typer.

Dans les langages à objet en typage statique, C++ est le principal langage en héritage multiple. Dans le cadre de la thèse, nous avons choisi d'étudier des langages en héritage multiple.

2.3.4 Monde ouvert et monde fermé

L'introspection et la réflexivité dans les langages permettent d'avoir accès au modèle du programme pendant l'exécution. L'introspection permet généralement d'avoir uniquement accès au modèle tandis que la réflexivité va plus loin en permettant de définir de nouvelles entités dans le code. Certains langages permettent même de charger de nouvelles unités de code (des classes) de manière programmatique pendant l'exécution du programme. L'introspection de Java offre cette possibilité.

Définition 7. Chargement dynamique Le chargement dynamique est le fait d'autoriser l'ajout de nouvelles classes au programme pendant son exécution.

Le chargement dynamique est une caractéristique des langages qui a des répercussions fortes sur l'implémentation dont nous parlerons ultérieurement. Le chargement dynamique entraîne un fonctionnement en *monde ouvert* (*Open-World Assumption*) de l'exécution d'un programme, l'ensemble de classes grandit dynamiquement. Il est par exemple possible dans certains langages de récupérer une classe sur une autre machine pendant l'exécution (Java par exemple). Par opposition, un langage qui ne permet pas ces fonctionnalités est généralement compilé en *monde fermé* (*Closed-World Assumption*).

2.4 Les machines virtuelles

Un système d'exécution permet d'exécuter un programme écrit dans un langage de programmation sur une machine. Pour cela, il faut passer du langage de programmation, qui est plutôt proche de l'humain (en tout cas du programmeur) jusqu'à la machine, qui a une représentation très basique avec des opérations arithmétiques très simples. Le processeur est l'unité calculatoire d'un ordinateur et possède son propre langage : l'assembleur.

Les différents systèmes d'exécution adoptent des stratégies diverses mais le but reste le même : il s'agit d'exécuter le programme.

2.4.1 Définitions

Ces systèmes peuvent être catégorisés en trois grandes familles :

— Compilateur

— Interpréteur

— Machine virtuelle

Définition 8. Compilateur Un compilateur est un programme qui lit un autre programme, écrit dans un langage de programmation, le langage *source* et qui le traduit en un programme équivalent dans un autre langage, le langage *cible* [Aho et al., 2007].

Les **compilateur globaux** compilent l'intégralité du programme en une seule étape, en ayant donc connaissance de toutes les classes. Généralement, ils nécessitent de recompiler l'intégralité d'un programme si un changement est effectué, Eiffel possède un compilateur en monde fermé.

Les **compilateurs séparés** (langage C++) sont composés de deux parties :

— Le compilateur : traduit le programme d'un langage cible à un langage source, toutes les adresses ne sont pas résolues.

— L'éditeur de liens : assemble les différentes portions du programme pour créer l'exécutable. Les adresses qui n'étaient pas résolues le sont à l'édition de liens.

Un compilateur génère généralement du code de plus bas niveau (C ou assembleur) que le langage source. La compilation est cependant coûteuse en temps : le programme compilé est efficace mais du temps doit être investi lors de la compilation.

Définition 9. Interpréteur Un interpréteur produit directement l'exécution du programme source en fonction des données fournies, sans générer de code assembleur.

La compilation d'un programme a un coût, la somme du temps de compilation et de l'exécution n'est pas forcément toujours rentable par rapport à une simple interprétation. À l'inverse, l'interprétation d'un programme est une opération rapide pour des programmes qui s'exécutent linéairement ou des programmes dont l'exécution est simple. Pour un programme complexe et/ou exécuté de nombreuses fois, la compilation sera plus

efficace que l'interprétation et sera donc un bon investissement. Cette caractéristique entraîne des machines virtuelles à adopter une approche mixte basée sur un interprétateur ainsi que plusieurs niveaux de compilations. Un système d'exécution peut donc mélanger les deux approches.

Définition 10. Machine virtuelle Les machines virtuelles ont un modèle hybride entre ces deux systèmes. Le programme en entrée d'une machine virtuelle est généralement dans un langage intermédiaire, appelé *bytecode*. Généralement, les machines virtuelles font de la compilation partielle ou totale de morceaux de programme avant de les exécuter, et ce pendant l'exécution.

Les machines virtuelles découvrent donc le programme au cours de l'exécution, elles permettent donc de mettre en œuvre le chargement dynamique et l'introspection. Mais il s'agit aussi d'une difficulté pour les implémenteurs de machine virtuelle car le chargement dynamique empêche d'effectuer les optimisations les plus efficaces (optimisations globales) à l'inverse des compilateurs.

2.4.2 Historique

Le concept de machine virtuelle (souvent abrégé VM pour *Virtual Machine*) est apparu dans les années 70 d'après [Gough, 2001]. Il s'agit de gagner en portabilité en compilant un programme dans un langage intermédiaire qui sera ensuite exécuté par la machine virtuelle qui elle-même aura été implémenté pour plusieurs plateformes. L'idéal est bien sûr d'être totalement indépendant de la plateforme cible du point de vue du langage intermédiaire. De plus, la machine virtuelle assure qu'un même programme fonctionne de manière identique sur plusieurs machines, et que ce programme soit correct, à travers de nombreuses vérifications lors de l'exécution du programme.

Le langage Smalltalk a utilisé une machine virtuelle dans les années 1980. Ce langage était le premier à la fois orienté objet et utilisant une machine virtuelle. Suite à cela, le langage de programmation Self [Ungar and Smith, 1987b] est sorti en 1987. Là encore, ce langage était orienté objet et utilisait une machine virtuelle.

De nombreuses techniques introduites pour implémenter le langage Self ont resservi pour spécifier le langage Java. Java sort dans les années 1990 : Sun conçoit le langage Java et spécifie la *Java Virtual Machine*, Microsoft l'imite en 2000 avec les langages fonctionnant avec .NET et le CLR (Common Language Runtime). Les performances de ces deux machines virtuelles sont assez proches [Singer, 2003]. La différence réside dans la philosophie de conception. La JVM n'est pas vraiment ouverte à d'autres langages que Java ni à des paradigmes différents. Le CLR a été prévu pour supporter plusieurs langages, y compris ceux venant de la programmation impérative (manipulations de pointeurs). Les deux plateformes restent toutefois assez proches et montrent que les machines virtuelles sont l'un des avenirs des langages de programmation de par leur portabilité et les avantages qu'elles offrent. Ces deux systèmes chargent les classes pendant l'exécution et les compilent à la volée : Compilation *Just-In-Time* (abrégé JIT). Le monde ouvert participe aussi à la réutilisabilité, c'est-à-dire qu'une classe est conçue indépendamment de ses usages futurs (en termes d'héritage par exemple). Mais ces avantages ont un coût,

le monde fermé de Eiffel permet intrinsèquement de meilleures performances. En effet, la connaissance lors de la compilation de l'ensemble du programme permet un niveau d'optimisation plus élevé. L'idée de la compilation JIT a des origines assez lointaines, d'après [Aycock, 2003] le premier article évoquant l'idée concerne LISP, il est écrit par McCarthy en 1960 [McCarthy, 1960]. Un système bash utilisé dans plusieurs universités américaines nommé *University of Michigan Executive System* contenait dans sa documentation technique une information importante : il était mentionné que l'assembleur et le chargeur du système pouvaient être utilisés pendant l'exécution. Il est possible de faire le parallèle avec le chargement dynamique et la compilation à la volée. Au fil du temps, ces idées se sont ensuite répandues et ont été considérablement développées.

2.4.3 Langage de bytecode

Le code en entrée de la machine virtuelle est produit depuis un langage source (Java, C#) qui est transformé en un langage intermédiaire de plus bas niveau. Ce langage est ensuite interprété par une machine virtuelle spécifique, dans le monde Java ce langage est appelé *bytecode*. Le terme de *bytecode* qu'on peut traduire par code à octet vient du fait que les instructions de ce langage sont encodées sur des octets. Aujourd'hui *bytecode* est un terme qui sert à désigner l'ensemble des langages qu'exécutent les machines virtuelles.

Il y a plusieurs types de compilations possibles pour le *bytecode* : le code peut être compilé à la volée (*Just-In-Time*) vers du code machine, ou interprété directement. Une autre approche intermédiaire existe, c'est celle choisie par la machine virtuelle Java de référence, HotSpot, qui utilise un mélange entre compilation et interprétation [Kotzmann et al., 2008]. Le programme commence par être interprété jusqu'à que la machine virtuelle détecte une portion de code nécessitant d'être optimisée. Ces portions de code seront compilées à la volée ainsi qu'optimisées. Les concepteurs de la machine virtuelle appellent de telles portions *hotspots*, c'est d'ailleurs l'origine du nom de la JVM de Sun/Oracle.

2.4.4 Caractéristiques

Selon les systèmes considérés, les caractéristiques varient. Cependant, la plupart des machines virtuelles possèdent les propriétés suivantes :

— Chargement dynamique

— Gestionnaire automatique de la mémoire

— Compilation à la volée

Définition 11. Chargement dynamique : Les classes du programme ne sont pas connues au début de l'exécution. Elles sont découvertes au fil de l'exécution, lors de l'instanciation d'un objet d'une classe pas encore chargée. Le chargement dynamique peut être rendu obligatoire par les spécifications d'un langage. Un langage réflexif devra être en chargement dynamique pour pouvoir charger les classes en passant par l'introspection.

Définition 12. Gestionnaire automatique de la mémoire : Un mécanisme qui libère automatiquement les objets inutilisés. De fait, le programmeur dans le langage

source n'a pas à gérer explicitement la mémoire, ce mécanisme est également nommé ramasse-miettes (*Garbage Collector* en anglais).

Définition 13. Compilation à la volée : Génère du code machine pendant l'exécution du programme. L'opération est généralement réalisée de façon paresseuse en compilant à la demande une méthode nouvellement rencontrée.

Ces trois caractéristiques ont des conséquences sur l'implémentation des machines virtuelles. De plus, ces mécanismes fonctionnent alors que le programme s'exécute, l'impact n'est pas négligeable sur les performances. Il faut alors interrompre l'exécution pour charger des classes ou compiler des méthodes.

2.4.5 Avantages et inconvénients des machines virtuelles

Les machines virtuelles ont quelques avantages sur les compilateurs classiques, mais aussi des inconvénients :

— La portabilité : le langage intermédiaire (bytecode) et le compilateur du langage source vers le bytecode permettent de ne pas se préoccuper de la portabilité vers différentes architectures. Il faut par contre implémenter une nouvelle machine virtuelle sur chaque système et architecture.

— Réduction de l'empreinte mémoire des exécutables : un programme compilé dans un langage de bytecode est généralement moins lourd qu'un programme compilé sous forme binaire. Il faut cependant prendre en compte le code produit par le compilateur à la volée. De plus, une machine virtuelle est cependant un programme complexe (donc lourd) qui doit accompagner les fichiers de bytecode.

— Mise en œuvre du chargement dynamique et monde ouvert. Les machines virtuelles permettent d'implémenter l'introspection et la réflexivité.

— Sur les machines virtuelles il est possible de récupérer du code via le réseau avec le chargement dynamique. Intrinsèquement, les machines virtuelles se prêtent bien aux systèmes dynamiques ou distribués (une architecture orientée service par exemple).

— Généralement dans les machines virtuelles, la gestion automatique de la mémoire est supportée. Il est également possible d'avoir un ramasse-miettes dans un langage utilisant un compilateur classique, bien que la majorité de ces langages ne choisissent pas cette approche.

— Possibilité d'avoir plusieurs stratégies d'exécution du code (interprétation, compilation, mixte) au sein d'une même machine virtuelle en fonction des performances souhaitées. La séparation des composants dans la machine virtuelle facilite aussi le développement de différentes stratégies de compilations.

2.4.6 La recherche dans les machines virtuelles

Depuis quelques années, la recherche dans le domaine des machines virtuelles s'est développée. Les langages non-objet et/ou à typage dynamique ont maintenant une place grandissante dans le monde scientifique, en particulier avec Javascript.

L'adoption croissante de Javascript sur les sites webs a conduit de nombreux chercheurs et entreprises à faire de la recherche pour optimiser l'exécution du langage.

De nombreuses machines virtuelles Javascript existent, V8 [Google, 2016] est la machine virtuelle Javascript actuellement utilisée par Google dans son navigateur. SpiderMonkey est la machine virtuelle Javascript utilisée par Mozilla dans Firefox. Des recherches ont été effectuées pour utiliser des compilateurs à la volée traceurs dans SpiderMonkey [Gal et al., 2009]. Les problématiques sont différentes des machines virtuelles pour des langages à objet, mais de nombreuses nouvelles techniques sont utilisées dans ce domaine. Ces recherches sortent du cadre de notre projet mais il est indéniable que la recherche scientifique dans ces domaines est très active aujourd'hui.

2.5 Les machines virtuelles pour des langages à objet à typage statique

2.5.1 Java Virtual Machine (JVM)

Initialement la JVM a été conçue pour exécuter du *bytecode* issu lui-même du Java. La JVM répond à une spécification, celle du langage de bytecode [Lindholm et al., 2012]. Puis, quelques années plus tard, certains concepteurs ont l'idée d'utiliser la JVM pour exécuter d'autres langages que Java, comme par exemple SCALA [Odersky, 2009]. Cette machine virtuelle prévue initialement pour des langages à objet à typage statique a également évolué avec la multiplication des langages fonctionnant sur la JVM. Une instruction spéciale pour appeler des méthodes dynamiquement a été rajoutée il y a peu. L'instruction `invokedynamic` [Rose, 2009] présente des défis d'implémentations décrits dans [Thalinger and Rose, 2010].

Ceci constitue à peu près la seule ouverture du Java à d'autres paradigmes, les spécifications de la JVM changeant assez peu depuis l'origine pour des questions de rétro-compatibilité notamment. L'essence de la JVM étant une spécification, plusieurs implémentations existent. La machine virtuelle Java de référence HotSpot existe par exemple à travers une version clients et une version serveurs pour lesquels les problématiques sont différentes.

Les caractéristiques essentielles de la JVM sont :

— Machine à pile
— Machine sécurisée (typage sûr, pas de manipulations de pointeurs, vérifications au chargement des classes pour éviter les incohérences et s'assurer que le typage est correct)
— Gestion automatique de la mémoire (*Garbage Collector*)
— Instructions du bytecode orientées objet
— Chargement dynamique
— Gestion du multitâches

Étant donné que la JVM est une sorte de processeur complexifié, des chercheurs se sont penchés sur l'implémentation d'un processeur dédié à l'exécution des instructions

bytecode, ceci dans le but d'accélérer l'exécution de Java. Sun a travaillé sur le sujet avec picoJava [O'connor and Tremblay, 1997]. Dans [Kent and Serra, 2000], les auteurs développent l'idée d'un processeur dédié à exécuter des instructions *bytecode* qui viendrait compléter le processeur classique qui effectue les tâches généralistes.

Dans l'article, la JVM est décrite comme ayant deux grandes composantes :

— les instructions de bas niveau

— le chargement de classes, ramasse-miettes et autres opérations de haut niveau

Le premier point peut être implémenté côté matériel tandis que le deuxième est trop complexe pour l'être seulement au niveau matériel. Les opérations constantes, arithmétiques, les manipulations de piles en passant par les chargements/sauts/comparaisons sont traités du côté matériel, il s'agit plus généralement de ce qui existe déjà dans les processeurs classiques.

Leur processeur dédié à exécuter du Java est capable de gérer l'instanciation des objets, la synchronisation, les appels de méthodes et l'accès aux attributs dans des cas simples qui n'entraînent pas de chargement dynamique de classes.

Toutefois, cette implémentation matérielle semble être une idée qui s'essouffle aujourd'hui bien que des recherches continuent.

Les processeurs dédiés à exécuter un langage ne sont pas des recherches limitées à Java. Dans les années 1980, des machines lisp sont apparues, elles étaient exclusivement dédiées à exécuter ce langage. Les machines Lisp n'étaient pas utilisées massivement, mais ce concept a inspiré les concepteurs de processeurs dédiés à Java. L'arrivée des processeurs généralistes (performants) à bas coût ont mis un terme aux recherches sur les processeurs dédiés.

2.5.2 Le langage Scala

Scala [Odersky, 2009] est un langage à part car il est exclusivement exécuté sur la machine virtuelle Java. Il est à noter que Scala est un langage à objet avec des classes. Mais le langage possède aussi des caractéristiques des langages fonctionnels (fonctions de premier-ordre, fermetures). Ce système a l'avantage d'avoir contribué à la popularité du langage puisque Scala est complètement compatible avec Java. Les premiers programmeurs Scala pouvaient donc utiliser des bibliothèques Java avant d'éventuellement complètement migrer vers Scala.

En contrepartie et d'un point de vue scientifique, l'utilisation de la JVM pour implémenter Scala présente plusieurs inconvénients pour ses concepteurs. Ils doivent en effet implémenter les fonctionnalités du langage en se basant sur ce qu'il est possible de faire dans la JVM. Ainsi, la généricité de Scala est effacée, comme celle de Java alors que ses concepteurs auraient pu implémenter la généricité comme C♯ qui n'est pas tributaire de la compatibilité avec la JVM. La généricité de Java a été implémentée pour être traitable uniquement du côté du compilateur du langage *javac*. La généricité est ensuite vérifié à la compilation uniquement. Dans la machine virtuelle Java, la généricité n'existe pas et aucune instruction la concernant n'a été rajouté au langage de bytecode. Les concepteurs

de Scala ont donc effectué un gros travail dans le compilateur du langage, en permettant par exemple l'inférence de types sans pouvoir modifier le langage de bytecode.

2.5.3 Langage C♯ et CLR

Microsoft a développé suite au succès de Java une architecture complète avec de grandes similitudes. Un format de bytecode sert de support à plusieurs langages, ce bytecode est le CIL *Common Intermediate Language*. L'ensemble des langages de la plate-forme .NET sont compilés vers ce bytecode. Il a la particularité, contrairement à Java, de supporter un large panel de langages. C♯, C++, F♯ ou encore Visual Basic ont des compilateurs vers CLI. La machine virtuelle qui exécute ce langage est la CLR *Common Language Runtime*.

C♯ est probablement le langage le plus connu qui utilise ce système. Le langage est orienté objet, statiquement typé, possède de la généricité ainsi que le même système d'interfaces que Java. Dans les deux cas, l'héritage multiple n'est pas supporté.

Comme noté par [Singer, 2003], il n'y a pas de grande différences entre les langages Java et C♯. De par sa postériorité, C♯ évite en contrepartie quelques erreurs présentes dans le méta-modèle de Java. La généricité a par exemple été implémentée complètement dans le bytecode CIL, à l'inverse de Java qui l'a implémentée uniquement dans le compilateur. La généricité de C♯ est donc plus performante à l'exécution, et plus sûre du point de vue des types. Les concepteurs de Java n'ont pas pu adopter la même approche à cause de la rétro-compatibilité que ses mainteneurs se doivent d'offrir au vu de la position du langage, ce qui parfois limite ses évolutions du langage.

Les bytecodes des deux langages possèdent des similitudes :

— Les deux sont sous forme de langage à pile

— Le modèle objet est similaire : classes en héritage simple et sous-typage multiple via des interfaces

— Les instructions sont relativement similaires

La différence essentielle est que le bytecode .NET a été conçu pour supporter plusieurs langages à l'inverse de Java. Cette différence entraîne également la présence d'opération de très bas niveau dans le langage CLI telles que des manipulations de pointeurs. Comme relevé par [Gough, 2001], le bytecode .NET contient aussi des instructions non-typées pour permettre d'implémenter des langages à typage dynamique sur la plateforme.

Du point de vue de la machine virtuelle, là encore, de grandes similarités existent avec Java. Les machines virtuelles pour CIL sont aussi un support de recherche, [Campanoni et al., 2008] ou encore [Bebenita et al., 2010] sont des exemples d'implémentations.

2.6 Implémentation des mécanismes objet

Un programme orienté objet utilise un grand nombre de mécanismes objets pendant son exécution. Les performances d'un langage sont donc très fortement liées à celles de l'implémentation efficace des trois mécanismes objets de base :

— Appel de méthode

— Accès aux attributs

— Test de sous-typage

L'implémentation de ces mécanismes requiert des structures de données spécifiques. De nombreuses implémentations des mécanismes objets ont été publiés au fil du temps. L'héritage simple est généralement trivialement implémenté dans les langages. Historiquement, l'héritage multiple a par contre toujours été une caractéristique difficile à implémenter efficacement dans les compilateurs. D'ailleurs, la littérature scientifique est assez riche en techniques d'implémentations de ces mécanismes, preuve qu'un réel consensus est difficile.

Le chargement dynamique des machines virtuelles complexifie encore la situation, notamment avec l'héritage multiple.

2.6.1 Sous-typage simple

La figure 2.2 présente une situation d'héritage simple dans un langage à objet. Trois classes sont représentées dans ce schéma : `A`, `B` et `C`. Les boîtes bleues représentent les tables des méthodes de chacune des classes. Pour la classe `A`, on affecte une position à chacune des méthodes introduites dans la classe. Dans la classe `B`, on recopie le bloc de méthodes de `A`. En cas de redéfinitions de méthodes introduites par `A` dans la classe `B`, on remplace l'adresse des méthodes concernées dans la table de méthodes de `B`. Ensuite, on accole le bloc de méthodes de `B` dans la table de méthodes.

Les méthodes qui sont éventuellement redéfinies dans les sous-classes sont donc positionnées à la même position (par rapport au début du bloc) que les autres définitions de la méthode dans les super-classes.

Cette situation introduit un **invariant de position** : une méthode se voit affecter une position, qui ne changera jamais en cas de spécialisation. L'appel de méthode devient trivial à implémenter. Il suffit d'accéder à la table de méthodes du receveur de l'appel, puis d'aller à la position de la méthode appelée pour y récupérer le code de la redéfinition appropriée. Pour les attributs, l'implémentation est similaire. La figure 2.3 présente une séquence de code simplifiée pour réaliser un appel de méthode en héritage simple.

En héritage simple, il est possible d'utiliser la technique du test de Cohen pour les tests de sous-typage [Cohen, 1991]. Cette technique est très efficace.

La technique de Cohen consiste à attribuer un identifiant à chacune des classes ainsi qu'une *couleur* (*offset*). La couleur d'une classe est sa profondeur dans la hiérarchie de classes. À l'origine, le test de Cohen était implémenté dans un tableau de chaque classe. Toutes les superclasses sont présentes dans ce tableau et sont ordonnées par leur couleur. Ce tableau était nommé *display*. Le tableau contiendra autant d'éléments que la classe a de superclasses.

Réaliser un test de sous-typage est alors une opération simple : il suffit d'aller à la position de la couleur de la cible et tester si l'identiant trouvé est celui attendu. Comme expliqué dans [Ducournau, 2008], il est possible d'implémenter le test en inlinant le

```
load [object + #tableOffset], table
load [table + #targetColor], classId
comp classId, #targetId
bne #fail
// succeed
```

FIGURE 2.1 – Code simplifié d'un test de sous-typage avec la technique de Cohen

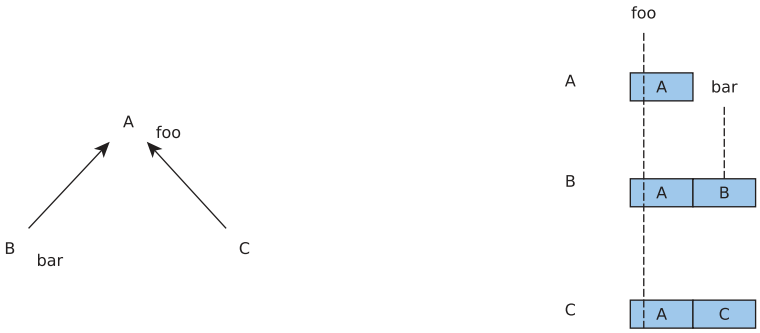

FIGURE 2.2 – Implémentation des méthodes en héritage simple

display dans la table de méthodes sans réaliser de test de bornes. La figure 2.1 présente un code simplifié d'un test de Cohen sans test de bornes.

Le formalisme de la figure 2.3 provient de [Driesen et al., 1995, Driesen and Hölzle, 1995, Driesen, 1999].

2.6.2 Héritage multiple

L'héritage multiple complexifie la situation car l'invariant de position ne peut-être tenu. Avec un schéma d'héritage en losange, les méthodes introduites dans les deux classes latérales ne seront pas à la même position dans la classe du bas de la hiérarchie. La figure 2.4 présente une situation d'héritage multiple. Les méthodes introduites dans

```
A x = new A()
x.foo


load [x + #tableOffset], table
load [table + #fooOffset], methAddr
call methAddr
```

FIGURE 2.3 – Séquence de code pour un appel en héritage simple

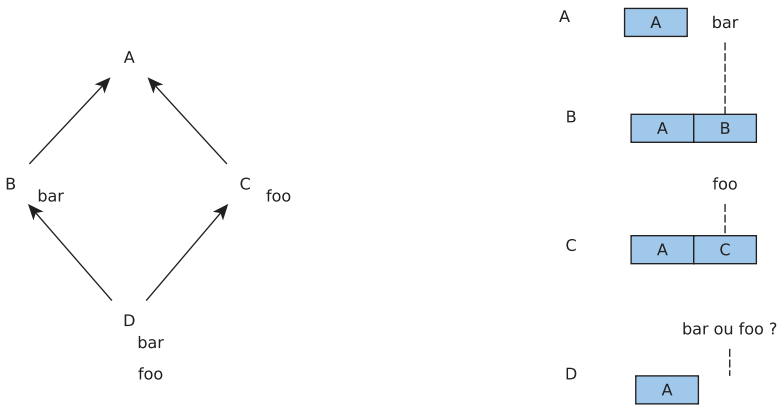

FIGURE 2.4 – Situation d'héritage multiple

les classes `B` et `C` ne peuvent pas toutes être à la même position dans la classe `D`. Dans la classe `D`, on ne peut en effet pas maintenir l'invariant de position pour `bar` et pour `foo`.

Avec cette hiérarchie de classes, il faudra utiliser des techniques d'implémentation objets compatible avec l'héritage multiple. Il faut noter que pour un langage en héritage multiple, il ne faut pas forcément toujours utiliser une technique compatible avec l'héritage multiple pour tous les sites d'appels. Certains appels pouvant très bien être implémentés comme en héritage simple.

2.7 Héritage multiple et machines virtuelles

L'héritage multiple de classes n'est généralement pas présent dans les langages en typage statique exécutés par des machines virtuelles. La plupart des langages s'exécutant sur des machines virtuelles possèdent une forme dégradée d'héritage multiple : le sous-typage multiple.

C'est le cas de Java et C♯, les interfaces utilisées ne permettent pas de définir des attributs dans les interfaces. De plus, les méthodes des interfaces ne peuvent pas contenir de code [1].

L'héritage multiple est fondamentalement plus complet qu'un système basé sur des interfaces. Il est reconnu que l'héritage multiple de classes apporte une facilité de modélisation. Le sous-typage multiple est une caractéristique très souvent présente dans les langages à objet. Les concepteurs du langage Ada ont par exemple rajouté un système d'interfaces avec la sortie de la version 2005.

1. À partir de Java 8, les méthodes des interfaces peuvent contenir du code.

FIGURE 2.5 – Implémentation de l'héritage multiple par coloration

2.7.1 Héritage multiple et chargement dynamique

La situation d'héritage multiple de la figure 2.4 aurait pu être implémenté différemment en monde fermé, c'est-à-dire sans chargement dynamique.

Sans chargement dynamique, une implémentation par coloration aurait pu être utilisé. Cette technique est inspirée du problème de coloration d'un graphe [Gary and Johnson, 1979].

Elle a été initialement proposée par [Dixon et al., 1989] pour l'appel de méthodes. L'accès aux attributs peut aussi être traité par coloration comme illustré dans [Pugh and Weddell, 1990, Ducournau, 1991]. [Vitek et al., 1997] l'utilise pour le test de sous-typage. [Ducournau, 2011a] propose la synthèse de ces différentes techniques autour de la coloration.

La figure 2.5 présente une telle implémentation. Avec cette approche, la classe `B` est positionnée en laissant la place pour le bloc de méthode de la classe `C`. Dans la classe `D`, l'invariant de position est maintenu et les méthodes sont toujours à la même position.

Cette implémentation est utilisable en monde fermé, mais pas en monde ouvert. Lors du chargement des classes `B` et `C`, on ne sait pas encore qu'une classe `D` sera chargée ensuite. Il est donc impossible de prévoir et de construire la table de méthodes de B de manière à préserver l'invariant de position dans `D`.

Le chargement dynamique impose donc d'utiliser des implémentations différentes de la coloration pour l'héritage multiple. Ces implémentations seront nécessairement moins efficaces que la coloration, les systèmes d'optimisations devront donc essayer de palier à cela en effectuant des optimisations.

2.7.2 Mixins et traits

Les mixins [Flatt et al., 1998, Findler and Flatt, 1998] et les traits [Schärli et al., 2003, Ducasse et al., 2006] sont parfois considérés comme des solution miracles permettant d'éviter les problèmes de l'héritage multiple tout en apportant les avantages. Ces caractéristiques des langages à objet permettent en effet une certaine modularité et réutilisation lors de l'écriture de programme. Les mixins sont des unités de code dans lesquelles il est possible de définir des attributs et des méthodes. Les mixins peuvent également être combinés entre eux pour en former des nouveaux. Il n'est par contre pas possible d'instancier un mixin, une classe doit être utilisée. Pour cela, la classe doit hériter du mixin. Les mixins peuvent être vus comme des sortes de classes abstraites.

Dans le langage Scala, qui possède des mixins [Odersky et al., 2008], les classes sont en héritage simple mais peuvent avoir plusieurs mixins. Un mixin peut spécialiser plusieurs mixins mais ne peut avoir qu'une seule superclasse. Par exemple on peut définir une classe `C` tel que `C` spécialise la classe `B` avec le mixin `M`. Une contrainte additionnelle est que l'unique superclasse directe de `M` doit être une superclasse de `B`.

La compilation du mixin permet de mieux comprendre l'implémentation, elle sera effectuée de la manière suivante :

— Une interface `I` est introduite qui contiendra les signatures des méthodes de `M`.

— Le mixin `M` est compilé en une classe abstraite `A` qui contiendra le corps des méthodes du mixin. Ces méthodes seront *statiques* et contiendront un paramètre supplémentaire pour le receveur.

— La classe `C` qui utilise le mixin implémentera l'interface `I` et pour les méthodes de `M`, une méthode identique est intégrée dans `C` qui fera appel à la méthode de la classe abstraite `A` en passant en plus le receveur.

— Les attributs du mixin `M` sont gérés en introduisant des accesseurs dans l'interface `I`. Les attributs sont eux introduits dans la classe `C` et sont accédés grâce aux accesseurs.

À la fin de l'opération, le contenu du mixin est accessible dans la classe `C` avec cette définition. Les mixins permettent donc bien de définir du code à l'intérieur, mais ce ne sont pas de vraies classes. Il s'agit donc plutôt d'une sorte d'implémentation ad hoc de l'héritage par génération de code (copie des attributs, méthodes statiques, génération des méthodes non statiques, interface, etc.). D'autres implémentations que celles décrites pour Scala sont possibles, par exemple [Bracha and Cook, 1990] utilise la généricité hétérogène de C++ pour implémenter les mixins.

Les mixins et les traits ne sont d'ailleurs pas exempts de problème d'héritage multiple. En reprenant l'exemple précédent, le mixin `M` et la classe `B` peuvent tous les deux introduire une même méthode `bar` ou bien redéfinir une méthode héritée depuis une superclasse commune. Des conflits similaires à ceux rencontrés avec l'héritage multiple devront être résolus dans ce cas.

Les traits sont très similaires aux mixins, ils ne peuvent par contre pas définir d'attributs mais peuvent être plus finement combinés entre eux.

2.8 Techniques d'implémentation utilisées dans les machines virtuelles

Cette section présente diverses techniques utilisées pour implémenter les mécanismes objets dans des machines virtuelles. Il s'agit des techniques qui reviennent le plus souvent ou qui sont implémentées dans des machines virtuelles très largement utilisées.

Comme décrit dans [Ducournau, 2008], la technique idéale d'implémentation de l'héritage multiple (pour les trois mécanismes objets) remplit les exigences suivantes :

— Temps constant

— Espace linéaire dans la taille de la relation de spécialisation

— Compatible avec le chargement dynamique

— La séquence de code du mécanismes est suffisamment courte pour être inlinée

2.8.1 Implémentation des interfaces par IMT

La technique basée sur les IMT (*Interface Method Table*) est utilisée pour implémenter l'instruction *invokeinterface* du bytecode Java dans JikesRVM, une machine virtuelle Java de recherche. Une telle instruction est utilisée lorsque le receveur de l'appel est typé par une interface et donc que la méthode appelée a été introduite dans une interface. Cette situation est donc similaire à un appel de méthode dans un langage en héritage multiple.

Cette implémentation est décrite dans [Alpern et al., 2001a]. L'implémentation utilise une table de hachage de taille fixe. À l'intérieur, sont hachées les identifiants des méthodes introduites par des interfaces. La résolution des conflits de hachage est réalisée avec une variante du *separate chaining* [Knuth, 1998] c'est-à-dire une liste chaînée à l'indice du tableau où apparaissent des conflits. Dans cette implémentation les conflits de hachage sont résolus en compilant le code qui résout le conflit et à la place d'une liste chaînée, un arbre binaire de recherche est utilisé. Cet arbre contient des identifiants qui correspondent aux méthodes introduites dans les interfaces de manière incrémentale à la découverte des méthodes.

Cette table de hachage est appelée *Interface Method Table*. L'objectif visé est d'avoir l'efficacité des tables à accès direct mais sans utiliser autant de mémoire. Pour optimiser l'espace utilisé dans les table de hachages, une coloration est effectuée.

Lors de l'appel de méthode il est donc possible d'obtenir directement l'adresse de la méthode ou alors d'effectuer une résolution d'un conflit de hachage en exécutant du code pour cela, le mécanisme n'est donc pas en temps constant.

Cette implémentation est donc relativement efficace tant qu'il n'y a pas trop de conflits ni trop d'interfaces. Néanmoins, un grand nombre d'interfaces ou une hiérarchie de classes avec plusieurs interfaces entraîneront beaucoup de conflits et donc une très forte dégradation des performances. De plus, les tables de hachage sont remplies de manière très inégales : certaines seront presque vides tandis que d'autres auront énormément de collisions.

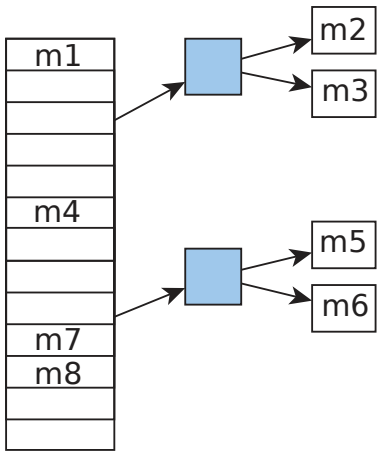

FIGURE 2.6 – Implémentation de la sélection d'une méthode des interfaces dans JikesRVM.

Dans la figure 2.6 ci-dessus, les identifiants des méthodes sont hachés et pointent directement vers la méthode. Dans le cas d'un conflit de hachage, les cases bleues représentent la portion de code compilé permettant de résoudre les conflits entre les méthodes. Plus précisément, la case de la table de hachage contient une arbre binaire plutôt qu'une simple liste chaînée.

2.8.2 Table à accès direct

Une table à accès direct a été utilisée dans SableVM [Gagnon and Hendren, 2001]. Cette table sert à effectuer la sélection des méthodes introduites par des interfaces. À notre connaissance, cette technique n'est pas utilisée dans d'autres machines virtuelles que SableVM. Mais elle a l'avantage d'être la technique la plus efficace en temps au détriment d'une consommation mémoire très importante.

Le postulat de base est d'effectuer une sélection de méthode définie dans une interface aussi rapidement que n'importe quelle autre méthode. Le coût final est donc analogue à celui d'une instruction *invokevirtual* du point de vue du temps uniquement.

La consommation mémoire est par contre importante : la table à accès direct représente une sorte de matrice avec les classes et les méthodes. Les méthodes sont restreintes à celles qui sont introduites par des interfaces implémentées par la classe considérée. Chaque méthode définie dans une interface se voit attribuer un identifiant unique qui servira à indexer la matrice. La table à accès direct est présente dans chaque classe. Cette table sera en grande majorité vide puisque elle ne contiendra que les adresses des méthodes définies dans les interfaces, la taille de la matrice correspondra à l'identifiant le plus élevé de la méthode introduite dans une interface.

Pour effectuer une sélection de méthode, il suffit de récupérer l'identifiant et d'aller

directement à l'indice donné pour appeler la méthode.

La matrice étant essentiellement creuse, il est possible d'utiliser les espaces vides pour allouer des objets à l'intérieur. Pour résumer, cette implémentation est très efficace en termes de temps mais au détriment d'une consommation mémoire démesurée (nombre de classes $\times$ nombre de méthodes des interfaces). La technique ne passe donc pas à l'échelle.

2.8.3 Test de sous-typage dans Hotspot

Hotspot est la machine virtuelle Java de référence, développée par Oracle. Le mécanisme de sous-typage de Hotspot est publié et décrit dans [Click and Rose, 2002].

La technique utilisée reprend le test de Cohen initialement adaptée à l'héritage simple. Des caches sont ajoutés ainsi qu'une extension pour supporter le sous-typage multiple. Du fait de la position de la JVM Hotspot, cette technique est souvent étudiée et reprise bien qu'elle contienne de nombreuses imperfections.

Ce test propose un traitement différent selon que la cible est une classe ou une interface. Ces deux catégories sont définies dans l'article :

1. **primary type** : Classe propre, tableau, type primitif ou tableau de types primitifs
2. **secondary type** : Interface ou tableau d'interfaces

Pour la première catégorie, le test de sous-typage utilisé est un test de Cohen. Un tableau est alloué contenant tous les super-types primaires d'une classe, il est trié du type le plus élevé dans la hiérarchie jusqu'à la classe courante. Ce tableau de super-types est appelé *display*.

Ainsi, si on veut tester si $\mathtt{S} \prec \mathtt{T}$[2]. La position de `T` dans $S.display$ dépend de la profondeur dans la hiérarchie de types de `T`. Le résultat est immédiat et en temps constant, si `T` n'est pas à la place prévue le test renvoie faux.

Mais, d'après les auteurs cette première version du test implique de faire un test des bornes du tableau pour vérifier si le type cible n'est pas à une position plus grande que le tableau, ce qui est assez coûteux. Les auteurs introduisent donc un *display* de taille fixe. La taille est choisie à l'origine en se basant sur les benchmarks specJdb 98, qui constituent une référence à l'époque. Ils observent que la profondeur moyenne de la hiérarchie de types (donc le *display*) est 5. Finalement, la taille 8 est arbitrairement choisie pour avoir de la place supplémentaire.

Bien entendu, une telle limite est fortement discutable. La taille de la hiérarchie des classes aujourd'hui dépasse souvent ce chiffre et ralentit le système. La limite a donc été augmentée depuis en passant à 13.

Avec ces limites de test de bornes du tableau, la définition des types primaires et secondaires est enrichie :

1. **restricted primary type** : type primaire si la profondeur de son *display* n'excède pas 8

2. Si `S` est sous-type de `T`

2. **restricted secondary type** : type secondaire (interface), ou type primaire avec une profondeur dans la hiérarchie supérieure à 8

Voici le schéma de ce test de sous-typage :

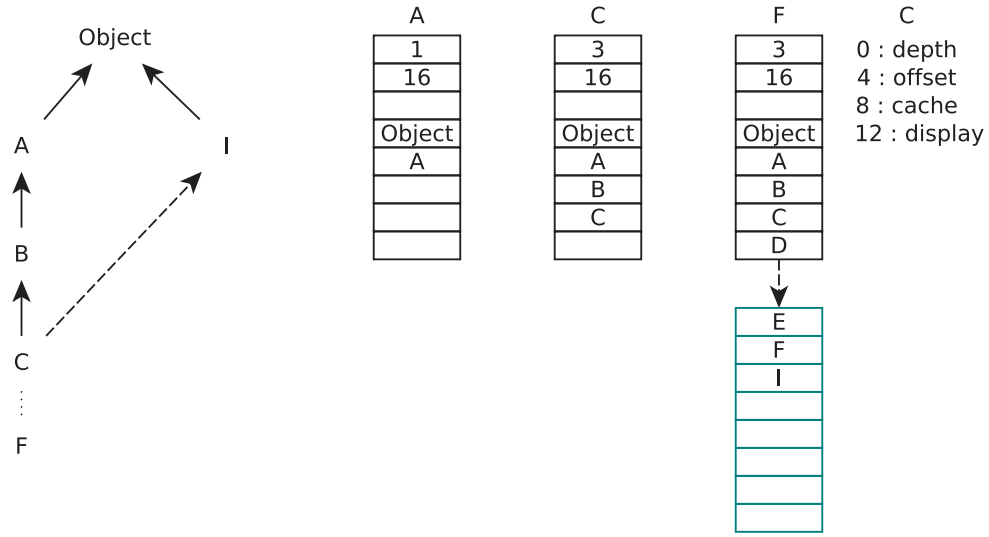

FIGURE 2.7 – Schéma du test de sous-typage dans Hotspot.

Ce tableau secondaire est donc constitué des interfaces implémentées par la classe ainsi que des classes dont la profondeur est supérieure à 8. Pour les grandes hiérarchies de types le test de sous-typage effectuera toujours une recherche linéaire. Un test de sous-typage avec une cible qui est une interface (ou une classe profonde dans la hiérarchie) entraînera une recherche linéaire dans un tableau.

