Université de Montréal

Compilation statique de Java

par

Étienne Bergeron

Département d'informatique et
de recherche opérationnelle

Faculté des arts et sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M. Sc.)
en informatique

Octobre 2002

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Compilation statique de Java.

présenté par :

Étienne Bergeron

a été évalué par un jury composé des personnes suivantes :

Jean Vaucher
(président-rapporteur)

Houari Sahraoui
(membre du jury)

Marc Feeley
(directeur de maîtrise)

Mémoire accepté le : 16 Decembre 2002

Sommaire

Mots-clés : Compilateur, statique, Java, Scheme.

JBCC est un compilateur natif statique que nous avons conçu pour un sous-ensemble presque complet de Java. J2S est une version de ce compilateur qui cible le langage Scheme. Nous présentons dans ce mémoire les mécanismes employés pour parvenir à compiler ce langage, les représentations intermédiaires ainsi que les analyses et optimisations effectuées.

Un des buts visés par notre travail est d'analyser les performances du langage lorsque compilé avec des analyses et optimisations globales. Les langages cibles Scheme, C et assembleur Intel ont été employés à divers stades du travail. Nous décrivons nos expériences de compilation avec les différentes versions du compilateur et nous analysons les performances obtenues.

Un autre but du travail est d'étudier l'implantation et la compilation de modules intégrés à la JVM ainsi que leurs coûts d'exécution. Par exemple, le système de threads de Java est crucial dans une application comme un serveur Web. Nous discutons des implantations de ces librairies critiques et de leurs impacts sur les performances des applications.

Nous concluons avec une discussion sur les modèles actuels et futurs des compilateurs.

Abstract

Keywords : Compilater, static, Java, Scheme.

JBCC is a static native compiler which we designed for an almost complete subset of Java. J2S is a version of this compiler which targets the Scheme language. We present in this thesis the mechanisms used to compile this language, the intermediate representations as well as the analyzes and optimizations carried out.

One of the goals of our work is to analyze the performances of the language when compiled with global analyzes and optimizations. The target languages Scheme, C and assembler INTEL were used at various stages of our work. We describe our experiments of compilation with the various versions of the compiler and we analyze their performances.

Another goal of our work is to study the implementations and the compilation of modules integrated to the JVM and their execution costs. For example, the threads system of Java is crucial in applications such as web servers. We discuss the implementation of these critical librairies and their impact on the applications performances.

We conclude with a discussion on current and future compiler models.

Table des matières

Remerciements **xv**

1 Introduction **1**

1.1 Objectifs 1

1.2 Propriétés recherchées 2

1.3 Motivation 2

1.4 Méthodologie 3

1.4.1 Déroulement du projet 3

1.4.2 Structure du document 3

1.5 Terminologie 4

2 Java **5**

2.1 Historique 5

2.2 Le langage 5

2.3 La machine virtuelle Java 8

2.3.1 Environnement d'exécution 9

2.3.2 Le format du fichier 14

2.4 Java et JVM 19

2.5 Compilation de Java 19

2.5.1 Compilateur dynamique 19

2.5.2 Compilation statique 21

2.5.3 Modèle de compilation 22

2.6 Conclusion 22

3 Compilateur de *bytecode* 24

3.1 Historique 24

3.2 Langage source : Java* 24

3.3 Processus de compilation 25

3.3.1 J2S : *Java to Scheme* 25

3.3.2 JBCC : *Java ByteCode Compiler* 26

3.4 Cheminement et représentation du code 27

3.4.1 Représentations intermédiaires 28

3.5 Langage cible 34

4 Le langage intermédiaire 36

4.1 JST : Java Syntax Tree 36

4.1.1 Syntaxe et sémantique 37

4.2 Importation du *bytecode* 37

4.2.1 Décodage des méthodes 38

4.2.2 Découpage en blocs de base 39

4.2.3 Importation des instructions et résolution de la pile d'exécution 40

4.2.4 Importation des variables locales et des exceptions 42

4.3 Analyses et optimisations 44

4.3.1 Élimination conservatrice de code mort 44

4.3.2 Analyse de type 47

4.3.3 Élimination précise du code mort 53
4.3.4 Analyse de variables vivantes 54
4.3.5 Transformation et substitution 55
4.3.6 Analyses de finalisation et de synchronisation 64
4.3.7 Améliorations futures 64
4.4 Analyse de performance 65
4.4.1 Élimination de code mort 65
4.4.2 Temps de compilation 66

5 Compilation vers Scheme **67**
5.1 Le langage 67
5.2 Gambit 69
5.3 Motivations 70
5.4 Génération du code Scheme 70
5.4.1 Mécanisme et fonctionnement 71
5.4.2 Vérification dynamique 76
5.5 Performance du compilateur 76
5.5.1 Autres compilateurs 77
5.5.2 Résultats 79
5.5.3 Analyse 83
5.6 Discussion 87

6 Compilation vers Intel x86 **88**
6.1 Structure du compilateur 88
6.2 Architecture 89
6.2.1 Sélection d'instructions 89

6.2.2 Allocation de registres 90
6.2.3 Optimisations 92
6.3 Librairies externes 94
6.3.1 Entrées/sorties 94
6.3.2 Système de threads 95
6.3.3 Ramasse-miettes 95
6.4 Performance 95
6.4.1 Résultats 95
6.4.2 Analyse 99
6.5 Discussion 101

7 Environnement d'exécution **103**
7.1 Système de threads 103
7.1.1 Processus lourds et processus légers 103
7.1.2 Modèle de mémoire 105
7.1.3 Implantation 106
7.1.4 Système de threads spécifique pour Java 106
7.2 Ramasse-miettes 107
7.3 Entrées/sorties 108
7.4 Conclusion 108

8 Conclusion **109**
8.1 Domaines d'application 109
8.2 Contributions 110
8.3 Analyses globales 110
8.4 Environnement d'exécution 111

8.5 Modèles de compilation 111

8.5.1 Compilation statique 112

8.5.2 Compilation dynamique 112

8.5.3 Modèle hybride 112

8.5.4 Compilation par couches 113

8.5.5 Futurs modèles 114

Bibliographie **118**

A Définition du langage JST **xvi**

A.1 Grammaire du langage JST xvi

A.2 Système de type du langage JST xviii

B Règles d'importation du *bytecode* vers JST **xix**

C Analyse de type pour domaines numériques entiers **xxv**

Table des figures

2.1 Exemple de la décompilation du corps d'une méthode 12

2.2 Structure ClassFile 15

2.3 Structure attribute_info 16

2.4 Structure field_info 16

2.5 Structure method_info 16

2.6 Structure Code_attribute 17

2.7 Exemple de la décompilation du corps d'une méthode 18

2.8 Chargement de classe rendant une optimisation incorrecte 20

3.1 Processus de compilation de J2S 25

3.2 Exemple de compilation avec J2S et ses compilateurs 26

3.3 Processus de compilation de JBCC 26

3.4 Exemple de compilation avec JBCC et ses compilateurs 27

3.5 Cheminement et représentation du code 28

3.6 Exemple de représentation JST 29

3.7 Exemple de représentation NAT 31

3.8 Exemple de représentation JASM 32

3.9 Exemple de représentation Assembleur Intel 34

4.1 Code source Java* 38

4.2 Bytecode Java décodé 38

4.3 *Bytecode* Java* découpé 40

4.4 Importation par interprétation abstraite et résolution de la pile d'exécution . . . 41

4.5 Langage JST obtenu après l'interprétation abstraite 41

4.6 Importation par interprétation abstraite et résolution de la pile d'exécution . . . 42

4.7 Langage JST obtenu après l'importation 43

4.8 Processus d'analyse et d'optimisation 44

4.9 Algorithme conservateur d'élimination du code mort 46

4.10 Analyse de type par l'algorithme du point fixe 48

4.11 Treillis par hiérarchie de classe 49

4.12 Treillis de sous-ensembles de classes 50

4.13 Code limitant la convergence d'un domaine numérique 51

4.14 Code limitant l'analyse par système de contraintes 52

4.15 Limite de la 0-CFA 53

4.16 Modifications à l'algorithme d'élimination du code mort 54

4.17 Code d'exemple utilisé pour les substitutions 55

4.18 Code après substitution des constantes 56

4.19 Code après propagation des constantes et des variables 57

4.20 Code après l'évaluation d'expressions constantes 58

4.21 Code après la transformation en forme SSA 59

4.22 Substitution d'une expression équivalente 59

4.23 Code après l'élimination d'expressions communes 60

4.24 Substitution d'un groupe d'expressions 61

4.25 Code après l'élimination du code mort 62

4.26 Substitutions de branchement 63

5.1 Exemple de calculs en Scheme 68

5.2 Fonction d'ordre supérieur 68

5.3 Programmation orienté-objet en Scheme 69

5.4 Représentation des instructions élémentaires en Scheme 71

5.5 Représentation des blocs de base en Scheme 72

5.6 Représentation des variables locales 72

5.7 Problème lié aux variables d'exceptions 73

5.8 Fermeture englobant les blocs de base et les gestionnaires d'exceptions 73

5.9 Représentation des variables globales en Scheme 73

5.10 Représentation des méthodes statiques et virtuelles en Scheme 74

5.11 Représentation des Tables des méthodes virtuelles et des types d'instances en Scheme 74

5.12 Représentation des Objet Java* en Scheme 74

5.13 Addition sous un domaine borné en *fixnum* 76

5.14 Vérification dynamique de domaine non borné en *fixnum* 76

5.15 Calcul numérique borné en *fixnum* 83

5.16 Calcul numérique non-borné en *fixnum* 83

5.17 Exemple de programme test 84

6.1 Registres et états de la pile 8087 92

6.2 Allocation des registres flottants 92

6.3 Exemples de substitutions localisées 93

6.4 Substitution d'instructions avec registres flottants 93

7.1 Système de threads à processus lourds 104

7.2 Système de threads à processus légers 104

7.3 Système de threads mixte 105

Liste des tableaux

2.1 Types de base supportés par la JVM 9

4.3 Résultat de l'élimination de code mort 65

4.4 Temps de compilation 66

5.1 jBYTEmark 79

5.2 Fibonacci (Threads) 81

5.3 Recherche de fichier (69 fichiers, 145 threads) 82

5.4 Comparaison des performances des *fixnums* et *bignums*. 85

5.5 Comparaison de performances des tests dynamiques. 86

6.1 jBYTEmark 96

6.2 Fibonacci 97

6.3 N-Queens 97

6.4 Recherche de fichier (69 fichiers, 145 threads) 98

6.5 Dimensions des exécutables 100

Remerciements

Je veux remercier mes parents, Denyse et Jean, qui ont toujours été présents et attentifs, non seulement au cours de ma maîtrise, mais aussi tout au long du périple que furent mes études et ma vie.

Je veux remercier mes amis Johanne et Claude qui m'ont fourni non seulement un support financier mais aussi un support moral. Ce support fait partie des fondations de ma motivation à poursuivre mes études.

Je veux également remercier mon directeur de recherche, Marc Feeley, qui, par sa maîtrise de son domaine et par son sens de la perfection, a su m'enterrer sous une pile de livres et d'articles m'ouvrant les portes de la connaissance et de la maîtrise de mon domaine.

Je remercie Annie, la femme de ma vie, pour avoir accepté les longues journées et soirées de solitude passées loin d'elle lorsque je croulais sous une pile de livres et d'articles. Je remercie Dieu qu'Annie soit étudiante et comprenne les sautes d'humeur qu'occassionne la vie d'étudiant. Je souhaite pour Annie les mêmes succès que les miens et je lui offre le même support.

Enfin, je désire remercier mes amis et collègues. Merci à Danny Dubé qui a su confronter ses idées aux miennes. Merci à mes collègues Éric Lesage, Jean-François Gagné et mes amis d'adolescence Chloé et Greg, lesquels m'ont permis, par moment, de m'évader des lieux d'étude pour de lointains lieux isolés où nous pouvions relaxer et discuter tranquillement... d'informatique. Merci à tous mes partenaires d'escalade, de plongée et de boisson[1].

Je voudrais particulièrement remercier ceux qui lisent les remerciements.

[1] Amaretto, Bailey's, bière, vin...

Chapitre 1

Introduction

La complexité grandissante des langages de programmation et le nombre élevé d'analyses et d'optimisations requises pour obtenir de bonnes performances rendent l'implantation complète d'un compilateur pour un langage de haut niveau trop laborieuse pour être réalisable en peu de temps.

Notre recherche consiste à étudier un modèle de compilateur qui repose sa compilation sur l'emploi d'un autre langage et de ses compilateurs pour parvenir à produire le programme exécutable final.

1.1 Objectifs

Le but principal de notre projet est de vérifier la faisabilité d'un compilateur Java via le modèle de couche de compilateurs en ciblant le langage Scheme [CR91] et spécifiquement le compilateur Gambit [Fee98] pour Scheme.

Un but secondaire de notre recherche est de comparer les avantages et inconvénients que présentent les compilateurs statiques et dynamiques par les analyses, optimisations et performances.

Un autre but, qui s'est ajouté pendant la réalisation du projet, est l'étude des différentes implantations des modules fondamentaux du langage pour en comprendre l'impact sur les techniques de compilation et les performances des applications.