Pour des tests dans le tableau secondaire, la propriété du temps constant est perdue. Pour limiter le coût de la recherche linéaire, des caches sont utilisés. Un mécanisme est introduit pour déterminer rapidement si le test est primaire ou secondaire. Pour cela, les auteurs utilisent un champs $offset$ dans la table de méthodes. Ce champs contient soit $display[T.depth]$ pour un type primaire ou alors l'offset du cache pour un type secondaire.

Le tableau 2.8.3 compare la résolution d'un test de sous-typage par rapport à une cible type primaire (classe, type primitif ou tableau de types primaires) et secondaire (interface, tableau d'interfaces).

Pour résumer, ce test est :

— Pas compatible avec l'inlining, la séquence de code en cas d'un test dans le tableau secondaire est trop longue

— Compatible avec le chargement dynamique

— Espace linéaire car le tableau principal est de taille fixe, le tableau secondaire est lui de taille linéaire

— Compatible avec le sous-typage multiple

— En temps non-constant

Les cas les plus défavorables sont ceux impliquant des classes avec une grande hiérarchie et des tests par rapport à des interfaces. En pratique, ces cas défavorables sont assez rares selon eux grâce au mécanismes des caches.

La consommation mémoire de cette technique n'est pas négligeable, comme relevé dans [Ducournau, 2011b], la taille des tables dépasse en moyenne celle des tables de hachages utilisées dans le hachage parfait.

2.8.4 Test de sous-typage basé sur des trits

L'article [Alpern et al., 2001b] présente le test de sous-typage implémenté dans JikesRVM.

Les types dans JikesRVM sont représentés par un *VM_Type*, ces derniers sont divisés en trois catégories emphVM_Primitive (type primitif), *VM_Class* (classe propre ou interface) et *VM_Array* (tableau). L'implémentation du test de sous-typage sera différente selon le *VM_Type*.

Il est précisé que le système d'optimisations tentera de supprimer le plus possible de test de sous-typage. Selon le niveau d'optimisation de la machine virtuelle, ce test est d'ailleurs systématiquement inliné ou bien exécuté dynamiquement. Les tests de sous-typage avec une cible qui est une classe finale [3] ne peuvent être positifs que si la source est égale à la cible.

Le fonctionnement du test de sous-typage avec des cibles qui sont des interfaces repose sur un système de *trit* (pour *Tertiary digit*). Un trit peut avoir trois valeurs : *yes*, *no* et *maybe* [4]. Chaque classe possède un tableau indexé par les identifiants des interfaces et contenant des trits. Chaque case du tableau contient le résultat du test de sous-typage par rapport à l'interface. Lors de la création d'une classe, son vecteur de trits pointe vers un tableau global rempli de valeurs *maybe*. Lors du premier test de sous-typage, le système détecte que la classe référence ce tableau global. Le système alloue alors un nouveau vecteur de trits (de taille fixée arbitrairement) et le remplit de valeurs *maybe*.

Ensuite, le test de sous-typage demandé est exécuté en interrogeant le modèle des classes stocké dans la machine virtuelle. Le résultat du test (*yes* ou *no*) est stocké à l'indice correspondant à l'identifiant de l'interface.

Si un test de sous-typage est provoqué par rapport à une interface dont l'identifiant est plus grand que les bornes de ce tableau de taille fixe, un nouveau tableau est alloué. Il est ensuite rempli de valeurs *maybe*, avant de stocker le résultat du test de sous-typage. Une telle opération de réallocation fait que la séquence de code est trop longue pour être inlinée.

3. Une classe finale est une classe qui n'a pas de sous-classes, également appelé *frozen* ou *sealed* selon les langages.

4. Le langage Eiffel possède une implémentation similaire

Pour économiser de la mémoire et optimiser le système, les interfaces les plus utilisées (*Cloneable*, *Serializable* par exemple) se voient affecter des identifiants de valeurs proches de 0. Une autre optimisation est de partager un même tableau de trits entre deux classes implémentant les mêmes interfaces.

Le système est généralement efficace par le jeu des numérotations favorables aux interfaces souvent utilisées, ainsi que grâce aux allocations de tableaux supplémentaires de manière paresseuse. Néanmoins, cette implémentation pour un programme avec beaucoup d'interfaces n'est pas optimale. Dans le pire des cas, la technique ne passe pas à l'échelle, elle est très ressemblante à la technique de la table à accès direct de SableVM.

2.8.5 Hachage parfait

Le hachage parfait ([Ducournau, 2008] pour son utilisation pour le test de sous-typage) est une technique d'implémentation des mécanismes objets s'appuyant sur une table de hachage. Cette table de hachage sera parfaite, c'est-à-dire qu'il n'y aura jamais de collisions.

Définition 14. Hachage parfait [Ducournau and Morandat, 2011] Pour I un ensemble non vide d'entiers. $hash : N \times N \rightarrow \mathrm{N}$ est une fonction telle que $hash(x, y) < y$ et $hash(x, y) \leq x$ pour tous $x, y \in N$. *Le paramètre de hachage parfait* de I est le plus petit $H \in N$ telle que la fonction h qui pour x fait correspondre $h(x) = hash(x, H)$ est injective sur I.

C'est à dire que pour tout $x, y \in I$, $h(x) = h(y)$ implique $x = y$. Cette définition est étendue aux ensembles vides en considérant que $H = 1$ quand $I = \emptyset$.

Définition 15. Hachage parfait de classe [Ducournau and Morandat, 2011] Chaque classe se voit attribuer un identifiant unique lors de sa construction. Ce que l'on cherche à hacher est l'ensemble des identifiants des superclasses de la classe courante. Cet ensemble est forcément immutable puisqu'on ne changera jamais les superclasses d'une classe.

Soit $(X, \preceq)$ une hiérarchie de classe possédant des identifiants de classes injectifs $id : X \Rightarrow N$. Le hachage parfait s'applique pour chaque classe c dans X en considérant l'ensemble $I_c = \{id_d | c \preceq D\}$ qui est immutable (les super-classes d'une classe sont connues et ne changeront jamais). Le paramètre H_c résultant est la taille de la table de hachage (représentée par un tableau) de la classe c. Et pour chaque superclasse d, la table contient id_d à la position $h_c(id_d)$. Toutes les autres positions j contiennent un entier quelconque l tel que $h_c(l) \neq j$.

$hash(x, y) \leq x$ est une contrainte qui n'est pas strictement nécessaire, mais elle a été vérifiée pour les fonctions de hachage testées dans [Ducournau and Morandat, 2011]. Ceci implique que $h_c(0) = 0$ pour tout c. Dans les hiérarchies de classes avec une racine 0 est un identifiant convenable pour la racine et c'est aussi une valeur pour les entrées vides de la table aux positions $j > 0$. Pour les hiérarchies de classes sans racine, n'importe quel entier non-nul peut représenter une entrée vide à la position 0 de la table. Mais cet entier ne doit pas être utilisé pour numéroter une classe.

La fonction ET binaire (& dans la syntaxe C) est choisie comme fonction de hachage car elle représente le meilleur compromis entre plusieurs autres fonctions testées. Le

masque de hachage est ensuite stocké dans les classes et sera utilisé pour comme fonction de hachage en plus de l'identifiant de la classe. La taille de la table de hachage ($masque+1$) est déduite depuis le masque de hachage. Les classes sont donc numérotées dans l'ordre de leur chargement. La table de hachage parfaite est remplie par les identifiants des super-classes. Le hachage parfait garantit ensuite qu'il n'y aura pas de collisions dans la table de hachage. Ceci signifie que l'accès est réellement en temps constant.

Après ces différentes étapes, une implémentation basée sur le hachage parfait est réalisée de la manière suivante :

1. Récupération de l'identifiant de la classe testée (la cible)
2. Récupération du masque de hachage de la source du test (le type dynamique de l'objet)
3. L'opération $masque\&id$ renvoie une position
4. Le test consiste à tester si l'identifiant prévu est situé à la position renvoyée dans la table de hachage

L'espace occupé par les tables reste faible : il est linéaire dans la taille de la relation de spécialisation, cette condition ayant été vérifiée empiriquement. Le hachage parfait remplit les cinq exigences pour le test de sous-typage énoncées dans [Ducournau, 2008].

Cette technique a l'avantage d'être utilisable pour implémenter tous les mécanismes objets : l'appel de méthode, l'accès aux attributs et le test de sous-typage. Elle est néanmoins plus lente que des implémentations en héritage simple, il faut donc essayer d'optimiser pour ne l'utiliser que quand elle est strictement nécessaire.

Le test de sous-typage est implémenté en testant si on trouve son identifiant dans la classe qui est source du test. L'appel de méthode nécessite de récupérer ensuite la position du bloc de classe qui a introduit la méthode appelée. Cette position est récupérée dans la table de hachage également. l'accès aux attributs est implémenté en récupérant la position du bloc de la classe qui a introduit l'attribut en question. Cette position est également stockée dans la table de hachage parfaite.

La figure 2.8 illustre l'utilisation du hachage parfait. Des pointeurs vers les *itable* des classes sont présents dans la table de hachage. Une *itable* est le début du bloc d'une classe dans la table de méthodes. La figure 2.9 présente la séquence de code abstraite pour un appel de méthode avec le hachage parfait.

La technique du hachage parfait impose une structure des tables de méthodes pour être utilisée. La table de méthode est partagée en deux parties : la partie négative contient la table de hachage parfaite et la partie positive contient les méthodes.

La table de hachage parfaite contient à la position appropriée des pointeurs vers la partie positive de la table de méthodes. La partie positive est partagée en plusieurs blocs, chaque bloc est constitué de la manière suivante :

— L'identifiant de la classe est à la première position

— Le Δ (noté `d(A)` sur le schéma) sert à implémenter l'accès aux attributs en héritage multiple. Le δ correspond au décalage du bloc d'attributs introduits par la classe dans la table d'attributs des objets de la classe.

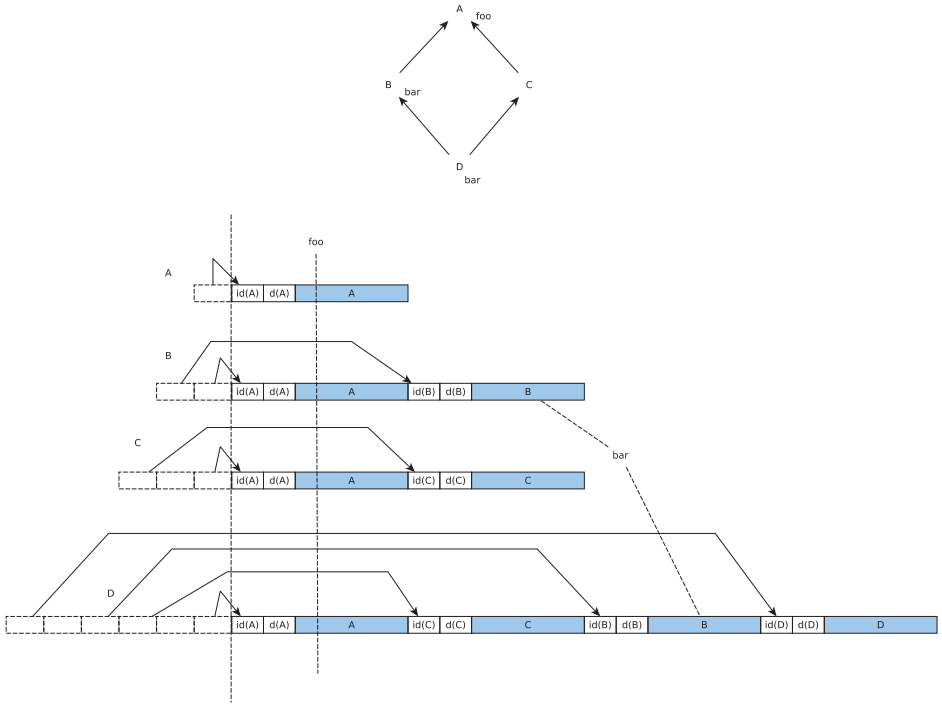

FIGURE 2.8 – Schéma d'implémentation des méthodes avec le hachage parfait

```
B x = new D()
x.bar()

load [object + #tableOffset], table
load [table + #hashingOffset], h
and #AId, h, hv
mul hv, #2*fieldLen, hv
sub table, hv, htable

itable = offset du bloc concerné

load [htable +#htOffset], itable
load [itable +#barOffset], method
call method
```

FIGURE 2.9 – Séquence de code abstraite d'un appel de méthode avec le hachage parfait

— Le code des différentes méthodes de la classe

Il est possible d'utiliser cette technique pour implémenter les interfaces et le test de sous-typage dans une machine virtuelle Java comme montré dans [Pagès, 2013].

2.9 Récapitulatif

Technique	Temps constant	Espace linéaire	Sous-typage multiple	Compatible inlining	Appel de méthode	Test de sous-typage
IMT	-	-	X	X	X	-
Accès direct	X	-	X	X	X	-
Test de Cohen	X	X	-	X	-	X
Trits	X	-	X	-	-	X
Test dans HotSpot	-	X	X	-	-	X
Hachage parfait	X	X	X	X	X	X

TABLE 2.1 – Récapitulatif de quelques techniques d'implémentation utilisées dans les machines virtuelles

Toutes les techniques du tableau sont compatibles avec le chargement dynamique. Le hachage parfait s'impose comme étant une technique efficace, qui remplit les cinq exigences présentées précédemment. Des expériences en attestent, notamment [Ducournau et al., 2009] et [Ducournau and Morandat, 2011] . De plus, c'est la seule technique qui peut être utilisée pour implémenter les trois mécanismes objets : appel de méthode, accès aux attributs et test de sous-typage.

2.10 Conclusion

Dans ce chapitre, nous avons présenté quelques langages à objet existants ainsi que leurs systèmes d'exécution. Les systèmes d'exécution sont divisés en trois catégories : les compilateurs, les interpréteurs et les machines virtuelles.

Nous nous sommes en particulier intéressés à la notion de machine virtuelle. Une machine virtuelle peut être vue comme un mélange entre un interpréteur et une machine virtuelle. Les machines virtuelles sont des systèmes très dynamiques, qui posent des difficultés à implémenter efficacement.

La seconde partie du chapitre est centrée sur les problématique d'implémentations, en particulier des mécanismes objets. Nous présentons des techniques existantes d'implémentations de machines virtuelles pour des langages à objet.

À travers l'état de l'art, nous avons montré que la combinaison machine virtuelle, héritage multiple et typage statique n'existe pas encore dans les différents langages.

Chapitre 3

Systèmes d'optimisations adaptatifs

Contents

3.1 Introduction et problématique

Dans les systèmes d'exécution comme les machines virtuelles Java, le langage de bytecode généré à la compilation n'est pas du tout optimisé. Cela oblige à déporter les problématiques d'optimisations du côté de la machine virtuelle. Par exemple si un appel `(new A).foo()` est présent dans un programme Java, le compilateur traduira en bytecode cette expression par un appel virtuel à la méthode `foo`. Autrement dit, le compilateur ne prend pas la décision d'utiliser un appel statique même si il est évident qu'il est possible de le faire à ce moment là.

Java avait à l'origine la réputation d'être un langage assez peu efficace en termes de performances, en effet, les premières machines virtuelles étaient n'étaient pas optimisées par rapport aux systèmes modernes. Des progrès considérables ont été réalisés depuis plusieurs années, les systèmes d'optimisations des machines virtuelles sont devenus efficaces.

Cependant, dans la littérature scientifique, les systèmes d'optimisations des machines virtuelles sont souvent assez peu décrits. On peut avancer quelques explications à cela :

— Certains nouveaux langages préfèrent être compatibles avec un système existant plutôt que de développer de nouveaux outils et se focalisent plutôt sur les fonctionnalités plutôt que les optimisations

— La complexité croissante des machines virtuelles ne permet pas de décrire facilement un système d'optimisations

— Les plus grosses machines virtuelles sont commerciales (par exemple HotSpot) et les développeurs ne veulent/peuvent pas forcément communiquer dessus

Les machines virtuelles doivent effectuer plusieurs opérations pendant l'exécution (gestion de la mémoire, compilation à la volée...). La question des performances est donc cruciale, les optimisations doivent avoir un bénéfice coût/gain favorable pour être considérées.

Le chargement dynamique complexifie encore le problème. La connaissance du programme n'est totale qu'à l'extrême fin de l'exécution, il faut donc se baser sur une vision partielle de l'exécution pour les optimisations. Ces dernières peuvent aussi être "agressives". De telles optimisations sont valides au moment où elles sont réalisées, mais plus tard, avec d'autres chargement de classes elles peuvent devenir incorrectes et fausser l'exécution du programme.

Il faut donc défaire ces optimisations pour maintenir une exécution correcte. Ces opérations sont appelées dé-optimisations [Hölzle et al., 1992]. Généralement, une dé-optimisation entraîne une recompilation partielle ou totale du code d'une méthode. Les systèmes d'optimisations doivent donc s'adapter en fonction de la situation.

Un système d'optimisations adaptatif doit donc être capable d'effectuer les opérations suivantes en parallèle :

— Optimiser le code selon les informations à sa disposition

— Dé-optimiser si besoin

Ces opérations nécessitent d'avoir des informations sur le code pour prendre des décisions d'optimisations. Dans ce chapitre, on cherchera à donner une définition structurée des différents éléments composant un système d'optimisations dans une machine virtuelle. On présentera quelques optimisations importantes dans les machines virtuelles ainsi que des systèmes complets tirés de la littérature.

3.2 Cadre des systèmes étudiés

Dans les sections suivantes, on considérera des systèmes de type machine virtuelle. Dans ce cadre, le **chargement dynamique** est une caractéristique nécessaire. Cela signifie que les classes ne sont pas connues au début de l'exécution mais qu'elles sont découvertes au fur et à mesures. Le chargement dynamique est d'ailleurs présent au niveau du langage via l'introspection ce qui oblige à l'implémenter dans le système d'exécution.

La compilation des méthodes est également **paresseuse** : les méthodes sont compilées au dernier moment, c'est-à-dire avant leur première exécution. Pour des raisons d'efficacité et compte tenu du chargement dynamique et des optimisations agressives, il est impératif de ne compiler que ce qui est nécessaire. Autrement, ce qui serait compilé en avance pourrait être invalidé avant même la première exécution. On peut par contre imaginer compiler en avance certaines méthodes à la condition de ne pas effectuer d'optimisations agressives.

Pour les machines virtuelles qui effectuent de la compilation des méthodes, la compilation paresseuse est généralement gérée par des **trampolines**. Un trampoline est une portion de code insérée dans les tables de méthodes (à la place de chacune des méthodes ou alors de manière globale).

Lorsque une méthode est exécutée pour la première fois, le trampoline présent dans la table déclenchera la compilation du code. Il s'agit donc d'une portion de code qui appellera le compilateur à la volée, qui compilera le code avant de remplacer l'adresse du trampoline dans la table de méthodes par l'adresse du code maintenant compilé.

3.3 Définition et composantes d'un protocole

Dans la littérature scientifique, le système d'optimisations n'est jamais précisément défini. Le terme de système d'optimisations adaptatif (*Adaptive optimisations system*) revient souvent pour désigner un tel système.

Nous utiliserons le terme de : *protocole de compilation/recompilation* pour désigner un système d'optimisations d'une machine virtuelle. Un protocole de compilation/recompilation est une boîte à outils avec un algorithme associé permettant de sélectionner les bons outils au bon moment. Un protocole est composé d'outils dans les catégories suivantes :

— Techniques de récolte d'informations

— Techniques d'optimisations

— Techniques de réparations suite aux dé-optimisations

Ces outils servent à choisir les implémentations des mécanismes objets du code. Ces mécanismes objets sont au nombre de trois :

— Appel de méthode

— Accès aux attributs

— Test de sous-typage

Dans un programme d'un langage à objet ces mécanismes sont très souvent utilisés, la performance du système d'exécution est donc liée à l'implémentation efficace de ces mécanismes. Par conséquent, il faut choisir les meilleures implémentations pour ces sites selon la situation.

Nous avons cherché à définir et nommer cette notion de protocole de compilation/recompilation pour en faire un objet d'étude. Nous chercherons donc à les comparer entre eux en mesurant leur efficacité. Bien entendu, le coût du protocole en lui même est aussi important : dans un système d'exécution dynamique comme une machine virtuelle, les optimisations trop coûteuses à mettre en place pour des bénéfices réduits doivent être évitées.

3.4 Différences avec des compilateurs en monde fermé

Les compilateurs tels que celui de Eiffel sont dits en *monde fermé* (*Closed-World Assumption*), SmallEiffel [Zendra et al., 1997] est un des compilateurs en monde fermé optimisant d'Eiffel. À la différence des machines virtuelles qui fonctionnent généralement en chargement dynamique, ils ont une connaissance globale du programme exécuté.

Cette connaissance entraîne une différence de taille avec les machines virtuelles puisqu'ils sont capables d'effectuer bien plus d'optimisations. Des stratégies poussées d'optimisations comme [Sonntag and Colnet, 2014] permettent de n'avoir aucun surcoût à l'exécution pour implémenter l'héritage multiple. Les auteurs utilisent un ensemble de techniques d'optimisations effectuant plusieurs passes d'optimisations jusqu'à avoir un état stable, le plus optimisé possible. Ces stratégies s'appuient sur l'analyse des types utilisés dans l'application de manière exhaustive. Il en résulte un exécutable très efficace au détriment du temps de compilation.

Comme montré par [Ducournau et al., 2009], un surcoût est nécessairement entraîné à la compilation pour implémenter l'héritage multiple de manière optimale. Ce n'est cependant pas un problème puisque l'objectif des compilateurs en monde fermé est de produire le code le plus efficace possible à l'exécution. Il est intéressant de noter que les techniques les plus efficaces d'implémentations comme la coloration requièrent des structures de données incompatibles avec le chargement dynamique.

3.5 Analyses statiques et dérivés

3.5.1 Analyses statiques en monde fermé

Les analyses statiques sont nombreuses dans la littérature. En général, les analyses statiques sont conçues pour fonctionner en monde fermé et ne s'appliquent pas au monde ouvert des machines virtuelles. Les analyses les plus poussées sont par ailleurs coûteuses à calculer en terme de temps, ce qui les rendraient de toutes façons difficilement utilisables dans des machines virtuelles. Les analyses peuvent être intra ou inter-procédurales, c'est-à-dire limitées à une méthode ou alors concerner l'ensemble du programme.

Les analyses statiques permettent de déterminer le **graphe d'appel** du programme. C'est à dire savoir quelles méthodes sont appelées à un endroit précis du programme. Cette information permet ensuite d'effectuer de nombreuses optimisations. De même, on cherchera aussi à faire des analyses de types c'est-à-dire de savoir quels sont les types possibles sur un site d'appel (pour le receveur par exemple). Cette information permet de réduire les méthodes appelables sur le site d'appel car l'ensemble de types est plus précis que le type statique du receveur.

CHA, *Class-Hierarchy Analysis* [Dean et al., 1995, Fernández, 1995] est une analyse de la hiérarchie des classes. Il s'agit en fait d'observer le modèle du programme. Les informations sont assez globales, mais on peut par exemple déterminer qu'une méthode n'est jamais redéfinie.

RTA, *Rapid Type Analysis* [Bacon and Sweeney, 1996] est une analyse effectuée en complément de CHA. Elle permet de connaître le programme vivant, c'est-à-dire les classes qui sont réellement instanciées. L'analyse est donc plus précise.

CFA, *Control-Flow Analysis* [Shivers, 1991, Shivers, 1988] est une analyse inter-procédurale très fine mais qui est très coûteuse en temps. Les machines virtuelles ne peuvent pas effectuer une telle analyse car elle nécessite d'avoir l'intégralité du flot de contrôle de l'application. Ces techniques sont donc réservées à des compilateurs statiques.

Interprétation abstraite

L'interprétation abstraite [Cousot and Cousot, 1977] est une théorie de l'approximation de sémantiques de langages de programmation. Plusieurs niveaux de précisions sont possibles avec l'interprétation abstraite [Cousot, 2002].

L'interprétation abstraite permet de simuler des exécutions d'un programme. On est capable de dire si le programme pourra arriver à un endroit précis du code ou encore si un état particulier du programme peut être suivi par un autre état du programme. Plusieurs sémantiques sont disponibles, elles peuvent répondre à certaines questions, mais pas toutes. Il faut donc être capable de sélectionner la bonne sémantique pour répondre à une question particulière, comme par exemple : "est-ce que ce programme termine dans certaines conditions ?".

Il est possible d'utiliser l'interprétation abstraite pour effectuer des analyses statiques. Dans ce cas, il faut généralement faire des approximations des données du programme. Par exemple, des intervalles d'entiers sont utilisés ou encore des expressions

régulières pour les chaînes de caractères.

Plutôt que d'utiliser des données très simples comme des entiers et les valeurs numériques d'un programme, on peut faire des interprétations abstraites sur les ensembles de types [Cousot, 1997].

3.5.2 Analyses statiques en monde ouvert

Certaines analyses statiques simples sont cependant adaptées et utilisables avec le monde ouvert. Généralement, on choisira plutôt des analyses intra-procédurales pour des questions de performances dans des machines virtuelles.

CHA est utilisable en monde ouvert en réalisant l'analyse uniquement sur le code chargé, cette analyse est de plus très légère, elle est très souvent utilisée dans les machines virtuelles. RTA nécessite normalement d'avoir tout le code du programme, mais on peut là encore brider l'analyse au contenu d'une seule méthode ou alors se baser sur le modèle vivant que l'on possède.

Il y a un écart entre des techniques inter et intra-procédurales en terme de temps et de précision. [Qian and Hendren, 2004] présente par exemple des techniques pour effectuer des analyses inter-procédurales dans un contexte de chargement dynamique. Des nuances existent, on peut imaginer utiliser l'analyse CFA bridée au code d'une méthode par exemple.

Cependant, comme montré par [Qian and Hendren, 2005], en monde ouvert, CHA fournit des résultats proches de l'idéal atteignable.

3.6 Récolte d'information

L'information sur le programme exécuté n'est jamais parfaitement complète et exhaustive dans une machine virtuelle avec le chargement dynamique.

Il faut donc récolter des informations au fil de l'exécution sur le programme pour pouvoir prendre les décisions d'optimisations. Plusieurs techniques sont possibles pour cela, le *profilage* permet une mesure fine à l'exécution. Cela consiste à observer l'exécution du programme pour trouver quelles sont les parties souvent exécutées.

Il est aussi possible d'effectuer des analyses statiques pour récolter des informations comme décrit dans la section précédente. Les analyses rapportent des données statiques sur le programme. Le méta-modèle apporte également des informations statiques sur le programme.

Des données peuvent être collectées de manière dynamique à travers le profilage de l'exécution. Cependant, pour être efficaces, ces données doivent être rattachées au méta-modèle, en d'autres termes, elles ne doivent pas être stockées sous forme de chaîne de caractère. On cherchera à enrichir le modèle avec les informations récoltées en les ordonnant de la manière la plus efficace possible.

3.6.1 Profilage de l'exécution

Il y a deux techniques importantes pour effectuer du profilage :

— L'échantillonnage (*sampling*) [JRockit, 2003, Arnold et al., 2000, Binder, 2006] : consiste à inspecter la pile d'exécution pour déterminer les méthodes présentes. Le nombre d'appels de chaque méthode est ensuite extrapolé à partir des méthodes présentes. L'opération doit être réalisée régulièrement pour être pertinente et donner une estimation fiable.

— Les compteurs [Hölzle and Ungar, 1994, Paleczny et al., 2001] : des compteurs d'exécution sont introduits dans le code généré des méthodes. À chaque exécution d'une méthode avec un compteur, ce dernier est incrémenté, on connaît donc de manière très précise le nombre d'exécutions de chaque méthode. Cette technique impose par contre un surcoût à chaque appel.

3.7 Techniques d'optimisations

Dans les machines virtuelles pour des langages à objet, les optimisations sont souvent centrées sur l'efficacité des mécanismes objet. L'appel de méthode est historiquement le principal mécanisme à optimiser. Dans un contexte d'héritage multiple, il faut également optimiser l'accès aux attributs et le test de sous-typage. Dans les machines virtuelles, il faut effectuer des optimisations spéculatives (aussi appelées agressives) pour des questions de performances. Ces optimisations sont correctes sur le moment mais pas forcément durant toute l'exécution. Nous sommes alors amené à effectuer des réparations et des recompilations du code durant l'exécution pour défaire des optimisations.

La *dévirtualisation* est une optimisation importante. Cela consiste à remplacer un appel de méthode classique par un appel statique, le coût du mécanisme de liaison tardive est donc supprimé.

Cet appel statique peut ensuite être transformé en *inlining*. Il s'agit de remplacer la séquence d'appel d'une méthode par le code de la méthode appelée. Le contenu de la méthode appelée est donc placé dans la méthode appelante. Le coût lié à l'appel d'une fonction qui implique d'empiler les paramètres sur la pile, stocker l'adresse de retour, est supprimé. Cette optimisation est classique en compilation et a toujours été très intéressante, elle est naturellement utilisée dans les machines virtuelles.

3.7.1 Dévirtualisation

Dans un programme écrit dans un langage à objet, la majorité des appels de méthodes sont en réalité *monomorphes*. Un appel est dit monomorphe quand une seule méthode est candidate sur le site d'appel. Selon les programmes considérés, le taux de sites d'appels monomorphes varie, mais peut être quantifié à hauteur de 70% environ. Ce taux est par exemple cité dans [Ishizaki et al., 2000] sur un corpus de programmes Java.

La dévirtualisation est une technique souvent utilisée dans les machines virtuelles mais pas nécessairement triviale à mettre en place. En effet, le chargement dynamique n'offre qu'une connaissance partielle de la hiérarchie de classes pendant l'exécution. Il faut donc considérer des techniques "agressives" de dévirtualisation. Cela consiste à

effectuer des optimisations spéculatives, qui peuvent obliger ultérieurement à défaire l'optimisation.

Une étude assez complète des techniques de dévirtualisation a été réalisée dans [Ishizaki et al., 2000]. Les auteurs considèrent deux classifications pour les dévirtualisations :

— Dévirtualisation gardée : la dévirtualisation est effectuée, sans garantie qu'elle soit pérenne.

— Dévirtualisation directe : en fonction de l'analyse complète de la hiérarchie de classes, des dévirtualisations sont effectuées de manière définitive.

La première catégorie est bien sûr compatible avec le chargement dynamique, en entraînant un surcoût lié au test qu'il faut insérer. À l'inverse, la seconde nécessite un compilateur en monde fermé tout en offrant une meilleure efficacité.

Les auteurs de l'étude prônent des dévirtualisations agressives en garantissant la correction du programme via des patchs du code si besoin. Ils se basent sur la connaissance courante de la hiérarchie de classes à travers l'analyse CHA. Pour obtenir des gains de temps d'exécution, la dévirtualisation est couplée à de l'inlining dans leur machine virtuelle.

3.7.2 Inlining

L'*inlining* (expansion en ligne en français) est une optimisation majeure dans la compilation. Cela consiste à remplacer un appel de méthode par le corps de la méthode appelée. Le code de la méthode appelée est donc inclus dans la méthode appelante. Il y a deux conséquences directes de cette optimisation :

— La liaison tardive n'est pas effectuée, plus généralement le mécanisme d'appel de fonction est évité

— La taille du code est bien sûr augmentée

— Le code inliné est alors spécialisé, ce qui permet de l'optimiser encore plus que dans la méthode d'origine

L'inlining est applicable principalement à des sites d'appels de méthodes qui sont statiques. En effet, il n'est pas vraiment raisonnable d'inliner toutes les méthodes candidates d'un site d'appel en mettant des gardes. Cela ferait perdre le bénéfice de l'optimisation en augmentant de manière disproportionnée la taille du code. Il est également possible dans des situations particulières d'effectuer des inlining de méthodes non-statiques : par exemple dans une boucle dans laquelle le type du receveur de l'appel est un invariant de la boucle.

De manière générale, cette optimisation concerne uniquement les petites méthodes, avec peu d'instructions. On peut bien sûr imaginer effectuer un inlining d'une méthode avec beaucoup de code si elle se trouve au milieu d'une boucle qui est exécutée un grand nombre de fois.

L'inlining est difficile à concilier avec le chargement dynamique, car il est compliqué de déterminer les sites d'appels qui sont statiques et qui le resteront (cf la dévirtualisation). Cette technique est souvent couplée à des mécanismes de réparations du code.

L'inlining ne concerne pas forcément que le code des méthodes. [Wimmer and Mössenböck, 2007] présente une technique permettant d'allouer un objet dans un autre objet dans la JVM HotSpot. Le but est d'accélérer l'accès aux attributs de l'objet inliné, si le système détecte que l'objet inliné est accédé uniquement par l'objet englobant. Ces approches sont intéressantes mais nécessitent d'avoir un contrôle très précis sur la gestion mémoire de la machine virtuelle.

Le sens commun du terme concerne cependant les séquences de code plutôt que les objets.

Inlining basé sur le profilage

[Suganuma et al., 2002] présente un système basé sur l'inlining. Leur système est complètement basé sur du profilage de l'exécution plutôt que des analyses statiques.

La décision d'inliner une méthode ou non est discutée. Dans leur système, les méthodes les plus petites telles que les accesseurs sont systématiquement inlinées. En particulier, les méthodes dont le code est environ de la taille du code de la séquence d'appel sont inlinées et ce, sans entraîner une augmentation de la taille du code.

3.7.3 Customisation et spécialisation

La spécialisation de code [Chambers and Ungar, 1989] est une optimisation importante. Comme dit précédemment, on peut spécialiser le code lors d'un inlining ou encore lors d'une optimisation de boucle pour laquelle on connait le type du receveur de l'objet manipulé.

La customisation s'applique par méthode, il s'agit d'une recopie de la méthode dans les sous-classes qui en héritent. Si cette recopie est systématique pour une méthode, alors le type du receveur est toujours connu dans les méthodes recopiées.

En connaissant le type du receveur, tous les appels avec `self` en receveur peuvent alors être optimisés par des appels statiques et de l'inlining. Il est possible d'avoir plusieurs duplications du code de la même méthode qui sera spécialisée pour plusieurs types possibles de `self`. La consommation mémoire est alors augmentée, mais le code spécialisé peut être beaucoup plus efficace que l'original.

3.8 Techniques de réparation

Les optimisations présentées précédemment peuvent être agressives c'est-à-dire qu'elles se basent sur la connaissance courante de la situation sans garantir qu'elles soient toujours correctes à l'avenir. Un chargement dynamique de classe peut alors invalider une optimisation située dans une méthode, il faut défaire l'optimisation, c'est-à-dire recompiler la méthode. Le problème est complexifié quand la méthode concernée est située dans la pile, elle est alors qualifiée d'*active*. Le chargement dynamique peut en effet avoir invalidé la méthode courante. Dans ce cas précis il faut **réparer** le code pour maintenir une exécution correcte. Les techniques de réparation peuvent être divisées en trois catégories :

```
if <guards> then
    <fast-path>
else
    <slow-path>
end
```

FIGURE 3.1 – Schéma général des gardes

— Les gardes

— Le patch du code

— Le patch de la pile

3.8.1 Gardes

Cela consiste à conserver deux versions du code : la version de base non-optimisée et la version optimisée. À l'exécution, un test est effectué pour savoir s'il est possible d'exécuter la version optimisée. Si le test échoue, il suffit d'exécuter l'ancienne version. En fait, ce mécanisme revient à avoir deux chemins dans le code : le *fast-path* et le *slow-path*. Le premier est emprunté si la situation le permet et le deuxième devient obligatoire quand l'optimisation n'est plus valide.

Les gardes provoquent une augmentation de la taille du code (consommation mémoire) en plus d'un test supplémentaire à faire à l'exécution (consommation de temps). Deux types de gardes sont présentés dans [Detlefs and Agesen, 1999] : les *method test* et les *class test*. Les tests de classes consistent à insérer un test avant la séquence de code optimisée. Le test vérifiera que le receveur de l'appel est du bon type. Si oui, alors la séquence contenant l'inlining sera exécuté, sinon la séquence classique qui n'est pas optimisée. La figure 3.2 présente une séquence de code contenant un tel schéma.

Ce schéma a l'inconvénient de ne pas être robuste si jamais l'inlining est possible pour plusieurs types de receveurs. Par exemple, la classe `B` sous-classe de `A` dans l'exemple, ne redéfinit peut-être pas la méthode `foo`. Il y a donc deux alternatives dans ce cas :

— La garde est complexifiée pour fonctionner avec plusieurs receveurs mais au prix d'une efficacité dégradée.

— Utiliser la méthode alternative des *method-test*

La garde sous forme de test de méthode consiste à garder selon l'adresse de la méthode qu'il aurait fallu exécuter avec un appel de méthode non-inliné. Si l'adresse sauvegardée correspond à l'adresse de la méthode appelée sans l'optimisation, alors l'inlining peut-être exécuté.

Cette technique est applicable si plusieurs classes sont candidates à l'inlining sans redéfinir la méthode inlinée, l'efficacité est améliorée dans le cas où plusieurs classes sont candidates. Dans le cas où la méthode inlinée est introduite dans une feuille de la hiérarchie de classes, le test classique est plus efficace (on évite d'effectuer une liaison tardive).

```
fun f(o: A)
do
    if o.methodTable == A.methodTable then
        o.bar() // <fast-path> avec inlining
    else
        o.foo() // <slow-path>
    end
end

...

fun foo(o: A)
do
    o.bar()
end
```

FIGURE 3.2 – Exemple de séquence de code contenant un inlining gardé

Il est possible d'utiliser des *thin guards* [Arnold and Ryder, 2002] pour alléger le mécanisme des gardes et gagner en efficacité. Une variable statique booléenne est utilisée en tant que garde. Quand il faut utiliser le chemin non-optimisé de la garde, cette variable passe à faux.

3.8.2 Patch du code

Le patch du code (*code-patching* en anglais) [Ishizaki et al., 2000] est une technique qui modifie localement le code compilé pour enlever une séquence de code obsolète. Il est possible de s'en servir pour revenir sur une optimisation effectuée.

Un tel patch peut être implémenté lors de la compilation d'une séquence de code. Quelques instructions vides sont positionnées au début de la séquence. Si besoin, la réparation pourra être effectuée de deux manières :

— Ces instructions sont remplacées par une nouvelle séquence de code plus longue

— Un saut inconditionnel est rajouté au début pour aller vers une version correcte du code qui a été pré-compilée.

La figure 3.8.2 illustre un exemple d'un patch de code avec le deuxième mécanisme.

3.8.3 Patch de la pile

Le patch de la pile (*On-Stack Replacement* en anglais [Fink and Qian, 2003, Soman and Krintz, 2006, Steiner et al., 2007]) est une technique qui modifie la pile d'exécution pour changer l'adresse de retour de la méthode. Cette technique est très complexe à mettre en œuvre mais a l'avantage de pouvoir effectuer une réparation "à chaud", c'est-à-dire pendant l'exécution de la méthode à réparer. La pile d'exécution est modifiée

```
<fast-path> // En cas de patch, on remplace par : jump alternative
suite:
    ...
    ...
    return

alternative:
    <slow-path>
    jump suite
```

FIGURE 3.3 – Exemple d'un patch de code

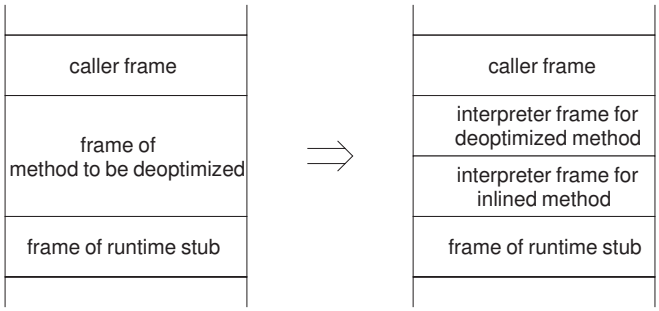

FIGURE 3.4 – Schéma du *On-Stack Replacement* dans HotSpot [Kotzmann et al., 2008]

pour que, lors du retour dans la méthode appelante après l'appel, l'exécution continue de manière correcte.

Cette technique implique de sauvegarder les états précédents de l'exécution pour pouvoir éventuellement revenir en arrière. En particulier, pour maintenir dans le contexte les variables passées en paramètre. De plus, pour effectuer une telle opération, l'exécution doit être stoppée. Cela signifie qu'un tel patch ne peut pas être effectué par un *thread* en parallèle de l'exécution.

La figure 3.4 présente l'implémentation dans la JVM HotSpot de ce mécanisme.

De façon plus générale, la figure 3.5 présente une réparation avec un On-Stack replacement de la séquence de code suivante qui aurait nécessité une réparation. Cette figure représente une pile d'exécution de l'exemple suivant, dans lequel on doit recompiler la méthode `foo` pendant l'exécution de `bar`.

```
fun foo()
do
    ...
    bar()
    ...
end
```

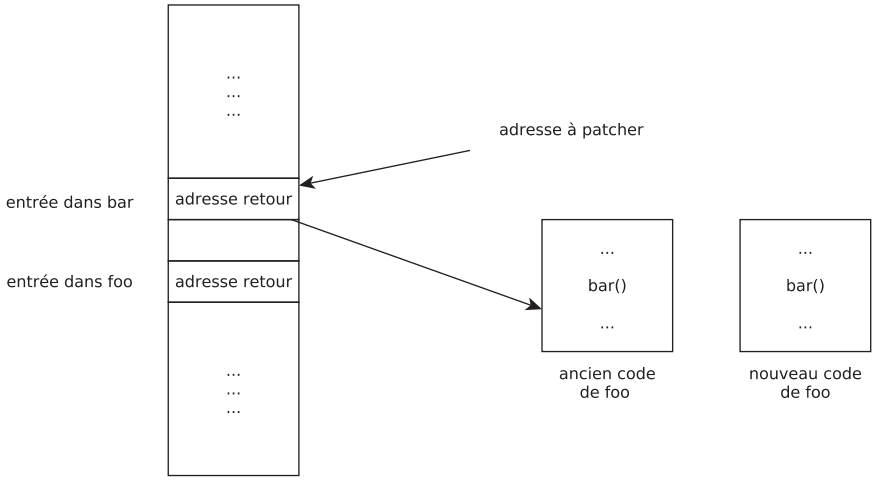

FIGURE 3.5 – Schéma général du *On-Stack Replacement*

Dans une telle situation, il faut recompiler la méthode `foo` de l'exemple. Pour cela, il faut lancer la recompilation de `foo` pour obtenir le nouveau code de la méthode. Ensuite, il faut patcher la pile pour la modifier :

— Remplacer l'ancienne adresse de `foo` dans les tables de méthodes qui référençaient l'ancien code

— Remplacer l'adresse de retour de `bar` pour prendre en compte le nouveau code compilé de `foo`

3.8.4 Préexistence

La préexistence [Detlefs and Agesen, 1999] est une alternative aux techniques de réparations précédemment citées.

Définition 16. Préexistence Une propriété du receveur d'un site d'invocation d'un mécanisme objet. La préexistence du receveur assure que les chargements de classes éventuels durant l'exécution courante de la méthode ne vont pas avoir d'effet sur la valeur du receveur (donc son type dynamique).

Un receveur préexistant assure que les chargements de classes éventuels le concernant ont déjà été réalisés avant l'appel de la méthode. La figure 3.6 présente une situation où le receveur `a` de l'appel `a.bar` est préexistant. Étant donné que le receveur provient d'un paramètre, il a nécessairement été créé avant le début de l'exécution de la méthode `foo`. Cela signifie que le modèle du programme (ses classes chargées) ne va pas évoluer d'une manière qui impacterait l'appel `a.bar` pendant l'appel courant. Il n'y a par contre pas de garanties sur l'évolution du modèle et il faudra peut-être recompiler quand même la méthode `foo`. Mais cette recompilation n'aura pas lieu à chaud, dans ce cas précis,

```
fun foo(a: A)
do
    a.bar
end
```

FIGURE 3.6 – Exemple d'un receveur préexistant

l'appel `a.bar` peut donc être inliné sans craindre de devoir défaire cette optimisation pendant l'appel à `foo`. La préexistence a donc un effet uniquement sur l'appel courant d'une méthode, ce n'est pas une garantie sur le futur.

La préexistence est une propriété très intéressante, elle permet déjà d'éviter d'avoir recours au patch de la pile qui est une opération complexe. Elle assure que des réparations à chaud ne seront pas nécessaire. Toutes les réparations ne sont néanmoins pas supprimées avec la préexistence.

Préexistence et réparation

La figure 3.7 illustre cette situation. Dans cet exemple, la méthode `f` est compilée sans que la classe `B` ne soit chargée. La classe `A` est par contre déjà chargée. Le système d'optimisations détectera que l'appel à `x.foo()` dans la méthode `f` est statique (une seule méthode est candidate pour l'instant). Cet appel sera donc optimisé et le receveur `x` est préexistant. Même si cet appel est invalidé, il ne faudra pas le réparer à chaud et on pourra positionner un trampoline pour le prochain appel.

Le receveur `x.foo()` n'est par contre pas préexistant, si jamais l'appel à `bar()` est optimisé. La suite de l'exécution de la méthode `f` entraîne ensuite le chargement de la classe `B` en exécutant `foo()`. Le `new B()` dans `foo()` provoque le chargement de la classe `B` qui par exemple redéfinit bar ce qui provoque l'invalidation de l'appel suivant à `bar`.

Dans cette situation, il faudra réparer à chaud l'appel à `bar()` en posant un patch dans `f` ou en utilisant le patch de la pile. Le premier appel `x.foo()` ne demandera par contre jamais d'effectuer une réparation à chaud grâce à la préexistence du receveur.

3.9 Systèmes d'optimisation complets

Comme dit précédemment, il existe assez peu de littérature scientifique sur des stratégies d'optimisations dans leur globalité. Dans cette section, nous présenterons divers systèmes dans leur totalité.

3.9.1 JikesRVM

JikesRVM est une machine virtuelle Java de recherche développée par IBM. Elle est développée en Java avec quelques extensions dans le langage pour supporter des opérations de bas-niveau nécessaires à l'implémentation.

```
fun bar()
do
    ...
end

fun foo(): A
do
    var b = new B() // Chargement de la classe B
    return b
end

fun f(x: A)
do
    x.foo().bar()
end
```

FIGURE 3.7 – Préexistence et réparation

Elle a été le support de nombreuses recherches autour de l'implémentation de Java au cours des précédentes années. En particulier, son système d'optimisations est très complet et très bien décrit dans [Arnold et al., 2004].

Architecture de la machine virtuelle

Le système se veut modulaire, la machine virtuelle est partagée entre quatre composants :

— *Runtime measurements* : le composant chargé de récolter des informations sur le programme exécuté et de fournir ces informations.

— *Controller* : chargé de désigner les données à collecter ainsi que les portions du code à compiler.

— *Recompilation* : le composant qui effectue les recompilations du code.

— *Knowledge repository* : ce composant est passif, il sert à centraliser les informations collectées par les différents composants.

Le *controller* est chargé d'élaborer deux éléments. Le premier est un plan de compilation, ce qui consiste à choisir quelles méthodes doivent être compilées. Il est également responsable de faire un plan de mesures, c'est-à-dire de choisir les méthodes qu'il faut observer ou les portions de code à instrumenter pour faire du profilage. La figure 3.8 présente l'architecture de la machine virtuelle avec ses différents composants.

Compilation à la volée dans JikesRVM

JikesRVM effectue uniquement de la compilation à la volée du code, l'approche uniquement compilation la distingue d'ailleurs de la plupart des autres machines virtuelles

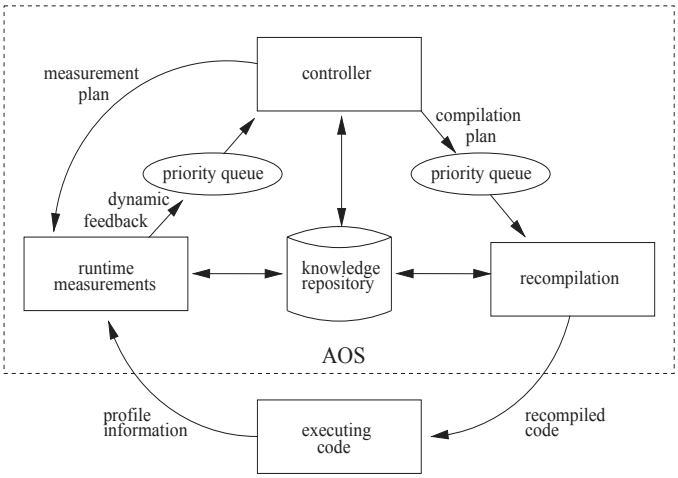

FIGURE 3.8 – Architecture des différents composants de JikesRVM [Arnold et al., 2004]

Java. Par contre, cette JVM possède plusieurs compilateurs :

— *baseline compiler* : un compilateur très simple qui génère du code non-optimisé, mais très rapidement

— *optimizing compiler* : génère du code optimisé

Le compilateur de base est utilisé de la même manière que l'interpréteur dans la JVM Hotspot, c'est-à-dire pour une première compilation des méthodes. Si jamais la méthode n'est quasiment pas utilisée alors elle restera compilée avec ce compilateur. Le compilateur optimisant est quant à lui capable de générer du code très optimisé. En contrepartie il nécessite plus de temps pour compiler une méthode.

Le compilateur optimisant possède plusieurs niveaux de compilations du code :

— niveau 0 : transformation en représentation intermédiaire, optimisations simples (inlining des très petites méthodes, élimination des redondances, etc.)

— niveau 1 : code le niveau 0 mais en effectuant plus d'inlining, les méthodes statiques et finales sont inlinées. Des méthodes virtuelles sont aussi inlinées, la préexistence est également utilisée pour effectuer des inlinings.

— niveau 2 : des optimisations des boucles sont rajoutées, SSA est calculé et exploité, les variables sont numérotées.

En fonctions des informations données par le profilage ou l'échantillonnage, les méthodes sont recompilées pour voir leur niveau d'optimisation augmenter.

3.9.2 JVM HotSpot

La JVM HotSpot est déclinée en deux versions : une client et une serveur. Elles ont des buts différents : la version serveur cherche à avoir le meilleur pic de performances une fois la machine virtuelle lancée au prix d'un temps de démarrage plus long. La version client doit être plus rapide à démarrer et produire rapidement du code, mais cela est fait au détriment des optimisations une fois la machine virtuelle "chauffée".

Version serveur

[Paleczny et al., 2001] présente le système d'optimisations de la version serveur de Hotspot. La stratégie de compilation est (comme pour la version client) mixte entre compilation et interprétation. De base, le code est interprété jusqu'à ce que le système détecte un point chaud qui nécessite d'être compilé.

Dans la version serveur, une version intermédiaire poussée du code est générée. Celle-ci est basée sur une représentation SSA du code. SSA permet d'effectuer de nombreuses optimisations telles que de la numérotation globale des variables, de la propagation de constante ou encore des éliminations des tests à null.

Des analyses statiques simples sont aussi effectuées : CHA permet d'obtenir des informations sur certains sites d'appels pour déterminer s'ils peuvent être statique. Du profilage à l'exécution complète les informations : certains sites ne sont pas statiques avec CHA mais sont détectés en tant que tel à l'exécution.

Dans ce cas des optimisations classiques sont effectuées : dévirtualisation et inlining sur les méthodes candidates. Une technique intéressante concernant l'inlining est décrite. Si le système détecte qu'une méthode doit être inlinée, alors la pile d'appel est inspectée pour déterminer la chaîne d'appel qui mène à cette méthode. Toute la chaîne d'appel peut alors être optimisée, par exemple en effectuant des inlinings récursifs.

La réparation suite à une dé-optimisation est effectuée grâce à un retour en mode interprétation. Pour cela, la pile d'appel est convertie en une *frame* de l'interpréteur et l'exécution revient à un point sûr (*safepoint*) précédent dans le code. Il faut donc en permanence maintenir une structure qui permette de revenir en arrière si besoin.

Le code généré peut aussi être plus finement optimisé à travers des *peephole optimisations*. Il s'agit de remplacer une séquence d'instructions assembleur par une meilleure séquence sur la machine cible.

Version client

HotSpot version client est décrite dans [Kotzmann et al., 2008]. Par rapport à la version serveur, l'objectif est de diminuer le temps de démarrage ainsi que l'empreinte mémoire. Ceci sera fait au détriment des performances une fois la machine virtuelle lancée.

Le fonctionnement de la version client est similaire à la version serveur. Le code commence par être interprété jusqu'à ce que le système détecte un point chaud. Il est aussi précisé que les méthodes contenant de longues boucles peuvent être compilées indépendamment de leur nombre d'appels.

Les performances sont assurées par globalement les mêmes optimisations que pour la version serveur. Les deux optimisations principales sont donc aussi l'inlining et la dévirtualisation. L'efficacité des ces techniques est mesurée dans l'article, l'inlining est décrit comme étant toujours efficace. Les gains apportés sont toujours positifs au prix d'un coût négligeable, ce qui confirme l'importance de cette optimisation.

La représentation intermédiaire des méthodes pour la version client est divisée en deux parties :

— HIR *High-level Intermediate Representation* qui est basée sur SSA.

— LIR *Low-level Intermediate Representation* qui est proche de la machine mais tout de même indépendant de la plateforme.

La forme HIR permet d'effectuer des optimisations comme par exemple :

— propagation de constantes

— l'élimination des tests à null [Kawahito et al., 2000] qui sont nécessaires en Java

— *global value numbering* [Briggs et al., 1997]

Une fois ces optimisations effectuées, le code est traduit en LIR qui sert notamment à l'allocation de registres, puis en assembleur.

La figure 3.9 présente l'architecture de la machine virtuelle HotSpot.

FIGURE 3.9 – Architecture de la JVM HotSpot d'après [Kotzmann et al., 2008]

3.10 Conclusion

Les optimisations dans les machines virtuelles sont très importantes pour obtenir de bonnes performances et de nombreuses techniques existent dans la littérature. Ces optimisations peuvent cependant devenir obsolètes avec le chargement dynamique et des réparations du code deviennent nécessaires.

Nous avons posé la définition d'un protocole de compilation/recompilation. Ce dernier est composé d'un algorithme qui choisit des outils entre plusieurs catégories :

— Récolte d'informations

— Techniques d'optimisations

— Techniques de réparations

Nous avons ensuite décrit des techniques existantes dans ces trois catégories qui sont utilisées dans les machines virtuelles. Ensuite, nous avons présenté deux systèmes d'optimisations complets dans deux machines virtuelles Java qui possédaient de la littérature sur le sujet.

Chapitre 4

Mesures de performances des protocoles

Contents

4.1 Introduction

Un des objectifs de la thèse est de mesurer précisément l'efficacité des protocoles de compilation/recompilation. En particulier, il serait intéressant de pouvoir comparer plusieurs protocoles entre eux. Dans la littérature, les mesures de performances sont généralement présentées de manière globale pour un seul protocole, sans comparaison.

Dans ce chapitre, nous présenterons les différentes techniques employées pour mesurer l'efficacité des optimisations dans des systèmes d'exécution. Celles ci sont diverses mais se limitent généralement à mesurer le temps d'exécution d'un programme avec le système

en question. Cela est limité, mais permet de voir si du temps est gagné par rapport à l'ancienne version.

4.2 Mesures théoriques

Il est possible de tester l'efficacité de mécanismes objet en faisant une analyse simplement théorique. On peut en effet calculer le nombre de cycles processeurs consommés pour effectuer une exécution d'un mécanisme objet. Ensuite, le programme peut être exécuté en comptant combien de fois les mécanismes devront être exécutés. La performance de la technique d'implémentation est mesurable en sachant combien de fois il faut l'utiliser et combien de cycles elle utilise.

Donald E. Knuth dans sa série de livres *The Art of Computer Programming* (dont le premier volume est [Knuth, 1997]) utilise aussi une analyse par cycles processeur. L'analyse de la complexité algorithmique des algorithmes permet d'avoir une estimation théorique de la complexité. Pour une estimation plus réaliste, il prône le comptage des cycles processeur.

[Driesen et al., 1995] discute de l'influence de l'architecture des processeurs en relation avec les implémentations des mécanismes objets. L'article utilise un processeur avec une architecture en *pipeline* pour un processeur dit *superscalaire*. Cette architecture permet d'exécuter plusieurs instructions en parallèle si elles sont indépendantes. Deux types de latence existent avec cette architecture de processeurs : la latence au chargement (nommée L), le processeur doit attendre que les données soient chargées dans les registres depuis les caches avant de pouvoir exécuter l'instruction et possiblement perdre quelques cycles. La deuxième latence est lié aux branchements (nommée B), le processeur va essayer de prédire l'instruction suivante qui devrait être exécutée. Par exemple dans le code de la figure 2.3 du chapitre précédent, la latence était de $2L+B$. Si la prédiction est correcte, il n'y aura aucun surcoût car l'exécution continue directement. Si par contre elle se révèle incorrecte alors des cycles seront perdus à aller chercher la bonne instruction avant de continuer l'exécution. Il est donc important de tenir compte de ces deux contraintes pour exploiter au mieux le parallélisme des instructions.

[Ducournau, 2011b] évalue également l'efficacité de techniques d'implémentations des mécanismes objets du point de vue des cycles processeur utilisés pour un grand nombre d'implémentations des mécanismes objet.

Ces analyses plus théoriques ont l'avantage de ne pas être sensible au bruit induit par le système d'exploitation. De plus, elles seront plus générales que des tests de performances basés sur des mesures de temps qui sont spécifiques au système d'exploitation, au système d'exécution ainsi qu'au programme testé et à l'architecture de la machine. Ainsi, l'utilisation d'un nouveau type de processeur avec la même architecture rendra possible une certaine prévision a priori de l'efficacité.

4.3 Mesures empiriques

Les tests de performance sont en général effectués par des mesures empiriques. Cependant, la problématique n'est pas triviale car de nombreux phénomènes peuvent fausser les mesures et leur comparaison, ou faire douter de la signification des résultats.

4.3.1 Problématique

La méthodologie pour réaliser des expériences est la suivante :

— On possède deux versions du système d'exécution à tester

— On possède un ou plusieurs benchmarks

— Avec le même protocole, on évalue la performance des deux versions sur chacun des programmes de tests

Les résultats sont ensuite comparés pour évaluer les différences et le gain éventuel. Dans un tel cadre, le protocole de tests doit être rigoureux pour que les résultats soient représentatifs.

Les compilateurs, et surtout les machines virtuelles, sont soumis à du bruit de plusieurs origines qui aura un impact sur les temps d'exécution. Afin de limiter le plus possible la variabilité on cherchera généralement à avoir un programme déterministe en tant que donnée. Par exemple, on attendra qu'un traitement sur des données se fasse dans le même ordre (comme une itération sur une collection), indépendamment de la version du système d'exécution à mesurer. Cela permet de limiter le bruit induit par des différences dans l'exécution et pouvoir mesurer plus précisément les performances du système d'exécution. Avoir un programme déterministe assure aussi que la localité mémoire du programme sera similaire entre deux exécutions, des différences seront malgré tout présentes car les adresses ne seront pas les mêmes entre deux exécutions, mais moins qu'avec un programme non-déterministe.

Mais le système d'exécution doit également être le plus déterministe possible. Un compilateur déterministe générera exactement le même binaire deux fois de suite, ce qui permet de supposer que la localité mémoire du programme sera meilleure. Par opposition, un compilateur non-déterministe pourrait générer le binaire en ordonnant les fonctions de façon aléatoire. Le système d'exécution sera en effet toujours soumis au bruit d'autres processus tournant en parallèle, le processeur pourra également ne pas toujours fonctionner à la même vitesse. Il est difficile de limiter ces bruits, il faut donc essayer de limiter les variations quand c'est possible à travers les programmes de test et les systèmes d'exécution.

[Ducournau et al., 2009] présente un compilateur qui est compilé de manière déterministe avec un exécutable toujours parfaitement identique entre plusieurs compilations. Cet idéal est possible dans un compilateur, mais paraît difficile avec les systèmes d'optimisations adaptatifs des machines virtuelles.

4.3.2 Benchmarks

Les mesures de performances sont réalisées en utilisant des programmes de test, appelés benchmarks. Les tests de performance ont beaucoup d'éléments variables qui ajoutent du bruit dans les résultats, le benchmark utilisé peut aussi rajouter du bruit.

Ces benchmarks doivent être représentatif, une amélioration doit être mesurable sur la majorité des benchmarks pour pouvoir en tirer des conclusion. Les programmes utilisés doivent donc être d'une taille raisonnable pour utiliser intensivement les mécanismes testés : on cherchera idéalement à avoir des informations sur le nombre d'invocation de ces mécanismes dans les programmes de tests (par analyse statique ou dynamique).

Utilisation de programmes générés

Lorsqu'on ne dispose pas de véritables programmes pour effectuer des expériences, il est possible d'en générer un. [Privat and Ducournau, 2005] présente par exemple des expériences réalisées avec un programme généré. Les auteurs utilisaient le même programme avec des langages différents pour comparer les performances. Un programme a donc été traduit dans un autre langage dans ce but. À l'origine, le programme était donc réaliste, mais il est aussi possible de générer de manière aléatoire des programmes pour faire des expériences sur des compilateurs, en particulier pour tester des points précis à travers des *microbenchmarks*.

Cette technique est le seul moyen de tester des systèmes d'exécution dans le pire des cas. Cela permet de concentrer le test sur un point critique, et aussi d'évaluer le passage à l'échelle d'un mécanisme particulier.

Utilisation de véritables programmes

Le cas le plus fréquent reste l'utilisation de programme, ou d'un corpus de programmes reconnu. Un véritable programme avec une taille conséquente sera représentatif des habitudes des programmeurs et constitue donc une donnée très intéressante.

De plus, l'utilisation des mêmes benchmarks entre différents systèmes d'exécution permet de comparer leur efficacité pour que les tests soient reproductibles. Ces benchmarks doivent donc être facilement disponibles (libre ou open source), et avoir souvent été utilisées.

En Java, il existe de nombreux corpus utilisés dans la littérature scientifique :

— DaCapo benchmarks [Blackburn et al., 2006]

— specsJVM benchmarks [specsJVM, 2008, Shiv et al., 2009]

— Qualitas Corpus [Tempero et al., 2010]

Ils contiennent tous des gros programmes Java divers qui correspondent aux programmes Java le plus souvent utilisés comme par exemple l'IDE Eclipse. Le fait que la machine virtuelle se comporte correctement sur cet ensemble de benchmarks donne un résultat empirique fiable. Par contre, il ne s'agit aucunement d'une preuve de l'absence de problèmes dans la machine virtuelle et cela ne met pas à l'abri non plus contre un programme qui aurait un comportement très défavorable aux stratégies d'optimisation.

4.4 Mesures de temps

Traditionnellement dans la littérature sur les compilateurs, les temps d'exécution sont mesurés pour évaluer les effets des optimisations. Il s'agit d'ailleurs souvent de la seule mesure fournie. Généralement, on effectue plusieurs exécutions successives, en faisant une moyenne ou en gardant le meilleur temps. Ces mesures sont parfois doublées d'un écart type et d'une élimination des valeurs extrêmes.

4.4.1 Mesures de temps des programmes compilés statiquement

[Ducournau et al., 2009] présente également des mesures de temps pour des implémentations de mécanismes objets. Ces mesures sont décrites comme étant difficiles à réaliser à cause de la pollution induite par le système d'exploitation. Cette pollution vient d'autres processus qui consomment du temps processeur de manière aléatoire et non-reproductible en imposant au programme testé une localité mémoire variable.

Leur technique pour pallier ce problème a été de lancer le système d'exploitation, en l'occurrence Linux, en *recovery mode* qui est un mode dans lequel le système démarre avec le minimum de processus requis et avec le moins de multi-tâches.

Cependant, toutes les pollutions ne sont pas éliminées, certains ordinateurs portables ont des processeurs dont la fréquence varie au cours du temps. La fréquence baisse quand l'ordinateur est chaud ou alors complètement inactif. La fréquence est ensuite augmentée jusqu'à atteindre un pic qui fait d'ailleurs chauffer l'ordinateur, elle descendra alors progressivement. Pour essayer de limiter ce problème, il est possible d'insérer une pause après une première exécution pour laisser le processeur refroidir et avoir ainsi deux exécutions dans des conditions similaires. Il faut noter que les processeurs pour ordinateur fixe ont également leur fréquence qui varient au cours du temps, ils ne sont donc pas exemptés de ce problème.

Ces expériences présentaient des résultats très peu bruitées avec les précautions prises.

4.4.2 Mesures de temps et machines virtuelles

Les mesures de temps sont généralement fournies pour attester de l'efficacité d'une machine virtuelle. La machine virtuelle HotSpot versions serveur [Paleczny et al., 2001] et client [Kotzmann et al., 2008] ont des publications avec des temps d'exécution. Plusieurs architectures sont testées pour plus de variations ainsi qu'un ensemble de benchmarks (généralement des gros programmes Java).

Deux mesures sont fournies pour la version serveur, elles sont comparées avec le meilleur temps d'exécution de leur nouvelle stratégie d'optimisation :

— Temps d'exécution le plus petit

— Temps d'exécution le plus long

L'article de la version client fournie juste quelques temps d'exécution sans plus de précisions sur la façon dont le temps est mesurée.

Généralement, les machines virtuelles avec des systèmes complexes d'optimisation mettent du temps à atteindre un niveau de performance intéressant. C'est souvent le cas pour les machines virtuelles qui font de l'interprétation et de la compilation ou alors celles qui ont plusieurs niveaux d'optimisations. [Arnold et al., 2004] contient beaucoup de mesures de temps d'exécution. Les auteurs ont effectués plusieurs mesures de temps :

— Temps d'exécution "total" : le temps est mesuré entre le lancement de la machine virtuelle jusqu'à réaliser un objectif fixé précédemment.

— Temps d'exécution "stable" : le benchmark est lancé pendant une période fixe pendant laquelle le travail effectué est mesuré (nombre de fois que le programme réalise sa tâche)

Pour la première mesure, l'approche est assez classique. Pour la seconde, les auteurs ont utilisés une période de chauffe de la machine virtuelle : le programme était lancé pendant 5 minutes puis l'efficacité était mesurée. Cela correspond donc à des performances à chaud, qui ignorent le temps passé à recompiler plusieurs fois la même méthode pour atteindre le pic de performances.

[Chevalier-Boisvert and Feeley, 2015] présente aussi des mesures de performances pour évaluer une technique d'optimisation. La technique employée consiste à mesurer 10 fois le temps d'exécution et à présenter la moyenne.

4.5 Mesures discrètes

La performance d'un compilateur peut aussi être mesurée en comptant de manière statique ou dynamique certains éléments qui caractérisent le benchmark. Les mesures dites statiques seront effectuées en comptant des éléments dans le code source du programme. Les mesures dynamiques compteront des événements à l'exécution comme le nombre d'appel de méthode par exemple.

4.5.1 Mesures statiques

La littérature sur l'implémentation des objets abonde en mesures statiques des programmes : nombre de classes, méthodes et attributs, nombre de sites d'appels de méthodes, taille des tables de méthodes, etc. Pour les travaux basés sur des optimisations, les sites sont décomptés selon le fait qu'ils sont optimisés ou pas. [Ducournau, 2011b] contient une synthèse de cette littérature en plus de nombreuses expérimentations et mesures détaillées sur un ensemble de benchmarks.

Dans l'optique de mesurer la performance d'une machine virtuelle, on cherchera surtout à évaluer la performance des implémentations. Pour un programme testé, on peut compter :

— Le nombre de sites d'appels non-optimisés (implémentés avec le hachage parfait)

— Les sites partiellement optimisés : implémentés comme en héritage simple

— Le nombre d'appels de méthodes optimisés (implémentés statiquement)

À partir de ces statistiques et pour plusieurs benchmarks testés, on peut arriver à mesurer le nombre de sites optimisés engendrés par un protocole donné. La comparaison des protocoles est donc déjà possible avec de telles mesures.

Ces mesures statiques ne sont cependant pas suffisantes : en effet, il peut arriver qu'une seule implémentation non-optimisée soit exécutée un très grand nombre de fois et produise de mauvaises performances. Il faut donc coupler ces résultats avec des mesures effectuées dynamiquement, dans le but de savoir si il y a adéquation.

4.5.2 Mesures dynamiques

Dynamiquement, il est possible de compter des éléments similaires à ceux mesurés statiquement. Par exemple :

— Nombre d'exécutions de sites optimisés (appels statiques par exemple)

— Nombre d'exécutions de sites non-optimisés

Il faut également mesurer le coût du protocole de compilation/recompilation et donc pouvoir mesurer les recompilations entraînées par un protocole. Suivant la stratégie choisie, on peut compter des éléments tels que :

— Nombre de gardes exécutées

— Nombre de patchs de code effectué

— Nombre de recompilations totale de la méthode

— Nombre de patchs de pile

Tous ces éléments constitueront le coût du protocole en terme de recompilations.

Comme expliqué précédemment, en connaissant le nombre de cycles processeur consommé par un mécanisme élémentaire ainsi que le nombre de fois où ce mécanisme est utilisé, on peut en théorie en déduire le temps d'exécution d'un système d'exécution. Cela permet en tout cas de comparer plusieurs protocoles à partir de ces informations et d'avoir une approximation du temps d'exécution sans le mesurer. D'après [Ducournau et al., 2009], une grande partie de la différence dans les mesures de temps s'explique à partir des mesures théoriques en nombre de cycles et des mesures dynamiques de nombre d'exécutions des sites.

Toutes ces mesures dynamiques ne permettent cependant pas d'évaluer le coût du protocole en lui même. C'est à dire le temps qu'il met à décider des optimisations et à mettre en œuvre les différentes stratégies.

4.5.3 Chargement aléatoire dans [Ducournau and Morandat, 2012]

Une simulation de machine virtuelle a été réalisée dans [Ducournau and Morandat, 2012]. L'efficacité des protocoles abstraits testés dépend grandement de l'ordre de chargement des classes dans l'interprétation. L'aléatoire permet d'obtenir des statistiques sur toutes les situations possibles et donc possiblement sur la pire, qui a peu de chances de se produire en réalité. Par exemple, des expériences basées sur le chargement aléatoire de classes ont permis de montrer que le hachage parfait se comportait de façon

approximativement linéaire en espace, et que le pire des cas restait acceptable. Cette simulation peut être vue comme une variante très lointaine d'interprétation abstraite où sont calculées les exécutions possibles.

De manière similaire, [Ducournau and Morandat, 2011] contient également des expériences réalisées par une interprétation aléatoire concernant des implémentations objets. Les mécanismes d'une machine virtuelle étaient simulés et un programme sous forme abstraite a été utilisé pour mesurer des optimisations. Les classes du programmes étaient chargées de manière aléatoire, et l'efficacité du protocole d'optimisation était mesurée en comptant les mécanismes objets utilisées sur les sites d'appels. Le coût des optimisations était contrebalancé par le nombre de recompilations qu'il fallait effectuer.

Cette technique permet de ne pas développer une vraie machine virtuelle pour réaliser des expériences et d'agir en simulant les mécanismes appropriés. D'après l'article, la simulation et les expériences menées ont été développées en quelques semaines (avec une plate-forme de tests déjà éprouvée).

4.6 Mesures dans les machines virtuelles

Étant donné l'aspect dynamique des machines virtuelles, il faut prendre en compte les changements d'implémentations au long du programme. Pour des mesures statiques, il faut alors effectuer les mesures statiques à deux instants de l'exécution :

1. Lors de la première compilation des méthodes
2. À la fin de l'exécution

Pour les mesures dynamiques, on peut aussi considérer deux types de compteurs :

1. Un compteur global, incrémenté lors de chaque exécution d'un site objet d'une implémentation particulière
2. Faire un compteur par site, comptant simplement le nombre d'exécutions

La première méthode est évidente et permet de voir l'exécution réelle de l'application. Avec le second type de compteur, on peut compter à la fin de l'exécution en fonction de la dernière implémentation du site. Ainsi, un site qui finira par être optimisé comptera comme si il avait toujours été optimisé avec cette méthode.

Ainsi, il sera possible de reconstituer les performances "à chaud" du benchmark sans relancer une deuxième fois l'exécution. Les changements d'implémentations lors de l'exécution n'étant alors plus pris en compte.

Les mesures en premières compilations fournissent des indications sur l'efficacité immédiate de la machine virtuelle. Par contre, elles ne présentent pas les performances atteintes "à chaud". Pour cela, il faut mesurer à nouveau ces informations en fin d'exécution, c'est-à-dire une fois l'intégralité du programme exécuté et les recompilations diverses effectuées. Dans la littérature scientifique sur les machines virtuelles, les performances sont d'ailleurs souvent mesurées une fois que la machine virtuelle a atteint son pic de performance, c'est-à-dire après une première exécution.

Pour les mesures de temps dans les machines virtuelles, les auteurs des expériences publient souvent plusieurs données. Les performances "à chaud" sont souvent présentées,

une fois que la machine virtuelle a démarré et qu'elle a déjà compilé et optimisé le code. De même, les temps de démarrage des machines virtuelles sont parfois présentés.

4.7 Discussions

Dans les techniques présentées, le chronométrage pour évaluer la performance est ce qui revient le plus souvent dans la littérature. Comme illustré, ce n'est pas nécessairement facile à faire : les machines virtuelles ont des performances variables au cours du temps, il est difficile d'isoler le processus etc.

Et ce n'est sans doute pas suffisant, vu le bruit ambiant induit par le système, il est difficile de vraiment savoir si une seule petite optimisation a des effets importants, et dans quelle mesure sur le système. En particulier, des systèmes d'optimisation complets sont évalués uniquement en mesurant le temps d'exécution, parfois en faisant des moyennes et parfois aussi en gardant le pire et le meilleur temps. Des écarts-types pour éliminer les valeurs extrêmes sont souvent réalisés.

On peut noter que le bruit engendré par le système est toujours positif, il est donc possible de considérer le temps minimum obtenu et de mesurer les écarts par rapport à ce minimum pour avoir des mesures un peu plus robustes.

Généralement, les outils statistiques ne sont pas utilisés même si ils existent et permettent de rendre plus robustes les conclusions tirées des mesures. On peut par exemple citer [Touati et al., 2013] qui résume des méthodes statistiques pour déterminer la taille de l'échantillon nécessaire pour évaluer correctement les performances ainsi que l'évaluation du risque d'erreur. Pour des mesures de temps d'exécution, ces outils statistiques semblent être la meilleure approche.

De manière générale, le problème de la reproductibilité des tests se pose : le programme testé n'est pas forcément déterministe, ce qui peut encore rajouter de la variabilité dans les résultats.

L'interprétation abstraite et ses variantes utilisables pour mesurer des exécutions est également une simulation et ne représente pas une exécution réelle d'un programme. Les méthodes qui se contentent de compter différents éléments pour mesurer l'efficacité restent des estimations, il est difficile de mesurer le coût nécessaire à la mise en place d'une optimisation. Mais elles ont l'avantage d'être plus simples à mettre en œuvre et plus robustes : pas de nécessiter de faire "chauffer" le système etc.

La meilleure manière de procéder est selon nous multiple : il faudrait mesurer le temps d'exécution pour avoir une idée réaliste de l'efficacité globale et pouvoir mesurer le coût induit par le protocole. En parallèle, il faudrait utiliser des mesures statiques et dynamiques pour mesurer finement des effets marginaux d'optimisations. Cela permettrait aussi d'avoir des mesures chiffrées et beaucoup plus précises pour comparer plusieurs protocoles entre eux.

4.8 Conclusion

Nous avons présenté quelques manières existantes de mesurer les performances dans le domaine des compilateurs :

— Mesure des cycles processeur

— Mesure du temps d'exécution

— Comptage d'éléments statiques ou dynamiques

Il faut mettre en relation ces techniques avec la problématique des benchmarks et de la reproductibilité des tests.

Pour mesurer l'efficacité des optimisations dans la littérature scientifique, l'approche mesurant le temps est souvent la seule utilisée. Elle n'est pourtant à nos yeux pas suffisantes pour tout mesurer et on gagnerait à la coupler avec d'autres techniques. En particulier, les mesures discrètes d'éléments de manière statique ou dynamique permettraient de mieux mesurer les effets et l'efficacité de plusieurs protocoles entre eux.

La solution idéale est selon nous mixte : le comptage du temps d'exécution de manière rigoureuse, avec éventuellement des méthodes statistiques. Cela doit alors être associé à des comptages d'éléments discrets de l'application. Ainsi, il est possible d'avoir des mesures objectives, et les expériences réalisées pourront être reproduites et plus facilement comparables avec d'autres protocoles de compilation/recompilation. Ces décomptes permettent de connaître le taux de sites qui ont une implémentation optimisée (par exemple un appel statique), ils permettront alors de décider de l'utilité d'une optimisation.

Depuis les premiers travaux sur les optimisations dans les langages à objet, on sait que le taux de sites dévirtualisables est très important. L'analyse par cycle a montré que les appels statiques sont beaucoup plus efficaces que les appels virtuels. Avec ces deux éléments, on peut alors mesurer l'efficacité des optimisations sans mesurer le temps d'exécution. Des expériences telles que [Ducournau et al., 2009] permettent de mesurer le gain de façon précise, qui se révèle proche et en accord avec les mesures théoriques (analyse par cycles et mesures discrètes).

Deuxième partie

La machine virtuelle Nit

Chapitre 5

État de l'art des outils

Contents

5.1 Introduction

Développer une machine virtuelle est un projet ambitieux. Ces systèmes représentent parfois jusqu'à plusieurs centaines de milliers de lignes de code. Dans le cadre d'un projet universitaire et d'autant plus dans la durée limitée d'une thèse, il était impératif de chercher à réutiliser le plus d'éléments possible.

Ce chapitre contiendra tout d'abord un récapitulatif des outils explorés puis utilisés pour mener à bien le projet. Ensuite, l'architecture de la machine virtuelle ainsi que son fonctionnement détaillé seront décris.

La principale difficulté du projet était de réaliser une machine virtuelle suffisamment réaliste pour effectuer des expériences sur les protocoles de compilation/recompilation. L'objectif du projet n'est pas d'avoir une machine virtuelle commerciale, il s'agit de spécifier une machine virtuelle pour l'héritage multiple ainsi que son compilateur à la volée. Pour cela, il était nécessaire d'effectuer beaucoup de développement, ce qui bien sûr est très chronophage. Le principal point consistait à ne pas réécrire de zéro l'intégralité des composants d'une machine virtuelle.

5.2 Le langage Nit

Cette thèse implique d'utiliser un langage à objets en héritage multiple et en typage statique en tant que langage source de la machine virtuelle. Comme présenté précédemment, il n'y a pas beaucoup de langages candidats à de telles expérimentations. Dans les langages à objet à typage statique, seul C++ et Eiffel présentent des caractéristiques proches et C++ n'est pas complètement objet.

Nous avons travaillé avec le langage Nit [Privat, 2008]. Initialement nommé *PRM* (*Programming with Refinement and Modules*), ce langage a été créé par Jean Privat [Privat, 2006] au sein du LIRMM puis à l'UQAM ensuite à partir de 2008. À ce moment là, le compilateur est écrit en Ruby et le langage est compilé vers du C. Par la suite, Floréal Morandat a continué de développer PRM dans sa thèse [Morandat, 2010] en réalisant notamment le bootstrap du compilateur. Ce compilateur bootstrapé a servi à effectuer des expériences sur l'implémentation des mécanismes objets.

Le langage a été renommé Nit en 2008, il est maintenant plus mature et son développement est actif bien que Nit reste un langage académique. Il possède actuellement plusieurs compilateurs et sert de support à la recherche.

Ce langage possède plusieurs propriétés intéressantes :

— Langage complètement orienté objet
— Héritage multiple de classes
— Types virtuels [Shang, 1996] ([Bruce et al., 1998, Torgersen, 1998] présentent des améliorations)
— Système de modules et de raffinement de classes [Ducournau et al., 2007]
— Types *nullables* [Gélinas et al., 2009]
— Gestion automatique de la mémoire via un ramasse-miettes : Boehm [Boehm and Weiser, 1988, Boehm, 1993]

5.3 Frameworks de machine virtuelle

Partant du constat que développer de zéro une machine virtuelle représente un grand effort (par exemple 250 000 lignes pour la JVM HotSpot), certaines équipes de recherches

ont tentées de créer des frameworks de machine virtuelle. Ces travaux restent tout de même peu nombreux et relativement marginaux. À notre connaissance, il n'existe aucune machine virtuelle importante qui ait été développée en utilisant un framework de machine virtuelle.

Plusieurs composants caractérisent une machine virtuelle :

— Gestionnaire de mémoire automatique (*Garbage collector*)

— Compilateur à la volée

— Gestionnaire de *threads*

— Représentation des objets en mémoire

Une étude plus poussée des différents frameworks utilisables pour développer une machine virtuelle a été réalisée dans [Saleil, 2013].

5.3.1 Open-Runtime Platform

Open-Runtime Platform est une machine virtuelle développé par Intel [Cierniak et al., 2002,Cierniak et al., 2005] pour plusieurs langages. Le bytecode Java ainsi que le CIL sont supportés par Open-runtime Platform. Ces deux langages étant relativement similaires, il a été possible de factoriser un grand nombre de mécanismes dans la machine virtuelle.

À l'origine le but était d'avoir une grande modularité dans les composants du système. Il était par exemple prévu de pouvoir changer facilement le ramasse-miettes, le compilateur à la volée ou d'autres éléments dans une machine virtuelle. Open-Runtime Platform n'est pas vraiment un framework de machine virtuelle, mais sa modularité aurait pu permettre de l'utiliser pour d'autres langages que Java et les langages .NET. La cible du projet semble être des langages en sous-typage multiple, il n'y a pas de garantie qu'Open Runtime Platform puisse supporter l'héritage multiple. Aujourd'hui, le développement du projet est arrêté.

5.3.2 Truffle et Graal

Truffle et Graal [Würthinger et al., 2013, Wimmer and Würthinger, 2012] sont des sortes de frameworks permettant d'implémenter des langages sur la machine virtuelle Java standard.

Du point de vue de l'utilisateur, un interpréteur pour le langage choisi doit être écrit en Java en utilisant l'API de Truffle. C'est la seule action à effectuer pour que le langage puisse être exécuté. Cet interpréteur est ensuite exécuté sur une machine virtuelle Java. Cette machine virtuelle Java est en fait la JVM de référence (HotSpot) avec un nouveau compilateur à la volée : *Graal*. Ce compilateur à la volée va essentiellement faire de la réécriture des nœuds de l'arbre syntaxique pour optimiser l'exécution de l'interpréteur.

Ce framework est un outil intéressant car il permet d'implémenter facilement un nouveau langage en ayant de bonnes performances. Plusieurs publications font état de l'utilisation de Truffle et Graal pour implémenter des langages, on peut par exemple citer [Bonetta, 2015] ou encore [Wimmer and Brunthaler, 2013].

Néanmoins, aucun contrôle n'est possible sur l'implémentation du langage dans la machine virtuelle. Il serait donc impossible d'effectuer des expériences sur les protocoles d'optimisation en l'utilisant, cette solution est donc à écarter.

5.3.3 Le framework VMKit

VMKit [Geoffray et al., 2010] est sans doute le projet le plus abouti dans ce domaine précis. Il s'agit d'un framework en C++ développé dans le cadre de la thèse de Nicolas Geoffray [Geoffray, 2009].