1.2 Propriétés recherchées

Nous avons choisi d'appliquer un modèle de compilation différent de celui des compilateurs Java populaires pour obtenir des propriétés intéressantes. Les propriétés recherchées par notre compilateurs sont :

- précision élevée de l'information nécessaire aux analyses,
- analyses et optimisations globales,
- vitesse d'exécution élevée de l'application générée,
- indépendance de l'application face à l'environnement d'exécution.

Pour obtenir ces propriétés, notre compilateur doit imposer des restrictions sur le langage. Les restrictions imposées ne causent aucun problème pour la majorité des applications Java.

Pour obtenir un niveau de précision élevé, notre compilateur doit prévoir des mécanismes de renseignement et doit effectuer des analyses globales. Ces analyses globales sont coûteuses en temps de compilation. Notre compilateur vise une compilation longue et précise pour obtenir des analyses et des optimisions précises. Il espère ainsi obtenir des gains sur la vitesse d'exécution.

1.3 Motivation

L'évolution rapide du matériel et des langages de programmation force les concepteurs de compilateurs à prévoir un modèle de compilateur qui peut s'adapter rapidement aux changements sans engendrer de coût trop élevé sur les performances. Vu la taille grandissante des compilateurs et leurs difficultés de maintenance, le modèle de compilateur recherché doit être en mesure de permettre son développement en parallèle. Le modèle de compilateur étudié possède les propriétés lui permettant de facilement s'adapter et de bien diviser formellement les entités de traitement permettant un développement concurrent. Nous croyons que l'emploi de ce modèle facilite le développement d'un compilateur sans engendrer une grosse perte de performance sur les applications générées.

Le langage Java, qui possède de nombreuses propriétés dynamiques, ne se prête pas facilement aux analyses statiques conventionnelles. Ces propriétés dynamiques offrent aux programmeurs des techniques de développement objet plus évoluées mais engendrent des coûts de compilation non négligeables. Cependant, beaucoup d'applications nécessitant de bonnes performances n'emploient pas ces propriétés dynamiques ; ce qui nous motive à prendre pour acquis leur absence pour permettre une compilation avec des analyses moins conservatrices et possiblement de meilleures optimisations.

1.4 Méthodologie

1.4.1 Déroulement du projet

Dans le cadre de notre recherche, nous avons réalisé deux versions de compilateur employant chacun un modèle différent : le premier sur le modèle de compilateur ciblant un langage de haut niveau et le second sur un modèle plus conventionnel. À l'aide de ces deux versions de compilateur, nous sommes en mesure de comparer les avantages et inconvénients des différents modèles.

Par la suite, nous avons comparé les performances de nos compilateurs avec celles obtenues par certains compilateurs dynamiques existants. Ces performances étant fortement influencées par les différents modules fournis par le langage, nous avons étudié les différentes implantations de ces modules pour faire ressortir les choix d'implantation qui doivent être faits lors de la conception d'un compilateur Java selon les propriétés recherchées.

1.4.2 Structure du document

Afin de justifier les choix d'implantation de notre compilateur, le chapitre 2 présente l'historique, les propriétés et l'environnement d'exécution du langage Java ainsi que les techniques de compilation présentement employées pour le compiler.

Le chapitre 3 décrit les différentes versions de notre compilateur et leurs propriétés. Un survol du processus de compilation y est effectué pour permettre une vue d'ensemble des différentes étapes de compilation et leurs langages intermédiaires.

Le langage JST, une représentation intermédiaire employée par notre compilateur, ainsi que les analyses et optimisations appliquées sur ce langage sont décrits dans le chapitre 4.

Les parties dorsales (*back-end*) générant les langages ciblés par le compilateur sont décrites dans les chapitres 5 et 6 pour le langage Scheme et l'assembleur Intel x86 respectivement. Les techniques de génération de code et les problèmes rencontrés y sont décrits ainsi que les performances résultant de chaque partie dorsale.

Le chapitre 7 décrit l'impact de l'environnement d'exécution fourni par le langage sur les performances des applications. Cette analyse permet de comprendre les choix d'implantation à prévoir lors de la conception d'un compilateur Java.

Nous concluons avec une discussion sur les modèles présents et futurs des compilateurs.

1.5 Terminologie

Dans cette section, nous définissons certains termes employés dans ce document pour bien décrire le concept visé lors de leur emploi et éviter toute ambiguïté.

Nous définissons *architecture* par les attributs matériels d'un système visibles par le programmeur et qui ont un impact direct sur l'exécution logique d'un programme.

Nous définissons *système d'exploitation* par l'application gestionnaire d'un ordinateur. Le système d'exploitation offre des services aux applications et effectue l'interface entre une application et le matériel. Dans ce document, ce concept inclut le noyau du système d'exploitation ainsi que toutes les librairies et applications standards fournies par ce système d'exploitation.

Nous définissons *environnement d'exécution* par l'environnement virtuel gérant l'exécution d'applications. Cet environnement fournit des services garantis aux applications. De plus, il contrôle l'exécution de l'application en la limitant aux actions accordées par l'utilisateur.

Nous définissons *fonction native* par une fonction définie dans le langage mais qui n'a pas été implantée dans le langage. Ces fonctions sont généralement fournies dans un autre langage via un interface natif.

Nous définissons *compilateur statique* par un compilateur qui effectue entièrement son traitement avant toute exécution.

Nous définissons *compilateur dynamique* par un compilateur qui effectue complètement ou partiellement son traitement pendant l'exécution de l'application.

Nous définissons *compilateur natif* par un compilateur dont le langage de sortie est l'assembleur ou code machine de l'architecture cible.

Chapitre 2

Java

2.1 Historique

Les origines de Java remontent à 1990 alors que le *World Wide Web* n'était qu'à un état embryonnaire. Naughton, Gosling et Sheridan, programmeurs chez Sun Microsystems, définirent quelques principes de base pour un nouveau projet. Leur objectif était de construire un environnement de petite taille pour une nouvelle génération de systèmes embarqués comme les machines à café, les laveuses et les magnétoscopes.

Gosling, déçu par le C++, conçoit un langage robuste, industriel et orienté objet nommé OAK. En janvier 95, OAK étant déjà une marque déposée, le projet est rebaptisé Java. Sun a officiellement présenté Java le 23 mai 1995. Java a beaucoup fait parler de lui d'abord par sa relation avec l'Internet puis par ses qualités intrinsèques qui en font un langage général.

2.2 Le langage

Sun s'est engagé à maintenir le langage et les outils Java de base dans le domaine public. Java est maintenant un langage largement employé dans l'industrie vu les propriétés intéressantes qu'il offre au programmeur et à l'utilisateur.

Sun en dit : *"Java est un langage simple, orienté objet. Java n'est pas déroutant pour le programmeur C++ mais en évite les écueils. Java est robuste, sécurisé. Il est indépendant de l'architecture, portable, il est performant, distribué, multi-threadé et dynamique".* [Gos95]

Java offre au programmeur un environnement de développement simple et complet pour la réalisation de projets de petite et de grande envergure. De plus, il offre à l'utilisateur d'une application Java des garanties de sécurité intrinsèques au langage.

Gosling discute des propriétés recherchées, que nous décrivons ci-dessous, lors de la conception du langage Java dans [Gos95].

Simple et stable

Java est un dérivé simplifié de C et C++ et ajoute des concepts modernes de langages plus récents comme Objective-C. Gosling le décrit comme *"C++ sans les couteaux ni les revolvers"* [Gos95]. Les éléments complexes de ce dernier, souvent mal utilisés et sources de nombreux bogues, ont été retirés pour obtenir un langage plus simple. Ainsi, les pointeurs, la surcharge d'opérateurs et l'héritage multiple ne sont plus des éléments ou constructions accessibles aux programmeurs Java. Pour faciliter la tâche du programmeur, un ramasse-miettes s'occupe de la gestion de la mémoire libérant ainsi en grande partie les programmes Java des fuites de mémoire et pointeurs fous dues au problème complexe de l'allocation et de la libération de la mémoire.

Orienté objet

Les principes orientés objets facilitent la structuration des programmes en modules, ce qui permet une abstraction plus naturelle pour les programmeurs. De plus, ils permettent une plus grande réutilisation de code. Java adopte les principes fondamentaux des langages objets : l'abstraction, l'encapsulation et l'héritage [Mey97].

Java n'utilise pas l'héritage multiple mais offre le mécanisme d'interfaces qui peut être utilisé pour implémenter de nombreuses fonctionnalités normalement mises en place grâce à l'héritage multiple.

Distribué

Les appels aux fonctions d'accès réseau (sockets) et les protocoles Internet les plus utilisés tels HTTP, FTP et Telnet sont intégrés dans les librairies Java. Il est simple d'écrire un programme avec une architecture client-serveur en Java.

Robuste

Le langage Java tire sa robustesse de ses contrôles de type stricts tant au moment de la compilation qu'à l'exécution. Son modèle interne de pointeur, non accessible par le programmeur, évite des erreurs introduites par la mauvaise manipulation de ceux-ci. Le programmeur ne peut que manipuler des références vers les objets. Des vérifications dynamiques sont faites pour vérifier que les données correspondent aux types spécifiés dans le programme. Il n'est pas possible de commettre des erreurs causées par des abus de langage comme les programmeurs C ont coutume de faire.

Sûr et sécurisé

De par son architecture, il est possible de télécharger une application Java distante pour ensuite l'exécuter localement. Le langage met l'emphase sur la sécurité afin d'éviter le risque d'infiltration ou de contamination par un virus. Il assure à l'utilisateur qu'une application n'a pas été altérée depuis sa compilation. Un contrôle serré sur l'exécution est fait pour éviter qu'un programme mal intentionné cause des dommages au système ou aux données présentes sur celui-ci. Un mécanisme de contrôle complet a été incorporé afin de vérifier la légitimité du code.

Portable

La diversité matérielle et logicielle a amené les concepteurs du langage à prévoir une architecture indépendante et facilement portable. Sur l'Internet, une multitude de systèmes hétérogènes cohabitent. Java se doit d'être indépendant des couches qu'il ne contrôle pas. C'est pourquoi le compilateur Java ne génère pas d'instructions spécifiques à une machine physique mais un *bytecode* spécifique aux JVM, soit un langage pour un processeur virtuel d'une JVM. Ce *bytecode* peut alors être exécuté par l'interprète d'une JVM.

Haute performance

L'expérience nous montre que les langages interprétés sont de l'ordre de dix à cent fois plus lents que les programmes compilés. L'avantage du *bytecode* est que les premières étapes de compilation, soit l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique et l'analyse de type ont déjà été effectuées par un compilateur. La machine virtuelle qui interprète le code possède *a priori* l'information nécessaire pour exécuter le code sans devoir l'analyser à nouveau. Des vérifications simples sont effectuées lors du chargement du programme. Elles ne seront pas

effectuées pendant l'interprétation accélérant ainsi l'exécution. Le format simple des instructions permet un décodage rapide comparativement aux interprètes classiques.

De plus, une nouvelle vague de machines virtuelles Java accélère le temps d'exécution pour se rapprocher du temps d'exécution de C. Ces machines virtuelles possèdent un compilateur dynamique générant du code machine pour l'architecture d'exécution pendant l'exécution du programme.

Multi-threadé

Le langage supporte directement les threads (ou processus légers) allégeant l'écriture de programmes concurrents. Il contient un jeu de primitives de synchronisation basé sur les travaux de Hoare [Hoa74]. Dans les programmes multi-threadés, plusieurs processus peuvent s'exécuter simultanément.

Réflexif

La réflexivité permet d'obtenir des informations dynamiques sur la nature des objets. Il est possible de connaître les classes chargées et d'obtenir des renseignements sur celles-ci. Il en va de même pour les méthodes et les champs d'une classe.

Dynamique

Java permet le chargement dynamique de classes. Il est possible de charger dynamiquement une librairie. Il n'est pas nécessaire de modifier le programme si une de ses librairies change.

2.3 La machine virtuelle Java

Pour réaliser les propriétés garanties par le langage, les développeurs ont dû prévoir une architecture d'exécution intégrant les mécanismes requis. Cette architecture d'exécution, appelée JVM (*Java Virtual Machine*) [LY99a], garantit les propriétés du langage Java [GJJB00] et offre un environnement portable, dynamique et sécurisé.

Dans cette section, nous expliquons les concepts nécessaires pour comprendre les services fournis par la JVM ainsi que le format des fichiers.

La spécification de la JVM par Sun est donnée dans [LY99a]. Cette spécification Sun man-

quant de formalisme, les travaux de Stärk, Schmid et Börger ont commencé à formaliser et valider l'architecture de la JVM [BS99, SSB]. Le but de leurs travaux étant d'éviter des incohérences comme mentionné dans [Sar97].

2.3.1 Environnement d'exécution

Afin d'être portable, Java a initialement employé le concept de machine virtuelle. Ainsi, la spécification de l'environnement d'exécution est la même pour toutes les applications peu importe l'architecture. Cette section présente les aspects importants de cet environnement virtuel.