VMKit est une agrégation de différents outils existants liés entre eux pour composer le framework :

— LLVM : compilateur à la volée et représentation intermédiaire du code
— MMTk : *garbage collector* utilisé initialement dans JikesRVM [Blackburn et al., 2004]
— Threads POSIX pour le multitâches
— Une base de code pour lier ces différents composants entre eux
— Un squelette minimal de machine virtuelle

VMKit a été utilisé pour développer deux machines virtuelles. La première, J3, est une machine virtuelle Java. Une autre machine virtuelle N3 exécutait du code CLR (plateforme .NET). Ces deux machines virtuelles sont plutôt des preuves de faisabilité que de véritables machines virtuelles efficaces. Les techniques d'implémentations utilisées pour leur conception étaient relativement classiques et correspondent à l'état de la littérature scientifique du domaine à l'époque de leur développement.

Ce framework est indéniablement un outil intéressant, et il était initialement prévu de l'utiliser pour développer notre machine virtuelle. Mais son développement a ralenti et le projet a été officiellement arrêté fin 2013.

5.4 Machine virtuelle utilisant VMKIT

5.4.1 Présentation

L'idée initiale était de développer une machine virtuelle avec VMKIT et de définir un sous-ensemble de Nit qui serait exécutable. Ce sous-ensemble ne devait pas contenir la notion de modules et de raffinement de classes qui est profondément incompatible avec le chargement dynamique.

En effet, du point de vue de la machine virtuelle, les modules sont problématiques : le raffinement de classes permet de définir une classe à travers plusieurs modules. Charger cette classe dans la machine virtuelle implique donc en théorie de "rassembler" tous ses morceaux dans les différents modules de l'application. Il n'est pas non plus possible de charger un module bout par bout à la volée car cela impliquerait de modifier la représentation des classes dans la machine virtuelle et de possiblement modifier des instances déjà créées. Une alternative est possible : aplatir les modules de manière globale avant l'exécution pour faire disparaître cette notion à l'exécution.

Pour développer intégralement la machine virtuelle avec VMKit impliquerait d'avoir un compilateur de Nit vers un langage de bytecode, qu'il faudrait également définir et spécifier. Il serait également possible de traiter directement des programmes en Nit en entrée mais il faudrait aussi développer un analyseur lexical et syntaxique.

5.4.2 Faisabilité

Le fait de spécifier un langage de bytecode est une tâche complexe : il faut en effet spécifier le langage et ensuite maintenir le compilateur de Nit vers le bytecode. Les nouveautés du langage devraient alors être supportées pour maintenir la compatibilité et pouvoir continuer à exécuter des benchmarks. En parallèle, il faudrait développer la machine virtuelle : au total deux systèmes sont donc à développer et maintenir.

Du point de vue des performances, il s'agirait du meilleur système possible car nous posséderions un compilateur à la volée très performant (LLVM) ainsi qu'une gestion mémoire poussée.

Par contre, cette approche oblige à réécrire tout le méta-modèle de Nit en C++, ce qui est long et redondant car il est déjà présent dans les compilateurs Nit. On peut également imaginer que la complexité de C++ et l'assimilation des différents composants de VMKit seraient des freins au développement.

Compte tenu des délais restreints et de la seconde partie du projet qui consiste à étudier des protocoles de compilation/recompilation, cette option pour développer la machine virtuelle semble complexe. Cependant, les spécifications seraient remplies avec VMKit. Historiquement, il était prévue d'utiliser cette méthode pour réaliser la machine virtuelle, mais l'arrêt du support de VMKit met définitivement un terme à cette option.

5.5 Machine virtuelle à partir de l'interprète Nit

5.5.1 Présentation

L'abandon du projet VMKit a demandé de chercher des alternatives. Le langage Nit dispose de plusieurs systèmes d'exécution dont un interpréteur. Une solution alternative est de se baser sur l'interpréteur Nit pour le transformer progressivement en machine virtuelle.

L'interpréteur de Nit est basé sur l'arbre syntaxique du programme (AST) qui est décoré par le modèle du programme. L'exécution est réalisée en parcourant arbre syntaxique, pour produire le résultat attendu, il n'y a pas de notion de compilation.

5.5.2 Faisabilité

Cet interpréteur n'effectue pas de chargement dynamique des classes, les structures nécessaires à l'évaluation sont donc créées globalement au début de l'exécution. Il est par contre déjà paresseux lors de l'exécution. Il faudrait donc commencer par ajouter le chargement dynamique.

Avec cette solution, il est est possible de commencer à travailler sur différents protocoles de compilation/recompilation après un relativement peu de temps passé à développer. La compilation du code à la volée est quelque chose de fondamental pour le travail sur les protocoles de compilation/recompilation. Cette compilation pourrait être réalisée de deux manières :

— Simulée : les implémentations des mécanismes objets seront décidées avant de les utiliser, sans produire de code machine

— Réelle : du code serait généré à la volée

Le développement d'un véritable compilateur à la volée est une tâche importante, au début du projet, il n'existait pas de possibilités d'utiliser LLVM en Nit pour réaliser ceci. Il faudrait donc développer un compilateur à la volée en utilisant d'autres moyens.

Cette solutions a également l'avantage de n'avoir pas besoin de spécifier un langage de bytecode ni de développer un compilateur de Nit vers ce bytecode. L'interpréteur est déjà capable d'exécuter des programmes Nit, du temps serait ainsi économisé.

De plus, le modèle du programme est également présent dans l'interpréteur et pourrait donc servir lors de l'étude des protocoles sans avoir besoin de le réécrire.

5.6 Compilateurs à la volée traceurs

Les compilateurs à la volée traceurs (*tracing-JIT* en anglais) sont une nouvelle approche permettant d'obtenir de relative bonnes performances dans un interpréteur. Cette approche a été par exemple été testée dans un interpréteur python [Bolz et al., 2009].

L'idée est qu'un programme passera beaucoup de temps dans les boucles, plusieurs itérations de ces boucles seront relativement similaires du point de vue du code exécuté. Un compilateur traceur va donc enregistrer les instructions exécutées dans une boucle, puis compiler ces instructions en code machine à la volée.

Aux prochaines itérations de cette même boucle, ce code compilé sera exécuté au lieu d'interpréter le code. Pour que l'approche soit viable, il faut bien sûr introduire des gardes dans le code compilé pour garantir une exécution correcte. Si jamais le code compilé diffère du code qui devrait être exécuté, l'exécution continue en mode interprétation seulement.

Cette technique est un fonctionnement mixte entre un interpréteur et un compilateur. De bonnes performances sont promises pour une complexité moindre que de développer un compilateur à la volée uniquement. Mais il faut tout de même posséder un compilateur à la volée capable de compiler vers du code machine des instructions élémentaires du code. Cette technique pourrait être utilisée en tant qu'approche mixte entre un interpréteur et un compilateur à la volée pour implémenter la machine virtuelle.

5.7 Comparatif des différentes approches

L'approche la plus naturelle, qui n'est pas la plus simple, est de re-développer une machine virtuelle de zéro. Dans un temps contraint et compte tenu des faibles ressources,

il n'est pas envisageable de suivre cette approche, de plus, elle demanderait trop de temps avant de pouvoir commencer à faire l'étude des protocoles de compilation/recompilation.

L'utilisation d'un framework de machine virtuelle serait probablement la meilleure solution pour avoir un système réaliste et performant tout en étant réalisable dans les temps. Il n'existe par contre plus à l'heure actuelle de framework qui pourrait servir à réaliser cette tâche.

Assez naturellement, la seule solution dont nous disposons est de développer la machine virtuelle à partir de l'interpréteur Nit. Elle n'est pas non plus exempte de problèmes car le système ne sera au début pas aussi réaliste qu'une vraie machine virtuelle, mais c'est la solution la plus rapide qui permette de travailler sur les protocoles de compilation/recompilation.

5.8 Outils utilisés

5.8.1 Discussions sur l'utilisation de Nit

Le langage Nit est un bon candidat pour l'étude prévue. Nit reste par contre un langage académique, ce qui pose en particulier des problèmes avec les benchmarks. Les programmes Nit ne sont pas très nombreux par rapport à des langages comme Java qui possèdent de très nombreux programmes et même benchmarks officiels.

Du point de vue de la diversité, les programmes Nit sont écrits par un nombre réduit de personnes qui peuvent avoir tendance à programmer de la même manière car travaillant ensembles. Il sera donc plus difficile d'avoir des programmes variés, et donc de tester spécifiquement certaines parties du système d'optimisations.

Problème des benchmarks

Le problème des benchmarks utilisables est grandement lié à l'utilisation du langage Nit. En particulier, il n'est pas possible de comparer les résultats avec ceux de machines virtuelles Java connues car nous n'avons pas de benchmarks en commun.

Il faudra donc utiliser des benchmarks issus uniquement du monde Nit qui seront de taille plus modeste par rapport aux benchmarks utilisés dans la recherche autour des machines virtuelles Java.

5.8.2 L'interpréteur Nit

L'interpréteur Nit a une structure très basique, il a aussi l'avantage d'être écrit en Nit. L'interpréteur commence par construire le modèle du programme de manière globale. l'arbre syntaxique est également généré et décoré par le modèle.

Cet interpréteur servira de base à la machine virtuelle Nit. Cette dernière sera progressivement développée en se basant sur l'interpréteur, en remplaçant progressivement ses composants.

La transformation de l'interpréteur Nit en machine virtuelle imposait de rajouter certaines caractéristiques :

— Chargement dynamique des classes

— Compilation du code à la volée

Ces deux caractéristiques imposait donc un fonctionnement paresseux dans la création des structures dans la machine virtuelle qui n'était pas présent dans l'interpréteur puisque tous les éléments dont on a besoin l'exécution sont présents avant son commencement. Il faut donc progressivement remplacer des éléments de l'interpréteur pour obtenir une machine virtuelle fonctionnelle et réaliste.

Travaux liés

Se baser sur un interpréteur pour développer une machine virtuelle est une approche qui a déjà été testée dans plusieurs publications. [Würthinger et al., 2012] présente un interpréteur qui est auto-optimisant. Cet interpréteur est capable d'effectuer des optimisations en réécrivant des nœuds de l'arbre syntaxique.

Des optimisations pour un interpréteur sont présentées dans [Kalibera et al., 2014]. L'implémentation officielle du langage R utilise un interpréteur, qui a de mauvaises performances. Un nouvel interpréteur (qui se veut simple) est développé, des optimisations sont ensuite effectuées pour assurer des performances correctes :

— Numérotation des variables

— Déroulement des boucles (*loop unrolling*)

— Spécialisation de code

Leur interpréteur a de meilleures performances que l'interpréteur officiel de R (jusqu'à 8 fois mieux sur certains benchmarks). De plus, la complexité de l'implémentation est bien moindre par rapport à un compilateur. Une approche basée sur un interpréteur semble donc être un choix raisonnable en présentant un bon compromis entre simplicité et performances.

Avantages d'une telle approche

Utiliser l'interpréteur Nit pour en faire une machine virtuelle présente plusieurs avantages :

— Pouvoir très rapidement exécuter des programmes avec la machine virtuelles

— Simplifier les tests, en particulier les tests de non-régression

— Facilité de développement par rapport à un compilateur à la volée

— Pouvoir travailler incrémentalement en remplaçant progressivement des éléments

— Ne pas re-développer un parseur pour un langage en entrée de la machine virtuelle

Cet interpréteur pose cependant quelques problèmes car il est difficile de s'affranchir de son architecture. Tout d'abord, le modèle ainsi que l'AST sont globalement construits, c'est-à-dire en monde fermé. Le chargement dynamique impliquant de découvrir progressivement le modèle vivant du programme, il faudra donc pour l'implémenter de manière réaliste ne se servir que des informations sur les classes **chargées**.

La simulation de la compilation à la volée du code implique aussi qu'il sera impossible de chronométrer en mesurant les temps d'exécutions de la machine virtuelle. Et ce par rapport à l'interpréteur ou même entre plusieurs protocoles de compilation/recompilation. Comme illustré dans le chapitre précédent, il faudra alors plutôt utiliser des méthodes statiques pour mesurer l'efficacité en comptant différents éléments.

Méthodes natives et interprétation

Certaines parties de la librairie de Nit sont implémentées en langage C via la *FFI* de Nit [Laferrière, 2012]. La FFI pour *Foreign Function Interface* permet d'écrire le code d'une méthode dans un autre langage que Nit. Ce mécanisme permet de passer des paramètres de Nit vers le langage C par exemple, il est possible d'accéder aux attributs, d'appeler des méthodes etc. L'implémentation de la FFI pour le C implique de faire appel à un compilateur C pour ces méthodes.

Ces méthodes sont dites *natives* et ne sont pas interprétables facilement par une machine virtuelle Nit (ou l'interpréteur). Le compilateur standard de Nit compile vers du C, il peut donc substituer du Nit par du C lors de la compilation. Une version limitée de la FFI, la *Light FFI*, est tout de même disponible pour ces deux systèmes d'exécution. Le compilateur C sera appelé directement par le système d'exécution (pendant l'exécution) pour compiler une méthode vers une librairie partagée, sous Linux, cela créera un *.so*. L'appel pourra ensuite être effectué pendant l'exécution. Pour l'instant, cette la *Light FFI* ne permet pas d'évaluer l'intégralité des programmes Nit et reste assez lente. En particulier, la machine virtuelle n'est pas méta-évaluable car elle contient des portions de C trop complexes pour une compatibilité avec la *Light FFI*.

Les benchmarks utilisables sont donc réduits ou ralentis par la *Light FFI*, mais il est tout de même possible d'exécuter l'interpréteur Nit ou son compilateur.

Problème du langage en entrée

Dans une machine virtuelle Java par exemple, un langage de bytecode est utilisé en entrée pour être exécuté. Le bytecode Java est construit sur le modèle d'une classe par fichier. Le chargement dynamique de classes lors de l'exécution de la machine virtuelle correspond donc à charger et parser un fichier lors de la première instanciation d'une classe jusqu'alors inconnue.

Le chargement dynamique dans une machine virtuelle Java est effectué en plusieurs étapes :

— Ouvrir le fichier de bytecode et le parser

— Vérifier que le bytecode Java est correct (conforme aux spécifications du langage)

— Construire en mémoire le modèle de la classe

Le modèle du programme et son AST étant complètement construits avant l'exécution, on a potentiellement accès à tous les éléments du programme de manière globale. Ce n'est pas un problème dans l'interpréteur Nit. Par contre, dans la machine virtuelle,

cela pose un problème de réalisme puisque nous avons potentiellement accès à tout le modèle du code et nous pourrions en théorie faire des optimisations en monde fermé.

Il faut donc s'imposer des contraintes : ne pas se servir des informations du modèle des classes qui n'ont pas encore été chargées.

Idéalement, il faudrait spécifier et développer un langage de bytecode complet. Ce langage serait similaire au bytecode Java avec bien sûr des extensions pour l'héritage multiple et le méta-modèle de Nit. Ce sujet dépasse néanmoins le cadre de ce travail.

5.8.3 Implémentation de l'héritage multiple

Comme expliqué dans le chapitre 2, l'implémentation de l'héritage multiple n'est pas un problème trivial en chargement dynamique.

La seule technique relativement efficace et compatible avec le chargement dynamique est le hachage parfait [Ducournau, 2008, Ducournau and Morandat, 2011]. C'est en effet la seule technique d'implémentation de l'héritage multiple qui possède les propriétés suivantes :

— Temps constant

— Espace linéaire (dans la taille de la relation de spécialisation)

— Compatible avec l'inlining (courte séquence de code)

— Compatible avec le chargement dynamique

Cette technique a également l'avantage d'être utilisable pour implémenter les trois mécanismes objets. Elle impose en contrepartie une certaine disposition mémoire des tables de méthodes.

5.9 Conclusion

Ce chapitre a présenté les différents outils dont nous disposons pour réaliser le projet de machine virtuelle.

Nous avons décidé de travailler avec le langage Nit. Pour développer la machine virtuelle, il était initialement prévu d'utiliser VMKit, un framework de machine virtuelle. Mais le projet a été abandonné, nous avons donc choisi de travailler à partir d'un des systèmes d'exécution de Nit : son interpréteur. Progressivement, une machine virtuelle sera dérivée de cet interpréteur et plusieurs éléments seront simulés pour s'affranchir des contraintes de ce système. La compilation à la volée sera donc simulée et par conséquent il faudra compter différents éléments pour mesurer l'efficacité des protocoles de compilation/recompilation plutôt que de mesurer le temps d'exécution.

Cette machine virtuelle sera centrée sur l'implémentation des mécanismes objet, le hachage parfait sera la technique d'implémentation de l'héritage multiple utilisée.

Chapitre 6

Architecture de la machine virtuelle

Contents

6.1 Introduction

Dans ce chapitre, nous présenterons l'architecture de la machine virtuelle Nit. Pour éviter d'écrire une machine virtuelle complète à partir de zéro, nous avons réutilisé l'interpréteur Nit en tant que base de code pour progressivement produire une machine virtuelle. Nous décrirons les différentes étapes qui consistent à rajouter des éléments pour en faire une machine virtuelle. Ensuite nous décrirons comment fonctionne les mécanismes de compilation du code dans la machine virtuelle.

De manière générale, l'implémentation se veut la plus paresseuse possible, et donc, en construisant au dernier moment les différentes structures dont nous avons besoin. Cette philosophie constitue un point important du design de la machine virtuelle. Le chapitre suivant a été partiellement publié dans [Pagès, 2015].

6.2 Méthodologie de développement

6.2.1 Développement progressif

L'utilisation de l'interpréteur Nit comme base de code pour développer la machine virtuelle a été un grand avantage. En effet, l'interpréteur était déjà fonctionnel au début du projet. Il a en effet un but de prototypage dans l'écosystème Nit. Les nouvelles fonctionnalités du langage sont souvent rajoutées d'abord dans l'interpréteur pour être testées avant d'être implémentées dans les différents compilateurs. Ce système est donc relativement simple à appréhender.

Le choix de travailler avec l'interpréteur de Nit implique que la machine virtuelle sera aussi écrite en Nit, ce qui constitue d'ailleurs une preuve de faisabilité supplémentaire pour le langage.

Pour remplir cet objectif, la philosophie générale a été de travailler progressivement, il fallait en effet garder un système complètement fonctionnel tout en rajoutant les éléments qui constituaient la machine virtuelle.

Dans les différents modules qui constituent la machine virtuelle, les composants ont été développés avec la chronologie suivante :

1. Coeur de la machine virtuelle, mécanismes de chargement dynamique et construction des tables de méthodes
2. Simulation de la compilation
   (a) Numérotation des variables locales dans les méthodes
   (b) Constructions des blocs de base
   (c) Algorithme SSA
3. Construction d'un modèle pour la représentation intermédiaire
4. Module de préexistence
5. Module des implémentations des entités du modèle
6. Module de statistiques (pour les expérimentations)
7. Développement des divers protocoles de compilation/recompilation

Bien entendu, des mise à jours des différents module de la machine virtuelle ont été réalisées au cours du développement. Le découpage des fonctionnalités a néanmoins facilité les phases de tests au cours du temps. Au cours des étapes du développement, l'objectif était d'utiliser les tests unitaires de l'interpréteur Nit avec la machine virtuelle, pour s'assurer du bon fonctionnement.

6.2.2 Modules, héritage et raffinement

Dans la mesure du possible, nous avons massivement utilisé les modules et le raffinement de Nit pour séparer les concepts de la machine virtuelle. Une hiérarchie de modules est prévue dans le but d'aider le développement, ainsi qu'au travail de débogage.

De manière générale, l'ordre chronologique de développement correspond à un nouveau module (au sens du langage Nit) dans la machine virtuelle. Un module correspond à l'ajout d'une fonctionnalité, pour bien séparer les concepts il paraissait important qu'un sous-module ajoute une fonctionnalité à l'application sans modifier les super-modules.

Ainsi, il est possible dans la machine virtuelle Nit de "débrancher" de la hiérarchie un module, ainsi que tout ses sous-modules, sans mettre en péril le fonctionnement de la machine virtuelle.

La figure 6.1 présente la hiérarchie des différents modules dans la machine virtuelle Nit. Les flèches montrent la relation de hiérarchie entre les modules. Un module plus bas dans la hiérarchie importera le module au dessus. Les classes du modules du bas peuvent contenir des héritages et raffinements par rapport aux classes du super-module. Les deux modules en bleu sur le schéma sont les deux exécutables, de la machine virtuelle et de l'interpréteur. Avant le développement de la machine virtuelle Nit, deux modules étaient déjà présents dans l'interpréteur : **niti** et **naive_interpreter**. Le premier permet juste de lancer les différentes phases de l'interprétation en commençant par l'analyse lexicale et syntaxique pour ensuite démarrer l'interpréteur à partir du point d'entrée du programme (la méthode `main`). Le deuxième module contenait tout ce qui était relatif à l'interprétation : les structures de données nécessaires ainsi que le parcours de l'arbre syntaxique avec les actions à effectuer pour chacun des nœuds.

Les modules de la machine virtuelle sont les suivants :

nitvm Ce module est le lanceur de la machine virtuelle. Il est très similaire au lanceur de l'interpréteur, le modèle ainsi que l'AST sont construits et l'exécution est démarrée.

niti Ce module est le lanceur de l'interpréteur.

naive_interpreter L'interpréteur original du langage Nit. Cet interpréteur existait avant cette thèse, il contient tous les mécanismes nécessaires à produire l'exécution du code. L'exécution est démarrée depuis le module lanceur de l'interpréteur, celui-ci demandera la construction du modèle ainsi que l'AST.

perfect_hashing Ce module contient toutes les opérations nécessaires au calcul des tables de hachage parfaites ainsi qu'à la numérotation parfaite. Le hachage parfait impose aussi de garder une structure qui stocke les identifiants libres encore disponibles, voir [Ducournau and Morandat, 2011].

virtual_machine Ce module introduit le chargement dynamique : les structures associées aux classes et aux objets sont construites. Les tables de méthodes, tables de hachages, structure des objets, etc. sont introduites dans ce module. Il redéfinit également tous les mécanismes de l'interpréteur pour utiliser ceux de la machine virtuelle, les nouvelles structures de données sont alors utilisées.

variables_numbering Une analyse locale des méthodes est effectuée de manière paresseuse. Les variables se voient attribuer un indice. Lors de l'appel d'une méthode

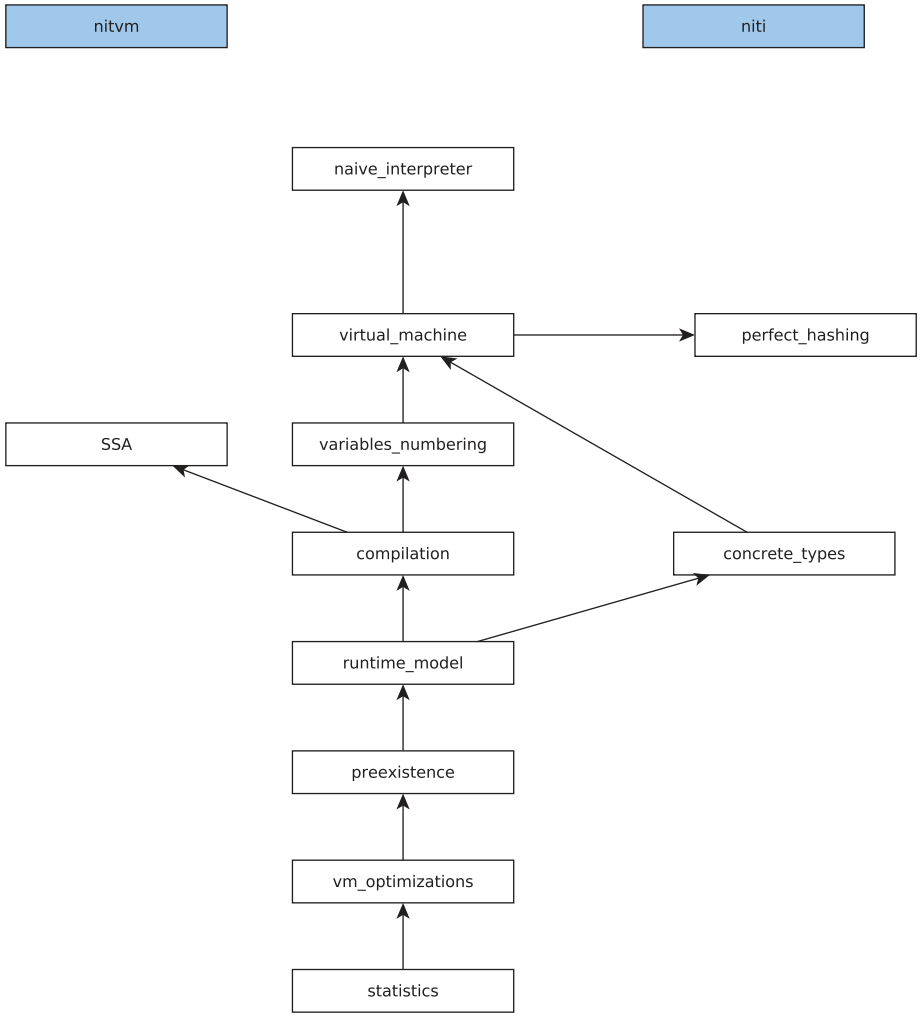

FIGURE 6.1 – Hiérarchie des modules dans la machine virtuelle Nit

un environnement est créé pour accéder aux valeurs des variables à partir de leur indice. Ce module est essentiellement une optimisation des mécanismes de l'interpréteur concernant les accès aux variables.

SSA Ce module implémente l'algorithme SSA. Il est d'ailleurs utilisable en dehors de la machine virtuelle : il présente donc une interface pour donner la version SSA du code d'une méthode particulière. Pour cela, il faut dans un premier temps construire les blocs de base (*basic blocks*) de la méthode, ce qui est fait dans ce même module.

compilation Ce module a pour but de faire le lien entre différents autres modules. Il sert également à ce que les sous-modules puissent simuler la compilation à la volée des méthodes. Il est également chargé de concilier la numérotation des variables avec l'algorithme SSA qui duplique certaines variables.

concrete_types Simule certaines informations que l'on pourrait obtenir dans un bytecode. Par exemple, si une classe est une classe finale ou si une méthode retourne un type final, voir chapitre 9 pour l'utilisation de ce module.

runtime_model Construit toute la représentation intermédiaire du code des méthodes ainsi que tout le modèle étendu décrit dans le chapitre 7. Ce modèle servira ensuite de base pour calculer les implémentations et tout ce qui est lié aux optimisations.

preexistence Calcule la préexistence des expressions du code, voir chapitre 8.

vm_optimizations Les implémentations du code sont calculées ici. En se basant sur les différents super-modules qui fournissent des informations sur le code, des décisions d'optimisations sont prises. Le module est aussi responsable des recompilations du code et plus généralement de l'implémentation des protocoles.

statistics Calcule toutes sortes de métriques sur l'exécution du programme. Différentes données sont collectées comme par exemple les implémentations des mécanismes objet du programme, le nombre de fois qu'un mécanisme objet est utilisé etc. Un export de certaines statistiques en `.csv` ou directement sous forme de tableaux latex est effectué.

6.3 Chargement dynamique

Comme dit précédemment, le chargement dynamique dans la machine virtuelle Nit impose de ne pas utiliser le modèle global afin de ne pas avoir accès à des informations de classes non-chargées. Nous utiliserons donc un modèle vivant du code, qui sera construit au fil de l'exécution par la machine virtuelle.

Le chargement dynamique de classe est par essence paresseux : le chargement d'une classe est donc effectué lorsqu'on exécute pour la première fois une instanciation d'une classe pas encore chargée. La sémantique du chargement est la suivante : charger une classe signifie charger récursivement d'abord ses superclasses.

Charger une classe est une opération assez complexe effectuée en plusieurs étapes :

1. Chargement récursifs des superclasses

2. Ordonnancement des superclasses pour le calcul de la table de méthodes de la classe chargée : on choisit l'ordre des superclasses en suivant une heuristique *la règle du préfixe* [Ducournau and Morandat, 2012] qui essaye de choisir un ordre favorisant la classe ayant introduit le plus de propriétés. Cela doit permettre de diminuer les changements de place des classes les plus importantes, et donc diminuer les appels en héritage multiple.
3. Calcul du paramètre de hachage et allocation d'un identifiant à la classe via la numérotation parfaite.
4. Mise à jour des propriétés locales de la classe (attributs et méthodes) et mise à jour du graphe d'appel.
5. Construction de la table de méthodes :
   (a) Calcul des delta pour le schéma mémoire des attributs, les delta sont les *offsets* des différents attributs groupés par classe d'introduction.
   (b) Allocation via le langage C et la FFI de Nit de la table des méthodes et initialisation de la table avec les identifiants de hachage, delta, et la table de hachage en partie négative.
   (c) Remplissage de la table des méthodes par les adresses des méthodes les plus spécifiques de cette classe. Les méthodes qui ne sont pas vivantes sont quand même positionnées dans la table de méthodes.

Dans un but d'optimisation nous considérons deux types de chargements de classes :

— Chargement explicite : la classe en question est instanciée

— Chargement implicite : une super-classe d'une classe explicitement instanciée

Lors d'un chargement implicite de classe, seules les étapes 1 à 4 sont effectuées. La classe chargée implicitement contient les informations nécessaires pour effectuer un test de sous-typage par rapport à cette classe. Pour cela, nous avons donc uniquement besoin de connaître son identifiant. Un chargement explicite peut par contre entraîner des appels de méthodes dont le receveur est la classe chargée, il faut donc une construction complète de sa table des méthodes. Cette optimisation de la construction des tables de méthodes s'inscrit à nouveau dans un fonctionnement "paresseux" de la machine virtuelle.

Une classe implicitement chargée peut par la suite être explicitement chargée, dans ce cas nous effectuons uniquement la construction de la table des méthodes.

Règle du préfixe La règle du préfixe est une heuristique qui optimise l'ordre des classes dans les tables de méthodes et les objets pour diminuer a priori les pertes de l'invariant de position des classes importantes. L'ordre des classes est calculé comme suit :

— Si il y a plusieurs superclasses directes, on choisit l'une d'elle qui devient la classe **préfixe**

— L'ordre des superclasses de la classe préfixe est le préfixe de l'ordre des superclasses de la classe considérée (celle que l'on charge actuellement)

— Les autres superclasses directes forment un suffixe dont l'ordre est quelconque
— La classe préfixe est choisie en suivant une heuristique basée sur le nombre de propriétés introduites pour maximiser le nombre de sites dont la position est invariante
— On aura un préfixe pour les attributs (dans les objets), et un préfixe pour les méthodes (table de méthodes) qui peuvent être différents

6.3.1 Représentation mémoire utilisée

Représentation mémoire des classes

Pour tout ces mécanismes objets, une représentation unifiée de la mémoire est utilisée. Cette représentation mémoire est imposée par l'utilisation du hachage parfait ainsi que l'inlining du test de Cohen (voir chapitre 2).

La figure 6.2 présente la représentation mémoire utilisée pour les tables de méthodes dans la machine virtuelle.

Dans cet exemple, nous avons quatre classes `A`, `B`, `C` et `D`. La figure 6.2 illustre la table de méthodes de la classe `D`. La table de méthode est déjà partagée en deux : la partie négative contient la table de hachage parfaite. La partie positive contient surtout les adresses des méthodes.

L'ordonnancement des superclasses dans l'exemple a sélectionné l'ordre suivant : `A C B D`, les blocs de chacune des classes sont donc positionnés dans cet ordre. Lors des premières étapes du chargement de la classe `D`, le hachage parfait a calculé un masque de hachage pour cette classe et lui a attribué un identifiant.

Le masque de hachage est situé à la première case mémoire de la table de méthodes, ensuite les blocs de chacune des classes sont présents. Ces blocs contiennent chacun trois informations :

1. L'identifiant de la classe
2. Le delta de la classe pour l'accès aux attributs (abrégé *d* dans la figure)
3. Toutes les adresses des méthodes introduites par la classe, en cas de redéfinition, les nouvelles méthodes seront positionnées au même emplacement que la méthode d'introduction

La partie négative de la table contient la table de hachage parfaite. Elle contient des pointeurs vers des blocs de méthodes dans la partie positive de la table. Ces pointeurs sont positionnés en utilisant le hachage parfait : *masque* et *id*. Le masque est situé au début de la table de hachage tandis que l'identifiant est celui d'une classe. Cette opération donnera un entier, qui est inférieur ou égal à la taille de la table de hachage, cet entier correspond à la position du pointeur vers la classe de l'identifiant.

Représentation mémoire des objets

La représentation des objets est plus simple que celle des classes. Elle est illustrée dans la figure 6.3. Un pointeur vers la table des méthodes est présent. Cette figure

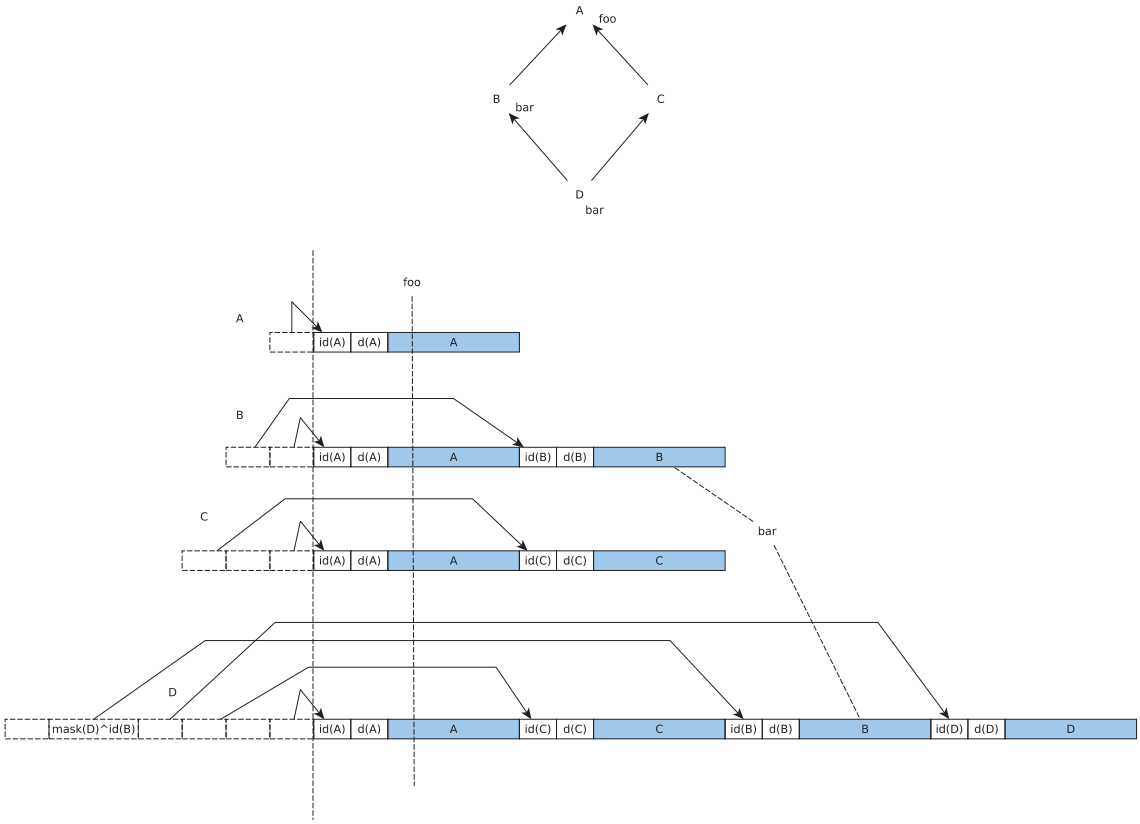

FIGURE 6.2 – Représentation mémoire de la table de méthodes de la classe `D`

présente le schéma mémoire d'une instance de la classe `D` en reprenant la hiérarchie de classes de l'exemple précédent.

Les objets contiennent uniquement des blocs mémoires avec un bloc pour chacune des classes. La taille d'un bloc correspond au nombre d'attributs introduits par la classe en question. Le delta est la position d'un bloc par rapport au début de la mémoire de l'objet.

L'objet contient un pointeur vers la table de méthodes de sa classe. Le premier bloc d'attributs est positionné au début de l'objet, la racine de la hiérarchie de classes aura donc toujours la valeur 0 comme delta. Dans cet exemple, si la classe `A` a introduit 5 attributs, alors le bloc de `B` aura la position 5 en tant que delta.

Ces différents blocs contiennent des pointeurs vers la valeur des attributs. En Nit, il est interdit d'accéder à un attribut qui n'est pas initialisé et il existe une instruction pour tester si un attribut est bien initialisé. Pour implémenter cette instruction, chacune des valeurs des attributs est initialisée avec une valeur spéciale qui permettra de savoir

si l'objet est initialisé ou pas.

Il faut noter que dans cet exemple, l'ordonnancement des classes dans la table de méthodes est le même que dans les objets, mais ce n'est pas nécessairement le cas. La machine virtuelle peut dissocier les deux ordres dans un but d'optimisation.

De plus, si jamais une classe n'introduit aucun attribut, alors elle n'aura pas de bloc réservé dans les objets, son delta sera alors strictement égal à celui de la classe située juste après dans l'ordonnancement. Dans cette représentation, la généricité a été traitée comme étant effacée, les objets qui sont instance de la classe `B[T]` et `B[U]` posséderont tous la même table de méthodes.

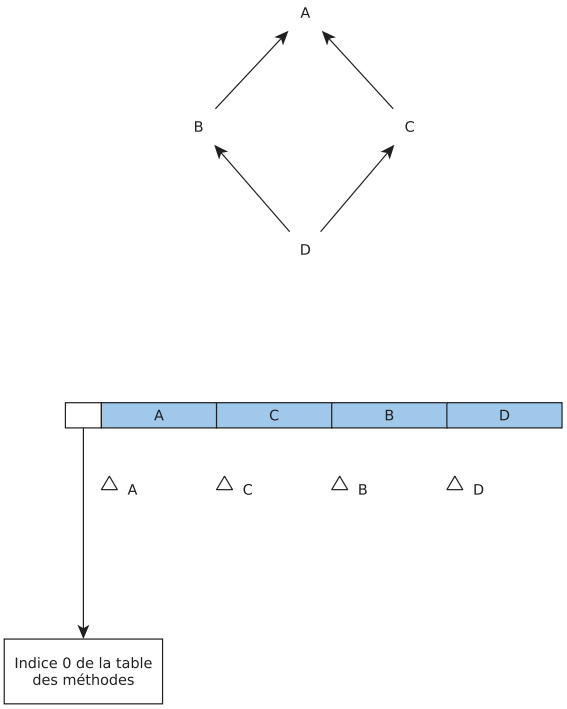

FIGURE 6.3 – Représentation mémoire des objets

6.4 Compilation à la volée

Développer un véritable compilateur à la volée qui produirait du code binaire en sortie s'est avéré être une tâche trop complexe et chronophage dans le cadre de cette thèse. Nous avons donc simulé, de la manière la plus réaliste possible, cette compilation à la volée. Cette section présente les différentes étapes de cette compilation.

Dans l'interpréteur Nit, nous n'avons pas de véritable représentation intermédiaire, nous devons donc travailler directement à partir de l'arbre syntaxique des méthodes.

La philosophie générale de ces simulations est d'avoir les mêmes contraintes qu'avec un véritable compilateur à la volée. Les différentes opérations sont effectuées de manière paresseuse, donc au premier accès à une méthode pas encore compilée.

6.4.1 Numérotation des variables

La première étape est la numérotation des variables locales. Il s'agit d'une étape de pure optimisation.

Dans l'interpréteur Nit, les accès aux variables locales des méthodes sont effectués via une table de hachage qui est créée au début de l'exécution de la méthode. Chaque fois qu'un accès en lecture ou écriture est effectué, la table de hachage est utilisée. Ces tables étaient un facteur significatif de mauvaises performances de l'interpréteur.

Nous avons donc numéroté les variables statiquement dans la machine virtuelle. À la compilation, un parcours du code local de la méthode est effectué pour récupérer toutes les variables locales de la méthode. Elles sont ensuite numérotées, en commençant par le receveur de la méthode. Les nœuds d'accès aux variables de l'AST sont modifiés pour stocker le numéro de la variable. On mémorise ensuite le nombre de variables locales qui peuvent exister en même temps dans la méthode : il s'agira de la taille de l'environnement.

À l'exécution de la méthode, un tableau de la taille de l'environnement est créé. Nous pouvons donc accéder directement au tableau en fonction de l'indice de la variable sans utiliser une coûteuse opération de hachage, les valeurs des variables sont contenues dans ce tableau. En pratique, des gains de performances jusqu'à 50% sur le temps d'exécution ont été obtenus grâce à cette optimisation. Cette opération de numérotation est définitive et réalisée une seule fois. Même si la machine virtuelle simule uniquement la compilation à la volée, cette optimisation permet de redéfinir du côté de la machine virtuelle les opérations d'accès aux variables qui n'est plus géré du côté de l'interpréteur.

6.4.2 Construction des blocs de base

Pour les étapes ultérieures de la compilation (analyses statiques) nous avons besoin de transformer le code du programme exécuté. La machine virtuelle travaille en effet avec l'AST du programme qui est une représentation arborescente par nature imbriquée.

Or, de nombreuses analyses statiques ont besoin de travailler sur une représentation de type graphe de flot de contrôle (CFG, *Control Flow Graph*) en entrée.

Un *bloc de base* (ou *basic block*) est défini dans [Aho et al., 2007] comme étant une structure maximale d'instructions ayant les propriétés suivantes :

— Le flot de contrôle ne peut entrer dans le bloc que par la première instruction, il n'y a aucun branchement vers l'intérieur du bloc

— Le flot de contrôle ne peut pas quitter le bloc par un branchement sauf à sa dernière instruction

Les blocs de base deviennent des nœuds du graphe de flot de contrôle, les arcs du graphe indiquent les blocs successeurs que peut atteindre un bloc.

Cette construction en bloc de base est propre à chaque méthode. Le premier bloc représente l'entrée dans la méthode, nous avons ensuite défini un bloc de retour, qui sera un point de passage obligé pour toutes les instructions qui permettent de retourner (sortir de la méthode).

La construction des blocs de base est implémentée par un visiteur qui effectue un parcours du code de la méthode méthode. La construction générale des blocs de base est faites par séparation successive des blocs de base :

1. Un premier bloc est créé au début de l'analyse, il contient toutes les instructions de la méthode
2. Le visiteur parcourt le code dans le bloc jusqu'à la rencontre d'un branchement (instruction conditionnelle, boucle, etc) :
   - le bloc courant est clôturé
   - Un nouveau bloc est créé, toutes les instructions suivantes du bloc sont recopiées dans le nouveau bloc créé
   - Le bloc précédent est chaîné au nouveau

La procédure de scission de blocs est ensuite effectuée récursivement dans les sous-blocs. La principale difficulté de cet algorithme consistait à gérer correctement les différentes structures pouvant apparaître dans le code telles que les boucles, tests ou les instructions *break* qui permettent de sortir d'une boucle.

6.4.3 Algorithme SSA

L'algorithme SSA *Single Static Assignment* [Cytron et al., 1991] est une transformation du code source (des variables locales) pour isoler les affectations dans les variables en créant des variables affectées une seule fois. Cet algorithme servait à l'origine à supprimer les affectations inutile ou redondantes entre les variables [Ferrante et al., 1987, Alpern et al., 1988], il sert également à construire le graphe de dépendance entre les variables. Il est aujourd'hui souvent utilisé en tant que base de nombreuses autres analyses statiques.

Pour construire la forme SSA du programme, on a besoin dans un premier temps de sa structure sous forme de bloc de base. L'algorithme consiste à fabriquer de nouvelles variables locales à chaque affectation d'une variable déjà affectée. Des ϕ-variables sont aussi utilisées, ce sont des entités représentant le point de jonction d'une même variable qui a été affectée dans deux branches du flot de contrôle. Ces ϕ-variables sont donc placées en sortie de constructions telles que des tests (if, else) ou des boucles.

L'algorithme SSA est souvent utilisé dans les représentations intermédiaires des machines virtuelles comme montré dans le chapitre 3.

Dans la machine virtuelle Nit, nous utilisons SSA exclusivement pour les dépendances entre les variables (nécessaires pour les analyses de préexistence). Généralement il est nécessaire de détruire les ϕ-variables pour avoir un code exécutable car ces entités sont abstraites. Du point de vue des dépendances des variables nous n'avons pas besoin d'effectuer cette étape de déconstruction car nous ne produisons pas de code exécutable. Nous ne supprimons donc pas les ϕ-variables : nous avons donc deux types de variables :

— Variable SSA : une seule dépendance

— ϕ-variable : plusieurs dépendances

SSA ne met pas à l'abri de dépendances circulaires dans les variables. Or, ces cycles peuvent poser problème pour certaines analyses que nous effectuons. Après la génération de SSA, nous passons donc en revue les dépendances des variables pour éliminer les cycles. Quand un cycle est détecté nous le supprimons en deux étapes :

1. Collecte de toutes les dépendances contenues dans le cycle (pour toutes les variables)
2. Fusionner cet ensemble et affecter le résultat de chaque fusion à toutes les variables du cycle

L'algorithme SSA utilisé dans la machine virtuelle est tiré du *SSA-book* [Auteurs multiples, 2016].

6.5 Représentation intermédiaire et implémentations

6.5.1 Génération de la représentation intermédiaire

L'arbre syntaxique du programme n'est pas parfaitement adapté au calcul des implémentations dans notre cas. Il contient en effet de nombreuses informations qui ne sont plus nécessaires au moment de la compilation. l'AST est également très générique et peut cacher des opérations élémentaires assez subtiles.

Nous avons utilisé une représentation intermédiaire qui contient uniquement les éléments nous intéressant pour la compilation et qui est liée au méta-modèle de Nit. Cette représentation sera précisément décrite au chapitre 7.

La représentation intermédiaire concerne essentiellement les mécanismes objets. Pour chacune des expressions d'un mécanisme objet de l'AST une expression de la représentation intermédiaire est générée. Notre représentation intermédiaire concerne essentiellement les instructions objet et les accès aux variables, les noeuds de type structure de contrôle n'ont pas de représentation. Nous profitons de divers parcours lors de la génération de la forme SSA pour récolter les différents noeuds qui nous intéressent pour la représentation intermédiaire.

Après le calcul de SSA, nous effectuons un parcours des noeuds spécifiques que nous avons récolté pour en générer la représentation intermédiaire, en étant aidé par la forme SSA du code. La génération est donc effectuée en deux temps : la génération de SSA ainsi que la récolte des noeuds puis ensuite la génération de la représentation intermédiaire. Finalement, les entités de la représentation intermédiaire créées son liées à leur nœud d'AST correspondant, ce qui permettra d'utiliser les entités à l'exécution plutôt que l'AST.

6.5.2 Calcul de la préexistence

Pour effectuer des optimisations, nous avons implémenté l'analyse de préexistence [Detlefs and Agesen, 1999]. La préexistence est calculée pour les entités de la représen-

tation intermédiaire.

Pour cela, nous nous basons sur les dépendances des variables construites à partir de SSA. Si la variable n'a qu'une seule dépendance alors il s'agit d'une variable SSA et sa préexistence est calculée pour la dépendance en question.

Si la variable a plusieurs dépendances, il s'agit alors d'une ϕ-variable et nous calculons la préexistence pour chacune des dépendances. La préexistence est une notion conservatrice : toutes les dépendances doivent être préexistantes pour que l'expression globale soit préexistante. La valeur de préexistence stockée pour une ϕ-variable représente la fusion des valeurs de toutes les dépendances de la variable. La préexistence sera une notion développée dans le chapitre 8.

6.5.3 Calcul des implémentations

La dernière étape de la compilation est le calcul des implémentations. Les implémentations sont calculées pour chacune des expressions de la représentation intermédiaire.

C'est le protocole de compilation/recompilation utilisé qui est responsable de ce calcul. L'implémentation peut donc être plus ou moins optimisée en fonction du choix du protocole. Il faut rappeler que les implémentations ne concernent que les mécanismes objets, par conséquent le calcul des implémentations cherchera à déterminer s'il faut utiliser le hachage parfait ou des implémentations optimistes plus efficaces comme SST ou des appels statiques.

Le calcul des implémentations sera largement détaillé au chapitre 9.

6.5.4 Implémentation et exécution

L'exécution est toujours effectuée par une visite de l'arbre syntaxique, comme dans l'interpréteur Nit. Cependant, les implémentations sont calculées avant l'exécution. Nous utilisons l'entité de la représentation intermédiaire dans un nœud d'AST pour récupérer cette implémentation puis l'exécuter. Ces implémentations sont calculée au premier appel d'une méthode, le protocole est ensuite responsable des mises à jours et recompilations éventuelles.

Le protocole de compilation/recompilation effectuera les changements d'implémentations selon sa politique. Malgré le fait que la machine virtuelle ne contienne pas un véritable compilateur à la volée, la simulation de compilation est réaliste puisque nous utilisons uniquement nos entités et les implémentations calculées.

Nous avons choisi d'exécuter réellement les implémentations calculées pour nous assurer qu'elles étaient correctement générées. Ainsi, lors de l'exécution il est possible de s'assurer que les mécanismes originels de l'interpréteur produisent bien le même résultat que nos implémentations. Par exemple, on s'assure qu'un appel de méthode renvoie bien la même méthode que celle que l'interpréteur aurait retournée.

Un véritable compilateur à la volée devrait, en plus des différentes étapes décrites dans ce chapitre, produire du code exécutable. Pour chaque site objet, il faudrait alors produire la séquence de code déterminée en fonction de l'implémentation choisie par le protocole.

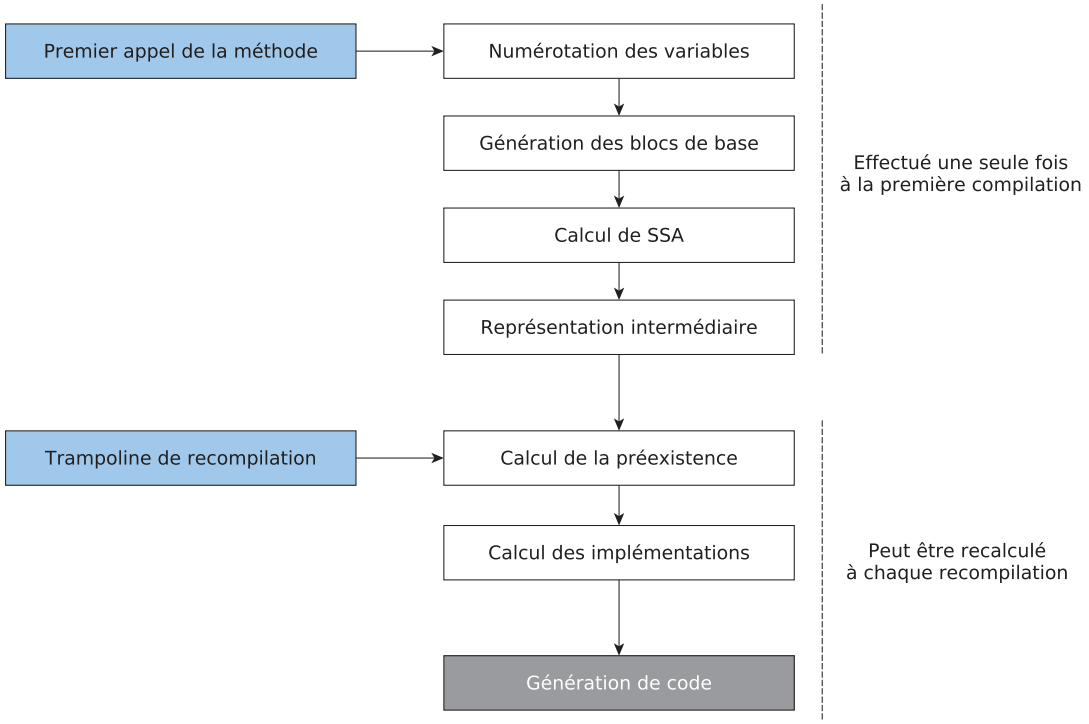

FIGURE 6.4 – Déroulement de la compilation d'une méthode

6.6 Récapitulatif de la compilation

Les différentes étapes de la compilation sont détaillées dans la figure 6.4. Les premières étapes de la compilation sont toujours effectuées une seule fois. Éventuellement selon sa politique, le protocole de compilation/recompilation peut recalculer différents éléments dans la préexistence ou recompiler certaines implémentations. Ces étapes sont par contre toutes préalables à la première exécution d'une méthode et sont donc effectuées de manière paresseuse.

La dernière boîte du schéma représente l'étape de production de code machine, qui n'est pas effectuée pour le moment.

6.7 Conclusion

Nous avons présenté dans ce chapitre l'architecture de la machine virtuelle Nit. L'architecture est séparée à travers plusieurs modules de Nit qui rajoutent chacun une couche dans la conception de la machine virtuelle.

Nous avons ensuite présenté les mécanismes de compilation dans la machine virtuelle. Cette compilation est effectuée de manière complètement paresseuse et en plusieurs étapes. La dernière étape étant le calcul des implémentations pour les différents mécanismes objets contenus dans la méthode. Cette compilation est simulée car la machine virtuelle ne produit pas de code machine, mais elle est réaliste pour effectuer des expérimentations autour des protocoles de compilation/recompilation.

Troisième partie

Protocoles de compilation et recompilation

Chapitre 7

Modèle des protocoles

Contents

7.1 Introduction

Représenter un protocole de compilation/recompilation est une tâche difficile. Nous avons cherché à intégrer un modèle pour représenter les protocoles dans le modèle de Nit existant.

Le méta-modèle existant de Nit permet de représenter le modèle du programme : ses classes et leurs propriétés. Le modèle présenté dans ce chapitre étend le méta-modèle existant pour représenter également les implémentations du programme.

En effet, dans une machine virtuelle, les implémentations sont assez souvent changeantes, nous avons donc cherché à les factoriser du mieux possible.

7.2 Méta-modèle de base des langages à objet

Cette section présente et rappelle le méta-modèle utilisé dans langage Nit. Il s'agit d'un travail déjà effectué dans [Privat, 2006] puis dans [Ducournau and Privat, 2011] qui spécifie un méta-modèle pour des langages à objet en typage statique (et héritage multiple).

Ce méta-modèle a deux objectifs principaux :

— Remplacer dans l'arbre syntaxique tous les noms par des objets, pour être libérés des ambiguïtéss liées à l'interprétation d'un même nom dans des contextes différents (para exemple avec l'héritage multiple ou la surcharge statique).

— Distinguer les messages, c'est-à-dire les signatures des appels, de leurs implémentations dans des classes.

Un des effets de ce méta-modèle est de distinguer une propriété introduite dans une classe, d'une propriété redéfinie dans une classe.

Chaque classe contient en effet un certain nombre de nouvelles propriétés. Il peut s'agir de méthodes ou d'attributs, qui sont **introduits** pour la première fois dans cette classe. Une classe peut également **redéfinir** une méthode d'une super-classe.

Deux entités du modèle distinctes permettent de ne pas mélanger l'introduction et la redéfinition : des propriétés globales et locales sont définies.

Définition 17. Propriété globale une propriété globale est introduite à la découverte d'une nouvelle propriété dans une classe. Elle regroupe toutes ses propriétés locales.

Définition 18. Propriété locale une définition spécifique d'une propriété globale dans une classe. Chaque définition d'une méthode est une nouvelle propriété locale (y-compris la première définition).

Une propriété locale possède plusieurs propriétés (tirées de [Privat, 2006]) :

— Définie dans une seule classe

— Appartient à une seule propriété globale

— Peut éventuellement redéfinir une propriété locale

— Est automatiquement héritée dans les sous-classes de sa classe de définition

Les propriétés locales et globales sont déclinées en plusieurs types :

— Méthode

— Attribut

— Type virtuel

— Paramètre de type

Pour chaque nouvelle méthode qui n'est pas une redéfinition, une propriété globale est dite **introduite** par la classe. Ensuite, chaque définition et redéfinition de cette méthode est une propriété locale de la propriété globale en question.

La figure 7.1 présente le méta-modèle de base de Nit. Une classe connaît (par héritage) ou introduit des propriétés globales. Une classe définit également des propriétés locales,

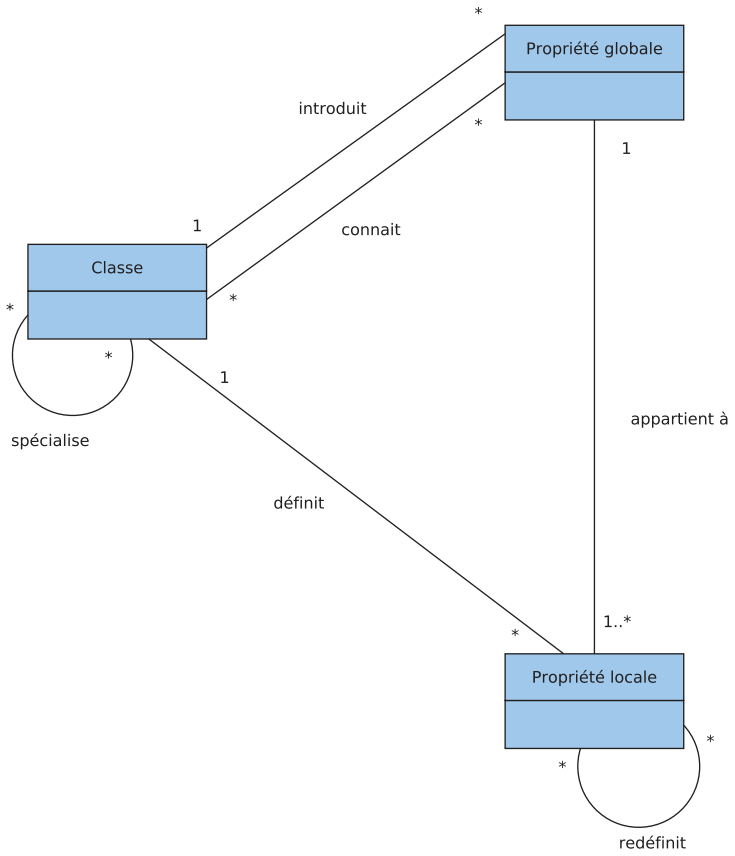

FIGURE 7.1 – Méta-modèle de base de Nit d'après [Ducournau and Privat, 2011]

en fournissant leur code. Tous les éléments présents dans ce modèle ont un nom (classes, propriétés locales et globales), la classe `Nom` n'est pas représentée pour des raisons de lisibilité sur le schéma.

Ainsi la **surcharge statique** qui existe par exemple en Java est expliquée plus clairement avec ce méta-modèle. Une méthode qui en surcharge une autre en Java n'a rien à voir avec la méthode qu'elle surcharge. En particulier, elles n'auront pas la même propriété globale mais possèdent juste le même nom. La surcharge statique est d'ailleurs interdite en Nit pour éviter les confusions, bien que les conflits de noms soient gérés.

7.3 Modèle du code et représentation intermédiaire

Avec le méta-modèle décrit précédemment, un appel de méthode est une opération qui consiste à trouver la bonne propriété locale à exécuter à partir de la propriété globale appelée.

Comme dit dans la section précédente, le méta-modèle existant de Nit représente les éléments d'un programme (classes, méthodes etc). Dans le cadre de la machine virtuelle Nit, nous avions besoin d'une représentation intermédiaire ainsi que d'une extension du modèle qui nous permette de traiter les implémentations du code.

Nous avons étendu le modèle en rajoutant trois entités, qui correspondent à trois niveaux de factorisation :

— PIC-Pattern : relation entre deux classes, la classe du type statique du receveur et la classe d'introduction de la propriété globale (PIC pour *Property Introduction Class*)

— GP-Pattern : relation entre une propriété globale et le type statique du receveur, un GP-Pattern connaît tous ses site-objets

— ObjectSite : un site particulier d'un mécanisme objet du programme

Tous les mécanismes objets : appel de méthode, accès à un attribut et test de sous-typage possèdent des entités du modèle. Néanmoins, les entités du modèle ne contiendront pas les même informations selon le mécanisme objet concerné. En particulier, le test de sous-typage est relativement différent des deux autres mécanismes objets.

Dans les sections suivantes, nous allons présenter progressivement ce modèle en présentant les différents niveaux de factorisation.

7.3.1 Sites objets

Un **site-objet** représente un noeud de l'arbre syntaxique qui réalise l'un des trois mécanismes objets.

Un site-objet est caractérisé par deux informations :

— Le type statique du receveur de l'appel

— La propriété globale appelée

Ce sont les deux informations nécessaires pour effectuer la compilation du site.

Le receveur de l'invocation d'un mécanisme objet correspond au type statique du receveur. Notre modèle ne traite pas de manière spécifique la généricité. Si le receveur de l'appel est une classe paramétrée nous traitons le site de manière homogène. Nous prenons donc la borne du type générique sans autre considération.

L'information importante est donc celle de la classe du receveur du mécanisme objet plutôt que le type. Le receveur de l'appel est donc stocké sous la forme d'une classe, et d'un type mais pour l'instant cette information ne sert pas.

Il y deux grandes catégories de site-objet, les sites de sous-typage et les sites d'accès à une propriétés globales.

Pour les sites-objet possédant une propriété globale, celle ci dépend de la catégorie du *PropSite* :

— Pour un site d'appel de méthode il s'agit de la méthode globale appelée

— Pour un site d'accès à un attribut c'est l'attribut global accédé

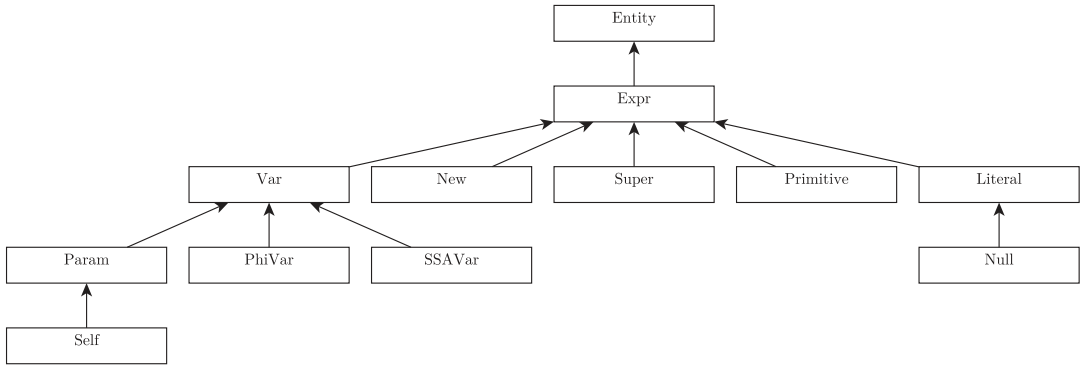

FIGURE 7.2 – Hiérarchie des expressions de la représentation intermédiaire

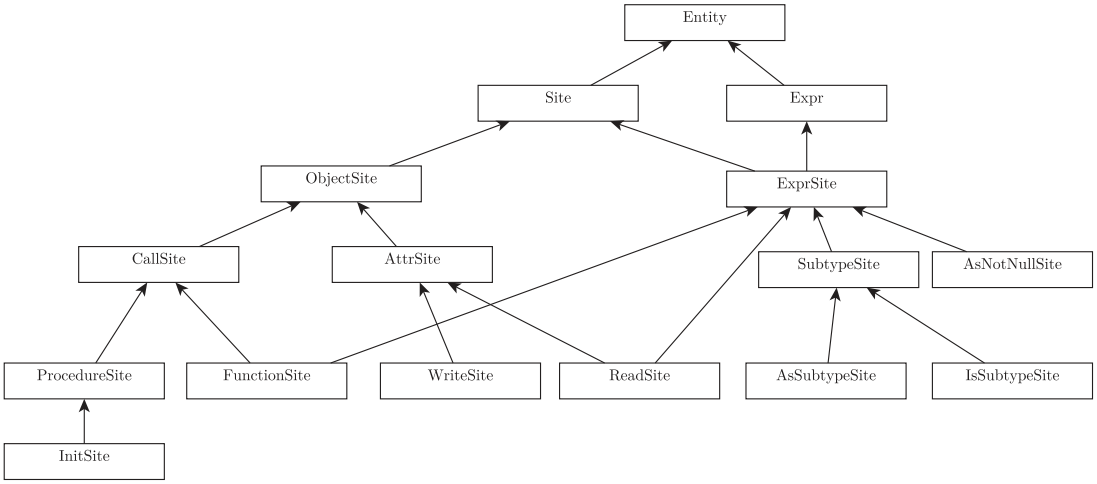

FIGURE 7.3 – Hiérarchie des entités de la représentation intermédiaire

Pour un test de sous-typage la situation est différente, il n'y a pas de propriété globale, la classe cible du test joue par contre un rôle similaire.

Les figures 7.2 et 7.3 présentent la hiérarchie des classes de la représentation intermédiaire utilisée. Voici la description rapide des entités qui sont exclusivement des expressions de ce modèle :

Entity : La classe racine de toute la hiérarchie.

Expr : La racine de toutes les **expressions** de la représentation intermédiaire. Les expressions ont la propriété de pouvoir retourner une valeur.

Var : La racine des classes qui représentent les variables du programme.

SSAVar : Une variable avec une dépendance unique.

PhiVar : Une variable avec plusieurs dépendances.

Param : Un paramètre est une variable locale particulière de la méthode. Il est nécessaire de les distinguer des autres variables pour les analyses de préexistence.

Self : Cette variable particulière représente le receveur de la méthode. Il s'agit aussi d'un paramètre, même s'il est passé de manière implicite dans les langages à objet.

New : Un site d'instanciation d'un objet.

Super : Une appel à la super méthode. Dans le monde Nit les appels à supers sont particuliers avec le raffinement de classes. L'appel à super peut être un appel à super classique dans le sens de Java (La méthode de la super-classe). Mais il peut aussi s'agir d'un appel à super dans le sens du raffinement, dans ce cas la super-méthode est la méthode dans le super module. Dans le cas d'un héritage ou d'un raffinement, la redéfinition d'une méthode masque la super-méthode. Un attribut particulier de *Super* permet de savoir s'il s'agit d'un raffinement ou d'un héritage.

Primitive : La classe des expressions primitives (Int, Char, Bool).

Literal : Les expressions littérales.

Null : Le cas particulier de l'expression nulle.

Les site-objets ont une hiérarchie parallèle présentée dans la figure 7.3 :

Site : La racine de la hiérarchie des site-objets. Tous ces sites ont donc une implémentation.

ExprSite : Les site-objets qui sont aussi des expressions.

AsNotNullSite : Un test vers le type non-nullable correspondant.

SubtypeSite : Les sites qui sont des tests de sous-typage. Une implémentation est présente pour cette classe et ses sous-classes.

AsSubtypeSite : Correspond à un cast, le type source est contraint d'être sous-type du type cible. Si jamais le test échoue, une exception est levée. Le type source est le type dynamique du receveur du test. Ces sites sont considérés comme des expressions du point de vue objet car le type cible est souvent plus spécifique que le type source. Cette information peut être intéressante pour des optimisations.

IsSubtypeSite : Un site qui teste si un type source est sous-type d'un type cible. Cela correspond à un `instanceof` en Java. Techniquement, il s'agit d'une expression puisque ce test renvoie une valeur booléenne, mais nous considérons que cette valeur booléenne est une expression primitive et donc peu intéressante.

PropSite : Les sites ayant une propriété globale à laquelle on accède.

AttrSite : Les sites d'accès aux attributs en lecture ou en écriture.

ReadSite : Un site de lecture d'un attribut, qui est aussi une expression.

WriteSite : Un site d'accès en écriture à un attribut.

CallSite : La racine de toutes les classes d'appel de méthode.

FunctionSite : Un site d'appel de méthode qui a une valeur de retour, et qui est donc une expression également.

ProcedureSite : Un site d'appel d'une procédure, sans valeur de retour.

InitSite : Le cas particulier des sites d'invocation de constructeurs. Dans Nit, un appel (explicite ou implicite/automatique) est toujours présent lors de la création d'un objet. Ces sites sont particuliers du point de vue des implémentations car on saura par exemple toujours quel est le constructeur qui sera appelé statiquement.

La figure 7.4 présente le diagramme de classes des site-objets intégrés avec l'ancien méta-modèle.

7.3.2 GP-Patterns

Les **patterns** sont une factorisation des site-objets. En effet, deux sites peuvent avoir les mêmes caractéristiques, par exemple pour un *PropSite*, même type statique du receveur et même propriété globale accédée.

Un GP-Pattern est une entité qui regroupe les site-objets qui ont les caractéristiques suivantes :

— Même type statique du receveur

— Même propriété globale appelée

Pour les tests de sous-typage, il s'agit des tests ayant la même source et la même cible du test.

Du point de vue des implémentations des mécanismes objets, tous les sites appartenant au même GP-Pattern partageront la même implémentation sauf cas particuliers liés à la connaissance du type concret du receveur du site. La figure 7.5 présente la hiérarchie des GP-Patterns. Les GP-Patterns sont stockés dans la classe du receveur de l'appel pour pouvoir les retrouver à la création des sites. Un site-objet ne possède qu'un seul GP-Pattern.

La figure 7.6 présente la hiérarchie des GP-Patterns dans le modèle étendu.

Pattern : Racine de la hiérarchie des GP-Patterns.

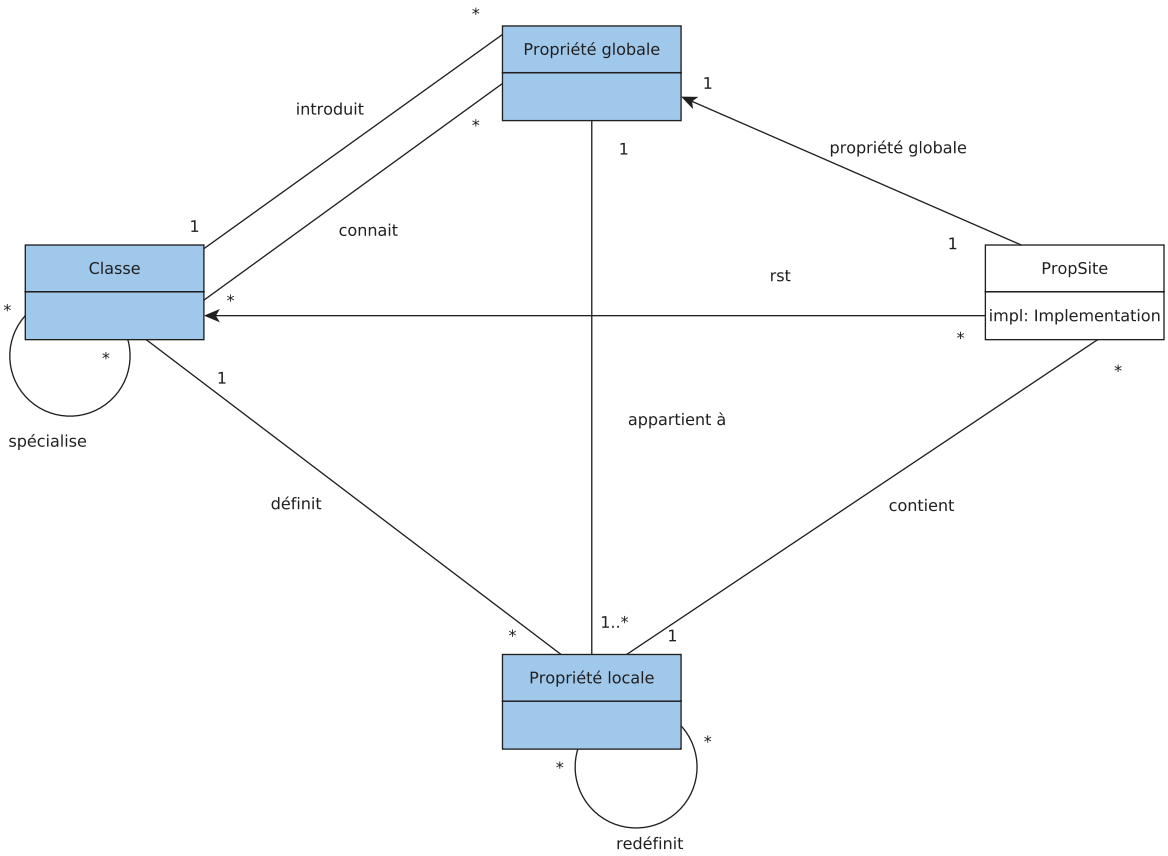

FIGURE 7.4 – PropSite dans le modèle étendu

NewPattern : Un tel GP-Pattern est créé quand une instanciation d'un objet est trouvée dans le code. L'instanciation est toujours accompagnée d'un appel à un constructeur (même implicite). La rencontre d'un new aura donc pour conséquence de créer un site d'appel du constructeur en question.

SitePattern : La super-classe de tous les autres GP-Patterns. Ces GP-Patterns possèdent des sites associés. Cette classe sert à factoriser la création paresseuse des GP-Patterns et d'autres opérations similaires.

ExprSitePattern : Le GP-Pattern de tous les sites qui sont des expressions, ces sites ont une valeur retournée.

GPPattern : Le GP-Pattern des **PropSite**. Ces GP-Patterns là sont caractérisés par une propriété globale en plus du type statique du receveur. Ils ont donc une implémentation.

SubtypeSitePattern : Regroupe les casts et les tests de sous-typage. Ces GP-Patterns

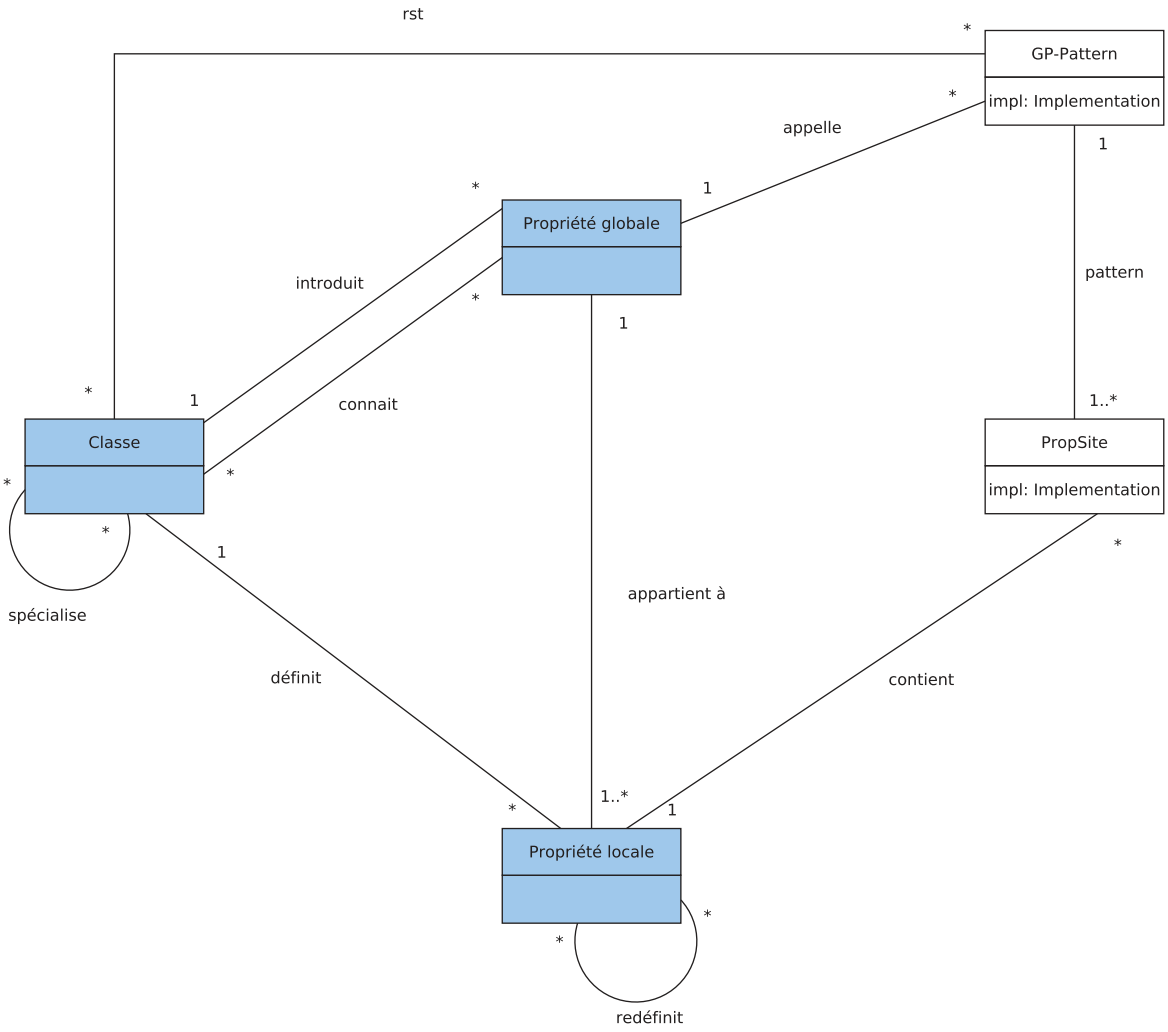

FIGURE 7.5 – GP-Patterns dans le modèle étendue

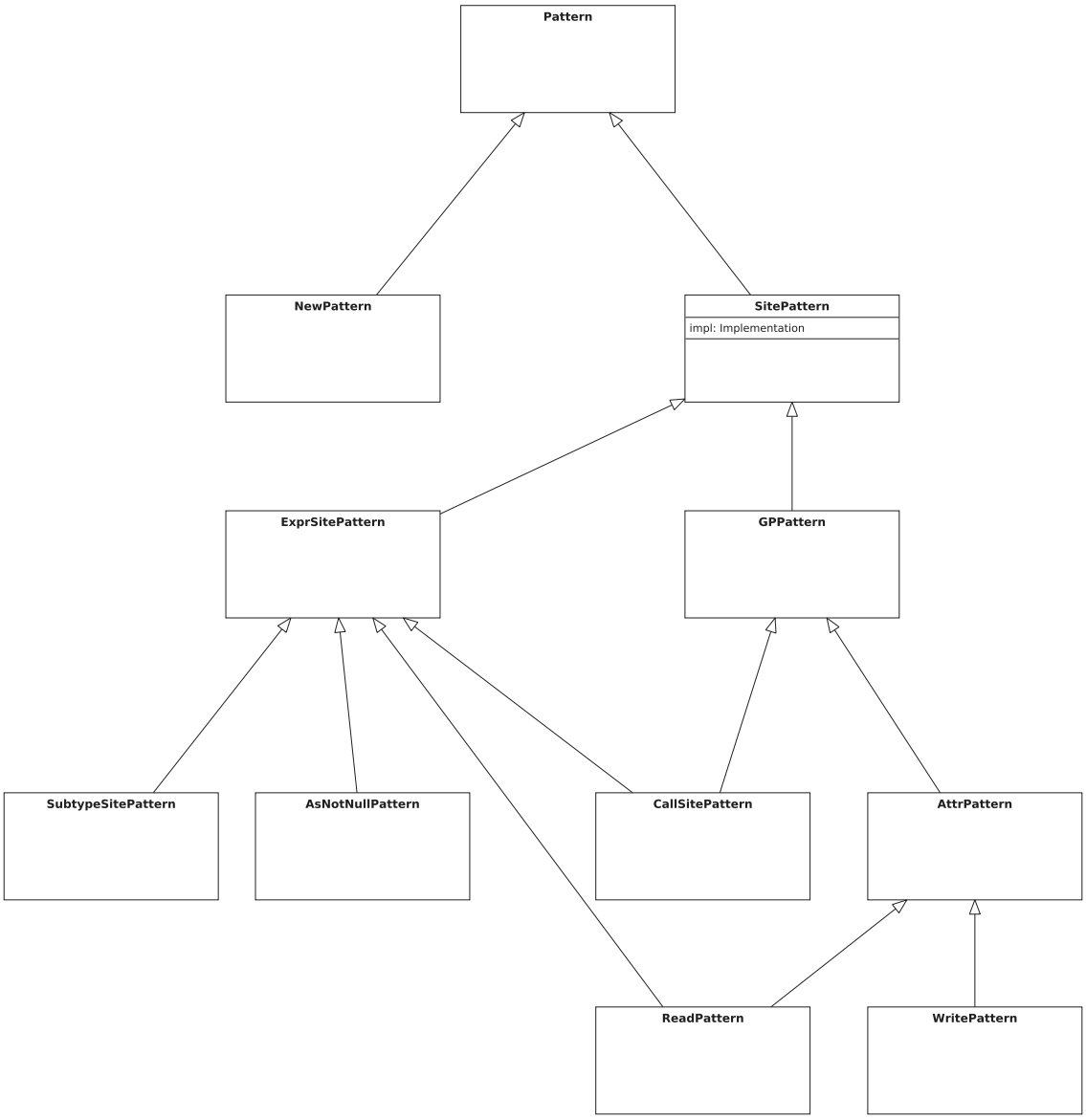

FIGURE 7.6 – Hiérarchie des GP-Patterns

sont caractérisés par deux classes : celle du type statique du receveur et la classe de la cible du test. Ces GP-Patterns sont ceux des **SubtypeSite**.

AsNotNullPattern : Pour la cohérence du modèle, ces GP-Patterns existent. Ils représentent des casts vers le type non-nullable de la source du test.

CallSitePattern : Ces GP-Patterns représentent les sites d'appels de méthodes. Ils sont caractérisés par la propriété globale appelée en plus de la classe du receveur. Ils possèdent aussi la liste des propriétés locales candidates à l'appel, les *callees*.

AttrPattern : GP-Pattern des sites d'accès aux attributs. Cela regroupe indépendamment les accès en lecture ou en écriture. Les appels de méthodes peuvent être chaînés entre eux, ce GP-Pattern est donc aussi sous-classe de *ExprSitePattern*.

WritePattern et ReadPattern : Ces deux GP-Patterns symbolisent respectivement les accès en écriture et en lecture aux attributs. Un GP-Pattern d'accès en lecture à un attribut est également une expression.

La figure 7.7 présente le cas particulier des sites d'appels de méthodes et de leurs GP-Patterns, par soucis de lisibilité, le schéma a été simplifié. Les GP-Patterns de méthodes contiennent deux informations importantes :

— les *callees* sont les propriétés locales candidates à l'appel.

— le *cuc* (callees uncompiled) est le nombre de propriétés locales candidates à l'appel qui ne sont pas encore compilées.

La relation inverse des *callees* est symbolisée par les **callers**, les propriétés locales ont la liste des GP-Patterns qui peuvent les appeler.

Pour les GP-Patterns, les **callees** sont construits à partir de l'analyse de la hiérarchie des classes (CHA). Toutes les méthodes vivantes sont donc situées dans cette liste, qui est mise à jour au fil de l'exécution et des chargements de classe.

Les **CallSite** contiennent eux aussi une liste des **callees**, qui est construite par un appel à une méthode. Si les sites contiennent des types concrets (les types possible du receveur), alors il est possible d'avoir une liste de propriétés locales plus restreinte qu'au niveau des GP-Patterns. Si les sites ne contiennent pas de types concrets, alors on réutilise la liste du GP-Pattern. Nous avons donc une hiérarchie parallèle entre les sites et les GP-Patterns. Un GP-Pattern d'appel de méthode sera en lien avec ses sites qui seront également des sites d'appel de méthode.

Les GP-Patterns des tests de sous-typage sont particuliers. Ils ne contiennent en effet pas de propriété globale à appeler. Par conséquent, ils sont caractérisés par deux classes : la classe du receveur du test et la classe de la cible du test. Du point de vue des classes uniquement, la situation est relativement similaire aux GP-Patterns qui possèdent une propriété globale (introduite dans une classe).

7.3.3 PIC-Pattern

Il est possible de factoriser encore au niveau des GP-Patterns. Deux GP-Patterns qui ont le même type statique du receveur et dont les classes d'introduction des propriétés appelées sont égales, peuvent être factorisés.