Types de données

Les types de base supportés par la JVM reflètent les types de base supportés par le langage Java. Pour chaque type de base du langage Java, il existe un type identique dans la JVM. La Table 2.1 donne les types supportés. Les types ne dépendent pas de l'implantation et du matériel car la spécification de Java indique précisement quelles sont les dimensions et le format des types de base. Un *int* est toujours un entier signé de 32 bits en représentation complément à deux. Les formats numériques respectent la norme IEEE 754 et les caractères respectent la norme Unicode.

Type	Dimension	Description
byte	8 bits	Entier signé en complément à deux
short	16 bits	Entier signé en complément à deux
char	16 bits	Entier non-signé
int	32 bits	Entier signé en complément à deux
long	64 bits	Entier signé en complément à deux
float	32 bits	Flottant IEEE 754 simple précision
double	64 bits	Flottant IEEE 754 double précision
reference	32 bits	Référence à un objet Java
returnAddress	32 bits	Adresse de retour pour les sous-routines.

TAB. 2.1 – Types de base supportés par la JVM

Les tableaux Java sont traités comme des objets, c'est-à-dire qu'ils sont accédés au moyen d'une référence. La valeur spéciale *null* indique que la référence ne pointe vers aucun objet. Il est à noter que le type *returnAddress* n'est pas un type du langage Java. Ce type est employé par les instructions de sous-routines nécessaires pour la compilation des clauses *finally* employées pour le traitement d'exceptions dans le langage.

Interprète

L'interprétation de code ralentit considérablement l'exécution de celui-ci par rapport à une exécution de code machine réel. Les concepteurs de la JVM ont choisi de compiler Java vers un *bytecode* pouvant être analysé rapidement par la JVM améliorant ainsi sa vitesse d'exécution. L'interprétation du *bytecode* offre à Java de meilleures performances que d'autres langages interprétés comme Perl et TCL.

Le *bytecode* a été conçu pour être vérifiable. Certaines propriétés sont vérifiables statiquement et d'autres dynamiquement. Puisque le code est vérifié avant l'exécution, l'interprète n'a pas à vérifier le code pendant son exécution, accélérant ainsi son travail. Il reste cependant certains tests dynamiques, tels que la vérification des bornes lors d'accès au contenu d'un tableau et la déréférence de *null*.

L'interprète exécute le *bytecode* sur un processeur virtuel possèdant un jeu d'instructions fixe, une pile d'exécution et des variables locales.

Processeur virtuel

Le *bytecode* est exécuté sur un processeur virtuel à pile qui ne possède aucun registre. Les propriétés du processeur sont indépendantes de l'architecture. Cette abstraction augmente la portabilité de la JVM et du langage Java. Les divers implanteurs de la machine virtuelle n'ont pas interprété les normes de Sun de la même façon, menant à certaines incompatibilités. Les travaux de Peter Berteksen [Ber97] essaient de définir formellement le fonctionnement des instructions.

Pile d'exécution

La pile d'exécution contient des *cases* de 32 bits. Elles servent à passer les paramètres dynamiques aux instructions. Les types de 32 bits occupent une case et les types 64 bits occupent deux cases. Il n'est pas possible d'accéder à la moitié d'une valeur de 64 bits. La disposition des deux cases pour les types 64 bits est donc laissée libre à l'implantation de la JVM.

Lorsqu'un type plus petit que 32 bits est ajouté sur la pile d'exécution, sa valeur est étendue vers un entier signé de 32 bits et devient de type *int*. Lorsqu'un type plus grand que sa destination est retiré de la pile, sa valeur est tronquée. Les seuls types possibles sur la pile d'exécution sont *int, long, float, double, reference et returnAddress.*

Chaque appel de méthode engendre la création d'une nouvelle pile d'exécution locale. Cette pile d'exécution est de taille fixe et connue statiquement. La pile n'est généralement pas profonde.

Elle excède rarement la profondeur de 5 cases et dans nos tests moins de 10% des méthodes possèdent des piles d'exécution supérieures à 10 cases, et nous n'avons jamais observé une méthode excédant 12 cases.

Variables locales

Un appel de méthode engendre la création d'un bloc d'activation où sont conservées les variables locales. Tout comme pour la pile d'exécution, les cases des variables locales sont de 32 bits et les types de 64 bits occupent deux cases adjacentes. Il n'y a pas de type associé à une case. Une même case peut contenir, lors de l'exécution d'une méthode, deux types différents. Par contre, le vérificateur accepte le code seulement s'il peut prouver par ses analyses qu'il n'y aura pas d'accès illégaux aux variables locales. Par exemple, il est impossible d'écrire un entier dans une variable locale et de s'en servir ensuite comme une variable de type *returnAddress*, ce qui entraînerait des problèmes de sécurité.

Le jeu d'instructions

Il existe 202 instructions dans le jeu d'instructions de la JVM. Les instructions possèdent un format d'encodage fixe et facilement décodable. L'unité de lecture pour décoder un fragment de code est l'octet (*byte*).

Le premier octet représente l'opérateur de l'instruction. Une instruction possède des paramètres statiques connus lors de la compilation et des paramètres dynamiques connus lors de l'exécution. L'opérateur, le premier octet lu, permet de déduire la taille et le nombre des paramètres statiques. Les paramètres statiques se trouvent encodés dans les octets qui suivent l'opérateur. Les paramètres dynamiques sont passés à l'instruction au moyen de la pile d'exécution. Chaque instruction comporte un nombre de paramètres dynamiques fixes. Ces paramètres sont retirés de la pile d'exécution et le résultat produit par l'instruction est remis sur la pile.

La plupart des instructions de la JVM sont typées. Par exemple, `iadd` effectue l'addition entière des deux derniers paramètres dynamiques sur la pile, c'est-à-dire que les deux valeurs au-dessus de la pile sont remplacées par leur somme.

Voici les catégories d'instructions de la JVM :

- Chargement de constantes sur la pile
- Accès aux variables locales
- Création de tableaux
- Accès aux tableaux
- Création d'objets
- Accès aux méthodes et champs des objets
- Gestion de la pile
- Opérations arithmétiques
- Conversion de type
- Comparaisons
- Branchement (conditionnel et inconditionnel)
- Branchement indirect via une table
- Retour de méthode et sous-routine
- Élargissement des index

La Figure 2.1 représente le corps d'une méthode avec ses attributs. L'attribut *code* représente le code à exécuter et a été décompilé afin de montrer un exemple employant les instructions de la JVM.

```
Code source :
     public static int carre(int x) {
       return x*x;
     }

Corps de methode :
     access_flags :9
     name_index :5  (carre)
     descriptor_index :6  ((I)I)
     Atributs :
        (Code)   7 :00 02 00 01 00 00 00 04
                    1A 1A 68 AC 00 00 00 01
                    00 08 00 00 00 06 00 01
                    00 00 00 05

Attribut code :
     max_stack :2
     max_locals :1
     Atributs :
        Attribut 8 :00 01 00 00 00 05

       CODE :
           000000 : ILOAD              0000
           000001 : ILOAD              0000
           000002 : IMUL
           000003 : IRETURN
```

FIG. 2.1 – Exemple de la décompilation du corps d'une méthode

Chargeur de classe

Le chargement dynamique est une propriété prévue pour faciliter la mobilité du code. Lors d'un chargement de classe, la JVM élargit son système de types et son ensemble de codes. Le chargement dynamique est permis car l'édition de liens se fait avec des références symboliques.

Le chargeur de classe a la responsabilité de trouver les classes et de les charger. Il doit vérifier

la validité du fichier. Le fichier doit être conforme au standard. Ainsi, il doit commencer par un *nombre magique* indiquant le type du fichier, la table des constantes doit contenir tous les éléments référés et nécessaires et, il ne doit contenir aucune donnée supplémentaire. Un fichier ne respectant pas les conditions de chargement est refusé par les mécanismes de sécurité de la JVM.

Vérificateur de classe

Les propriétés de sécurité fournies par Java ne sont pas dépendantes du système d'exploitation, mais du langage et de la JVM. Pour éviter tout code malicieux, le vérificateur de classe doit s'assurer que la classe chargée n'enfreint aucune de ces propriétés. Par exemple, toutes les classes doivent avoir une classe parent, (exceptée pour la classe `java.lang.Object`), et ce parent doit être défini et accessible. La classe chargée ne peut pas surcharger une méthode déclarée avec l'attribut *final* dans une classe ancestrale. Une vérification doit être faite parallèlement à l'édition de liens pour vérifier les propriétés d'accès aux méthodes et aux champs inter-classes.

Vérificateur de *bytecode*

Les instructions de la JVM sont fortement typées, ce qui permet à l'interprète de ne pas vérifier dynamiquement les types. Le vérificateur de *bytecode* s'assure, avant la première exécution de chaque méthode, que son *bytecode* respecte les contraintes de manipulation des types. Cette vérification implique une analyse de flux de données et une analyse de flux de contrôle. Par exemple, une analyse est effectuée pour vérifier la profondeur maximale de la pile, déclarée statiquement dans le fichier, afin de s'assurer que cette limite est respectée et éviter tout problème d'exécution relié à une pile erronée. Les accès aux variables locales sont aussi contrôlés pour s'assurer qu'il n'y a pas de tentatives d'accès à des renseignements contenus dans une case non initialisée. Par exemple, une ancienne pile d'exécution peut laisser des références possiblement privées dans les cases d'une nouvelle pile d'exécution. Toutes les adresses de branchement sont vérifiées pour s'assurer qu'elles pointent vers du code valide et toutes les contraintes de types sont vérifiées statiquement.

Gestionnaire de mémoire

Pour allouer un objet, l'interprète doit communiquer avec le chargeur de classe de la JVM afin que les vérifications nécessaires soient effectuées. Le langage permet le chargement dynamique de classes et lors de la création d'un objet le gestionnaire de mémoire doit s'assurer que la classe est chargée et initialisée.

Il n'est pas possible de demander explicitement de libérer la mémoire qu'occupe un objet. Tous les objets sont détruits par un ramasse-miettes. Le programmeur peut explicitement appeler le ramasse-miettes mais aucune garantie de libération ou même d'exécution du ramasse-miette n'est promise. Les objets alloués par la JVM pourront être manipulés uniquement avec le type *reference*.

Gestionnaire de sécurité

Un gestionnaire de sécurité est inclus dans la JVM pour garantir un environnement d'exécution sécuritaire. Ce gestionnaire permet un environnement qui empêche une mauvaise exécution d'un programme, malicieuse ou non. La sécurité est une propriété importante du langage, et pour l'obtenir, la JVM effectue des vérifications dynamiques de sécurité.

Le gestionnaire de sécurité peut varier pour chaque instance de machine virtuelle modifiant les contraintes selon les besoins de sécurité. Il varie selon le type d'application et est créé par l'application qui instancie la JVM.

Le gestionnaire de sécurité effectue des vérifications dynamiques afin de limiter les actions d'une application à celles qui lui sont permises. Ainsi, une application web qui n'a pas été authentifiée par le gestionnaire de sécurité peut se voir refuser l'accès aux fichiers locaux.

Le système de threads

L'environnement fourni aux applications Java permet de créer des threads et d'exécuter des tâches en parallèle. Des primitives, similaires à celles des moniteurs [Hoa74], permettent de synchroniser les tâches. Ces mécanismes sont indépendants de l'architecture de la machine hôte. Il existe plusieurs implantations de ce système offrant des propriétés différentes que nous verrons dans la Section 7.1.

2.3.2 Le format du fichier

Les concepteurs du format de fichier du *bytecode* l'ont prévu pour que le chargement de classes soit simple et rapide et pour permettre la possibilité d'analyse et de décodage partiel. Cette section décrit les notions essentielles pour comprendre le format du fichier supporté par les JVM. La spécification formelle est décrite dans [LY99a].

Chaque fichier contient la définition d'une classe ou d'une interface. Comme le nom l'indique, les fichiers de *bytecode* sont une suite d'octets qui sont découpés en structures. L'octet étant la

base de traitement, les structures emploient des types basés sur l'octet. Nous y retrouvons les types `u1`, `u2` et `u4` représentant respectivement un, deux et quatre octets qui forment un entier non signé.

ClassFile

La structure *ClassFile* contient toutes les informations sur la classe décrite dans le fichier. En outre, elle indique ses attributs, son parent, ses interfaces, ses méthodes et ses champs. Une structure *ClassFile* contient une table de constantes utilisée par les structures ou les instructions requérant des paramètres statiques. L'accès à toutes les constantes utilisées dans le fichier se fait par l'entremise d'un index faisant rérérence à une entrée dans la table des constantes.