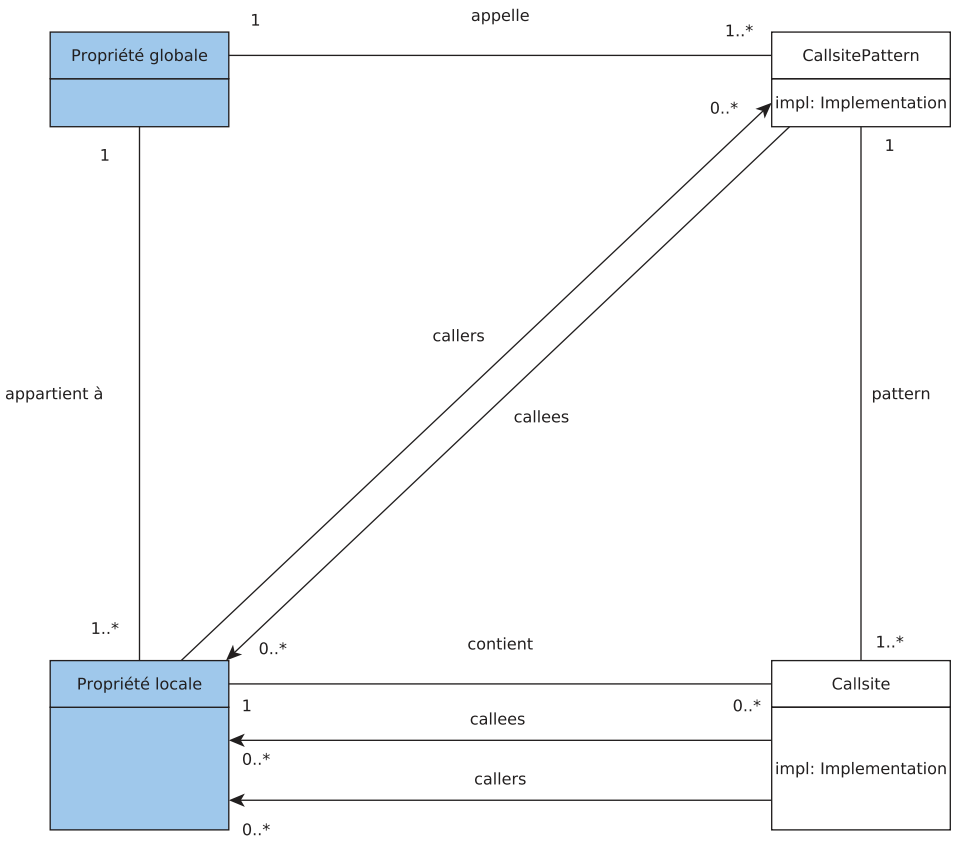

FIGURE 7.7 – Sites et GP-Patterns d'appel de méthode

En effet, en prenant deux *CallSitePattern*, qui sont des GP-Patterns d'appel de méthode. Si deux de ces GP-Patterns ont le même type statique du receveur ainsi que la même classe d'introduction de la propriété globale, on constate que les implémentations de ces GP-Patterns seront les mêmes.

Les PIC-patterns représentent donc cette factorisation. Il s'agit d'une association entre deux classes :

— la classe du receveur de l'appel : appelée *rst* pour *Receiver Static Type*

— la classe d'introduction de la propriété globale accédée : appelée *pic* pour *Property Introduction Class*

Avec ces deux classes, il est déjà possible d'avoir des informations sur l'implémentation. En particulier, il est possible de savoir si ces deux classes sont concernées par des situations d'héritage multiple qui feraient perdre l'invariant de position.

Les PIC-patterns induisent donc d'avoir une relation de sous-typage entre les deux classes concernées. La classe d'introduction de la propriété globale doit être $\geqslant$ à la classe

du type statique du receveur. Si une relation de sous-typage entre les deux classes existe, nous pouvons déjà savoir si on peut utiliser une implémentation en héritage simple ou en héritage multiple. S'il n'y a pas de relation de sous-typage alors cette factorisation ne nous apporte rien car il n'est pas possible de calculer une quelconque implémentation. Les tests de sous-typages n'ont donc pas de PIC-pattern. Cependant, les GP-Patterns des tests de sous-typage contiennent des informations très similaires à celles stockées dans les PIC-Patterns des attributs et des méthodes.

La figure 7.8 présente l'intégration des PIC-patterns avec les autres entités du modèle. Le modèle est maintenant complet avec toutes ses entités.

Les PIC-patterns concernent deux sortes de GP-Patterns uniquement, les GP-Patterns d'attributs ou et les GP-Patterns de méthodes :

PICPattern La super-classe de tous les PIC-Patterns. Elle contient une implémentation ainsi que la classe ayant introduit la propriété **pic** et la classe du receveur **rst**. À partir de ces deux classes il est déjà possible de déterminer si l'implémentation sera SST ou PH.

MethodPICPattern Spécialisation pour les méthodes. En plus de l'implémentation SST ou PH, on peut déterminer si le type statique du receveur est une classe finale. Si c'est le cas, la méthode ne sera jamais redéfinie et toutes les implémentations seront statiques.

AttributePICPattern Spécialisation pour les attributs. Si la classe du receveur est finale, on peut également choisir sûrement une implémentation SST même si le PIC a plusieurs positions dans la hiérarchie.

7.3.4 Autres entités du modèle

Le modèle étendu ne possède pas que des mécanismes objets. Certaines entités sont pas exemple primitives.

7.4 Création des entités du modèle à partir de l'AST

Lors du calcul de SSA, la structure manipulée est essentiellement de l'AST ainsi que les blocs de base qui ont été générés. Nous profitons d'effectuer plusieurs visites de l'AST en calculant SSA pour récolter toutes les instructions objets qui nous intéressent.

Une liste d'instruction objets contenant les nœuds d'AST est construite au fur et à mesure du parcours des méthodes. À la fin de l'analyse d'une méthode, le constructeur du modèle prend la main pour créer les entités du modèle. À partir de la liste d'instructions au sens de l'AST, la représentation intermédiaire ainsi que le modèle sont construits.

Pour transformer l'instruction de la figure 7.9, on commencera par traiter l'appel de méthode `.foo()`. L'objet correspondant du modèle sera créé s'il n'existe pas déjà. Ensuite, on appellera récursivement la méthode de création du modèle sur le receveur de l'appel : `a.bar()`. Pour finir, on traitera le premier receveur de l'appel chaîné : la variable `a`.

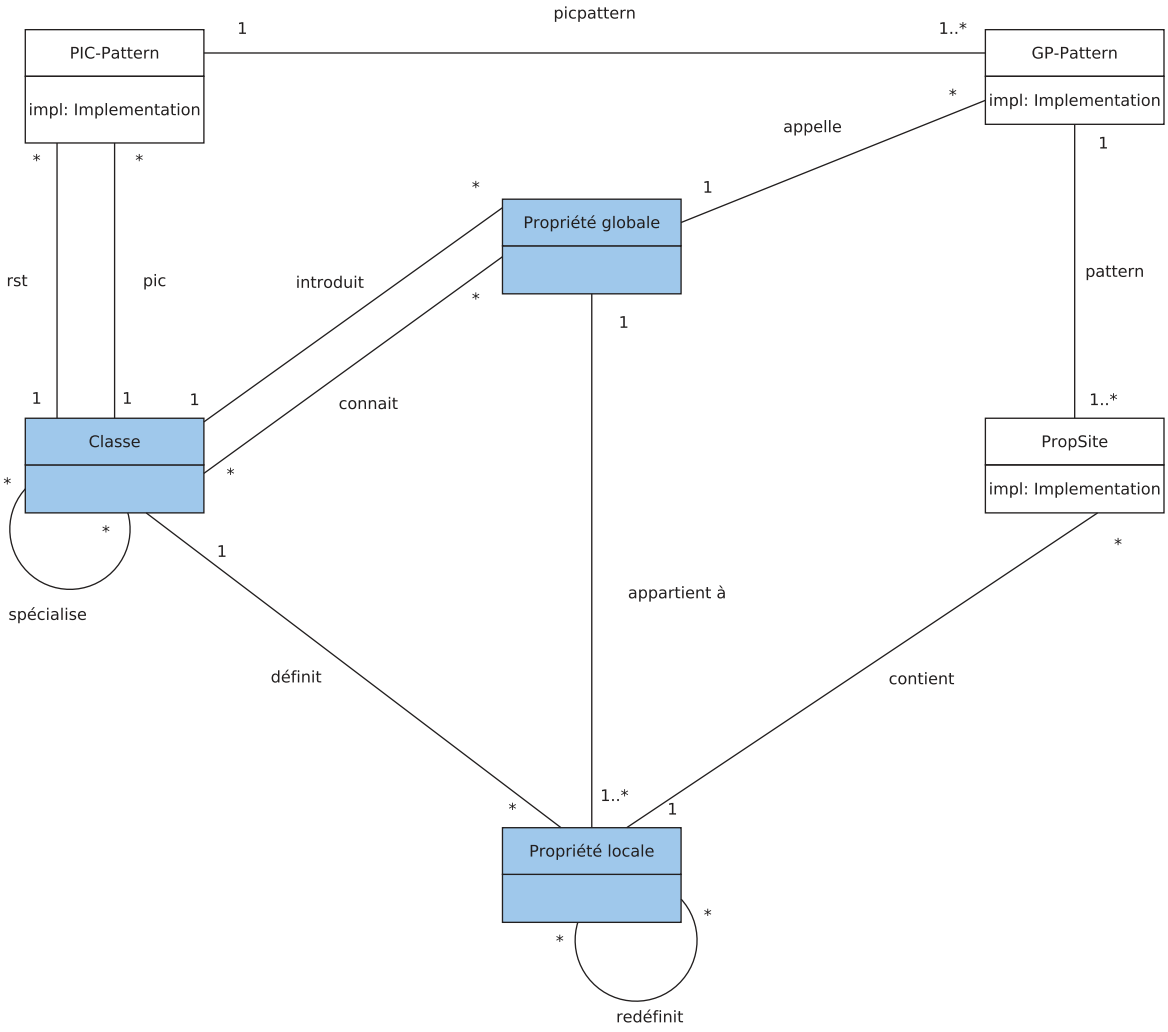

FIGURE 7.8 – Hiérarchie complète du modèle étendu

```
a.bar().foo()
```

FIGURE 7.9 – Portion de code à transformer vers la représentation intermédiaire

La cohérence entre les entités du modèle et l'AST est assurée par association. Le nœud d'AST contient un attribut vers l'entité du modèle correspondant et inversement.

Ainsi, pour revenir à l'exemple, si jamais l'appel `a.bar()` possède déjà son entité du modèle, cette dernière sera affectée en tant que receveur de l'appel à la méthode `foo`. À la fin de l'étape de la construction du modèle, tout est cohérent. À ce stade, aucune implémentation n'a encore été calculée.

Une fois qu'un ObjectSite du code a été créé, on va chercher à lui associer son GP-Pattern. Les GP-Patterns sont stockés dans le *rst* du site.

On va alors chercher si le GP-Pattern correspondant au site existe déjà pour l'affecter. S'il n'existe pas, il sera alors créé. Les PIC-Pattern sont également stockés de la même manière dans la classe du receveur de l'appel. Ils sont créés de manière paresseuse s'ils n'existent pas déjà : on ne crée que les GP-Patterns qui apparaissent dans le code à compiler.

Toute l'opération de génération des entités du modèle intermédiaire commence par créer les sites avant de remonter aux GP-Patterns puis aux PIC-Patterns. Cette étape de construction de la représentation intermédiaire et du modèle étendu est complètement paresseuse.

7.5 Implémentations et représentation intermédiaire

Les PIC-patterns et les GP-patterns permettent de factoriser des informations sur la représentation intermédiaire. Le principe général des implémentations est qu'elles peuvent être **propagées** des PIC-patterns vers les GP-patterns puis les sites dans certains cas.

De plus, un site-objet pourra toujours utiliser l'implémentation du GP-pattern, et le GP-pattern pourra toujours utiliser l'implémentation de son PIC-pattern. De manière générale, il faut éviter de recalculer de manière inutile une implémentation, le calcul doit être le plus paresseux possible. On utilisera donc l'implémentation du niveau supérieur, sauf si les informations présentes au niveau courant permettent d'avoir une meilleure efficacité. Les sections suivantes décriront le principe des implémentations pour les différents niveaux de la représentation intermédiaire.

7.6 Implémentation des mécanismes dans les entités du modèle

7.6.1 Implémentations dans les PIC-Patterns

À partir d'un PIC-Pattern, il est déjà possible de calculer des implémentations. Les deux classes peuvent être dans une situation d'héritage simple entre elles, ou de l'héritage

multiple peut intervenir dans leurs hiérarchie.

Le PIC-Pattern est une relation entre deux classes : la classe du type statique du receveur *rst* et la classe qui a introduit la propriété globale accedée *pic*. Si parmi toutes les sous-classes chargées du *rst*, le *pic* est toujours à la même position, alors l'implémentation du PIC-pattern est en SST par défaut. D'un point de vue implémentation, cela signifie que toutes les tables de méthodes dans la hiérarchie sous le rst auront le bloc de méthodes du pic à la même position. Si ce n'est pas le cas, il faut utiliser du hachage parfait.

Au chargement d'une classe, un PIC-Pattern anciennement en implémentation SST peut devoir changer d'implémentation suite au chargement d'une classe qui rajoute une situation d'héritage multiple. En effet, il est possible de charger une sous-classe du receveur qui "bouge" le bloc de propriétés du PIC. Dans ce cas, une implémentation SST ne peut plus être utilisée et il faut changer l'implémentation du PIC-pattern vers PH.

Pour les PIC-Patterns d'attributs, la situation est identique, deux implémentations sont également disponibles : SST et le hachage parfait.

7.6.2 Implémentation dans les GP-Patterns

L'implémentation de base du GP-Pattern est celle du PIC-Pattern. Mais dans certains cas il est possible de faire mieux.

Pattern de méthodes

Comme décrit précédemment, les GP-Patterns de méthodes contiennent la liste des propriétés locales candidates à l'appel ainsi que le nombre de ces propriétés qui ne sont pas encore compilées. S'il n'y a qu'une seule propriété locale candidate à l'appel, on utilise une implémentation statique dans le GP-Pattern, sinon nous reprenons l'implémentation du PIC-Pattern.

Lors des chargements de classes, nous mettons à jour les *callees* dans le GP-Pattern, et cela peut amener à recalculer l'implémentation du GP-Pattern. Ce changement d'implémentation devra ensuite être propagé aux site-objets qui utilisaient l'implémentation du GP-Pattern.

Pattern des attributs

Dans le cas des attributs, l'appel statique n'existe pas. Les attributs sont des accès en lecture/écriture et il n'est pas possible de mettre "en cache" une quelconque valeur comme avec l'appel statique des méthodes.

Au niveau du GP-Pattern, nous ne pouvons pas faire mieux qu'au niveau du PIC-Pattern.

7.6.3 Implémentation des PropSite

Pour un PropSite, si nous disposons du type concret du receveur, on peut envisager une implémentation plus efficace que celle du GP-Pattern. L'algorithme est le même que

pour les GP et PIC-Patterns mais en remplaçant les sous-classes vivantes du rst par le type concret.

Si on ne dispose pas du type concret du receveur au niveau du site, nous reprenons l'implémentation du GP-Pattern.

Implémentation des sites d'appel de méthode

Si on possède un type concret sur le site, on peut utiliser une implémentation statique si :

— Un seul receveur est possible pour ce site

— Plusieurs receveurs sont possibles, mais une seule méthode est candidate

Dans tous les autres cas, l'implémentation est soit SST soit PH selon la situation et les receveurs possibles, de manière analogue au calcul de l'implémentation du GP-Pattern.

7.6.4 Test de sous-typage

Le test de sous-typage peut être implémenté de trois manières différentes. La sémantique est par contre légèrement différente par rapport aux autres mécanismes objets.

Il y a quatre cas possibles pour un GP-Pattern d'un test de sous-typage :

— Implémentation statique : le type cible et source sont disjoints (pas de sous-classe commune), ou alors les sous-classes chargées du type source sont toutes sous-classes du type cible

— Implémentation finale : le type cible n'a pas de sous-classe chargée

— Le type cible a une unique position dans les sous-classes du type source : SST

— Dans tous les autres cas, on utilise le hachage parfait

Une implémentation statique d'un test de sous-typage signifie que l'on peut calculer statiquement le résultat du test. Ce résultat est alors soit toujours vrai, ou alors toujours faux. Il est pas exemple possible que le type statique du receveur soit un sous-type de la cible du test. Dans ce cas, on parle d'un *cast* ascendant. Un tel test de sous-typage est bien sûr inutile, mais il peut être implémenté statiquement : il sera toujours vrai. L'implémentation statique d'un cast consiste simplement à supprimer le test pour exécuter le code dans la branche appropriée selon le résultat calculé statiquement.

L'implémentation finale est un cas particulier de l'implémentation statique. Un cas particulier de l'implémentation existe à travers une implémentation **finale** du test. Si la cible du test est une classe finale (une feuille de la hiérarchie de classe), alors une optimisation est possible. La seule possibilité que le test soit vrai est que la source du test soit la même classe que la cible du test. Nous gardons alors l'adresse de la table de méthodes de la cible du test, à l'exécution nous comparerons l'adresse de la table de méthodes de la classe source avec la classe cible. Si les adresses sont égales, alors le test réussira, dans tous les autres cas il échouera.

Système de types de Nit Le système de types de Nit est assez complexe, il inclut en effet les types virtuels, la généricité ainsi que les types nullables. Toutes ces caractéristiques complexifient l'implémentation du test. Le test de sous-typage est donc composé de deux parties dans la machine virtuelle Nit.

Une première partie traite toutes les caractéristiques telles que nullables, généricité etc. On cherche à déterminer si la source du test est compatible avec la cible avec les types nullables etc. Lorsque cette première partie générique du test est effectuée, les différentes annotations de types sont résolues et traitées. Il se peut qu'une incompatibilité soit présente : par exemple la source du test est le type null mais la cible du test n'est pas un type nullable. Dans ce cas le test de sous-typage renvoie directement faux. Cette première partie d'un test de sous-typage est réalisée dans la machine virtuelle par une méthode qui n'utilise pas d'implémentations.

Si la partie générique du test est concluante et qu'il n'y a pas d'incompatibilités dans le système de types, alors il faut effectuer un test entre deux classes : la source et la cible. Seule la partie du test entre deux classes est concernées par les implémentations.

Implémentation SST Au chargement d'une classe, un identifiant est alloué à chaque classe via le hachage parfait et la numérotation parfaite. En parallèle, on affecte une **couleur** à chaque classe, il s'agit de la position de l'identifiant de la classe dans la table de méthodes au moment de sa création. Un test implémenté en SST nécessite d'aller à la position indiquée par la couleur de la cible du test. Il faut ensuite tester si on trouve dans la table de méthodes de la source l'identifiant de la cible à la position appropriée, voir section 2.6.

Implémentation avec le hachage parfait L'implémentation d'un test de sous-typage avec le hachage parfait est similaire au début d'un appel de méthode en hachage parfait, voir section 2.8.5. Il faut tout d'abord récupérer l'identifiant de la classe cible ainsi que le masque de hachage du receveur.

La première opération consiste à calculer l'entrée appropriée dans la table de hachage. Pour cela, on utilise l'opération binaire **et** entre le masque et l'identifiant cible. La table de hachage est contenue dans la partie négative de la table de méthodes. Il faut ensuite aller à la position indiquée par l'opération de hachage pour y récupérer le pointeur vers le bloc de la classe cible dans la table. Si le test est positif, alors l'adresse récupérée pointera directement sur l'identifiant de la classe cible. Tout autre configuration renverra faux.

Implémentation des sites de test de sous-typage

Au niveau des sites, si on peut calculer des types concrets, on refait un calcul similaire à celui réalisé au niveau du GP-Pattern, mais avec les classes du type concret. Ces classes du type concret remplacent les sous-classes chargées du rst dans le GP-Pattern.

7.6.5 Propagation

Les changements d'implémentations et propagations sont déclenchés par le chargement d'une classe ou la première compilation d'une méthode.

Quand une classe est chargée, on doit effectuer les étapes suivantes :

— Pour toutes les classes qui apparaissent dans un suffixe, on parcourt les PIC-Patterns dont ces classes sont le pic, pour détecter celles qui étaient en SST et qui doivent passer en PH. Si l'implémentation change, il faut propager ce changement aux GP-Patterns, qui re-propageront jusqu'aux sites.

— Pour toutes les méthodes définies dans la classe nouvellement chargée, on rajoute la méthode aux *callees* des GP-patterns dont le rst est une super-classe de la classe chargée. Cette opération peut amener à recalculer l'implémentation des GP-Patterns et à la propager jusqu'aux sites.

La factorisation en PIC-Patterns et GP-Patterns sert aussi à simplifier le recalcul des implémentations en séparant les concepts. Le passage d'une implémentation SST vers PH car toutes les classes ne sont plus à la même position est une situation qui est gérée par les PIC-Patterns. Lors d'une propagation, nous vérifions d'abord que la propagation est pertinente : si jamais l'entité vers laquelle on propage possède sa propre implémentation optimisée et n'est pas concernée, nous n'effectuons pas de propagation.

Il faut noter que pour les tests de sous-typage, cette situation est gérée dans les GP-Patterns puisqu'ils ne possèdent pas de PIC-Patterns.

Ainsi, les propagations et les mises à jour des implémentations sont gérées de haut en bas : des PIC-Patterns aux GP-Patterns puis jusqu'aux sites.

7.7 Conclusion

Nous avons rappelé le méta-modèle de Nit basé sur des propriétés globales et locales. L'essentiel des performances d'une machine virtuelle pour un langage à objet vient de l'implémentation efficace de mécanismes objets. Les appels de méthode, accès aux attributs et tests de sous-typage ont donc été implémentés avec un soin particulier.

Ces trois mécanismes sont implémentés de plusieurs façons dans la machine virtuelle. Le protocole d'optimisation sera responsable de choisir l'implémentation adaptée suivant le contexte.

Nous avons ensuite présenté les extensions apportées au méta-modèle pour aider à représenter les implémentations du code, ces extensions constituent la représentation intermédiaire du code. Trois principales entités ont été ajoutées pour factoriser les implémentations :

— Les PIC-Patterns

— Les GP-Patterns

— Les site-objets (ObjectSite)

Ces extensions sont basées sur la notion de site d'invocation d'un mécanisme objet. Ces sites possèdent une implémentation et constituent la représentation intermédiaire

du code. Un site-objet possédant une propriété globale est constitué du type statique du receveur ainsi que de la propriété globale appelée. Ils sont ensuite factorisés à travers des GP-patterns qui regroupent plusieurs sites ayant le même type statique et la même propriété globale appelée.

Une autre factorisation est également introduite : les PIC-patterns, ils sont constitués de deux classes. La classe du receveur de l'appel *rst* ainsi que la classe d'introduction de la propriété globale appelée *pic*.

Tout ce modèle est généré paresseusement à partir de l'arbre syntaxique du programme. Nous avons ensuite présenté le principe des implémentations des entités du modèle.

Chapitre 8

Préexistence et types concrets

Contents

8.1 Introduction

Le principe de la préexistence [Detlefs and Agesen, 1999] a été précédemment décrit dans le chapitre 3. La préexistence est une propriété d'un site d'appel de méthode. Cette propriété permet d'affirmer qu'il est possible d'optimiser sûrement (sans avoir à faire une réparation à chaud) un appel de méthode.

Dans ce chapitre, nous présenterons plus en détail la préexistence selon sa définition originelle. Nous présenterons ensuite des extensions que nous avons réalisées pour étendre la définition ainsi que l'usage de la préexistence. Ce chapitre a été présenté une première fois dans [Ducournau et al., 2015] puis publié dans [Ducournau et al., 2016].

8.2 Préexistence originelle selon Detlefs et Agesen

La préexistence a été définie de la façon suivante dans [Detlefs and Agesen, 1999].

Définition 19. Préexistence La préexistence est une propriété d'un receveur d'un site d'appel de méthode. Elle atteste que le receveur a nécessairement été créé avant l'appel de la méthode considérée.

Dans l'exemple minimal suivant, `x` est un paramètre de la méthode `foo`. `x` a donc nécessairement été créé avant l'exécution de `foo` car il provient de la méthode appelante. Selon la définition de la préexistence, l'appel `x.bar()` possède un receveur préexistant, `x`.

```
fun foo(x: A)
do
    x.bar()
end
```

Cette propriété est très importante dans le contexte des machines virtuelle : elle atteste qu'il n'y aura pas besoin d'effectuer une réparation à chaud d'un appel de méthode avec un receveur préexistant. Il est donc possible d'effectuer des optimisations sans avoir recours à des gardes ou à d'autres techniques de réparation à chaud, voir chapitre 3.

Pour reprendre l'exemple précédent, il n'y aura pas besoin de réparer l'appel `x.bar()` **pendant** l'exécution de `foo`. Le receveur `x` a déjà été créé, par conséquence, il n'y aura pas de chargement de classes pendant l'exécution de `foo` qui provoquera une recompilation de la méthode poussant à changer l'implémentation de `x.bar()`. Si l'implémentation du site est invalidée, alors il suffit de positionner un trampoline pour recompiler la méthode lors du prochain appel, l'exécution de la méthode courante peut continuer sans risque.

À l'inverse, l'exemple suivant contient un appel chaîné `x.baz().bar()`. Le receveur (`x.baz()`) n'est pas préexistant. Si jamais l'appel `x.baz()` retourne une valeur d'une classe qui n'était pas chargée jusqu'ici, alors il faudra recompiler la méthode `foo` pendant son exécution, en ayant recours à un mécanisme de réparation à chaud.

```
fun foo(x: A)
do
    x.baz().bar()
end
```

La définition de la préexistence de [Detlefs and Agesen, 1999] est basée sur deux règles :

— Le receveur est un paramètre qui n'a pas été ré-affecté

— Le receveur provient de la lecture d'un attribut immutable

Plus formellement, nous avons défini un ensemble de règles pour la préexistence :

Parameter-P : Tous les paramètres d'une méthode qui n'ont pas été ré-affectés sont préexistants.

ImmutableAttribute-P : Une expression de lecture d'attribut est préexistante si son receveur est préexistant et que l'attribut lu (la propriété globale) est immutable. Par exemple, l'attribut est caractérisé par le mot-clef `final` en Java ou `val` en Scala. Dans un langage tel que Nit qui n'offre pas une telle fonctionnalité, il serait possible d'effectuer une analyse lors de la compilation vers un bytecode pour déduire une telle annotation. En l'état actuel des choses, il n'est pas possible de qualifier un attribut d'immutable en Nit. Cette règle ne sera donc pas utilisée dans l'analyse de préexistence implémentée dans la machine virtuelle.

Les deux premières règles de préexistence sont en fait des préexistences de valeur : le receveur a été créé avant l'appel de la méthode.

Définition 20. Préexistence de valeur Un receveur d'un site d'appel de méthode est préexistant si sa valeur a été créée avant l'exécution de la méthode courante.

8.3 Préexistence étendue

Nous avons cherché à étendre la notion préexistence dans plusieurs directions :

— Considérer toutes les expressions, pas seulement les receveurs

— Tous les sites d'invocation de mécanismes objet, et pas seulement les appels de méthode

— Préexistence de type et pas seulement de valeur

La préexistence de valeur est en fait une indication que le type dynamique de l'objet est une classe qui a été chargée avant l'appel à la méthode courante. Mais nous avons réalisé que la préexistence ne repose pas uniquement sur la valeur des objets : la véritable information intéressante pour la préexistence est de savoir si la classe du receveur a été chargée avant d'exécuter la méthode. Cela nous amène à considérer une définition élargie de la préexistence à travers la préexistence de type.

Définition 21. Préexistence de type Un receveur est préexistant si sa classe a été chargée avant l'exécution de la méthode courante.

Par simplification, on considère une implémentation homogène de la généricité pour la préexistence de type. Par exemple, pour un receveur typé par `List<T>` on ne considère que `List` et pas l'instance générique précise. À l'avenir, il serait possible de traiter plus finement la généricité en enregistrant les instances génériques pour faire de la spécialisation de code par exemple.

Il faut noter que la préexistence de valeur implique la préexistence de type, la préexistence de type est plus "limitée" que la préexistence de valeur. Avec cette définition, un paramètre d'une méthode aura une préexistence de valeur et donc également de type. Dans un contexte d'héritage multiple, nous n'avons pas que les appels de méthodes à considérer mais également les accès aux attributs et les tests de sous-typage. De plus, la

définition originelle interdisait de considérer les appels chaînés comme étant possiblement préexistants.

Avec la définition de la préexistence de type, l'exemple suivant permet d'augmenter le nombre de sites dont les receveurs sont préexistants.

```
foo (x: A)
do
    var y: A
    if <condition> then
        y = new B()
        y.bar1() // Préexistence de type, si B est déjà chargée à la compilation
    else
        y = x
        y.bar2() // Préexistence de valeur
    end
    y.bar3()
end
```

Dans cet exemple, l'appel `y.bar1()` devient préexistant avec la préexistence de type, à la condition que la classe `B` soit chargée au moment de la compilation de `foo`. Le deuxième appel `y.bar2()` a lui aussi un receveur préexistant puisque `y` est un paramètre. Le dernier appel `y.bar3()` nécessite une analyse statique plus fine pour déterminer ses receveurs possibles et donc sa préexistence. Dans cet exemple, cet appel sera aussi préexistant car son receveur dépend de deux valeurs préexistantes.

8.3.1 Représentation intermédiaire et préexistence

La préexistence de type implique des analyses plus fines que les règles originelles. La première analyse nécessaire se limite à l'aspect intra-procédural, il faut déjà connaître les dépendances des variables.

Pour cela, nous avons utilisé SSA, la forme SSA d'une portion de code permet de déterminer de nombreuses choses mais nous nous sommes limités aux dépendances des variables. Lors du calcul de la forme SSA, nous calculons les expressions dont dépendent une variable, voir le chapitre 6 pour les détails.

Dans notre représentation intermédiaire, chaque méthode possède alors les éléments suivants :

- Une liste des paramètres d'entrée de la méthode, avec le receveur (`self` en Nit)
- Des littéraux si elle en contient.
- L'ensemble des sites d'invocation des mécanismes objet.
- L'ensemble des sites d'instanciations, qui incluent automatiquement un site d'appel au constructeur.
- Un ensemble de variables, dont chacune dépend d'une ou plusieurs expressions. La dépendance représente le fait que l'expression est affectée à la variable.

- Une variable spéciale représente les valeurs retournées par la méthode.

Les appels de procédures et les écritures d'attributs ne sont pas des expressions (ce sont des instructions) mais les autres sites objet sont des expressions. Les sites objets qui ont un type primitif en tant que receveur ne sont pas considérés car il ne s'agit pas vraiment de mécanisme objet. Les types primitifs sont des types finaux, chargés au début de l'exécution, il n'y a donc pas de difficulté à les implémenter.

La même entité peut être vue comme un site-objet ou comme une expression : vue comme un site objet, on considère son implémentation et l'éventuelle préexistence de son receveur. Vue comme une expression, on considère sa préexistence, et son éventuel rôle en tant que receveur. L'expression d'appel d'une méthode qui a une valeur de retour considère la **valeur** retournée par la méthode appelée. Cette distinction permet de calculer la préexistence en cas d'appels chaînés : on veut pouvoir calculer la préexistence du receveur d'un appel avant de calculer la préexistence de ce que retourne cet appel. Il faut donc pouvoir considérer un appel de méthode en tant que site d'appel ou expression.

8.3.2 Règles de la préexistence étendue

Nous avons étendu les règles de préexistence en ajoutant les propriétés suivantes :

Variable-P : Une variable est préexistante si toutes les expressions dont elle dépend sont préexistantes. Il faut noter que cette règle n'était pas explicite dans l'article [Detlefs and Agesen, 1999], mais il est possible qu'elle ait été implémentée.

Literal-P : Un littéral est toujours préexistant (de valeur) car sa classe correspondante est chargée au début de l'exécution.

Comme dit précédemment, lors de la construction de SSA nous calculons les dépendances des variables. Il y a deux types de variables dans la représentation intermédiaire : les variables SSA avec une seule dépendance et les ϕ-variables avec plusieurs dépendances.

Dans l'exemple 8.1, la variable `y` a deux dépendances :

— La variable `x`, un paramètre

— L'attribut immutable `z`

Ici, l'attribut `z` est considéré comme immutable. La préexistence de l'appel `y.bar()` sera calculée en fusionnant deux valeurs de préexistence qui sont les deux affectations dans la variable `y`. Par conséquent, l'appel `y.bar()` aura un receveur préexistant car le receveur provient de deux expressions préexistantes. Il faut noter que la préexistence est une propriété conservatrice, toutes les dépendances doivent être préexistantes pour que l'expression considérée soit préexistante.

ConcreteType-P : Le type concret d'une expression est l'ensemble des types dynamiques possibles de l'expression si cet ensemble peut être connu statiquement. L'expression est alors type-préexistante si toutes les classes du type concret sont chargées. De nombreuses règles sont possibles pour obtenir des types concrets sur une expression, elles seront décrites ultérieurement.

```
var z: B is immutable

fun foo(x: A)
do
    var y
    if ... then
        y = x
    else
        y = x.z
    end

    y.bar()
end
```

FIGURE 8.1 – Exemple d'une préexistence originelle multiple

Return-P : Le retour d'une méthode a une valeur de préexistence correspondant à celle de sa variable de retour. La variable de retour a comme dépendances toutes les valeurs retournées dans la méthode. Par exemple dans la figure 8.2, la préexistence de la variable de retour aura les valeurs `new B` et `y`. En effet, il y a deux expressions de retour dans la méthode foo, statiquement on peut donc dire qu'un des deux retours sera exécuté : on considère donc els deux pour le calcul de la préexistence.

Call-P : Une expression d'appel de méthode est préexistante (c'est-à-dire qu'elle retourne une valeur préexistante) si :

- Le receveur de l'appel est type-préexistant.
- Les expressions de retour de toutes les méthodes candidates à l'appel sont préexistantes. Cela signifie que si un paramètre de la méthode est retourné, il faut vérifier si l'argument correspondant dans l'appel est préexistant.

Pour que cette règle s'applique, il faut avoir connaissance de toutes les méthodes candidates, et qu'elles soient toutes compilées. Dans tous les cas contraires, l'expression est considérée comme non-préexistante.

Cast-P : Une expression de cast est préexistante si son receveur est préexistant. En effet, pour que le cast réussisse et que le programme continue de s'exécuter, la cible du cast est nécessairement une classe chargée puisque le receveur doit être sous-type de la cible.

8.3.3 Sites monomorphes ou préexistants

Définition 22. Site Monomorphe Un site-objet pour lequel on connaît le type dynamique de son receveur statiquement et facilement (sans analyse complexe). Plusieurs cas de monomorphisme sont possibles pour un site :

```
var z: B is immutable

fun foo(x: A): A
do
    var y
    if ... then
        y = x
    else
        return new A
    end

    return y
end
```

FIGURE 8.2 – Préexistence de la variable de retour

— Le receveur provient d'un unique *new* dont la classe correspondante est chargée.

— Le type statique du receveur est celui d'une classe finale (une feuille de la hiérarchie de classes).

— La méthode appelée est finale (elle ne sera jamais redéfinie). Ce cas précis ne s'applique pas au langage Nit de manière générale, mais un langage comme Java permet de définir une méthode finale. Certaines méthodes internes de Nit sont cependant non-redéfinissables.

Les sites monomorphes sont très nombreux dans des programmes d'un langage à objet. Par exemple, tous les appels à des constructeurs suite à un *new* sont des sites monomorphes.

Il faut noter qu'un site-objet monomorphe sera toujours implémenté avec la meilleure des implémentations possible. Par conséquent, nous avons choisi de les classer à part des autres sites objets.

Si jamais un site n'est pas monomorphe, il est alors considéré comme étant *polymorphe* et on cherchera à déterminer sa préexistence en utilisant les différentes règles. Un site-objet sera dit **préexistant** si son receveur est préexistant (de type ou de valeur). On considérera qu'un site monomorphe possède un unique type concret, le type concret correspondant à celui du *new*, ou à celui de la classe finale qui type l'expression. Le modèle reste ainsi uniforme entre les sites monomorphes et les sites polymorphes.

Définition 23. Site préexistant Un site-objet qui a son receveur préexistant (de type ou de valeur)

8.4 Préexistence étendue et types concrets

Il existe de nombreuse manières d'obtenir des types concrets. Le but de la préexistence est d'optimiser sûrement des sites d'invocations de mécanismes objet.

8.4.1 Types concrets

Les types concrets entraînent de la préexistence. Obtenir le type concret d'un receveur d'un site-objet permet de connaître les types dynamiques possibles du receveur de l'appel. Par conséquent, si les classes correspondantes sont déjà chargées, le receveur sera préexistant, avec une préexistence de type. Calculer la préexistence d'un type concret revient donc à vérifier que les classes le composant sont bien chargées.

Les types concrets sont représentés de la manière suivante :

— Un ensemble de classes. Pour l'instant, on ne s'intéresse qu'aux implémentations en ne considérant pas encore l'optimisation des types paramétrés dont l'implémentation est considérée comme étant homogène.

— Le caractère immutable ou non. Selon les règles, un type concret peut être immutable. Pour faciliter le calcul et la propagation des types concrets, il est utile de savoir si le type concret est susceptible d'évoluer. Un type concret mutable signifie qu'il est possible que ce type concret soit augmenté dans le futur. Par exemple : le type concret calculé à partir du retour d'un appel de méthode est potentiellement mutable si une nouvelle méthode candidate est rajoutée à l'appel.

Si la phi-variable x dépend de e1 et e2, pour que x ait un type concret, il faut que e1 et e2 en aient un, et le type concret résultant est l'union des deux.

Le type concret est un **ensemble** (il ne contient donc pas de doublon) et n'est jamais vide, il est doublé d'une condition booléenne qui dit que le type concret est connu. Fusionner deux types concrets (par exemple à travers une ϕ-variable) consiste à faire l'union de leurs ensembles respectifs et la conjonction de de leurs conditions.

Pour qu'un type concret soit préexistant dans l'analyse, il faut que toutes les classes le composant soient chargées. Un type concret dont toutes les classes sont chargées entraînera donc de la préexistence de type, cf la règle **ConcreteType-P** définie précédemment.

8.4.2 Règles des types concrets

Les règles suivantes produisent des types concrets immutables :

FinalType-CT : Quand le type statique d'une expression est une classe finale. Le type concret de l'expression est immutable, et correspond à la classe.

New-CT : Une expression d'instanciation d'objet a un type concret immutable correspondant à la classe instanciée.

PrivateWrite-CT : L'attribut global est déclaré privé en écriture. (au sens de la visibilité). L'écriture est alors réservée à l'unité de code courante (le module en Nit, la classe en Java). Le type concret de l'attribut est la fusion de tous les types concrets

des expressions affectées dans l'attribut. Pour que le type concret de l'attribut soit considéré, les types concrets affectés dans l'attribut doivent être immutables.

PrivateMethod-CT : Cette règle est équivalente à la précédente, mais pour les méthodes. La règle s'applique quand une méthode est définie comme étant privée, et que son type de retour a un type concret immutable.

Toutes les règles suivantes sont susceptibles d'introduire de la mutabilité dans les types concrets.

Variable-CT : Le type concret d'une variable est la fusion des types concrets de toutes les expressions dont elle dépend. Si une des expressions a une préexistence mutable, alors l'ensemble du type concret de la variable est mutable. Dans le cas contraire, la variable aura un type concret immutable.

SelfWrite-CT : Cette règle est similaire à la règle **PrivateWrite-CT**. Les écritures d'attributs ne sont plus seulement réservées à l'unité de code courante mais limitées au receveur courant, y compris dans les sous-classes (comme en Eiffel).

Self-CT : Le type concret de `self` (`this` en Java) est calculé. Pour cela, il faut tenir compte des redéfinitions de méthodes. Dans une méthode `foo` définie dans la classe `A`, le type concret de `self` est l'ensemble des sous-classes de A (incluse) qui ont été instanciées et qui héritent de la méthode `foo` sans la redéfinir.

Si une classe `T` redéfinie la méthode foo en faisant un appel à `super` à l'intérieur, le type concret de `self` dans `B` est rajouté au type concret de `self` dans la classe A.

En effet, si la méthode est redéfinie dans les sous-classes, `self` ne sera pas du type de ces sous-classes puisque la méthode de la super-classe A sera masquée par la redéfinition. Le type concret de `self` doit donc être calculée dans chacune des méthodes de chaque classe. En pratique, pour ne pas surcharger l'analyse, il ne faut le faire que dans les cas où cette information serait exploitable, c'est-à-dire quand le type concret de `self` n'explose pas. Cette information doit être beaucoup plus précise que la totalité des classes chargées en dessous de `self` pour être pertinente.

Return-CT : Si le type de retour d'une méthode est final, la règle **FinalType-CT** s'applique, même si la méthode n'a pas été compilée. Sinon, si la méthode a été compilée, alors la règle **Variable-CT** s'applique.

Call-CT : Le type concret d'une expression d'appel de méthode est la combinaison des types concrets de toutes les méthodes candidates à l'appel. Pour considérer le types concret d'un tel appel, le receveur de l'appel doit être préexistant. Sinon, le graphe d'appel pourrait s'agrandir, invalidant alors le type concret du retour. Cette règle entraîne une mutabilité des types concret. Chaque fois qu'une nouvelle méthode candidate est ajoutée sur le site d'appel, il faut recalculer le type concret de l'expression.

Cast-CT : Le type concret d'un cast est l'ensemble des types concrets du receveur qui

sont sous-types de la cible du cast. Le receveur doit être type-préexistant pour considérer cette règle.

Les règles **Call-P** et **Call-CT** obligent à considérer une analyse inter-procédurale pour être calculées.