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

FIG. 2.2 – Structure ClassFile

Le champ *magic* contient la valeur `0xCAFEBABE` identifiant le fichier. Les champs *minor_version* et *major_version* donne l'information sur la version du *bytecode*. Le tableau *constant_pool* est une table de constantes de *constant_pool_count* constantes. La table des constantes permet la résolution des informations à partir d'un index. Le champ *access_flags* donne les propriétés d'accès de la classe contenue dans le fichier. Les champs *this_class* et *super_class* sont des index de la table des constantes qui permettent de déterminer la classe présente et son parent. Le tableau *interfaces* contient des index de la table de contantes permettant de déterminer les interfaces de la classe. Les tableaux *fields* et *methods* contiennent des structures avec les informations sur les champs et les méthodes de la classe. Les informations de ces structures sont décrites ci-dessous.

attribute_info

Les attributs permettent de définir une propriété et de donner une valeur à cette propriété.

```
attribute_info {
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_length];
}
```

FIG. 2.3 – Structure attribute_info

Un attribut possède un nom, une longueur et une série d'octets de cette longueur. Le nom de l'attribut sert à déterminer l'utilité du contenu de l'attribut et se trouve via la table des constantes. Certains attributs, comme l'attribut *code* d'une méthode, sont requis. Par contre, d'autres attributs, comme les attributs *LineNumberTable*, *SourceFile* et *LocalVariableTable*, sont optionnels et standardisés. Il est possible d'ajouter d'autres attributs pour des besoins spécifiques de l'utilisateur. Ces attributs sont ignorés par les JVM ne les reconnaissant pas.

field_info

La structure définissant un champ possède les renseignements sur le champ et un tableau d'attributs.

```
field_info {
 u2 access_flags;
 u2 name_index;
 u2 descriptor_index;
 u2 attributes_count;
 attribute_info attributes[attributes_count];
}
```

FIG. 2.4 – Structure field_info

Le champ *access_flags* donne les droits d'accès à ce champ. Le champ *name_index* contient un index qui permet de déterminer le nom du champ à l'aide de la table des constantes et le champ *descripteur_index* permet de déterminer le type du champ. Le tableaux d'attributs défini les attributs supplémentaires du champs. Aucun attribut n'est requis.

method_info

Chaque méthode est à son tour une structure contenant des renseignements tels que le nom et le descripteur de type.

```
method_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

FIG. 2.5 – Structure method_info

Le champ *access_flags* donne les droits d'accès de la méthode. Les champs *name_index* et *descriptor_index* sont les index permettant de résoudre le nom et la signature de la méthode.

La structure contient un tableau d'attributs qui doit absolument contenir l'attribut *code* pour être valide, excepté pour les méthodes natives ou abstraites. Cet attribut permet d'obtenir l'information pour exécuter le corps de la méthode. D'autres attributs peuvent être présents comme les renseignements laissés par un compilateur pour le débogage de l'application ou des références vers le code source utilisées lors du traitement d'exceptions.

Nous pouvons observer qu'il n'est pas nécessaire d'analyser le corps des méthodes pour charger le fichier car l'analyse et le décodage de l'attribut *code* peuvent se faire lors de sa première utilisation.

Code_attribute

L'attribut *code* contenu dans les attributs de méthode contient les informations sur le code de la méthode. Nous y retrouvons les renseignements utiles pour l'exécution du corps de la méthode : la profondeur maximale de la pile et le nombre de variables locales utilisées pour créer le bloc d'activation de la méthode. La table d'exceptions permet de déterminer les routines de traitement lorsqu'une exception est levée.

```
Code_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 max_stack;
  u2 max_locals;
  u4 code_length;
  u1 code[code_length];
  u2 exception_table_length;
  { u2 start_pc;
    u2 end_pc;
    u2 handler_pc;
    u2 catch_type;
  } exception_table[exception_table_length];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
 }
```

FIG. 2.6 – Structure Code_attribute

La Figure 2.7 montre l'attribut *code* décodé de la méthode `java.lang.String.charAt(I)`.

```
    public char charAt(int index) {
        if ((index < 0) || (index >= count)) {
            throw new StringIndexOutOfBoundsException(index);
        }
        return value[index + offset];
    }


max_stack :3
max_locals :2
Atributs :
Attribut 131 :00 03 00 00 00 D5 00 0C
00 D6 00 15 00 D8

Attribut code :
Offset  Instruction       Static Parameters
000000 : ILOAD             0001
000001 : IFLT              0000 000B
000004 : ILOAD             0001
000005 : ALOAD             0000
000006 : GETFIELD          0000 001A
000009 : IF_ICMPLT         0000 000C
00000C : NEW               0000 000D
00000F : DUP
000010 : ILOAD             0001
000011 : INVOKESPECIAL     0000 0012
000014 : ATHROW
000015 : ALOAD             0000
000016 : GETFIELD          0000 0032
000019 : ILOAD             0001
00001A : ALOAD             0000
00001B : GETFIELD          0000 0025
00001E : IADD
00001F : CALOAD
000020 : IRETURN
```

FIG. 2.7 – Exemple de la décompilation du corps d'une méthode

cp_info

La structure cp_info contient une entrée dans la table des constantes. Il existe plusieurs variantes de cette structure dépendant du type de constante qu'elle contient. Comme nous ne jugeons pas important de comprendre cette structure, nous ne la détaillerons pas.

Voici la liste des différents types de constante présents dans la table des constantes :

- CONSTANT_Class
- CONSTANT_Fieldref
- CONSTANT_Methodref
- CONSTANT_InterfaceMethodref
- CONSTANT_String
- CONSTANT_Integer
- CONSTANT_Float
- CONSTANT_Long
- CONSTANT_Double
- CONSTANT_NameAndType
- CONSTANT_Utf8

2.4 Java et JVM

La majorité des propriétés du langage Java sont directement attribuables à la JVM. Il est important de dissocier le langage Java de la JVM. Le langage Java est un langage orienté objet sûr mais ne contient aucune primitive intrinsèque pour en faire un langage dynamique, réflexif, sécuritaire ou parallèle car ces propriétés proviennent de la JVM.

De nouvelles propriétés se sont ajoutées à la JVM depuis sa première version. Par exemple, les nouvelles versions de JVM possèdent la notion de référence faible. Cette notion n'est pas accessible par le langage Java mais l'est par les librairies de la JVM.

2.5 Compilation de Java

L'utilisation du *bytecode* a permis au langage Java d'être interprété efficacement et d'être plus rapide que d'autres langages interprétés depuis un langage de plus haut niveau, mais son interprétation reste plus lente que l'exécution du même programme compilé par un compilateur statique natif. Le coût associé à la portabilité de Java a été sa vitesse. Par ses nombreuses propriétés intéressantes le *bytecode* Java est rapidement devenu une représentation intermédiaire pour les compilateurs. Pour obtenir de meilleures performances, une nouvelle vague de compilateurs Java est née : les compilateurs JIT (*Just In Time compilers*) ou compilateurs dynamiques.

2.5.1 Compilateur dynamique

Les compilateurs dynamiques [Kra98] compilent durant l'exécution le code vers le langage natif de l'hôte de la JVM. Par exemple, la JVM *Hotspot* [Sun99] de Sun effectue de la compilation dynamique. Ces compilateurs réussissent à obtenir de très bonnes performances en améliorant leurs analyses avec l'information dynamique (disponible pendant l'exécution).

Les analyses et optimisations effectuées par un compilateur dynamique sont restreintes par le temps et la complexité. Elles se doivent d'être rentables en terme de temps d'exécution comparé au temps de compilation. Le temps consacré à la compilation d'une méthode est lié à son temps d'exécution et sa fréquence d'exécution. Plusieurs travaux de recherche récents ont été effectués pour déterminer le niveau d'optimisation et le moment idéal pour effectuer une compilation ou compilation partielle. Il existe aussi des projets de compilation dynamique possédant un langage source de bas niveau tel l'assembleur [BDB00].

L'architecture de la JVM facilite la compilation dynamique. Elle fournit un langage intermédiaire d'un niveau suffisamment bas pour ne pas entraîner un coût trop élevé au moment de

la compilation et elle fournit un code vérifié et valide. Un standard d'interface entre la JVM et les JIT existe pour faciliter la greffe d'un compilateur dynamique à une JVM [Yel96].

Le langage Java se prête bien à la compilation dynamique au niveau du corps des méthodes. Chaque méthode peut être compilée nativement sans avoir d'effet sur les autres méthodes. Les appels de méthodes ou accès aux champs d'un objet se font via le mécanisme de la JVM, un mécanisme similaire à l'interface vers JNI (*Java Native Interface*) [Lia97]. Un gain de performance est obtenu pour les applications dont les points fréquemment exécutés du programme se retrouvent entièrement dans une méthode. Cependant le style de programmation objet et modulaire de Java rend la compilation dynamique native basée sur les méthodes inefficaces car les applications Java incluent beaucoup d'appels de fonction pour des corps de méthode relativement petits.

Les compilateurs dynamiques doivent faire des analyses globales pour optimiser les programmes Java ou doivent effectuer des artifices pour parvenir aux mêmes performances.

Les meilleurs compilateurs dynamiques arrivent à effectuer des analyses globales incrémentales et conservatrices qui souvent demandent d'éliminer une partie de code préalablement compilé pour remettre une version interprétée ou recompilée correspondant aux nouvelles propriétés et contraintes du système. Par exemple, l'optimisation d'*inliner* une méthode peut devenir incorrecte au chargement d'une nouvelle classe définissant une méthode virtuelle entrant en conflit avec l'optimisation. Le compilateur dynamique peut remplacer la méthode par l'ancienne méthode interprétée et la recompiler ultérieurement ou modifier la méthode dynamiquement pour s'assurer de la validité des optimisations. La Figure 2.8 montre un exemple d'optimisation dynamique, l'*inlining* de la fonction *calcul*, qui doit être modifiée au chargement d'une nouvelle classe et un exemple de code que le compilateur dynamique pourrait générer en remplacement.

Code original

```
class A {
  int valeur;

  int calcul( x ) {
    return x*x;
  }

  int bidon( A objet ) {
    return calcul( objet.valeur );
  }
}
```

Code optimisé

```
class A {
  int valeur;

  int bidon( A objet ) {
    int tmp = objet.valeur;
    return tmp*tmp;
  }
}
```

Après le chargement de la classe B

```
class A {
  int valeur;

  int calcul( x ) {
    return x*x;
  }

  int bidon( A objet ) {
    if( objet strict-instanceof A ) {
      int tmp = objet.valeur;
      return tmp*tmp;
    }
    else
      return calcul( objet.valeur );
  }

class B extends A {
  int calcul( x ) {
    return x*x*x;
  }
}
```

FIG. 2.8 – Chargement de classe rendant une optimisation incorrecte

Ce type d'optimisation nécessite des mécanismes compliqués pour remplacer une méthode, possiblement pendant son exécution. La JVM *Hotspot* possède ces mécanismes, ce qui explique en partie ses bonnes performances.

2.5.2 Compilation statique

L'évolution des langages présente aux compilateurs des langages de plus en plus modulaires et orientés objets ne permettant pas des analyses évoluées et rendant complexes leurs optimisations. Pourtant, en tant qu'utilisateur, on s'en remet à eux pour générer du code efficace pour des langages de niveau de plus en plus haut.

Les propriétés du langage Java et de la JVM ne permettent pas d'en effectuer facilement une compilation statique. La compilation statique de Java est confrontée au même problème que la compilation de modules C. Deux approches de compilations sont possibles : séparée et globale.

Compilation séparée

La compilation séparée consiste à compiler séparément les modules et à effectuer l'édition de liens après la compilation de tous les modules. L'édition de liens peut se faire statiquement ou dynamiquement.

La compilation séparée est plus proche du modèle modulaire et favorise le développement de librairies. Pour le développement d'une application, ce mode de compilation est privilégié parce qu'il offre un temps de compilation plus court qu'une compilation globale.

Le défaut de la compilation séparée est qu'elle rend le compilateur aveugle par l'ignorance des modules externes et limite ses optimisations. Les analyses doivent rester très conservatrices. Le même problème se pose avec les librairies fournies par les environnements d'exécution, tel que les librairies fournies par les systèmes d'exploitation.

Compilation globale

La compilation globale consiste à recompiler tout le code à chaque exécution et à se servir de l'information obtenue par une analyse globale pour améliorer la compilation. Ces analyses peuvent aller jusqu'à la simulation du code pour obtenir des renseignements dynamiques. Les analyses complexes rendent le temps de compilation trop long pour la phase de développement d'applications. Généralement, ce type de compilation est effectué pour générer la version finale d'un projet.

Il est difficile de compiler globalement une application si le langage permet le chargement dynamique et la réflexivité. Ces propriétés limitent les analyses et les optimisations du compilateur qui doit trop souvent rester conservateur au détriment de l'efficacité.

2.5.3 Modèle de compilation

L'architecture des compilateurs modernes sépare en couches le traitement nécessaire. Chaque couche possède ses propriétés, ses analyses et ses optimisations. Généralement, les premières couches sont dépendantes du langage et les dernières, de l'architecture.

Un modèle par couche de compilateur utilise la même idée qu'un compilateur par couche mais divise le traitement dans des entités distinctes (c'est-à-dire des programmes entiers). Ce modèle offre une meilleure division de traitement et comporte peu de dépendances entre les couches. Les couches de compilateurs ne partagent aucune autre structure de données que le langage intermédiaire.

Le modèle classique de compilation pour les applications Java est divisé en deux étapes. La première étape effectue une compilation séparée statique qui produit un *bytecode*. La seconde étape, qui s'effectue pendant l'exécution, est l'interprétation ou la compilation dynamique de ce *bytecode* vers l'architecture hôte.

2.6 Conclusion

Il est important de considérer le langage Java et la JVM comme deux entités distinctes. Ils possèdent des propriétés distinctes que les programmeurs rattachent souvent de manière erronée au langage. Ainsi, le chargement dynamique et la réflexivité sont des propriétés fournies par la JVM et non par le langage.

À l'intérieur du modèle classique de compilation, les étapes de compilation d'une classe pouvant être effectuées statiquement et indépendamment des autres classes sont effectuées par un compilateur statique pour produire du *bytecode*. Vu la facilité de chargement et les informations qu'il contient, le *bytecode* est devenu un langage source pour les compilateurs. Un interprète ou un compilateur dynamique exécute le *bytecode* en temps voulu.

Le modèle de compilation des applications Java qui gagne en popularité effectue de la compilation dynamique à partir du *bytecode*. Notre compilateur essaie un autre modèle de compilateur pour le langage Java que nous discutons dans le prochain chapitre. Il vise une compilation globale statique contrairement à une compilation séparée dynamique.

Pour simplifier notre compilateur et pour mieux le comparer à un compilateur dynamique, le langage source du compilateur est le *bytecode*. Deux versions différentes de ce compilateur, que nous décrivons plus loin, utilisent un modèle classique de compilateur et un modèle par couche de compilateur.

Chapitre 3

Compilateur de *bytecode*

Dans cette section, nous décrivons le langage source et l'architecture du compilateur que nous avons développée. Nous expliquons les représentations intermédiaires qu'il utilise et nous effectuons un survol du processus de compilation.

Stephan Diehl donne une bonne introduction aux techniques de compilation de Java [Die97].

3.1 Historique

La première version du compilateur, nommée J2S, compile du *bytecode* Java vers le langage Scheme. Ce compilateur fut développé dans le cadre d'un projet de recherche subventionné par Ericsson visant l'étude du compilateur Gambit [Fee98] et de son environnement d'exécution. La seconde version du compilateur, JBCC, est une version subséquente du compilateur. Le langage intermédiaire du compilateur J2S est réutilisé et un module de génération de code natif pour Intel est ajouté. Les deux versions de compilateurs, J2S et JBCC, partagent les mêmes langage source, représentation intermédiaire, analyses et optimisations ; seuls la génération de code et l'environnement d'exécution varient.

3.2 Langage source : Java*

JBCC et J2S compilent statiquement des applications Java à partir de leurs *bytecode*. Nous avons choisi d'utiliser comme langage source le *bytecode* à cause de sa simplicité de chargement et de l'information qu'il contient. Étant donné les propriétés dynamiques et modulaires du

langage, Java tend à favoriser la compilation dynamique contrairement à notre compilateur qui recherche une compilation statique. Pour permettre une compilation statique, nous avons enlevé des propriétés de la JVM telles que la réflexivité et le chargement dynamique. Ce sous-ensemble de Java, que nous appelons Java*, conserve les propriétés du langage, mais ne conserve pas toutes les propriétés de la JVM. Pour faciliter notre implantation, nous supposons que le *bytecode* fourni est déjà vérifié et valide. Nous supposons qu'il ne contient pas de code malicieux et que sa provenance est sûre. Nous pouvons justifier notre approche par le fait que nous avons isolé le langage de la JVM. Notre compilateur vise une compilation globale et précise améliorant les optimisations.

Nous avons appelé le langage Java* afin de dissocier le langage de la JVM. Lorsque nous parlons de Java, nous incluons le langage et la JVM.

Le respect des restrictions supplémentaires imposées par le langage Java* sur le langage Java est la responsabilité du programmeur. Les applications ne respectant pas les contraintes peuvent ne pas être compilées ou même donner des résultats imprévisibles à l'exécution.

3.3 Processus de compilation

Nous décrivons dans cette section le processus de compilation de J2S et JBCC pour obtenir un exécutable à partir d'une application Java*. En plus de fournir le code source de l'application, l'utilisateur doit fournir au processus de compilation les librairies natives habituellement fournies par la JVM ainsi que les renseignements sur les effets de bord de ces librairies.

3.3.1 J2S : *Java to Scheme*

J2S emploie une cascade de compilateurs pour effectuer la compilation complète d'une application. L'idée de ce modèle de compilation est de séparer le problème en sous-problèmes dont certains sont déjà résolus. Chaque couche de compilateur possède des propriétés distinctes. La Figure 3.1 montre l'architecture de compilation de J2S. J2S est une couche de compilateur entre deux autres couches.

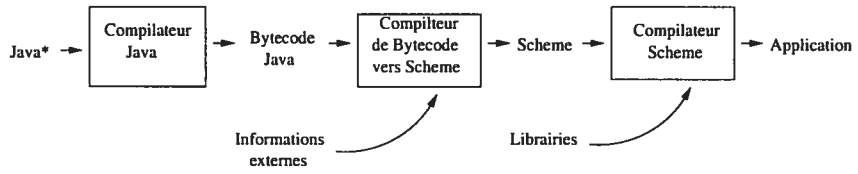

FIG. 3.1 – Processus de compilation de J2S

J2S repose sur le langage Scheme, un langage de haut niveau, pour générer l'application finale ; ainsi les propriétés du langage Scheme sont héritées. L'application hérite d'un environnement d'exécution concurrent, sûr et stable possédant les modules de gestion de mémoire et d'entrées/sorties.

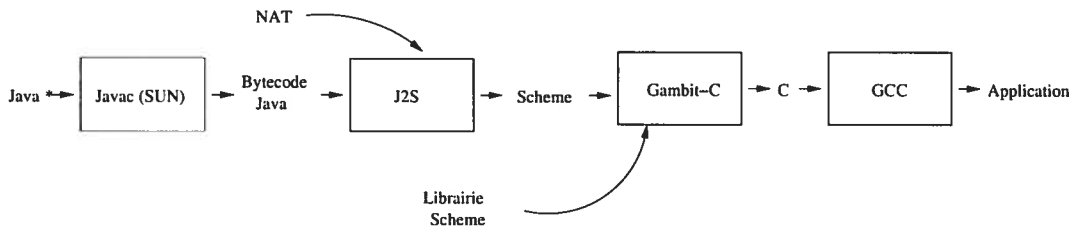

FIG. 3.2 – Exemple de compilation avec J2S et ses compilateurs

La Figure 3.2 montre un flux de compilation avec les compilateurs impliqués et les représentations intermédiaires pour obtenir un programme exécutable sous le modèle de compilation de J2S. Nous discutons plus loin des avantages et inconvénients de ce modèle de compilation.

3.3.2 JBCC : *Java ByteCode Compiler*

JBCC intègre toutes les couches nécessaires à la compilation de haut, moyen et de bas niveau du langage. Son architecture est similaire aux compilateurs modernes. Il produit du code assembleur à partir du langage Java*.

Pour obtenir une application à l'aide de JBCC, il faut employer d'autres compilateurs et fournir des librairies natives avec la spécification de leur impact sur l'environnement d'exécution. La Figure 3.3 montre le modèle du compilateur et la Figure 3.4 montre le flux d'information pour chaque compilateur impliqué et les représentations intermédiaires..

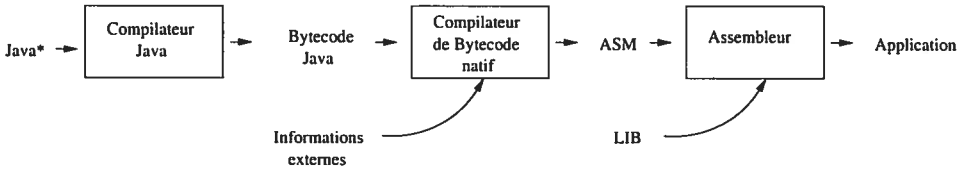

FIG. 3.3 – Processus de compilation de JBCC

Le *bytecode* est obtenu par un compilateur Java standard ou peut provenir d'une librairie précompilée. Par exemple, les librairies fournies par une JVM peuvent être précompilées, mais

l'emploi de propriétés non supportées par Java* provoque un comportement imprévisible et non spécifié. Le *bytecode* est compilé par JBCC vers du code assembleur. L'utilisateur fournit à JBCC les informations sur les librairies natives qui seront utilisées pour produire l'application finale. Après l'étape de compilation par JBCC, l'utilisateur peut effectuer l'assemblage de son programme et l'édition de liens avec les librairies natives.

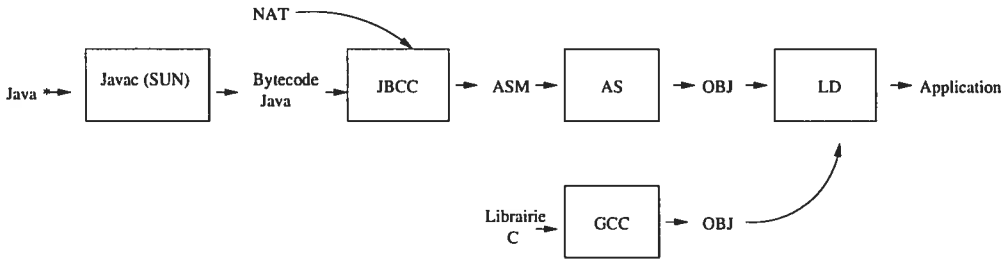

FIG. 3.4 – Exemple de compilation avec JBCC et ses compilateurs

La Figure 3.4 montre un exemple de compilation avec les compilateurs impliqués pour obtenir un programme exécutable sur Linux. Si l'utilisateur désire que l'application roule sous un autre système d'exploitation ou qu'elle emploie des librairies natives différentes, il doit fournir les nouveaux fichiers d'informations externes à JBCC et effectuer l'édition de liens avec ses librairies. JBCC fournit les librairies et les fichiers d'informations pour un système compatible POSIX [ISO96, UNI].

3.4 Cheminement et représentation du code

Dans cette section, nous décrivons le processus de compilation de notre compilateur et effectuons un survol des représentations intermédiaires utilisées.

L'utilisateur du compilateur fournit en entrée le *bytecode* de l'application, les librairies natives et la description des librairies natives. La Figure 3.5 montre le cheminement du code à l'intérieur des compilateurs (la ligne du haut pour J2S et toutes les lignes pour JBCC). Les processus de compilation sont encadrés et les flèches décrivent les représentations en entrée et en sortie des processus. Les langages JBC, NAT, JST, JASM et ASM sont les représentations intermédiaires du compilateur que nous détaillons à la Section 3.4.1.

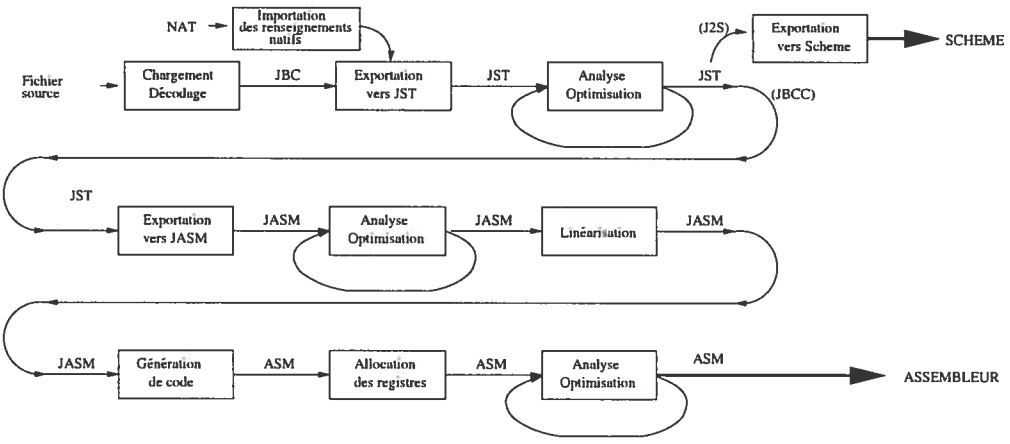

FIG. 3.5 – Cheminement et représentation du code

Les deux compilateurs partagent les premières étapes et représentations. Le compilateur J2S dérive la sortie du langage JST vers un module d'exportation Scheme tandis que JBCC continue le traitement du langage JST vers des langages de plus bas niveau au lieu de laisser un langage de plus haut niveau et ses compilateurs effectuer la tâche.

Le premier processus de compilation consiste à charger toutes les classes requises par le programme et à décoder leurs méthodes. Cette représentation, JBC, est exportée vers le langage JST. Les analyses et optimisations de haut niveau sont effectuées à ce stade. Le langage JST optimisé est converti vers une représentation de plus bas niveau : JASM. Les analyses et optimisations de bas niveau sont effectuées. Le code est ensuite linéarisé pour perdre toute notion de bloc. L'étape de génération de code exporte les instructions JASM vers l'assembleur natif. Le code obtenu est de l'assembleur natif avec des registres virtuels. Le compilateur alloue les registres selon les contraintes spécifiées par les registres virtuels. Une dernière étape d'analyse et d'optimisation au niveau des instructions termine le processus de compilation.

3.4.1 Représentations intermédiaires

Plusieurs langages sont employés dans le processus de compilation et chacun d'eux fournit des propriétés facilitant certains traitements. Les compilateurs modernes emploient plusieurs représentations intermédiaires spécialisées pour chaque traitement et pour permettre la compilation de plusieurs langages sources vers plusieurs architectures cibles. JBCC a été conçu pour compiler le *bytecode* vers différentes architectures. Présentement, seule l'architecture Intel est supportée.

JBC : *Java ByteCode*

Le langage JBC est obtenu par le chargement des classes et par le décodage des méthodes. La Figure 2.7 montre un exemple de code JBC. Ce langage est la première représentation du programme source. Il est de bas niveau et proche d'une machine virtuelle. Il contient beaucoup d'informations locales mais peu d'informations globales. Une compilation naïve et directe à partir de ce langage est possible.

Nous ne détaillons pas ce langage puisqu'il est bien défini dans la spécification de Sun [LY99a]. Les informations nécessaires aux lecteurs sont décrites dans la Section 2.3.

JST : *Java Syntax Tree*

Le langage JST est une représentation de haut niveau du code source. Il a été conçu pour faciliter les analyses et optimisations statiques et globales de haut niveau. Il permet la représentation de code structuré avec les renseignements de type et de profilage. Cette représentation rend abstraite la notion de machine et tend vers une représentation fonctionnelle fortement orientée objet. La représentation JST se retrouve sous forme d'arbre en mémoire. Il existe plusieurs variantes de cette représentation selon les informations qu'elle contient.