8.4.3 Mutabilité de la préexistence

La préexistence originelle n'était pas mutable. En effet, les règles étaient simples, la préexistence était calculée une seule fois et donnait définitivement la valeur de préexistence.

La préexistence étendue est par essence mutable : certaines expressions sont en effet immutables par défaut et peuvent le devenir plus tard. C'est par exemple le cas de la règle **New-CT** qui par nature introduit de la mutabilité. Lorsque la classe du new n'est pas encore chargée (le new n'a pas encore été exécuté) les expressions qui dépendent de ce new ne seront pas préexistantes. Le fait d'exécuter le new entraînera un changement de préexistence des expressions en question, si jamais la préexistence est recalculée.

Les changements de préexistence du sens non-préexistant vers préexistant ne sont en fait pas un problème. La préexistence a un effet conservatif, ne pas la recalculer dans ce cas précis ne posera pas de problème lors de l'exécution.

Le changement de préexistence dans le sens préexistant → non-préexistant est plus problématique. Le protocole de compilation peut prendre une décision d'optimisation en se basant sur un receveur préexistant. Le fait que ce receveur ne soit plus préexistant doit entraîner la recompilation pour maintenir une exécution correcte. Les règles pouvant entraîner un changement de préexistant vers non-préexistant sont les suivantes :

— **Return-CT**
— **SelfWrite-CT**
— **Self-CT**
— **Call-CT**
— **Cast-CT**

Pour simplifier l'analyse et la propagation, nous restreignons l'utilisation de **Return-CT** et **Return-P** aux cas où le type concret et la préexistence sont immutables. Cela évite de faire une propagation inter-procédurale. Toutes les règles mutables citées ci-dessus sont sensibles au chargement de classes. Pour la règle **Cast-CT**, un chargement peut introduire une nouvelle classe dans la source du test. Pour les autres règles, un chargement de classe peut introduire une nouvelle méthode candidate sur un site d'appel.

8.4.4 Mise en œuvre des différentes règles

Mise en œuvre de la règle PrivateWrite-CT

En Nit, les attributs peuvent être lus et écrits dans le module où ils sont définis. En dehors du module d'introduction, par défaut, les attributs peuvent uniquement être

accédés en lecture. L'accesseur en écriture est défini comme privé de base sauf annotation contraire du programmeur lors de la définition de l'attribut [1]. Un grand nombre d'attributs peuvent donc avoir des types concrets dans un langage tel que Nit.

Pour calculer le type concret de l'attribut, il faut donc parcourir tout son module de définition pour récupérer toutes les expressions d'affectation de l'attribut. Cette opération est effectuée par un visiteur au chargement de chacune des classes. Ce mode de fonctionnement est bien sûr une simulation de ce qu'il faudrait faire pour implémenter correctement cette règle.

Idéalement, lors de la compilation statique d'une classe, il faudrait parcourir tout son code pour déterminer le type concret des attributs définis de manière privé. On peut imaginer que le type concret serait encodé dans le bytecode. Cette opération serait compatible avec le chargement dynamique car une classe représente une unité de code connue et limitée. En effet, une fois calculé, le type concret d'un attribut privé est bien sûr immutable.

FinalType-CT

Le mot-clé `final` (`frozen` en C♯ et Eiffel) n'existe pas en Nit, nous avons donc simulé l'existence d'un tel mot-clé.

Dans les expériences, on utilisera principalement des outils de manipulation de code Nit en tant que benchmarks. Dans ce cadre, on considère que les classes du parser et de l'arbre syntaxique de Nit sont finales. Cela est simulé en parcourant le modèle en monde fermé de ces classes pour déterminer si elles sont finales. Toutes les classes définies dans le module du parser ou de les classes de l'arbres syntaxiques sont potentiellement finales.

En effet, les classes du parser sont générées, et si le mot-clé final existait en Nit, on peut imaginer que ces classes aient cette propriété. Les classes de l'AST sont quant à elles connues à la compilation car le compilateur est bootstrapé régulièrement. Les nombreuses classes feuille de la hiérarchie de l'AST pourraient être déclarées finales car elles sont contenues dans le bootstrap.

Pour toutes les autres classes de la librairie, il n'est pas possible d'inférer une telle annotation.

8.5 Calcul et encodage de la préexistence

Le calcul et le stockage des valeurs de préexistence peut entraîner des problèmes de performances. De nombreuses portions du code, c'est-à-dire toutes les expressions, doivent stocker leur préexistence. Pour optimiser au mieux le calcul, nous avons utilisé un encodage binaire de toutes les valeurs de préexistence. Les opérations de combinaison de deux préexistence et de calcul sont alors facilitées et plus efficaces.

1. Une fonctionnalité de Nit `intrude import` permet d'importer un module en outrepassant la visibilité du module. Cette caractéristique n'est cependant pas souvent utilisée et doit l'être avec parcimonie.

8.5.1 Encodage de la préexistence

L'encodage utilisé est le suivant :

2^0	préexistence
2^1	immutable (si préexistant)
2^2	préexistence de valeur
2^3	non-préexistence
2^4	immutable (si non-préexistant)
2^5	préexistence récursive [2]
2^6	`self`, le receveur de la méthode
$2^7 >$	dépendances des autres paramètres de la méthode

Il est souvent utile de fusionner deux valeurs de préexistence, par exemple pour considérer la préexistence d'une *phi*-variable. Dans ce cas, on ne considère l'expression préexistante que si toutes les dépendances sont préexistantes.

Dans l'encodage, les bits 1 à 4 sont fusionnés si besoin avec un `et` binaire, pour l'aspect conservatif. Les bits qui encodent les dépendances avec les paramètres sont fusionnés avec le `ou` binaire, ils représentent les dépendances envers les paramètres d'entrée de la méthode. Pour s'assurer qu'on obtient bien une valeur correcte de préexistence, un test était effectué lors du développement pour s'assurer que tout calcul de préexistence fournissait une valeur plausible de préexistence. Par exemple, une expression ne pouvait pas être la fois préexistante et non-préexistante dans son encodage.

8.6 Conclusion

Dans ce chapitre, nous avons présenté des extensions à la définition de la préexistence d'après [Detlefs and Agesen, 1999]. Nous avons étendu la définition et les usages possibles pour considérer des expressions et plus seulement des receveurs d'un site d'appel. De plus, nous avons étendu la préexistence aux tests de sous-typage ainsi qu'à l'héritage multiple pour considérer également les sites d'accès aux attributs. La préexistence étendue considère enfin les types en plus des valeurs.

Pour cette préexistence étendue, nous avons effectué une analyse des dépendances des variables avec SSA, nous avons également réalisé marginalement une analyse interprocédurale. Un encodage binaire des valeurs de préexistence est aussi utilisé pour diminuer le temps de calcul.

La préexistence étendue ouvre la voie à un plus grand nombre d'expressions préexistantes mais introduit aussi de la mutabilité dans la préexistence dans certains cas. Dans le chapitre suivant, nous décrirons un protocole basé uniquement sur la préexistence, qui permettra de voir l'efficacité de l'extension de la préeexistence sur les implémentations.

Chapitre 9

Protocoles et expérimentations

Contents

9.1 Introduction

Un protocole de compilation/recompilation contient plusieurs éléments. Il est possible de faire un certain nombre de choix dans l'implémentation d'un protocole. Par exemple, il n'est pas obligatoire de provoquer une recompilation pour optimiser. Par contre, dans tous les cas il faudra faire des réparations pendant l'exécution si nécessaire pour maintenir une exécution correcte. Ce chapitre décrit les différents choix possibles et présentera les expérimentations réalisées. Le chapitre courant a été partiellement publié dans [Ducournau et al., 2016].

9.2 Principe des expérimentations

La machine virtuelle Nit n'est pas capable d'effectuer de la compilation à la volée du code. Ce mécanisme est d'ailleurs simulé dans la machine virtuelle, les mesures de temps d'exécution sont donc à exclure. Nous allons suivre la méthodologie décrite dans le chapitre 4 pour compter des éléments à l'exécution. Nous compterons par exemple les éléments suivants :

— Le nombre de site-objets suivant leur préexistence et leur implémentation : PH, SST, statique

— Les recompilations de méthodes et les invalidations d'implémentation des sites

— Le nombre d'exécutions des sites suivant leur implémentation

Notre méthode basée sur le comptage de différents éléments est également compatible avec le fait de compter à chaud. Nous allons mesurer à deux moments différents de l'exécution :

— Au cours de l'exécution, c'est-à-dire lorsque les méthodes sont compilées pour les éléments statiques et lors de l'exécution des sites pour les éléments dynamiques.

— À la fin de l'exécution nous recompterons tous ces éléments.

Pour compter ces éléments, nous mettrons en place plusieurs types de compteurs, par exemple, pour mesurer les implémentations dynamiquement nous aurons :

— un compteur global par type de sites, nous comptons le nombre d'appel de méthode en hachage parfait par exemple en incrémentant à chaque appel exécuté.

— un compteur local à chaque site qui est incrémenté à chaque exécution.

Avec les compteurs globaux, on obtiendra un état réel de l'exécution du programme. Ces chiffres présenteront le nombre exacts de chacune des implémentations et donne donc une idée de l'efficacité de l'exécution du programme. Les compteurs locaux sont une approche différente : à la fin de l'exécution, nous reconstruirons des chiffres globaux en fonction de la dernière implémentation du site. Par exemple : un site d'appel de méthode peut initialement être implémenté en hachage parfait et être exécuté deux fois puis voir son implémentation évoluer en statique et être exécutée deux fois. Globalement, nous compterons donc 4 appels statiques au total car c'est la dernière implémentation du site à la fin de l'exécution. Cela donne donc une idée de l'efficacité du programme si jamais

nous relançons l'exécution en gardant les implémentations finales : nous obtenons les performances "à chaud" sans compter les changements d'implémentations comme avec les compteurs globaux.

9.3 Présentation des benchmarks

Nit étant un langage académique, il ne dispose pas benchmarks nombreux et reconnus pour effectuer des tests.

Pour des raisons techniques, tout programme Nit ne peut pas forcément être exécuté dans la machine virtuelle. En effet, comme précisé dans le chapitre 5, certains programmes utilisent l'interface native de Nit pour écrire dans un autre langage que Nit certaines méthodes.

La machine virtuelle Nit n'est pas capable d'exécuter de telles méthodes, elle supporte uniquement une forme limitée de la FFI de Nit. En particulier, il n'est pas possible de méta-évaluer la machine virtuelle Nit puisque elle contient certaines méthodes complexes écrites en C.

Les programmes Nit utilisés sont les suivants :

nitc Le compilateur officiel de Nit, ce programme contient la construction de l'AST et du modèle ainsi que toute la partie compilation. C'est le plus gros programme dont nous disposons pour les expériences

niti L'interpréteur Nit sur lequel est basé la machine virtuelle. Il possède des parties communes avec *nitc*.

nitdoc L'outil qui génère la documentation de Nit à partir d'un fichier source. Il y a donc des analogies avec un compilateur dans la forme du programme.

jwrapper Un outil qui génère des classes Nit à partir d'un code source en Java.

nitiwiki Un outil de génération d'un site web sous la forme d'un wiki. Une syntaxe est définie, les fichiers sont ensuite parsés pour rendre le site web.

En entrée de ces différents outils, nous utilisons un petit programme de test rédigé en Nit, qui calcule une fonction de fibonacci en effectuant quelques chargements de classes à travers un pattern de programmation *factory*. L'entrée est commune pour tous les programmes cités plus haut. Ce programme de test a été rédigé pour tester plusieurs fonctionnalités de Nit et effectuer des chargements de classes successifs.

Ces programmes ont été choisis car ce sont ceux contenant le plus de code que nous pouvons exécuter avec la machine virtuelle. Cependant, la base de code de *nitc*, *niti* et *nitdoc* ont des points communs car le modèle du programme est construit pour les trois. Ces programmes sont de taille importante, sont profondément objets et ont été développés par plusieurs personnes.

jwrapper et *nitiwiki* sont de taille plus modeste, mais ils ont l'avantage d'avoir une base de code différente des trois autres benchmarks.

nitc est le plus gros programme du corpus de benchmarks, nous allons donc particulièrement détailler ses résultats.

```
class A

    fun foo do end
end

class B
        super A
end

class C
        super B

        redef fun foo
        do
            var i = 1
            while i < 5 do
                i += 1
            end
        end
end

class Amaker

        fun factory: A do
            return new A
        end
end

class Bmaker
super Amaker

        redef fun factory do
            return new C
        end
end

redef class Int
        # Calculate the self-th element of the fibonacci sequence.
        fun fibonacci: Int
        do
            if self < 2 then
                return 1
            else
                return (self-2).fibonacci + (self-1).fibonacci
            end
        end
end

class ToCastTest
        var a: A

        fun subtypetest(obj: A)
        do
            if obj isa C then
                obj.foo
```

9.4 Variantes de chargement de classes et de première compilation des méthodes

Une classe peut être uniquement implicitement *chargée* : c'est à dire chargée car elle est superclasse d'une classe instanciée, on qualifiera d'*implicite* son chargement, cf chapitre 6, la construction de sa structure sera partielle. Elle peut aussi être directement *instanciée*, c'est-à-dire qu'une instance de la classe sera créée via un *new*, dans ce cas la construction de sa structure est complète, cela correspond au chargement *explicite*. Le chargement de classes peut être spécifié de différentes façons :

Variante 1

1. Une classe `A` est chargée de manière *explicite* quand un site `new A` est exécuté pour la première fois.
2. Quand la classe `A` est réellement instanciée pour la première fois, on alloue sa table de méthode avant de la remplir. Les propriétés locales rajoutées dans `A` sont propagées aux sites qui ont besoin de cette information.
3. Quand une classe est chargée explicitement, ses superclasses directes sont d'abord récursivement chargées de manière implicite si besoin.
4. Une méthode est compilée la première fois qu'on doit l'exécuter.

Variante 2

Avec cette variante, la règle 1 est remplacée de la manière suivante :

1. Une classe `A` est chargée quand un site `new A` est compilé.

Cette règle permet d'éviter des recompilations ultérieures si le `new A` est effectivement exécuté. Mais s'il ne l'est pas, cela peut polluer les analyses ultérieures avec des classes considérées comme vivantes alors qu'elles ne le sont pas, et des méthodes jamais compilées qui sont un obstacle à une partie de l'analyse de préexistence. Aussi, la règle a été implémentée pour que la classe A considérée soit chargée lors de la compilation de *new A* uniquement si l'expression *new A* ne figure pas dans la portée d'une conditionnelle.

Pour implémenter cette variante, nous annotons des bloc de base lors de leur création. Le premier bloc de la méthode est bien sûr inconditionnel. L'aspect inconditionnel est maintenu jusqu'à ce qu'on rencontre un bloc avec une boucle ou alors un test. Dans ce cas, l'intérieur de tels blocs seront conditionnels [1].

Après un éventuel test unique dans une méthode, nous annotons également inconditionnel le bloc. Globalement, tous les chemins du code où on est sûr de passer à l'exécution sont donc annotés inconditionnels.

1. Des exceptions à cette affirmation existent, par exemple avec des boucles de la forme `loop` directement à l'entrée de la méthode. Nous ne traitons pas ces cas précis pour l'instant et nous annotons donc ces blocs conditionnels même s'il ne le sont pas.

```
fun foo()
do
    ...
    ...
    if ... then
        new A
    else
        new B
    end

    new C
    ...
end
```

FIGURE 9.2 – Sites d'instanciation conditionnels

Dans le schéma de code 9.2, les sites d'instanciation `new A` et `new B` sont conditionnels car situés à l'intérieur d'un test. Par contre, l'expression `new C` est située dans un bloc inconditionnel car le flot de contrôle passera obligatoirement par cette portion de code.

Variante 3

Avec cette variante, toutes les méthodes appelées statiquement sont compilées quand leur appelant est compilé. La règle 4 devient donc : une méthode est compilée quand on compile une méthode qui l'appelle statiquement.

Comme pour la variante 2, on peut imaginer un compromis qui effectuerait cette opération uniquement pour les appels situés dans des blocs inconditionnels. Cependant, alors que la variante 2 n'a pas d'effet de bord, la variante 3 peut entraîner une situation problématique : une méthode compilée car appelée statiquement peut voir sa compilation invalidée avant même son premier appel.

Discussions

Dans ces différentes variantes, la variante 3 est donc potentiellement problématique car il serait difficile de mesurer ses effets et elle pourrait même se révéler contre-productive. La variante 2 semble intéressante à tout point de vue car elle réduit automatiquement le nombre de recompilations sans risques.

Nous avons donc choisi de l'utiliser dans tous les protocoles, tous les chiffres présentés ultérieurement suivent cette variante :

— Une classe A est chargée quand un site *new A* est exécuté pour la première fois, ou lors de la compilation de ce site s'il est inconditionnel.

— Une méthode est compilée juste avant sa première exécution.

9.5 Préexistence étendue

Les différentes règles de préexistence étendues ont été implémentées dans la machine virtuelle. Pour mesurer leur efficacité et leur apport, nous avons aussi implémenté la préexistence selon la définition de [Detlefs and Agesen, 1999]. Toutes les règles de préexistence originelle ne sont pas implémentés car le langage Nit ne le permet pas. La règle concernant les attributs immutables n'est donc pas présente.

Une option de la machine virtuelle `-no-preexist-ext` utilise uniquement les règles originelles de préexistences.

Nous présenterons donc d'abord les résultats avec les règles originelles de préexistence, puis ceux avec les extensions pour pouvoir observer le gain.

9.6 Protocoles implémentés

L'implémentation d'un site objet n'est pas lié à la préexistence. La préexistence assure juste qu'il n'y aura pas de réparations nécessaires pendant l'appel (des recompilations à chaud). Par conséquent, il est possible d'implémenter les sites de la manière la plus efficace possible sans tenir compte de la préexistence, mais au prix de nombreuses recompilations. Les expériences présentées permettront de quantifier les bénéfices et l'intérêt de la préexistence étendue sur les recompilations. Nous avons implémenté trois protocoles différents pour quantifier cela.

9.6.1 Sites monomorphes et primitifs

Nous avons distingué les sites monomorphes des autres sites objets. Classiquement, un site d'appel monomorphe est un site d'appel de méthode qui appelle toujours la même méthode.

Dans ces expériences, on essayera de détecter en amont le plus possible de sites monomorphe. Dans ce cadre, un site sera dit monomorphe si on peut trivialement déterminer que le receveur est d'un seul type, et donc que nous serons capable de l'implémenter statiquement.

Les sites monomorphes ont donc l'une de propriétés suivantes :

— Le receveur provient d'un new dont la classe est chargée

— Le type statique du receveur est final

— Le receveur est une variable SSA avec l'une des précédentes propriétés

Ces sites monomorphes seront toujours préexistants et implémenté avec la meilleure implémentation possible. Ces sites seront donc classés à part des autres sites-objets des expériences pour ne pas fausser les résultats.

Une deuxième catégorie classée à part est celle des sites primitifs. Les sites primitifs sont des site objets dont le receveur est typé par un type primitif de Nit. Ces types là ne possèdent pas de sous-classes et généralement ils sont situés assez haut dans la hiérarchie, les types concernés ont une relation de sous-typage avec les types suivants (avec leur variante nullable) :

— Bool
— Int
— Char
— Float

Des variantes des types de nombres existent en Nit avec plusieurs tailles disponibles pour les entiers, toutes ces classes sont considérées comme étant des types primitifs.

Les sites monomorphes et primitifs apparaîtront donc dans des colonnes à part dans les résultats.

9.6.2 Protocole basé sur la préexistence

Ce protocole vise à n'optimiser que les sites qui sont préexistants pour éviter les réparations. La définition de l'implémentation des sites peut être précisée avec deux notions. Parmi les implémentions possibles pour un site, on peut distinguer :

Définition 24. Implémentation optimiste L'implémentation optimiste est la meilleure implémentation que le site peut avoir au moment considéré, et donc peut être invalidée par des chargements de classes ultérieurs ou des recompilations.

Définition 25. Implémentation conservatrice L'implémentation conservatrice est la meilleure implémentation du site qui ne nécessitera jamais de recompilation. Cette implémentation ne pourra jamais être invalidée.

Ces deux définitions permettent d'avoir une nuance dans les implémentations des sites objets. Un site est par ailleurs dit *optimisé* si son implémentation utilisée est plus efficace que son implémentation conservatrice. Pour un appel de méthode nous avons la hiérarchie suivante des implémentations :
$statique \leq optimiste \leq optimisée \leq conservatrice \leq hachage\ parfait$

La meilleure implémentation possible pour un appel de méthode est l'appel statique. La pire implémentation est le hachage parfait, c'est la plus coûteuse des implémentations mais elle fonctionnera dans tous les cas.

Entre les deux extrêmes, l'implémentation peut être conservatrice. Si jamais la méthode est introduite dans `Object`, la racine de la hiérarchie, l'implémentation *conservatrice* sera *SST*.

Le protocole basé sur la préexistence stocke deux implémentations dans les sites-objets, la conservatrice et l'optimiste. Si le receveur est préexistant, alors l'implémentation optimiste est utilisée, sinon la conservatrice.

La recompilation est gérée par une recompilation globale de la méthode quand nécessaire : un changement d'implémentation optimiste sur un site qui l'utilise. Cette recompilation totale fonctionne de la manière suivante : lorsque on détecte qu'un site qui utilise une implémentation optimiste doit être recompilé, on positionne un *flag* sur la méthode contenant le site. La prochaine fois que l'on appellera cette méthode, il faudra recompiler tous les sites dans la méthodes. Cette recompilation totale de la méthode est donc totalement paresseuse.

9.6.3 Protocole basé sur le patch de code

Ce protocole est basé exclusivement sur le *code-patching*, il n'utilise pas la préexistence. L'implémentation utilisée dans les sites-objets est toujours la plus efficace possible.

Dans ce protocole, la recompilation des méthodes est toujours partielle et exclusivement basée sur le patch des implémentations. On s'attend donc à ce que ce protocole soit le plus efficace en terme d'implémentation utilisées, mais au détriment d'un très grand nombre de patchs. L'implémentation utilisée dans les sites est par conséquent toujours *optimiste*.

Ces recompilations à base de patch ne sont par contre pas paresseuse. Quand on détecte qu'il faut recompiler un site, l'opération est appliquée immédiatement, même si le site en question ne sera plus jamais exécuté à l'avenir ou si il devra être recompilé une deuxième fois avant sa prochaine exécution.

9.6.4 Protocole mixte

Le protocole mixte combine les deux approches précédentes. Étant donné qu'effectuer du patch de code est plus facile pour les méthodes que pour les attributs, on utilisera des patchs pour les appels de méthodes.

Les autres sites qui devront être recompilés entraîneront une recompilation totale de la méthode comme dans le protocole basé sur la préexistence. Les implémentations utilisées sont donc les suivantes :

— Indépendamment de la préexistence, les sites d'appels de méthode utilisent l'implémentation optimiste

— Les autres sites (tests de sous-typage et accès aux attributs) utilisent l'implémentation optimiste si le receveur est préexistant

— les autres sites utilisent l'implémentation conservatrice si le receveur n'est pas préexistant

9.7 Résultats des expériences

Les résultats présentés ici ont été récoltés en suivant le protocole d'expérimentations basé sur des comptages d'éléments de l'exécution. Techniquement, un important module de la machine virtuelle Nit est dédié à la récolte et à la mise en forme de toutes les statistiques. Il redéfinit différentes méthodes pour effectuer du profilage de l'exécution et compter les éléments dynamiques.

Pour toutes les mesures statiques, les méthodes gérant la première compilation, les recompilations etc. sont redéfinies pour gérer le comptage. De plus, en début et en fin d'une exécution, nous effectuons divers parcours des éléments de la représentation intermédiaire et du modèle pour récolter des données.

	Méthodes	Attributs	Casts	Total
monomorphes	5063	2711	0	7774
préexistants	5762	3537	360	9659
non préexistants	4422	1899	819	7140
total polymorphes	10184	5436	1179	16799
taux de préexistence	56%	65%	30%	57%

TABLE 9.1 – Préexistence originelle pour *nitc* en fin d'exécution

	Méthodes	Attributs	Casts	Total	%
Read	2268	1420	97	3785	53
New	75	3	0	78	1
Call	1933	407	709	3049	43
Cast	11	13	0	24	0
other	135	56	13	204	3
total	4422	1899	819	7140	100

TABLE 9.2 – Raisons de la non-préexistence des sites dans *nitc*

9.7.1 Préexistence originelle

La table 9.1 contient les taux de préexistence de *nitc* avec les règles originelles de préexistence (y compris avec l'analyse des variables). *nitc* présente un taux de préexistence avec les règles originelles qui est dans la moyenne des résultats obtenus dans l'article de base [Detlefs and Agesen, 1999]. Les règles de préexistence originelles sont parfaitement immutables, par conséquent, les résultats en cours et à la fin de l'exécution sont identiques. Avec ces règles, 57% des sites objets polymorphes sont préexistants à la fin de l'exécution. Nous obtenons également 31% de sites polymorphes, pour rappel ces sites seront forcément implémentés avec la meilleure implémentation et seront également préexistants.

Parmi les sites qui ne sont pas préexistants avec les règles de base, on obtient la répartition suivante présentée dans la table 9.2. Il s'agit de sites-objets qui ont pour receveur un retour de méthode, une lecture d'attribut etc. Les pourcentages présentés dans ce tableau sont relatifs au total de la table et servent à illustrer la répartition. Les chiffres présentés dans chacune des lignes sont ceux des sites qui proviennent exclusivement de la règle indiquée. La colonne *other* présente toutes les combinaisons, il s'agit d'un site qui provient d'une combinaison des règles précédentes, donc à travers une variable avec des dépendances multiples : une ϕ-variable.

La plupart des sites qui ne sont pas préexistants proviennent donc de retour de méthodes ou alors de lecture d'attributs.

	Méthodes	Attributs	Casts	Total	%
Read	707	168	0	875	23
New	75	1	0	76	97
Call	396	86	9	491	16
Cast	1	4	0	5	21
other	44	6	6	56	27
total improved	1223	265	15	1503	21

TABLE 9.3 – Effets des différentes règles de préexistence étendue pour *nitc* à la première compilation

	Méthodes	Attributs	Casts	Total	%
Read	719	170	0	889	23
New	75	3	0	78	100
Call	410	96	9	515	17
Cast	1	4	0	5	21
other	46	14	2	62	30
total improved	1251	287	11	1549	21

TABLE 9.4 – Effets des différentes règles de préexistence étendue pour *nitc* en fin d'exécution

9.7.2 Préexistence étendue

Dans le détail, les tables 9.3 et 9.4 présentent l'apport des différentes règles implémentées de la préexistence étendue. Pour tous les sites qui sont maintenants préexistants avec la préexistence étendues, ces deux tableaux présentent quelle est l'origine de leur préexistence. Nous pouvons donc voir quelles sont les règles de préexistence qui fournissent le plus de préexistence dans nos exemple, et celles qui sont plus marginales.

On constate déjà que les chiffres sont relativement similaires entre la version en cours d'exécution et celle en fin d'exécution. Le taux de préexistence entre les deux versions augmente, ce qui fait que les sites qui deviennent préexistants sont plus nombreux que ceux qui perdent leur préexistence.

Ce point est important pour les recompilations, selon le protocole, il n'est pas nécessaire de recompiler pour optimiser. Il est par contre obligatoire de recompiler pour défaire une optimisation qui est devenue incorrecte.

La table 9.4 montre le nombre de site-objets qui sont devenus préexistants par rapport à la table 9.2. Il est intéressant de noter que les deux plus gros contingents étaient les receveurs provenant d'un retour de méthode et d'une lecture d'attributs. Dans ces deux catégories, nous obtenons maintenant respectivement 17% et 23% de préexistence. Ces deux règles sont les plus intéressantes car elles améliorent sensiblement le taux de préexistence global. De plus, les sites améliorables par ces deux règles sont aussi les plus nombreux, ce qui prouve le potentiel d'optimisation.

Les *new* qui n'étaient pas préexistants en étant étiquetés monomorphes lors de leur

	Méthodes	Attributs	Casts	Total
monomorph	5063	2711	0	7774
preexisting	7013	3824	371	11208
non preexisting	3171	1612	808	5591
total polymorph	10184	5436	1179	16799
preexistence rate	68%	70%	31%	66%

TABLE 9.5 – Récapitulatif de la préexistence étendue pour *nitc* en fin d'exécution

compilation sont tous devenus préexistants à la fin de l'exécution. Les *casts* sont améliorés à hauteur de 21% mais ce chiffre n'est pas très représentatif compte tenu du très faible nombre de casts (24 originellement). Les *other* sont améliorés à hauteur de 30%, ce qui prouve déjà qu'il existe des variables avec plusieurs dépendances qui voient leur préexistence fusionner.

Globalement, toutes ces améliorations apportent 21% de préexistence dans ce qui n'était pas préexistant avec la définition originelle de préexistence. Cela représente environ 10% de sites qui deviennent préexistants avec les nouvelles règles par rapport au total des sites polymorphes.

La table 9.5 présente les taux de préexistence avec la préexistence étendue dans *nitc*. On y observe que les extensions de la préexistence apportent un gain indéniable puisque globalement le taux de préexistence passe à 66% (au lieu de 57%).

9.7.3 Implémentations des PIC-patterns

Pour tous les protocoles, l'implémentation des PIC-Patterns est la même. Leur nombre de recompilations est également identique entre les différents protocoles, qui n'agissent pas au niveau des PIC-Patterns.

	MethodPIC-Pattern	AttributePIC-Pattern
monomorphes	992	528
polymorphes	1592	751
sst	1538	741
ph	22	5
null	32	5
Borne théorique des PIC-Patterns	3357	1493

TABLE 9.6 – Implémentation des PIC-Patterns pour *nitc* en fin d'exécution

La table 9.6 présente les implémentations des PIC-Patterns. Que ce soit pour les méthodes ou pour les attributs, la plupart des PIC-Patterns sont candidats à des implémentation en héritage simple (SST). L'implémentation null pour les PIC-Patterns correspond au cas suivant : le pic n'est pas chargé, et le rst non plus, dans ce cas le receveur est forcément nul si jamais un tel site devait être exécuté. Là encore, nous retrouvons un grand nombre de PIC-Patterns monomorphes (dont tous les GP-Patterns

sont monomorphes et donc les sites).

On peut tirer quelques conclusions et explications de cette observation :

— L'héritage multiple n'est pas très présent dans la hiérarchie de classes considérée ou peut être optimisé

— Les optimisations de l'ordonnancement des superclasses sont utiles, voir la règle du préfixe et ses effets plus loin

Nous avons d'ailleurs observé expérimentalement qu'en modifiant l'heuristique du choix de la classe préfixe, le nombre de PIC-Patterns en PH augmentait.

La dernière ligne de la table 9.6 présente la borne théorique des PIC-Patterns. Il s'agit d'évaluer à quel point il est utile de créer paresseusement les PIC-Patterns par rapport à une création systématique au début de l'exécution. Pour les PIC-Patterns de méthodes, la borne théorique est la taille de la fermeture transitive de la relation de spécialisation (un PIC-Pattern pour chaque paire rst/pic). Pour les attributs, le calcul est similaire, en excluant les classes qui n'introduisent pas d'attributs, il y a donc moins de PIC-Patterns en théorie pour les attributs.

Les PIC-Patterns réels sont ceux créés paresseusement, pour lesquels il existe un GP-Pattern de même type statique, où la propriété globale a été introduite par l'autre classe du PIC-Pattern. On observe donc qu'on créé environ la moitié des PIC-Patterns par rapport à leur nombre maximal. Ainsi, la création paresseuse est justifiée et encouragée par rapport à une création systématique au début de l'exécution.

9.7.4 Implémentation des GP-patterns

Les GP-Patterns sont implémentés de la même manière entre les différents protocoles.

La préexistence est sans effet sur l'implémentation des GP-patterns, puisqu'il n'y a pas d'expressions en jeu. Marginalement, les implémentations des patterns changent au cours de l'exécution. La table 9.7 présente les implémentations des GP-Patterns pour *nitc* en fin d'exécution.

Les GP-Patterns qui sont étiquettés monomorphes ne possèdent que des sites qui sont monomorphes (exclusivement). Les sites en question sont donc tous monomorphes. Ces GP-Patterns auront donc la meilleure implémentation possible selon la catégorie du GP-Pattern. Pour des GP-Patterns d'appel de méthode, les constructeurs fournissent un grand nombre de GP-Patterns qui sont monomorphes : on peut en effet toujours calculer statiquement la méthode `init` qui sera appelée par un constructeur.

Parmi les GP-Patterns avec des sites qui ne sont pas monomorphes, on peut quand même avoir la meilleur implémentation possible. Par exemple, pour les appels de méthodes, la plupart des GP-Patterns possèdent une implémentation statique.

Ces résultats peuvent paraître surprenants par le très faible nombre d'implémentations autres que statique pour les appels de méthodes. En réalité, une grande partie de ces patterns sont des patterns d'appel à un constructeur. Ils possèdent donc automatiquement une implémentation statique. Nous avons 2931 GP-Patterns qui sont monomorphes : c'est-à-dire que tous leurs sites sont monomorphes. Cela représente 53%

	CallSitePattern	AttrPattern	SubtypeSitePattern
monomorphes	1484	1447	0
polymorphes	1382	1186	276
static	1118	0	204
sst	170	974	70
ph	70	204	2
null	24	8	0

TABLE 9.7 – Implémentation des GP-Patterns pour *nitc* en fin d'exécution

du nombre total de GP-Patterns, ces patterns seront implémentés statiquement pour les méthodes et en SST pour les attributs.

Pour les patterns d'accès aux attributs, les implémentations statiques n'existent pas mais on retrouve quand même un très grand pourcentage de patterns qui ont une implémentations SST, la plus optimisée possible. Ces chiffres sont satisfaisants, les optimisations de l'ordonnancement des superclasses ont un effet non négligeable sur ces résultats.

Les patterns de tests de sous-typage ont des implémentations mieux réparties que les deux autres catégories. En réalité, les implémentations statiques de ces patterns occultent surtout des implémentations finales.

9.7.5 Protocole en patch de code

Dans ce protocole, la préexistence n'a presque pas d'influence sur les implémentations. Cependant, certaines règles de préexistence permettent d'exploiter des types concrets supplémentaires. Ces types concrets peuvent permettre d'améliorer quelques implémentations effectives.

Dans ce protocole, il y a deux choses à distinguer :

— la préexistence n'est pas utilisée pour choisir l'implémentation effective

— mais les règles sur les types concrets, basées sur la préexistence, améliorent marginalement les implémentations optimistes

Pour les tables suivantes, et celles similaires des autres protocoles, les chiffres sont ceux en fin d'exécution, il s'agit donc des performances "à chaud".

Dans la table 9.8, on obtient les meilleures implémentations de tous les protocoles avec le protocole basé patchs. Cette table est commune à tous les protocoles. Dans le protocole base patch, on utilisera l'implémentation dans cette table indépendamment de la préexistence, donc les totaux des lignes static/SST/PH.

La table 9.9 présente le nombre d'exécutions des sites-objets dans ce protocole. Cela correspond donc au nombre d'exécution des implémentations optimistes. Le nombre d'appels dans cette table est exprimé en millier d'unités. On peut déjà observer que le protocole basé sur les patchs entraîne un coût très faible de l'héritage multiple. Une très faible part des sites sont implémentés avec PH, les autres ont été optimisés. Pour les protocoles des sections suivantes, les résultats seront présentés de la même manière.

	Méthodes	Attributs	Casts	Total
monomorph	5063	2711	0	7774
static	8620	0	603	9223
static preexisting	4911	0	160	5071
static non-preexisting	3713	0	443	4156
SST	1475	5330	573	7378
SST preexisting	815	3439	197	4451
SST non-preexisting	660	1891	376	2927
PH	87	98	3	188
PH preexisting	38	91	3	132
PH non-preexisting	49	7	0	56
Null	2	8	0	10
Null preexisting	1	7	0	8
Null non-preexisting	1	1	0	2
total	10184	5436	1179	16799

	Méthodes	Attributs	Casts	Total
monomorph	5063	2711	0	7774
static	8691	0	603	9294
static preexisting	6105	0	162	6267
static non-preexisting	2590	0	441	3031
SST	1427	5332	573	7332
SST preexisting	876	3730	206	4812
SST non-preexisting	551	1602	367	2520
PH	64	96	3	163
PH preexisting	38	91	3	132
PH non-preexisting	26	5	0	31
Null	2	8	0	10
Null preexisting	1	7	0	8
Null non-preexisting	1	1	0	2
total	10184	5436	1179	16799

TABLE 9.8 – Implémentations optimistes des site-objets dans les protocoles en préexistence originelle en haut, et étendue en bas dans *nitc*

	Méthodes	Attributs	Casts	Total
monomorph	33998	1835	0	35833
static	81394	0	663	82057
SST	45489	66322	4056	115868
PH	574	548	322	1445
total	161456	68706	5042	235204

TABLE 9.9 – Nombre d'exécutions en milliers des sites-objets dans le protocole basé patch en préexistence étendue dans *nitc*

	Méthodes	Attributs	Casts	Total
monomorph	33994	1835	0	35829
static	62637	0	663	63301
SST	42757	63477	3625	109860
PH	21996	3372	752	26121
total	161386	68685	5041	235112

TABLE 9.10 – Nombre d'exécutions en milliers des sites-objets dans le protocole basé préexistence en préexistence étendue dans *nitc*

9.7.6 Protocole basé sur la préexistence

Les résultats présentés pour ce protocole sont ceux concernant l'implémentation effective des sites. Pour le protocole basé sur la préexistence cela signifie que l'implémentation utilisée est conservatrice si le receveur est non-préexistant et optimiste sinon.

La figure 9.8 présente les résultats de ce protocole en considérant que nous utiliserons l'implémentation optimiste uniquement si le site est préexistant, ce sont par exemple les lignes statique préexistants pour les méthodes et SST préexistants pour les attributs. Si les sites ne sont pas préexistants, alors on utilisera l'implémentation conservatrice (au pire PH). Les implémentations sont moins bonnes que dans le protocole basé sur le patch de code, mais toutes les implémentations optimisées sont préexistantes, on s'assure donc d'un faible nombre de recompilations. Les chiffres sont en accord avec ceux à l'exécution de la table 9.10.

On observe d'ailleurs un coût plus élevé de l'héritage multiple à l'exécution par rapport au protocole basé patch, mais cela est du aux implémentations conservatrices.

9.7.7 Protocole mixte

Les résultats présentés pour ce protocole sont aussi ceux concernant l'implémentation effective des sites. L'implémentation utilisée est donc toujours optimiste pour les appels de méthodes. Les autres sites objets utilisent leur implémentation optimiste uniquement si le receveur est préexistant sinon ils utilisent l'implémentation conservatrice.

La figure 9.8 présente également les implémentations des sites dans ce protocole. Pour les méthodes, les résultats sont similaires à ceux du protocole basé patch de code. Par contre pour les attributs et les tests de sous-typage, on retrouve les résultats du protocole basé sur la préexistence. Le compromis semble donc très intéressant. Ces chiffres et proportions se retrouvent dans la table 9.11 avec des chiffres à l'exécution similaires. On peut d'ailleurs observer que le coût de l'héritage multiple est très faible à l'exécution par rapport au protocole basé sur la préexistence. Le nombre d'exécution des accès aux attributs en PH est quant à lui plutôt faible, on peut alors dire que le surcoût est raisonnable au vu des faibles recompilations.

	Méthodes	Attributs	Casts	Total
monomorph	34000	1835	0	35835
static	81438	0	663	82101
SST	45577	63480	3627	112685
PH	574	3434	752	4761
total	161590	68750	5042	235384

TABLE 9.11 – Nombre d'exécutions en milliers des sites-objets dans le protocole mixte en préexistence étendue dans *nitc*

9.8 Recompilations dans les différents protocoles

Ces trois protocoles ont un impact sur le nombre de recompilations. La table 9.12 présente le nombre de recompilations pour chacune des entités du modèle.

Les deux premières lignes du tableau présentent le nombre de recompilations des PIC-Patterns et des GP-Patterns. Ces chiffres sont identiques pour les trois protocoles car ils sont indépendants. Il s'agit du nombre de fois qu'une de ces deux entités doit changer d'implémentation.

Dans le cas des PIC-Pattern, le nombre de recompilations est le nombre de fois où deux classes qui étaient en position invariante l'une par rapport à l'autre deviennent variantes. En clair, la classe du receveur a maintenant plusieurs positions par rapport à la classe d'introduction de la propriété. L'implémentation du PIC-Pattern en question passe donc de SST à PH.

Les GP-Patterns sont aussi concernés par ce changement de position puisqu'il sera propagé jusqu'à eux par leur PIC-Pattern.

Les recompilations des GP-Patterns sont aussi entraînées par l'ajout d'une propriété locale candidate à l'appel. Dans ce cas, l'implémentation peut passer de statique à SST et il faut recompiler ce pattern. Il est aussi possible qu'une méthode soit compilée et que le compteur du nombre de méthodes non compilées dans le pattern atteigne 0. Dans ce cas, l'implémentation doit aussi changer et cela provoque une recompilation du GP-Pattern.

Les recompilations des sites peuvent être provoquées par de multiple éléments :

— Des propagations depuis les PIC-patterns

— Des recompilations provoquées par un changement au niveau des GP-Patterns

— Des évolutions dans les types concrets

— Un changement de préexistence du site dans le sens préexistant -> non-préexistant

Les chiffres de recompilations sur les sites représentent le nombre de fois où l'exécution invalide l'implémentation. Dans le cas des protocoles utilisant des patchs de code, la recompilation du site est immédiate. Pour le protocole de préexistence, en revanche, seuls les sites préexistants ont été optimisés et nécessitent une recompilation. On rappelle que la recompilation dans le protocole de préexistence consiste à recompiler la méthode complète et donc tous ses sites.