```
1. CLASS java/lang/Integer extends java/lang/Number
2.     FIELD value :<I 0 567>
3.
4.     METHOD toString ()Ljava/lang/String;
5.     l#1048576-0#781 <REF : OBJ(java/lang/Integer) >
6.     : v#785 <REF : OBJ(java/lang/String) >
7.
8.     {
9.       let l#1048576-0#781 <REF :OBJ(java/lang/Integer) >
10.            l#1048576-0#781 <REF :OBJ(java/lang/Integer) >
11.    [...]
12.      let v#2291 <I -inf +inf>
13.            fieldref java/util/Hashtable . count v#2290 <REF :OBJ(java/util/Properties) >
14.      let v#2293 <I -inf +inf>
15.            add v#2291 <I -inf +inf> 1 <I 1 1>
16.    [...]
17.      jump BasicBlock#8243  ( l#1048576-0#781 <REF :OBJ(java/lang/Integer) > )
18.    }
19.    [...]
```

FIG. 3.6 – Exemple de représentation JST

La Figure 3.6 montre un exemple de représentation JST. Nous remarquons que le langage possède la notion d'objet, de champ et de méthode avec les mots clés `CLASS`, `FIELD` et `METHOD`. De plus, il contient de l'information riche pour permettre de bonnes analyses.

À la première ligne de l'exemple, nous observons la définition de la classe `java.lang.Integer`. La deuxième ligne définit le champ `value` de cette classe et nous observons que le compilateur a borné son domaine entier entre 0 et 567. Les lignes 4 à 6 montrent la définition de la méthode `toString`. Cette méthode est virtuelle et par conséquent reçoit comme

premier paramètre la variable locale `l#1048576-0#781` représentant la référence *this*. Nous observons que le compilateur a prouvé que *this* est une référence et ne peux contenir que des instances de la classe `java.lang.Integer`. Cette méthode retourne le contenu de la variable `v#785`, une instance de la classe `java.lang.String`. Les lignes 8 à 18 montrent le corps de cette méthode. Les lignes 9 et 10 montrent une création d'environnement (expression `let`) pour les instructions suivantes du même bloc. Un nouvel environnement est créé et contient la variable `l#1048576-0#781` à laquelle est affectée l'évaluation de l'expression dans l'ancien environnement. Les lignes 12 et 13 montrent un accès au champ non statique `count` de la classe `java.util.Hashtable`. Le compilateur a prouvé que la référence `v#2290` ne contient que des instances de la classe `java.util.Properties`, une classe qui hérite de la classe `java.util.Hashtable`. Le contenue du champ `count` n'a pas pu être borné et contient un domaine de l'infini négatif (`-inf`) à l'infini positif (`+inf`). Les renseignements globaux que notre compilateur prouve rendent les optimisations précises et ciblées.

Nous expliquons plus en détails les mécanismes d'exportation, les analyses et les optimisations de ce langage dans le Chapitre 4. La syntaxe et la sémantique du langage JST est défini dans l'Annexe A.1.

NAT : Interface Native

Ce langage permet de spécifier au compilateur l'interface des fonctions natives et l'impact de celles-ci sur le programme. La création de ce langage est justifiée par le fait que nous désirons un compilateur moins aveugle. Les informations apportées par ce langage rendent les analyses plus précises. Généralement, les compilateurs évaluent de manière conservatrice les effets de bord des fonctions natives, polluant ainsi les analyses globales.

```
1.  ;# INCLURE core.nat
2.  ;# OPTION jasm-call-convention C
3.
4.  ;#DEFINE java/lang/System initProperties
5.  ;#                    (Ljava/util/Properties;)Ljava/util/Properties;
6.  ;#REQUIRE-METHOD java/util/Hashtable put
7.  ;#               (Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
8.  ;#TYPE CALL-VIRTUAL-METHOD java/util/Hashtable put
9.  ;#               (Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
10. ;#               PARAM-TYPE 0
11. ;#               CLASS java/lang/String
12. ;#               CLASS java/lang/String
13. ;#TYPE RETURN PARAM-TYPE 0
14. ;#END
15.
16. [...]
17.
18. ;#DEFINE java/lang/Math sin (D)D
19.  fldl 4(%esp)
20.  fsin
21.  fstp %st(1)
22.  ret
23. ;#INLINE
24.  fldl 4(%esp)
25.  fsin
26.  fstp %st(1)
27. ;#END
```

FIG. 3.7 – Exemple de représentation NAT

La Figure 3.7 montre un exemple de fichier de renseignements externes fourni à notre compilateur. Nous y observons la définition de fonctions externes écrites en assembleur et qui peuvent être insérées dans le code, les renseignements sur les méthodes natives et l'impact de ces fonctions sur le programme.

Si le programmeur omet de définir un effet de bord d'une méthode native sur le programme, le résultat des optimisations peut être faux et le programme généré aura un comportement imprévisible.

Les informations du fichier sont données sur des lignes en commentaire car les autres lignes contiennent généralement du code source d'un autre langage. Ainsi, les syntaxes de commentaires de Scheme, C/C++ et Assembleur sont acceptées à l'intérieur des fichiers natifs. Ce mécanisme de définition d'information permet d'inclure dans un même fichier le code natif et ses renseignements tout en conservant un fichier qui peut être compilé séparément avec un compilateur spécifique au langage.

La première ligne demande au compilateur d'inclure un autre fichier contenant d'autres informations natives. La seconde informe le compilateur sur la convention d'appel utilisée pour les méthodes natives définies dans ce fichier. Ainsi, le programmeur peut choisir entre la convention d'appel de C, la convention d'appel avec passage de paramètres par les registres et la convention d'appel interne à notre compilateur.

Les lignes 4 à 14 renseignent le compilateur sur l'existence d'une méthode native `java.lang.System.initProperties`. Les lignes 6 et 7 informent le compilateur sur la dépendance entre cette méthode et la méthode `java.util.Hashtable.put`. Ces renseignements sont

utilisés par l'algorithme d'élimination de code mort. Les lignes 8 à 13 informe l'impact de l'appel de cette méthode sur le système de type. Ces renseignements sont utilisés lors de l'analyse de type. Le corps de cette méthode est défini dans un fichier séparé que le programmeur doit fournir.

Les lignes 18 à 27 définissent la méthode `java.lang.Math.sin`. Nous observons une double définition du corps de cette méthode. Le compilateur choisira quelle version utiliser selon la convention d'appel, possiblement modifiée à la suite d'analyses détaillées. La convention d'appel `inline` permet au compilateur de glisser le code au site d'appel de la méthode en respectant le passage de paramètres par registre.

Nous ne détaillons pas davantage ce langage que nous avons créé. Il faut simplement noter qu'il existe un langage capable d'informer le compilateur sur les effets de bord externes.

JASM : Assembleur Java

Ce langage est une représentation abstraite de l'assembleur permettant les analyses et optimisations de bas niveau communes à toutes les architectures.

```
1.  ;;- Java ASSEMBLER
2.  .start _java_clinit
3.  mti__java_lang_NullPointerException__init___V ==> 28
4.  fdi__java_util_Hashtable_threshold ==> 16
5.  fdi__java_util_Hashtable_count ==> 24
6.  fdi__java_util_Hashtable_table ==> 32
7.  [...]
8.
9.  java_util_Hashtable_put_Ljava_lang_Object_Ljava_lang_Object__Ljava_lang_Object_ :
10. ptr.def     Pp32.r-1
11. ptr.def     Pp32.r-2
12. ptr.def     Pp32.r-3
13. ptr.ld      p32.r40  Pp32.r-1
14. ptr.ld      p32.r41  Pp32.r-2
15. ptr.ld      p32.r42  Pp32.r-3
16. ptr.set     p32.r40  p32.r40
17. ptr.set     p32.r41  p32.r41
18. ptr.set     p32.r42  p32.r42
19. ptr.push    p32.r40
20.     call    java_lang_System__mutex_lock__I_V
21. ptr.set     p32.r42  p32.r42
22. ptr.set     p32.r41  p32.r41
23. ptr.set     p32.r40  p32.r40
24.     goto    bb__25201__
25. bb__25201__ :
26. ptr.set     p32.r42  p32.r42
27. ptr.set     p32.r41  p32.r41
28. ptr.set     p32.r40  p32.r40
29.     goto    bb__25202__
30. [...]
31.
```

FIG. 3.8 – Exemple de représentation JASM

La Figure 3.8 montre un exemple de code JASM. Nous observons aux lignes 3 à 6 la définition de constantes potentiellement utilisées par des librairies externes. Nous y retrouvons la définition des index des champs des objets ainsi que la définition des index des méthodes dans les tables virtuelles.

Les lignes 9 à 29 définissent la méthode `java.util.Hashtable.put`. Nous observons que toutes les instructions et registres sont typés. Le langage supporte un nombre infini de registres facilitant les transformations de code de bas niveau. La notion de structure de code est limitée aux blocs de base. Les notions de haut niveau tels les objets, les appels de méthodes et les moniteurs ne font plus partie des informations conservées.

Ce langage étant de bas niveau, il se prête mal aux analyses globales statiques et par conséquent, les traitements effectués sur ce langage sont similaires aux traitements effectués par d'autres compilateurs. Nous ne décrivons pas davantage ce langage ainsi que ses analyses et optimisations dans ce document.

ASM : Assembleur Intel

La représentation finale de notre compilateur est sous la forme de l'assembleur Intel. Cette représentation est compilable par les assembleurs supportant la syntaxe AT&T.

```
.globl java_util_Hashtable__init__IF_V
  .type java_util_Hashtable__init__IF_V,@function

java_util_Hashtable__init__IF_V :
  pushl     %ebp
  movl      %esp,%ebp
  subl      $32,%esp
  movl      %ebx,-12(%ebp)
  movl      %esi,-28(%ebp)
  movl      8(%ebp),%ebx
  movl      12(%ebp),%eax
  flds      16(%ebp)
  fst       %st(1)
  fstps     -16(%ebp)
  pushl     %ebx
  call      java_util_Dictionary__init___V
  addl      $4,%esp
  flds      -16(%ebp)
  fst       %st(1)
  fstp      %st(2)
  fldl      nd__219303__
  fld       %st(2)
  fcompp
  fnstsw    %eax
  sahf
  movl      $1,%eax
  jp        fcmp0
  movl      $-1,%eax
  jb        fcmp0
  movl      $1,%eax
  ja        fcmp0
  movl      $0,%eax
fcmp0 :
  cmpl      $0,%eax
  jng       bb__135__
  fld       %st(0)
  fstps     -16(%ebp)
  jmp       bb__137__
bb__135__ :
  pushl     $16
  call      java_lang_System__memory_alloc__I_I
  addl      $4,%esp
  movl      $vt__java_lang_IllegalArgumentException__,%ebx
  movl      %ebx,0(%eax)
  movl      %eax,%ebx
  pushl     %ebx
  call      java_lang_IllegalArgumentException__init___V
  addl      $4,%esp
  pushl     %ebx
  call      java_lang_System__throw__Ljava_lang_Throwable__V
  addl      $4,%esp
  movl      -28(%ebp),%esi
  movl      -12(%ebp),%ebx
  leave
  ret
bb__137__ :
  flds      -16(%ebp)
  fst       %st(1)
  fstps     24(%ebx)
  pushl     $20
  call      java_lang_System__memory_alloc__I_I
  addl      $4,%esp
  movl      $vt__java_lang__array___,%ecx
  movl      %ecx,0(%eax)
  movl      %eax,%esi
  movl      $101,%eax
  movl      $4,%ecx
  imull     %ecx,%eax
  pushl     %eax
  call      java_lang_System__memory_alloc__I_I
  addl      $4,%esp
  movl      $8,%ecx
  movl      %ecx,4(%esi)
  movl      $101,%ecx
  movl      %ecx,12(%esi)
  movl      %eax,16(%esi)
  movl      %esi,20(%ebx)
  movl      $101,-4(%ebp)
  fildl     -4(%ebp)
  fstp      %st(3)
  flds      -16(%ebp)
  fst       %st(2)
  fld       %st(3)
  fmulp
  fstp        %st(1)
  fstcw     -4(%ebp)
  fstcw     -8(%ebp)
  orl       $3072,-4(%ebp)
  fldcw     -4(%ebp)
  fld       %st(0)
  fistpl    -4(%ebp)
  fldcw     -8(%ebp)
  movl      -4(%ebp),%eax
  movl      %eax,12(%ebx)
  movl      -28(%ebp),%esi
  movl      -12(%ebp),%ebx
  leave
  ret
```

FIG. 3.9 – Exemple de représentation Assembleur Intel

Un exemple est donné à la Figure 3.9. Nous observons dans cet exemple la définition du constructeur de la classe `java.util.Hashtable`.

3.5 Langage cible

Toutes les versions de compilateur que nous avons implantées avaient pour langage intermédiaire le langage JST et les informations NAT. La première version de notre compilateur, J2S, avait pour langage cible Scheme tout comme le compilateur ETOS de Erlang vers Scheme [FL98]. Nous discutons plus en détail de ce langage cible et des résultats obtenus dans le Chapitre 5. Par la suite, nous avons décidé de générer du code C tout comme Gambit-C et [BDBV94], mais cette

version resta inachevée et nous avons décidé de générer du code natif comme dans [HGmWH]. Nous discutons dans le Chapitre 6 de la dernière version de notre compilateur, JBCC, qui génère du code natif pour processeur Intel (IA32).

Chapitre 4

Le langage intermédiaire

Dans ce chapitre nous décrivons la syntaxe et la sémantique de notre langage intermédiaire. Ensuite, nous examinons les mécanismes d'importation du *bytecode* Java vers le langage intermédiaire JST. Nous discutons de certaines analyses et optimisations effectuées sur le langage ainsi que des analyses et optimisations futures du compilateur.

4.1 JST : Java Syntax Tree

Le langage JST a été conçu pour permettre des analyses de haut niveau précises et il est fortement inspiré du langage intermédiaire ILL du compilateur Trail [HJU00]. Il ne contient aucune primitive dépendante du système d'exploitation ou de l'architecture. Les propriétés nécessaires pour valider le code ne sont pas conservées puisque nous supposons un code valide et vérifié. Par exemple, les propriétés d'accès aux variables ne sont pas conservées dans notre langage intermédiaire. Une étape de vérification et de validification pourrait facilement être ajoutée, mais elle n'a pas d'impact sur la performance du code produit et nous avons décidé de ne pas l'inclure dans notre prototype. Le langage est fortement typé et toutes les conversions de types nécessaires doivent être explicitement présentes. Les conversions de types implicites du *bytecode* et de la sémantique de Java sont ajoutées lors du processus d'importation du *bytecode*. Les vérifications dynamiques doivent explicitement être présentes et sont ajoutées lors de l'importation. Le langage supporte une portée lexicale, mais comme le compilateur effectue une numérotation des variables, aucun conflit entre variables n'est possible.

4.1.1 Syntaxe et sémantique

Nous décrivons dans cette section la syntaxe et la sémantique du langage intermédiaire de notre compilateur. Le premier langage ciblé par notre compilateur étant Scheme, la syntaxe et la sémantique du langage JST lui est similaire.

Comme les programmes encodés dans ce langage ne se retrouvent qu'en mémoire lors de la compilation, il ne possède pas de syntaxe textuelle et toutes les représentations textuelles générées, qui varient selon les paramètres passés au compilateur, sont représentées par la grammaire conceptuelle donnée dans l'Annexe A.1. Les différentes représentations disponibles du langage servent uniquement à vérifier le fonctionnement du compilateur et la validité de ses analyses et optimisations lors de son développement.

La sémantique du langage est fortement inspirée de Scheme. La modification majeure apportée à Scheme est la modification de l'évaluation numérique pour l'adapter à Java* car leur système numérique est incompatible. En plus d'inclure les instructions logiques et arithmétiques adaptées, le langage prévoit des opérations spécialisées pour la manipulation d'objets.

Trois différentes sortes de variables sont présentes dans le langage : les variables d'exceptions, les variables locales et les variables temporaires. Les noms des variables sont formés par une lettre et un numéro d'identificateur. Pour permettre de dissocier la sorte de variables, la lettre du nom indique la sorte de variables. Les lettres `v`, `l` et `e` représentent respectivement les variables de sorte temporaire (`v#1`), locale (`l#1`) et d'exception (`e#1`). Les variables temporaires ne possèdent qu'un point d'affectation. Les variables locales et les variables d'exception possèdent plusieurs points statiques d'affectation et doivent respecter les contraintes du mécanisme d'exceptions. La seule différence entre les variables locales et les variables d'exceptions est que les variables d'exceptions ont été introduites par le mécanisme d'exceptions et influencent les règles d'analyse du traitement parallèle.

4.2 Importation du *bytecode*

Dans cette section nous décrivons le mécanisme d'importation du *bytecode* décodé et le langage intermédiaire en résultant. Nous ne donnons pas la liste exhaustive des mécanismes d'importation, mais l'idée générale et les cas spéciaux.

L'importation du *bytecode* consiste à importer toutes les classes potentiellement utilisées dans l'application compilée. Toutes les classes référencées doivent passer par la table des constantes du *bytecode* de la classe. Comme aucune analyse n'est encore effectuée au début de l'importation, le compilateur ne connaît pas les classes requises et emploie un mécanisme conservateur pour

charger les classes : le compilateur charge récursivement toutes les classes référencées par une structure `ClassInfo` de la table des constantes. Le compilateur élimine plus tard les classes qu'il peut prouver inutiles. La première classe chargée est la classe principale, donnée en paramètre au compilateur. Lors de l'importation d'une classe, le compilateur importe tous ses champs ainsi que toutes ses méthodes. Les informations utiles pour les analyses sont conservées dans les structures du langage JST.

L'importation du corps des méthodes est plus complexe : le code source est du *bytecode*, un langage à pile, tandis que le langage JST est un langage à registres. La première phase consiste à découper le programme pour obtenir les *blocs de base* (séquence d'instructions se terminant par un branchement conditionnel ou non). La deuxième phase résoud statiquement la pile d'exécution. Une phase pour activer et désactiver les gestionnaires d'exceptions selon les parties de code qu'ils couvrent et les moniteurs termine l'importation du *bytecode*.

Nous expliquons les mécanismes d'importation dans les prochaines sections. Comme exemple de code à compiler, nous nous servons du code de la Figure 4.1.

```
public synchronized int loop (int n) {
  int sum = 0;
  for (int i=0; i<n; i++)
    sum += i;
  return sum;
}
```

FIG. 4.1 – Code source Java*

4.2.1 Décodage des méthodes

La première phase consiste à découper le corps des méthodes en blocs de base. Le code source est préalablement décodé par le chargeur de classes de notre compilateur et arrive sous la forme d'un tableau d'instructions décodées. La Figure 4.2 représente l'entrée du module d'importation du langage JST.

```
0000 : ICONST_0          0000
0001 : ISTORE            0002            ; sum = 0
0002 : ICONST_0          0000
0003 : ISTORE            0003            ; i=0
0004 : GOTO              0000 000A       ; jump a l'adresse 0000E
0007 : ILOAD             0002            ; chargement de sum
0008 : ILOAD             0003            ; chargement de i
0009 : IADD                              ; t := i+sum
000A : ISTORE            0002            ; sum := t
000B : IINC              0003 0001       ; i++
000E : ILOAD             0003            ; chargement de i
000F : ILOAD             0001            ; chargement de n
0010 : IF_ICMPLT         00FF 00F7       ; branchement a l'adresse 0007 si i<n
0013 : ILOAD             0002            ; chargement de sum
0014 : IRETURN                           ; return sum
```

FIG. 4.2 – Bytecode Java décodé

4.2.2 Découpage en blocs de base

Les blocs de bases débutent aux points de branchements d'un bloc de code (c'est-à-dire tous les points étant la cible d'un branchement explicite ou implicite). Selon la définition d'un bloc de base, il ne peut y avoir de branchements dont la cible est au milieu d'un bloc de base. Ce regroupement en blocs facilite les analyses.

Les points nécessitant un bloc de base sont :

- l'entrée d'une méthode,
- l'activation d'un gestionnaire d'exception,
- la désactivation d'un gestionnaire d'exception,
- l'instruction suivant un branchement conditionnel,
- la cible d'un branchement conditionnel,
- la cible d'un branchement inconditionnel,
- la cible d'un appel à une sous-routine,
- l'instruction suivant un appel à une sous-routine,
- l'instruction suivant une création de tableau.

Une autre approche valide serait de créer un bloc de base pour chaque instruction et laisser à une transformation subséquente le soin de les regrouper en blocs plus larges. Cette approche, quoique valide, ralentit les analyses.

Pour effectuer le découpage en blocs de base, le compilateur applique un algorithme de découpage récursif. L'appel initial à l'algorithme s'effectue sur l'entrée de la méthode (et les entrées des gestionnaires d'exception). Par la suite, l'algorithme parcourt le bloc courant et détermine les nouveaux points de découpage selon les instructions du bloc. Il applique récursivement l'algorithme à tous les points de code nécessitant un découpage. La condition d'arrêt de la récursion est rencontrée lorsque l'algorithme a déjà analysé le point de découpage.

Par exemple, l'appel initial à l'algorithme pour découper le code de la Figure 4.2 commence le découpage à l'adresse `0000` et crée le bloc de base `BasicBlock#1`. L'algorithme parcourt le bloc découpé et détermine un nouveau point de découpage à l'adresse `000E` (la cible du branchement inconditionnel de l'adresse `0004`). L'appel récursif crée le bloc de base `BasicBlock#2` et détermine deux nouveaux points de découpage aux adresses `0007` et `0013` (les cibles du branchement conditionnel de l'adresse `0010`). Les appels récursifs sur les deux nouveaux points de découpage crée les blocs de base `BasicBlock#3` et `BasicBlock#4`.

Après cette phase, le compilateur obtient le code représenté à la Figure 4.3.

```
BasicBlock#1 {
  0000 : ICONST_0          0000
  0001 : ISTORE            0002
  0002 : ICONST_0          0000
  0003 : ISTORE            0003
  0004 : GOTO              0000 000A
}
BasicBlock#3 {
  0007 : ILOAD             0002
  0008 : ILOAD             0003
  0009 : IADD
  000A : ISTORE            0002
  000B : IINC              0003 0001
}
BasicBlock#2 {
  000E : ILOAD             0003
  000F : ILOAD             0001
  0010 : IF_ICMPLT         00FF 00F7
}
BasicBlock#4 {
  0013 : ILOAD             0002
  0014 : IRETURN
}
```

FIG. 4.3 – *Bytecode* Java* découpé

4.2.3 Importation des instructions et résolution de la pile d'exécution

La seconde phase consiste à générer les instructions du langage et à résoudre la pile d'exécution. La résolution de la pile consiste à associer les variables dynamiques du *bytecode* à des registres du langage JST. Le but de cette analyse est d'éliminer la notion de pile d'exécution dans notre langage intermédiaire et permettre des analyses plus précises.

Le compilateur effectue une exécution abstraite parcourant toutes les branches du corps de la méthode en simulant la pile d'exécution. La génération de code est triviale pour la majorité des instructions du *bytecode*. Les règles d'interprétation abstraite et de génération de code sont fournies dans l'Annexe B. Lors de l'exécution abstraite, le compilateur crée une variable temporaire pour chaque donnée dynamique déposée sur la pile d'exécution et retourne cette variable lors de sa consommation. Lors d'un branchement, les données dynamiques non consommées par le bloc de base sont passées en paramètre au bloc de base cible. Ainsi, des branchements différents au même bloc de base peuvent posséder des variables différentes. La Figure 4.4 montre l'exécution abstraite pour le premier bloc de base.

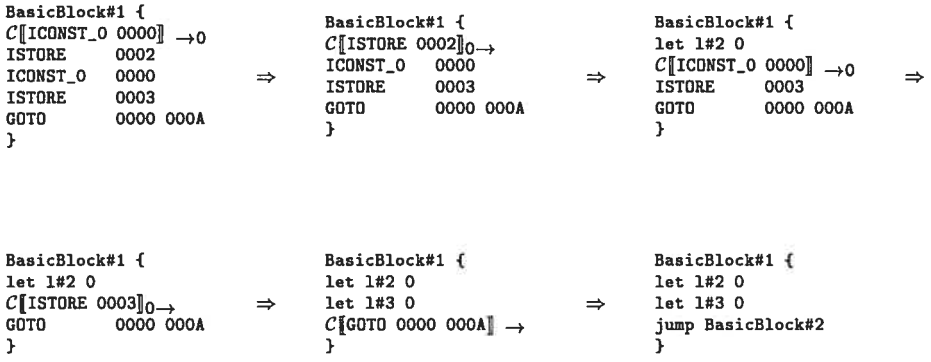

FIG. 4.4 – Importation par interprétation abstraite et résolution de la pile d'exécution

La Figure 4.5 montre le code d'exemple obtenu après cette phase. Dans cet exemple, aucun paramètre dynamique passé par la pile n'est échangé entre les blocs de base. Il est à noter qu'à ce stade le passage des variables locales n'est pas encore effectué.