Recompilation	Méthodes	Attributs	Casts	Total
PIC-Pattern	42	27	0	69
GP-Pattern	358	136	19	513
Sites protocole basé-patch	1560	358	56	1974
Sites protocole préexistence	1332	183	27	1542
Sites protocole mixte	793	182	27	1002

TABLE 9.12 – Nombre de recompilations des entités du modèle pour les différentes catégories de sites selon les protocoles *nitc*

Dans le protocole mixte, le chiffre de recompilations des sites d'appels de méthode correspond au nombre de fois où nous effectuons des patchs de code pour changer les implémentations. Les chiffres des attributs et des casts représentent, comme dans le protocole basé préexistence, le nombre de fois où on positionne le trampoline de recompilation de la méthode.

Enfin, la figure 9.13 présente le nombre de recompilations totale d'une méthode dans le protocole de préexistence et dans le protocole mixte. La première ligne pour chacun des protocoles indique le nombre de méthodes compilées en première compilation puis ensuite le nombre de recompilations.

La deuxième ligne indique le coût de la recompilation, ce coût est calculé en multipliant le nombre de compilations par le nombre de sites dans la méthode. Cela donne une indication sur la taille des méthodes (re)-compilées : recompiler une méthode avec 100 sites d'appels n'a pas le même impact qu'une recompilation d'une méthode avec quelques sites.

Sans surprise, le protocole mixte a un coût plus faible, car nous utilisons des patchs de code pour changer les implémentations des appels de méthodes. Il faut donc mettre en relation ces chiffres avec le nombre de patchs effectués qui constituent un surcoût. Le protocole mixte provoque 159 recompilations totales des méthodes pour un coût de 8174. Ce protocole provoque donc environ 3 fois moins de recompilations que le protocole basé sur la préexistence, par contre son coût est uniquement deux fois plus faible. Cela signifie que ce sont plutôt des grosses méthodes avec de nombreux sites qui sont recompilées.

En plus de ce coût de recompilations totales, il faut ajouter 1002 sites qui sont patchés : cela représente environ 2 fois moins de sites patchés que le protocole uniquement basé sur le patch de code. Du point de vue de l'efficacité, les performances du protocole mixte sont similaires pour les appels de méthodes à celles du protocole basé patch. À savoir que nous éliminons quasiment tous les appels qui devraient être exécutés avec du hachage parfait : environ 20 millions d'appels en moins pour des appels statiques à la place. Il est difficile de pousser plus loin l'analyse car nous n'avons pas pu mesurer en temps le coût d'un patch par rapport au coût d'une recompilation de méthode. Malgré cela, ce protocole présente des caractéristiques très intéressantes du point de vue de l'efficacité résultante et nous semble être le meilleur compromis.

	Compilations	Recompilations
recompilations préexistence	3039	498
coût	27616	16129
recompilations mixte	3039	159
coût mixte	27616	8174

TABLE 9.13 – Nombre de recompilations totale des méthodes pour les différents protocoles dans *nitc*

9.9 Analyses et discussions des résultats

9.9.1 Extensions de la préexistence

Les expériences ont montré que les extensions de la préexistence étaient pertinentes. Nous avons augmenté le nombre de site-objet dont le receveur est préexistant, ce qui réduit le nombre de recompilations ultérieures en exploitant cette information.

Certaines des règles de la préexistence étendue introduisent de la mutabilité dans la préexistence contrairement aux règles originales. Les expériences ont prouvé qu'en pratique assez peu de site-objets voient leur receveur passer de préexistant à non-préexistant. Les extensions n'introduisent donc pas un surcoût de ce point de vue là.

Les mutations dans le sens non-préexistant vers préexistant sont moins problématique. En effet, nous ne sommes pas forcés d'optimiser tous les sites et il est possible de ne pas recompiler pour optimiser dans ce cas. Les changements dans l'autre sens obligent par contre à recompiler si la préexistence avait servi à optimiser. On peut donc dire que les extensions sont pertinentes.

Cependant, le coût général du calcul des extensions de la préexistence n'a pas été mesuré. Il faudrait réaliser ces expériences pour en tirer des conclusions définitives.

9.9.2 Implémentations optimisées et préexistence

Les implémentations optimisées dans le protocole de préexistence sont nombreuses, et les résultats sont satisfaisant de ce point de vue.

La table 9.14 présente les statistiques pour les appels de méthodes qui ont la meilleure implémentation possible (statique) dans le protocole de préexistence. La troisième colonne de la table présente le pourcentage d'amélioration du nombre de ces appels de méthodes entre la préexistence originelle et étendue. En préexistence originelle, nous avons 48% des appels de méthodes qui sont implémentés statiquement en moyenne sur tous nos benchmarks. En comparaison, nous obtenons 60% en préexistence étendue, le pourcentage d'amélioration entre ces deux chiffres est donc de 24% environ, ce qui n'est pas négligeable.

De manière similaire, la table 9.15 présente les accès aux attributs préexistants avec la meilleure implémentation possible (SST). En préexistence originelle, nous avons déjà 66.5% de taux de préexistence, ce qui est plus que pour les appels de méthodes. Par conséquent, l'amélioration est moins importante avec seulement 7% d'amélioration. En

Benchmark	Original	Étendu	Amélioration	Total
nitc	4911	6101	24%	10184
niti	3682	4381	19%	7918
nitdoc	1723	2397	39%	4525
jwrapper	2061	2356	14%	2950
nitiwiki	267	391	46%	680
Total	12644	15626	24%	26228

TABLE 9.14 – Appels de méthodes statiques dans le protocole pure-préexistence

Benchmark	Original	Étendu	Amélioration	Total
nitc	3439	3726	8%	5436
niti	2274	2428	7%	3494
nitdoc	2293	2456	7%	3463
jwrapper	912	958	5%	1083
nitiwiki	363	379	4%	466
Total	9281	9947	7%	13942

TABLE 9.15 – Sites compilés statiquement en SST dans le protocole pure-préexistence

préexistence étendue, nous avons maintenant 71% de taux de préexistence.

Globalement, pour les attributs et les méthodes, les extensions sont intéressantes puisque le gain est non-négligeable. Pour les sites d'accès aux attributs, le taux de préexistence est assez élevé avec les extensions.

Le ratio des différences : (étendue − originelle)×100÷(total − originelle) représente le gain relativement aux améliorations possibles. Ce ratio indique un taux d'amélioration de : $(15626 - 12644) \times 100 \div (26228 - 15626) = 28\%$ pour les appels de méthode sur la moyenne des benchmarks.

Pour les attributs ce même ratio est de : $(9947-9281)\times100\div(13942-9947) = 16.5\%$.

9.9.3 Coût de l'héritage multiple

Le coût de l'héritage multiple avec les optimisations demeure finalement assez faible. Le nombre de sites implémentés en hachage parfait reste très faible. Pour le protocole basé sur la préexistence, pendant l'exécution, seulement 5% des accès aux attributs sont effectués avec le hachage parfait. Ce chiffre est de 13.5% pour les sites d'appels de méthodes. La grande majorité des implémentations contient donc des implémentations optimale ou différente du hachage parfait. Nous pouvons avancer plusieurs hypothèses à ce faible taux de sites utilisant le hachage parfait, en plus des optimisations présentées précédemment :

— L'héritage multiple est marginalement utilisé dans les benchmarks considérés

— L'heuristique du préfixe est efficace

Pour la première hypothèse, il y a 881 classes chargées en exécutant *nitc*. Sur ces

classes, seulement 31 ont plusieurs superclasses directes. Le programme testé contient donc beaucoup de classes en héritage simple, ce qui n'est pas surprenant pour un programme contenant beaucoup de feuilles dans la hiérarchie (essentiellement des classes du parseur et de l'arbre syntaxique).

Pour la deuxième hypothèse, 40 classes différentes sont positionnées dans un suffixe, 29 dans un suffixe de l'ordre des méthodes, et 32 dans le suffixe de l'ordre des attributs. L'heuristique va privilégier les classes qui introduisent beaucoup de propriétés, et sacrifier celles qui en introduisent moins.

Ces chiffres sont à mettre en relation avec le nombre de PIC-Patterns qui utilisent le hachage parfait : ils ne sont que 27, ce qui ne représente que 1% par rapport au nombre total de PIC-Pattern, l'heuristique s'avère donc pertinente.

Au niveau des GP-Patterns, on dénombre 44 GP-Patterns qui ont des positions multiples par rapport aux 2568 GP-Patterns polymorphes. Pour conclure, la chance ou le hasard entrent aussi en jeu, l'heuristique n'est pas parfaite et une situation particulièrement défavorable (ou favorable) pourrait arriver à l'exécution avec un très grand nombre de sites en hachage parfait exécuté.

9.9.4 Protocoles de compilation/recompilation

Dans les protocoles implémentés, le protocole basé sur les patchs de code est celui qui est le plus efficace comme prévu. Il sert en effet de comparatif avec les autres protocoles. Si un programme ne contient pas du tout d'héritage multiple, l'implémentation produite par ce protocole sera optimale pour les attributs si nous générons uniquement le `fast-path` dans le code, voir chapitre 3. En héritage multiple, la situation est moins favorable à cause du grand nombre de patchs, ou alors si jamais les `slow-path` sont générés.

Le protocole basé sur la préexistence uniquement donne des résultats satisfaisant, les recompilations sont assez faibles et l'efficacité est également intéressante. Finalement, le protocole mixte est probablement le plus intéressant, les recompilations sont assez faibles également et il permet d'augmenter sensiblement les appels de méthodes optimisés.

9.10 Conclusion

Dans ce chapitre, nous avons présenté les expérimentations menées dans le cadre de ce projet. Nous utilisons un ensemble de benchmarks qui sont des programmes Nit, leur nombre bien que limité permet d'avoir des programmes de taille assez importante.

Nous avons mesuré et montré les résultats des extensions apportées à la préexistence. Ces extensions se révèlent intéressantes. Ensuite nous avons présenté les protocoles de compilation/recompilation implémentés avec les résultats. Pour montrer leur efficacité, la méthodologie utilisée est basée sur le comptage d'éléments dans le code tels que le nombre de recompilations, le nombre de sites d'appels qui ont une implémentation optimisée. Nous effectuons aussi des mesures dynamiques en comptant en cours d'exécution le nombre d'appel pour chaque implémentation.

Les résultats présentés sont encourageants et les protocoles implémentés semblent intéressants. Cependant, il n'est pas possible de mesurer le coût induit par le protocole en lui même en mesurant le temps processeur consommé pour prendre les décisions. De plus, il faudrait pouvoir mesurer précisément les effets de la préexistence étendue avec de l'inlining, ce qui n'a pas encore été réalisé. Les études présentées, en particulier concernant les protocoles demandent donc à être confirmées.

Chapitre 10

Conclusion générale

Contents

10.1 Conclusion

Aujourd'hui, la combinaison machine virtuelle et langage à objet est devenue courante, comme nous l'avons montré. Les machines virtuelles présentent en effet de nombreux avantages en matière de portabilité, de découpage des problématiques, etc. Nous avons présenté ensuite les optimisations qui sont nécessaires pour assurer de bonnes performances en monde ouvert. Les performances d'une machine virtuelle pour un langage à objet sont très dépendantes, entre autres, de l'implémentation efficace des trois mécanismes objet : appel de méthode, accès aux attributs et tests de sous-typage. Nous avons défini la notion de **protocole de compilation/recompilation**, qui est le système chargé d'effectuer les optimisations en récoltant de l'information pour ensuite effectuer si besoin des réparations ou recompilations du code. Nous avons aussi présenté un état de l'art des techniques et méthodes employées pour mesurer la performance des systèmes d'exécution.

L'objectif de cette thèse était double :

— Spécifier et implémenter un prototype d'une machine virtuelle pour un langage à objet en héritage multiple, ce qui n'a jamais été réalisé à notre connaissance

— Implémenter et tester des protocoles de compilation/recompilation dans cette machine virtuelle

10.2 Contributions

Les principales contributions de cette thèse sont :

— La réalisation d'un prototype de machine virtuelle pour un langage en héritage multiple et en typage statique
— La simulation d'un compilateur à la volée dans cette machine virtuelle
— Une extension de l'analyse de préexistence associée à une analyse de types concrets
— La spécification de protocoles de compilation/recompilation/réparation basés sur la préexistence ou des patchs de code, dans un contexte où nous nous intéressons principalement à la généralisation de la dévirtualisation
— Des expérimentations pour mesurer l'efficacité de ces protocoles en utilisant des mesures discrètes à l'exécution

La spécification de la machine virtuelle est très ressemblante à celle de la majorité des machines virtuelles Java. Nous avons utilisé le langage Nit [Privat, 2008], un langage à objet en héritage multiple et en type statique en tant que langage source de la machine virtuelle. De façon totalement annexe, l'utilisation du langage Nit a d'ailleurs permis de contribuer au design du langage, à travers une étude de la covariance de la généricité de Nit [Ducournau et al., 2014]. La machine virtuelle a également été développée en Nit, en se servant de l'interpréteur Nit existant en tant que base de code. La machine virtuelle Nit possède les propriétés suivantes :

— Chargement dynamique
— Simulation de compilation à la volée
— Système d'optimisations adaptatif

Dans le cadre de ce travail, nous nous sommes concentrés sur les implémentations et optimisations associées aux mécanismes objet bien qu'il ne s'agisse pas des seuls éléments à optimiser dans les machines virtuelles. La compilation à la volée a été seulement simulée dans la machine virtuelle. Cela signifie que nous choisissons *a priori* les implémentations de tous les sites d'invocation des mécanismes objet du code mais que nous ne produisons pas de code machine.

Une fois le premier objectif rempli, il était possible de travailler sur les protocoles de compilation/recompilation. Nous avons commencé par enrichir le méta-modèle existant de Nit basé sur les propriétés locales et globales pour créer une représentation intermédiaire ainsi que des éléments permettant de factoriser les implémentations du code. Nous avons ensuite étendu la définition et l'usage de la préexistence pour la généraliser à l'héritage multiple puis en augmentant son champ d'application. La préexistence étendue est maintenant applicable à toutes les catégories de site-objets et plus seulement aux appels de méthodes. Ensuite, nous avons considéré les expressions de manière générale pour ne plus se limiter aux receveurs. Enfin, nous avons présenté une analyse permettant de déduire des types concrets en explicitant leurs liens avec la préexistence. Cette contribution a aussi l'avantage de n'être pas limitée aux langages en héritage multiple, dans la mesure où elle s'appliquerait aussi aux langages comme Java et C♯ ou similaires.

Nous avons implémenté plusieurs protocoles de compilation/recompilation. Trois stratégies différentes étaient présentées : un protocole basé uniquement sur le patch de code, qui utilise toujours les meilleures implémentations possible. Un deuxième protocole était basé sur la préexistence dans lequel seulement les sites préexistants étaient

optimisés. Le dernier protocole était une stratégie mixte : les sites d'appels de méthode se comportaient comme ceux du protocole basé sur les patchs, et tous les autres sites n'étaient optimisés que s'ils étaient préexistants. Ce protocole est probablement le meilleur compromis entre les trois en matière d'efficacité et de recompilations.

L'efficacité des protocoles a été mesurée en comptant des éléments à l'exécution : de manière statique, dans le code source, et de manière dynamique en utilisant des compteurs à l'exécution. Nous avions ainsi des mesures très précises sur les sites et leur préexistence et implémentation ainsi que les recompilations.

La problématique initiale était de déterminer si on pouvait avoir la même efficacité dans l'implémentation de l'héritage multiple dans une machine virtuelle en monde ouvert par rapport à des langages en sous-typage multiple. De façon absolue, la réponse est non. De façon relative, nous avons montré que le surcoût était faible et comparable à celui de Java.

Les optimisations présentées assurent que du point de vue des implémentations, la non-présence de l'héritage multiple dans un programme source donnera un surcoût nul avec le protocole basé patchs. Le programme utilisera les mêmes implémentations que celles produites par un compilateur global. Cela ne sera pas le cas dans le protocole utilisant la préexistence, bien que le surcoût soit faible. Enfin, le faible nombre de réparations du protocole mixte ainsi que son surcoût très faible au niveau des implémentations en font certainement le meilleur compromis. En présence d'héritage multiple, le protocole en pur code-patching n'offre pas le meilleur compromis : il est donc préférable de choisir l'approche mixte.

Sur les benchmarks utilisés, le surcoût est assez faible quels que soient les protocoles et les programmes grâce aux optimisations.

10.3 Perspectives

La machine virtuelle a été développée dans un temps contraint. Elle constitue un prototype et reste assez inefficace quant à son temps d'exécution. Le travail effectué peut alors être enrichi dans plusieurs directions.

Tout d'abord, les protocoles de compilation/recompilation implémentés sont surtout basés sur la préexistence et les patchs de code. Il faudrait diversifier les protocoles en intégrant de nouvelles optimisations. La problématique reste ouverte, mais les opportunités offertes par l'analyse des types concrets et la préexistence constituent un bon support à d'autres optimisations. Pour mesurer encore plus finement l'efficacité de nos optimisations, il serait intéressant de comparer l'efficacité de nos implémentations par rapport à celles produites par un compilateur statique. Il serait possible de calculer toutes les implémentations en fin d'exécution en ayant connaissance de l'intégralité du programme, et de comparer ces résultats avec ceux fournis par les différents protocoles. L'inlining est une des optimisations les plus efficaces, nous avons montré que notre analyse de préexistence augmentait théoriquement les possibilités d'inlining en augmentant le nombre de dévirtualisations. Il faudrait maintenant intégrer pleinement l'inlining au sein des protocoles.

Cependant, les effets de l'inlining et de la préexistence peuvent être difficilement conciliables. En effet, si nous considérons deux méthodes `foo()` et `bar()`, le fait d'inliner `bar()` dans `foo()` peut poser problème. Si `foo()` est ensuite inliné, la préexistence du receveur de `bar()` sera cassée. Il faut alors déterminer s'il vaut mieux conserver le premier inlining ou alors le défaire pour inliner le deuxième appel. La réponse se trouve peut-être dans la structure du code, l'un des appels peut être situé dans une boucle et il faudra alors favoriser son inlining. Des études complémentaires seraient nécessaires pour mesurer ces effets, et trouver des compromis intéressants. L'extension de la préexistence n'est pas limitée aux langages en héritage multiple, il serait intéressant d'étudier l'effet de ces extensions à des langages comme Java ou C♯.

À plus long terme, la principale extension de ce travail consisterait à développer un véritable compilateur à la volée qui produirait du code machine. Plusieurs options sont possibles pour cette génération de code. La première possibilité serait d'écrire à la main un générateur d'assembleur, ce qui serait très spécifique à un type de processeur. La deuxième possibilité consisterait à utiliser un outil tel que LLVM [Lattner and Adve, 2004] qui permettrait de générer du code plus efficace sans se soucier de la portabilité. Il faudrait par contre préalablement intégrer cet outil au langage Nit.

Cette extension permettrait de réaliser certaines optimisations de manière parfaitement réaliste, telles que la spécialisation de code ou encore l'inlining. De manière générale nous nous sommes focalisé sur les implémentations des mécanismes objet, mais il faudrait également travailler sur toutes les autres optimisations qui seraient intégrables dans la stratégie des protocoles de compilation/recompilation.

Des optimisations qui ne sont pas limitées au monde objet telles que l'optimisation des boucles ou des optimisations liées à la génération de code (propagation de constante, ordre des instructions) seraient également des pistes à explorer.

Enfin, de manière annexe à la machine virtuelle Nit, il serait intéressant de développer un compilateur de Nit vers un langage de bytecode. Ce langage serait d'ailleurs à spécifier, mais on peut imaginer y intégrer des éléments dedans qui aideraient la machine virtuelle à effectuer des optimisations. Comme par exemple, certaines analyses concernant les types concrets.

Bibliographie

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D., Deschamp, P., Lorho, B., Sagot, B., and Thomasset, F. (2007). *Compilateurs : principes, techniques et outils*. Pearson Education.

[Alpern et al., 2001a] Alpern, B., Cocchi, A., Fink, S., and Grove, D. (2001a). Efficient implementation of java interfaces : Invokeinterface considered harmless. *SIGPLAN Not.*, 36(11) :108–124.

[Alpern et al., 2001b] Alpern, B., Cocchi, A., and Grove, D. e. a. (2001b). Dynamic type checking in jalapeno. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 41–52.

[Alpern et al., 1988] Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM.

[Arnold et al., 2000] Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. (2000). Adaptive optimization in the jalapeno jvm. In *ACM SIGPLAN Notices*, volume 35, pages 47–65. ACM.

[Arnold et al., 2004] Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. F. (2004). Architecture and policy for adaptive optimization in virtual machines. Technical report, Technical Report 23429, IBM Research.

[Arnold and Ryder, 2002] Arnold, M. and Ryder, B. G. (2002). Thin guards : A simple and effective technique for reducing the penalty of dynamic class loading. In *ECOOP'02*, pages 498–524. Springer.

[Auteurs multiples, 2016] Auteurs multiples, I. (2016). *Static Single Assignment Book*. Springer.

[Aycock, 2003] Aycock, J. (2003). A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2) :97–113.

[Bacon and Sweeney, 1996] Bacon, D. F. and Sweeney, P. F. (1996). Fast static analysis of c++ virtual function calls. *ACM Sigplan Notices*, 31(10) :324–341.

[Barnes, 2006] Barnes, J. (2006). *Programming in Ada 2005 (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc.

[Bebenita et al., 2010] Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., and Venter, H. (2010). Spur : a trace-based jit compiler for cil. In *ACM Sigplan Notices*, volume 45, pages 708–725. ACM.

[Binder, 2006] Binder, W. (2006). Portable and accurate sampling profiling for java. *Software : Practice and Experience*, 36(6) :615–650.

[Blackburn et al., 2004] Blackburn, S. M., Cheng, P., and McKinley, K. S. (2004). Oil and water ? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146. IEEE Computer Society.

[Blackburn et al., 2006] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., et al. (2006). The dacapo benchmarks : Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM.

[Bobrow et al., 1988] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. (1988). Common lisp object system specification. *ACM Sigplan Notices*, 23(SI) :1–142.

[Boehm, 1993] Boehm, H.-J. (1993). Space efficient conservative garbage collection. In *ACM SIGPLAN Notices*, volume 28, pages 197–206. ACM.

[Boehm and Weiser, 1988] Boehm, H.-J. and Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software : Practice and Experience*, 18(9) :807–820.

[Bolz et al., 2009] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. (2009). Tracing the meta-level : Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA. ACM.

[Bonetta, 2015] Bonetta, D. (2015). 4.6 parallel javascript in truffle. *Concurrent Computing in the Many-core Era*, page 22.

[Bracha and Cook, 1990] Bracha, G. and Cook, W. (1990). Mixin-based inheritance. *ACM Sigplan Notices*, 25(10) :303–311.

[Briggs et al., 1997] Briggs, P., Cooper, K. D., and Simpson, L. T. (1997). Value numbering. *Software-Practice and Experience*, 27(6) :701–724.

[Bruce et al., 1998] Bruce, K. B., Odersky, M., and Wadler, P. (1998). A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming*, pages 523–549. Springer.

[Campanoni et al., 2008] Campanoni, S., Agosta, G., and Reghizzi, S. C. (2008). A parallel dynamic compiler for cil bytecode. *ACM Sigplan Notices*, 43(4) :11–20.

[Chambers and Ungar, 1989] Chambers, C. and Ungar, D. (1989). Customization : Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Notices*, volume 24, pages 146–160. ACM.

[Chevalier-Boisvert and Feeley, 2015] Chevalier-Boisvert, M. and Feeley, M. (2015). Simple and effective type check removal through lazy basic block versioning. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 101–123.

[Cierniak et al., 2005] Cierniak, M., Eng, M., Glew, N., Lewis, B., and Stichnoth, J. (2005). The open runtime platform : a flexible high-performance managed runtime environment. *Concurrency and Computation : Practice and Experience*, 17(5-6) :617–637.

[Cierniak et al., 2002] Cierniak, M., Lewis, B. T., and Stichnoth, J. M. (2002). Open runtime platform : Flexibility with performance using interfaces. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, pages 156–164, New York, NY, USA. ACM.

[Click and Rose, 2002] Click, C. and Rose, J. (2002). Fast subtype checking in the hotspot jvm. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 96–107, New York, NY, USA. ACM.

[Cohen, 1991] Cohen, N. H. (1991). Type-extension type test can be performed in constant time. *TOPLAS*, 13(4) :626–629.

[Cousot, 1997] Cousot, P. (1997). Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–331. ACM.

[Cousot, 2002] Cousot, P. (2002). Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1) :47–103.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM.

[Cytron et al., 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4) :451–490.

[Dean et al., 1995] Dean, J., Grove, D., and Chambers, C. (1995). Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95*, pages 77–101. Springer.

[Detlefs and Agesen, 1999] Detlefs, D. and Agesen, O. (1999). Inlining of virtual methods. In *ECOOP'99*, pages 258–277. Springer.

[Dixon et al., 1989] Dixon, R., McKee, T., Vaughan, M., and Schweizer, P. (1989). A fast method dispatcher for compiled languages with multiple inheritance. In *ACM SIGPLAN Notices*, volume 24, pages 211–214. ACM.

[Driesen, 1999] Driesen, K. (1999). *Software and hardware techniques for efficient polymorphic calls*. PhD thesis, University of California at Santa Barbara, Santa Barbara, CA, USA.

[Driesen and Hölzle, 1995] Driesen, K. and Hölzle, U. (1995). Minimizing row displacement dispatch tables. *ACM SIGPLAN Notices*, 30(10) :141–155.

[Driesen et al., 1995] Driesen, K., Hölzle, U., and Vitek, J. (1995). Message dispatch on pipelined processors. In *European Conference on Object-Oriented Programming*, pages 253–282. Springer.

[Ducasse et al., 2006] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits : A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2) :331–388.

[Ducournau, 1991] Ducournau, R. (1991). Y3 : Yafool, le langage à objets, et yafen, l'interface graphique. *Sema Group, Montrouge*.

[Ducournau, 2008] Ducournau, R. (2008). Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6) :33 :1–33 :56.

[Ducournau, 2011a] Ducournau, R. (2011a). Coloring, a versatile technique for implementing object-oriented languages. *Software : Practice and Experience*, 41(6) :627–659.

[Ducournau, 2011b] Ducournau, R. (2011b). Implementing statically typed object-oriented programming languages. *ACM Computing Surveys (CSUR)*, 43(3) :18 :1–18 :48.

[Ducournau, 2016] Ducournau, R. (2016). Programmation par objets : des concepts fondamentaux à leur application dans les langages. `http://www.lirmm.fr/~ducour/Publis/objets2.pdf`. Support de cours.

[Ducournau and Morandat, 2011] Ducournau, R. and Morandat, F. (2011). Perfect class hashing and numbering for object-oriented implementation. *Software : Practice and Experience*, 41(6) :661–694.

[Ducournau and Morandat, 2012] Ducournau, R. and Morandat, F. (2012). Towards a full multiple-inheritance virtual machine. *Journal of Object Technology*, 11(3) :6 :1–29.

[Ducournau et al., 2007] Ducournau, R., Morandat, F., and Privat, J. (2007). Modules and class refinement : a metamodeling approach to object-oriented languages. *Rapport de Recherche LIRMM-07021, Université Montpellier*, 2 :1–37.

[Ducournau et al., 2009] Ducournau, R., Morandat, F., and Privat, J. (2009). Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 41–60.

[Ducournau et al., 2014] Ducournau, R., Pagès, J., Privat, J., and Vidal, C. (2014). Genericity and (Co)variance, an Empirical Study. Rapport de recherche, LIRMM ; Université de Montpellier.

[Ducournau et al., 2015] Ducournau, R., Pagès, J., Vidal, C., and Privat, J. (2015). Preexistence revisited. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '15, pages 1–1. ACM.

[Ducournau et al., 2016] Ducournau, R., Pagès, J., and Privat, J. (2016). Preexistence and concrete type analysis in the context of multiple inheritance. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform : Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, pages 10 :1–10 :12.

[Ducournau and Privat, 2011] Ducournau, R. and Privat, J. (2011). Metamodeling semantics of multiple inheritance. *Science of Computer Programming*, 76(7) :555–586.

[Fernández, 1995] Fernández, M. F. (1995). Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 103–115, New York, NY, USA. ACM.

[Ferrante et al., 1987] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3) :319–349.

[Findler and Flatt, 1998] Findler, R. B. and Flatt, M. (1998). Modular object-oriented programming with units and mixins. In *ACM SIGPLAN Notices*, volume 34, pages 94–104. ACM.

[Fink and Qian, 2003] Fink, S. J. and Qian, F. (2003). Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization*, pages 241–252. IEEE Computer Society.

[Flanagan and Matsumoto, 2008] Flanagan, D. and Matsumoto, Y. (2008). *The ruby programming language*. O'Reilly Media, Inc.

[Flatt et al., 1998] Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998). Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM.

[Gagnon and Hendren, 2001] Gagnon, E. M. and Hendren, L. J. (2001). Sablevm : A research framework for the efficient execution of java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, volume 1, pages 27–39.

[Gal et al., 2009] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., et al. (2009). Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6) :465–478.

[Gary and Johnson, 1979] Gary, M. R. and Johnson, D. S. (1979). Computers and intractability : A guide to the theory of np-completeness.

[Gélinas et al., 2009] Gélinas, J., Gagnon, E., and Privat, J. (2009). Prévention de déréférencement de références nulles dans un langage à objets. *Langages et Modèles à Objets*, 3 :5–16.

[Geoffray, 2009] Geoffray, N. (2009). *Fostering system research with managed runtimes*. PhD thesis, Paris 6. Thèse de doctorat dirigée par Folliot, Bertil Informatique Paris 6 2009.

[Geoffray et al., 2010] Geoffray, N., Thomas, G., Lawall, J., Muller, G., and Folliot, B. (2010). Vmkit : a substrate for managed runtime environments. *ACM Sigplan Notices*, 45(7) :51–62.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.

[Google, 2016] Google (2016). Google v8. `https://developers.google.com/v8/`.

[Gosling, 2000] Gosling, J. (2000). *The Java language specification*. Addison-Wesley Professional.

[Gough, 2001] Gough, K. J. (2001). Stacking them up : a comparison of virtual machines. *Aust. Comput. Sci. Commun.*, 23(4) :55–61.

[Hejlsberg et al., 2003] Hejlsberg, A., Wiltamuth, S., and Golde, P. (2003). *C# language specification*. Addison-Wesley Longman Publishing Co., Inc.

[Hölzle et al., 1992] Hölzle, U., Chambers, C., and Ungar, D. (1992). Debugging optimized code with dynamic deoptimization. In *ACM Sigplan Notices*, volume 27, pages 32–43. ACM.

[Hölzle and Ungar, 1994] Hölzle, U. and Ungar, D. (1994). A third-generation self implementation : reconciling responsiveness with performance. In *ACM SIGPLAN Notices*, volume 29, pages 229–243. ACM.

[Ingalls, 1978] Ingalls, D. H. (1978). The smalltalk-76 programming system design and implementation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 9–16. ACM.

[Ishizaki et al., 2000] Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., and Nakatani, T. (2000). A study of devirtualization techniques for a java just-in-time compiler. In *ACM SIGPLAN Notices*, volume 35, pages 294–310. ACM.

[JRockit, 2003] JRockit, B. (2003). Java for the enterprise. *Technical White Paper*.

[Kalibera et al., 2014] Kalibera, T., Maj, P., Morandat, F., and Vitek, J. (2014). A fast abstract syntax tree interpreter for r. In *VEE '14*, pages 89–102. ACM.

[Kawahito et al., 2000] Kawahito, M., Komatsu, H., and Nakatani, T. (2000). Effective null pointer check elimination utilizing hardware trap. *ACM SIGOPS Operating Systems Review*, 34(5) :139–149.

[Kent and Serra, 2000] Kent, K. B. and Serra, M. (2000). Hardware/software co-design of a java virtual machine. In *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on*, pages 66–71. IEEE.

[Knuth, 1997] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[Knuth, 1998] Knuth, D. E. (1998). *The art of computer programming : sorting and searching*, volume 3. Pearson Education.

[Kotzmann et al., 2008] Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., and Cox, D. (2008). Design of the java hotspot^{TM}client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1) :7 :1–7 :32.

[Laferrière, 2012] Laferrière, A. (2012). L'interface native de nit, un langage de programmation à objets. Mémoire de maîtrise, Université du Québec à Montréal.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). Llvm : A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-directed and Runtime Optimization*, CGO '04, pages 75–88, Washington, DC, USA. IEEE Computer Society.

[Leroy et al., 2014] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. (2014). *The OCaml system release 4.02 : Documentation and user's manual*. Inria.

[Lindholm et al., 2012] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2012). The java virtual machine specification : Java se 7 edition.

[Matsakis and Klock II, 2014] Matsakis, N. D. and Klock II, F. S. (2014). The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM.

[McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4) :184–195.

[Meyer, 2002] Meyer, B. (2002). *Eiffel*. Bildarchiv der ETH-Bibliothek Prod.

[Morandat, 2010] Morandat, F. (2010). *Contribution à l'efficacité de la programmation par objets : évaluation des implémentations de l'héritage multiple en typage statique*. PhD thesis, Montpellier 2.

[O'connor and Tremblay, 1997] O'connor, J. M. and Tremblay, M. (1997). picojava-i : The java virtual machine in hardware. *Micro, IEEE*, 17(2) :45–53.

[Odersky, 2009] Odersky, M. (2009). The scala language specification, version 2.8. *EPFL Lausanne, Switzerland*.

[Odersky et al., 2008] Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala : a comprehensive step-by-step guide*. Artima Inc.

[Pagès, 2015] Pagès, J. (2015). A virtual machine for testing compilation/recompilation protocols in multiple inheritance. In *ECOOP Doctoral Symposium '15*, pages 1–10.

[Pagès, 2013] Pagès, J. (2013). Étude de machines virtuelles java existantes et adaptation au hachage parfait. Mémoire de master 2 recherche, LIRMM - Université de Montpellier.

[Paleczny et al., 2001] Paleczny, M., Vick, C., and Click, C. (2001). The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–12. USENIX Association.

[Privat, 2006] Privat, J. (2006). *De l'expressivité à l'efficacité : une approche modulaire des langages à objets : le langage PRM et le compilateur prmc*. PhD thesis, Montpellier 2.

[Privat, 2008] Privat, J. (2008). Nit language. `http://nitlanguage.org/`.

[Privat and Ducournau, 2005] Privat, J. and Ducournau, R. (2005). Link-time static analysis for efficient separate compilation of object-oriented languages. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005*, pages 20–27.

[Pugh and Weddell, 1990] Pugh, W. and Weddell, G. (1990). Two-directional record layout for multiple inheritance. In *ACM SIGPLAN Notices*, volume 25, pages 85–91, New York, NY, USA. ACM.

[Qian and Hendren, 2004] Qian, F. and Hendren, L. J. (2004). Towards dynamic interprocedural analysis in jvms. In *Virtual Machine Research and Technology Symposium*, pages 139–150.

[Qian and Hendren, 2005] Qian, F. and Hendren, L. J. (2005). A study of type analysis for speculative method inlining in a jit environment. In *CC*, pages 255–270. Springer.

[Rigo and Pedroni, 2006] Rigo, A. and Pedroni, S. (2006). Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM.

[Rose, 2009] Rose, J. R. (2009). Bytecodes meet combinators : Invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2 :1–2 :11, New York, NY, USA. ACM.

[Saleil, 2013] Saleil, B. (2013). Étude de kit de développement de compilateurs et machines virtuelles. Stage de master 2, Université Montpellier 2 - LIRMM.

[Sallenave and Ducournau, 2012] Sallenave, O. and Ducournau, R. (2012). Efficient compilation of .net programs for embedded systems. *Journal of Object Technology*, 11(3) :1–28.

[Schärli et al., 2003] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. P. (2003). Traits : Composable units of behaviour. In *ECOOP 2003–Object-Oriented Programming*, pages 248–274. Springer.

[Shang, 1996] Shang, D. (1996). Are cows animals. *Object Currents*, 1(1).

[Shiv et al., 2009] Shiv, K., Chow, K., Wang, Y., and Petrochenko, D. (2009). Specjvm2008 performance characterization. In *SPEC Benchmark Workshop*, pages 17–35. Springer.

[Shivers, 1988] Shivers, O. (1988). Control flow analysis in scheme. In *ACM SIGPLAN Notices*, volume 23, pages 164–174. ACM.

[Shivers, 1991] Shivers, O. (1991). *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer.

[Singer, 2003] Singer, J. (2003). Jvm versus clr : a comparative study. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169. Computer Science Press, Inc.

[Soman and Krintz, 2006] Soman, S. and Krintz, C. (2006). Efficient and general on-stack replacement for aggressive program specialization. In *Software Engineering Research and Practice*, pages 925–932.

[Sonntag and Colnet, 2014] Sonntag, B. and Colnet, D. (2014). Efficient compilation strategy for object-oriented languages under the closed-world assumption. *Software : Practice and Experience*, 44(5) :565–592.

[specsJVM, 2008] specsJVM (2008). Spec jvm 2008 Benchmarks. `https://www.spec.org/jvm2008/`.

[Steiner et al., 2007] Steiner, E., Krall, A., and Thalinger, C. (2007). Adaptive inlining and on-stack replacement in the cacao virtual machine. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 221–226. ACM.

[Stroustrup, 1995] Stroustrup, B. (1995). *The C++ programming language*. Pearson Education India.

[Suganuma et al., 2002] Suganuma, T., Yasue, T., and Nakatani, T. (2002). An empirical study of method inlining for a java just-in-time compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 91–104.

[Tempero et al., 2010] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The qualitas corpus : A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345. IEEE.

[Thalinger and Rose, 2010] Thalinger, C. and Rose, J. (2010). Optimizing invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 1–9, New York, NY, USA. ACM.

[Torgersen, 1998] Torgersen, M. (1998). Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, volume 544, pages 1–9.

[Touati et al., 2013] Touati, S.-A.-A., Worms, J., and Briais, S. (2013). The speedup-test : a statistical methodology for programme speedup analysis and computation. *Concurrency and computation : practice and experience*, 25(10) :1410–1426.

[Ungar and Smith, 1987a] Ungar, D. and Smith, R. B. (1987a). Self : The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA. ACM.

[Ungar and Smith, 1987b] Ungar, D. and Smith, R. B. (1987b). Self : The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA. ACM.

[Vitek et al., 1997] Vitek, J., Horspool, R. N., and Krall, A. (1997). Efficient type inclusion tests. *SIGPLAN Not.*, 32(10) :142–157.

[Wimmer and Brunthaler, 2013] Wimmer, C. and Brunthaler, S. (2013). Zippy on truffle : a fast and simple implementation of python. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications : software for humanity*, pages 17–18. ACM.

[Wimmer and Mössenböck, 2007] Wimmer, C. and Mössenböck, H. (2007). Automatic feedback-directed object inlining in the java hotspot^{TM} virtual machine. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 12–21. ACM.

[Wimmer and Würthinger, 2012] Wimmer, C. and Würthinger, T. (2012). Truffle : a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications : software for humanity*, pages 13–14. ACM.

[Würthinger et al., 2013] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., and Wolczko, M. (2013). One vm to rule them all. In *Symposium on New Ideas in Programming and Reflections on Software, Onward ! 2013, part of SPLASH '13*, pages 187–204. ACM.

[Würthinger et al., 2012] Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., and Wimmer, C. (2012). Self-optimizing ast interpreters. In *Proceedings of DLS*, pages 73–82. ACM.

[Zendra et al., 1997] Zendra, O., Colnet, D., and Collin, S. (1997). Efficient dynamic dispatch without virtual function tables : The smalleiffel compiler. *ACM SIGPLAN Notices*, 32(10) :125–141.

Résumé

Cette thèse traite des langages à objets en héritage multiple et typage statique exécutés avec des machines virtuelles. Des analogies sont à faire avec Java bien que ce langage ne soit pas en héritage multiple. Une machine virtuelle est un système d'exécution d'un programme qui se différencie des classiques compilateurs et interpréteurs par une caractéristique fondamentale : le chargement dynamique. Les classes sont alors découvertes au fil de l'exécution.

Le but de la thèse est d'étudier et de spécifier une machine virtuelle pour un langage à objets en héritage multiple, pour ensuite spécifier et implémenter des protocoles de compilation/recompilation. Ces derniers devront mettre en place les optimisations et les inévitables mécanismes de réparations associés. Nous présenterons d'abord l'architecture et les choix réalisés pour implémenter la machine virtuelle : ceux-ci utilisent le langage Nit en tant que langage source ainsi que le hachage parfait, une technique d'implémentation de l'héritage multiple. Ensuite nous présenterons les spécifications pour implémenter des protocoles de compilation/recompilation ainsi que les expérimentations associées. Dans ce cadre, nous avons présenté une extension des analyses de préexistence et de types concrets, pour augmenter les opportunités d'optimisations sans risque. Cette contribution dépasse la problématique de l'héritage multiple.

Mots clés : Machine virtuelle, langage à objet, héritage multiple, chargement dynamique, optimisations, inlining

---

This thesis is about object-oriented languages in multiple inheritance and static typing executed by virtual machines. We are in the context of a Java-like language and system but in multiple inheritance. A virtual machine is an execution system which is different from static compilers and interpreters since they are in dynamic loading. This characteristic makes classes to be discovered during the execution.

The thesis' goal is to study, specify and implement a virtual machine for an object-oriented language in multiple inheritance and then in a second step to specify and implement compilation/recompilation protocols. These protocols are in charge of optimizations and unavoidable repairing. We will present the architecture of the virtual machine : the used language is Nit, and perfect hashing as the multiple inheritance implementation technique. Then we will present the protocols and the experiments. In this thesis, we have presented an extension of preexistence and concrete types analysis to increase optimization opportunities. This contribution is not limited to multiple inheritance object-oriented languages.

Keywords : Virtual machine, object-oriented language, multiple inheritance, dynamic loading, optimizations, inlining