```
1.  {
2.    jump BasicBlock#1
3.  }
4.  BasicBlock#1 {
5.    let l#2 0
6.    let l#3 0
7.    jump BasicBlock#2
8.  }
9.  BasicBlock#3 {
10.   let v#3 l#2
11.   let v#4 l#3
12.   let v#5 add v#3 v#4
13.   let l#2 v#5
14.   let v#6 l#3
15.   let v#7 add v#6 1
16.   let l#3 v#7
17.   jump BasicBlock#2
18. }
19. BasicBlock#2 {
20.   let v#1 l#3
21.   let v#2 l#1
22.   if (lt v#1 v#2)
23.     jump BasicBlock#3
24.   else
25.     jump BasicBlock#4
26. }
27. BasicBlock#4 {
28.   let v#8 l#2
29.   return v#8
30. }
```

FIG. 4.5 – Langage JST obtenu après l'interprétation abstraite

Comme mentionné dans la spécification de la JVM, un *bytecode* dont la pile ne se résoud pas statiquement est un code invalide ; nous supposons donc que notre compilateur ne sera pas confronté à cette situation.

Pour montrer la technique de passage des valeurs restantes sur la pile de l'interprète abstrait au bloc de base, nous avons choisi un exemple simple et valide selon les règles du *bytecode* mais qui n'a pas été généré par un compilateur Java. La structure impérative des méthodes Java donne un *bytecode* qui échange peu d'information par la pile d'exécution entre les différents blocs de bases.

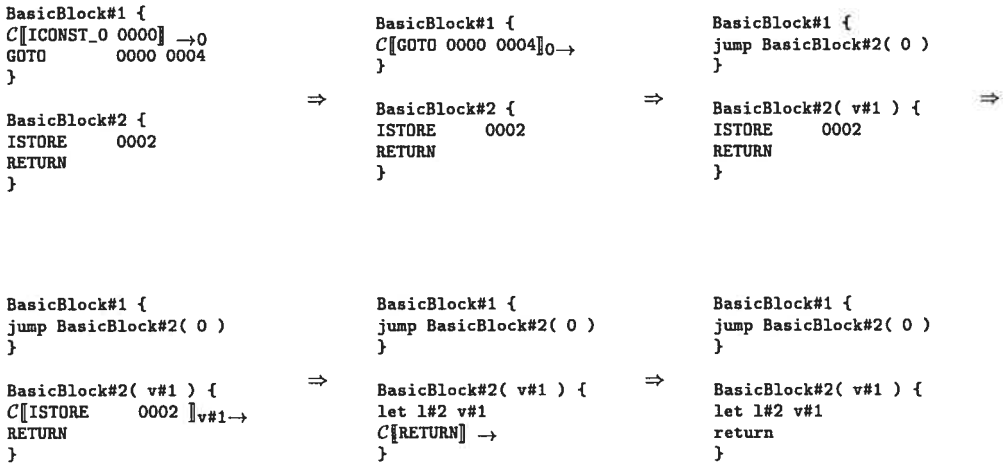

FIG. 4.6 – Importation par interprétation abstraite et résolution de la pile d'exécution

La Figure 4.6 montre l'interprétation abstraite de notre exemple selon les règles d'importation donnée en Annexe. Nous pouvons y observer le passage de la valeur `0` au bloc de base `BasicBlock#2`. Le bloc de base reçoit cette valeur par la variable `v#1` permettant ainsi plusieurs sites de branchement pour le même bloc de base. Cette technique ne serait pas possible si la taille de la pile d'exécution pouvait varier dynamiquement.

4.2.4 Importation des variables locales et des exceptions

Les variables locales sont ajoutées à tous les blocs de base et tous les branchements. Les variables *mutables* (vivantes à l'intérieur d'un gestionnaire d'exception) ne peuvent pas être passées en paramètre à un bloc de base. Par conséquent, il n'existera qu'un registre ou espace mémoire disponible pour contenir toutes les instances de la variable peu importe son lieu et son moment d'utilisation. Les variables passées en paramètre aux blocs de base seront transformées sous la forme SSA (*Static Single-Assignment*) excepté pour les variables mutables. Une méthode est sous la forme SSA si chaque variable recevant une valeur est la cible d'une seule instruction. Cette forme découpe l'intervalle de vie d'une variable en sous intervalle permettant des analyses plus précises. Nous discutons de cette optimisation dans la Section 4.3.5.

Après cette phase, le compilateur obtient le code représenté à la Figure 4.7. Nous y voyons maintenant le passage des variables locales des branchements vers les blocs de base cibles, la gestion des mécanismes de synchronisation créés par le mot clé `synchronized` et la gestion des exceptions qui en découle.

```
1.  {
2.    let l#0 l#0
3.    let l#1 l#1
4.    let l#2 0
5.    let l#3 0
6.    lock l#0
7.    let e#1 0
8.    methodref java/lang/Thread . <set_exception_handler>(A)V ( &BasicBlock#5  )
9.    jump BasicBlock#1 ( l#1 l#2 l#3 )
10. }
11. BasicBlock#1 ( l#1 l#2 l#3 ) {
12.   let l#2 0
13.   let l#3 0
14.   jump BasicBlock#2  ( l#1 l#2 l#3 )
15. }
16. BasicBlock#3 ( l#1 l#2 l#3 ) {
17.   let v#3 l#2
18.   let v#4 l#3
19.   let v#5 add v#3 v#4
20.   let l#2 v#5
21.   let v#6 l#3
22.   let v#7 add v#6 1
23.   let l#3 v#7
24.   jump BasicBlock#2
25. }
26. BasicBlock#2 ( l#1 l#2 l#3 ) {
27.   let v#1 l#3
28.   let v#2 l#1
29.   if (lt v#1 v#2)
30.     jump BasicBlock#3 ( l#1 l#2 l#3 )
31.   else
32.     jump BasicBlock#4 ( l#1 l#2 l#3 )
33. }
34.
35. BasicBlock#4 ( l#1 l#2 l#3 ) {
36.   let v#8 l#2
37.   return v#8
38. }
39. BasicBlock#5  {
40.   let v#9 methodref java/lang/Thread . currentThread()Ljava/lang/Thread;
41.   let v#10 checknull v#9
42.   let v#11 fieldref java/lang/Thread . <exception> v#10
43.   if (e#1)
44.     jump BasicBlock#6  ( v#11  )
45.   else
46.     jump BasicBlock#6  ( v#11  )
47. }
48. BasicBlock#6 ( v#12 )  {
49.   methodref java/lang/Thread . <unset_exception_handler>(A)V ( &BasicBlock#1531  )
50.   unlock l#0
51.   throw v#12
52. }
53. }
```

FIG. 4.7 – Langage JST obtenu après l'importation

Nous pouvons oberver le passage des variables locales `l#1 l#2 l#3` avec le branchement vers le `BasicBlock#1` à la ligne 9. Les variables locales sont reçues par le bloc de base `BasicBlock#1` à la ligne 11. Nous observons que le bloc de base `BasicBlock#2` a plusieurs sites d'appel (lignes 14 et 24) et reçoit ses paramêtres de plusieurs sites.

Nous observons aux lignes 8 et 49 que la gestion des mécanismes d'exception a introduit des instructions pour activer et désactiver le gestionnaire d'exception de la méthode. Le point d'entrée du gestionnaire d'exception de la méthode est le bloc de base `BasicBlock#6`. Le gestionnaire d'exception va chercher l'exception levée et cherche le gestionnaire d'exception actif qui doit la gérer. La variable d'exception `e#1` contient une valeur booléenne et permet de déterminer si le premier gestionnaire d'exception est actif. Dans notre exemple, l'exception levée n'est pas gérée par le programmeur. Le bloc de base `BasicBlock#6` effectue le traitement nécessaire pour une exception non gérée (relacher le mutex).

4.3 Analyses et optimisations

Dans cette section nous décrivons les analyses et optimisations effectuées sur le langage de haut niveau de notre compilateur. La Figure 4.8 montre le cheminement suivi par le compilateur lors du processus d'analyse et d'optimisation. Le langage source et cible de cette phase est le langage JST. Cette phase est par conséquent optionnelle. Cependant, certaines analyses particulières générant les tables d'informations sur les classes de l'application sont requises. Une version future du compilateur pourrait permettre une génération de code naïve sans analyses ni optimisations complexes.

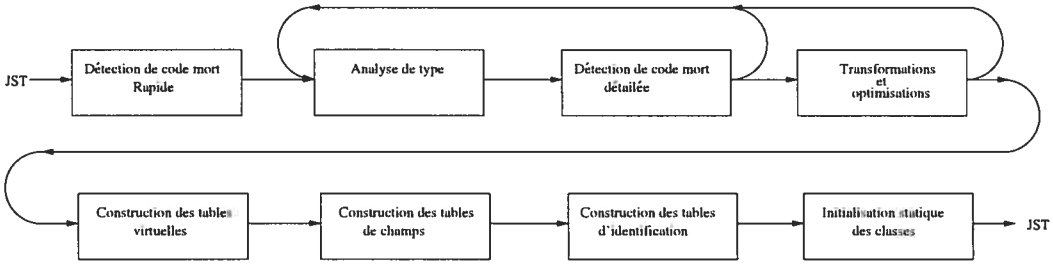

FIG. 4.8 – Processus d'analyse et d'optimisation

Les processus d'analyse de type, de détection de code mort détaillée et de transformations et optimisations s'effectuent en boucle puisqu'ils sont interdépendants. Une optimisation peut rendre inaccessible une partie de code qui devra être retirée pour que ce code mort ne pollue pas l'analyse de type. Le nombre d'itérations effectuées par le compilateur est paramétrable selon le niveau d'analyse et d'optimisation désiré par l'utilisateur.

4.3.1 Élimination conservatrice de code mort

La première étape de compilation consiste à minimiser la taille du code fourni au processus d'optimisation. Le compilateur, ayant effectué le chargement des classes de manière conservatrice, se retrouve avec un ensemble de code et de données dont une grande partie risque de ne pas être utilisée. Le processus rapide d'élimination de code mort élimine les classes, les méthodes et les champs inutilisés sans demander d'analyses complexes. L'algorithme est très conservateur puisqu'aucune information précise sur les types n'est disponible. L'algorithme employé est similaire à l'algorithme décrit dans [ST98].

L'algorithme se divise en trois étapes. La première étape initialise l'ensemble de code vivant, la seconde étape détermine itérativement le code vivant et la dernière étape élimine le code mort. La Figure 4.9 décrit l'algorithme employé. La fonction prend en paramètre un ensemble de définition de classe et retourne cet ensemble nettoyé du code mort. Cet algorithme est à point

fixe, conservateur et vorace.

Un algorithme à point fixe permet de déterminer itérativement les ensembles solutions du problème. L'algorithme réitère tant que les ensembles s'élargissent et arrête lorsqu'un état stable est atteint. Comme les ensembles ne décroissent jamais, l'algorithme effectue à chaque itération un progrès et, par conséquent, est un algorithme vorace.

fastclean(C) : $Classes \rightarrow Classes$

$Lives \leftarrow [\,]$
$Mark(C[java/lang/Object])$
$Mark(C[java/lang/String])$
$Mark(C[java/lang/Class])$
$Mark(C[java/lang/ClassLoader])$
$Mark(C[java/lang/\ <DefaultClassLoader>])$
$Mark(C[java/lang/\ <array>])$
$Mark(C[java/lang/Thread])$
$Mark(C[\ll MainClass \gg])$
$Mark(C[\ll MainClass \gg].Methods[main([Ljava/lang/String;)V])$
`do`
  $mark \leftarrow false$
  $\mathcal{MC}(c)$ $\quad \forall c \in C : c \in Lives$
  $\mathcal{MM}(m)$ $\quad \forall c \in C, \forall m \in c.Methods : c \in Lives \wedge m \in Lives$
`while` $mark ==$ `true`
$C \leftarrow C/\{c\}$ $\quad \forall c \in C : c \notin Lives$
$c.Fields \leftarrow c.Fields/\{f\}$ $\quad \forall c \in C, \forall f \in c.Fields : f \notin Lives$
$c.Methods \leftarrow c.Methods/\{m\}$ $\quad \forall c \in C, \forall m \in c.Methods : m \notin Lives$

c1 extends $c2$: $Classes \times Classes \rightarrow boolean$

$c1 == c2 \ \vee$ c1.$super$ extends $c2 \ \vee$
$\exists i \in c1.Interfaces : i$ extends $c2$

$Mark(x)$: $Classes \cup Fields \cup Methods \rightarrow VOID$

`if` $x \notin Lives$ $\left\{ \begin{array}{l} Lives \leftarrow Lives \cup \{x\} \\ mark \leftarrow \texttt{true} \end{array} \right.$

$\mathcal{MC}(c)$: $Classes \rightarrow VOID$

$Mark(c.super)$
$Mark(i)$ $\quad \forall i \in c.interfaces$
$Mark(c.Methods[<clinit>\ ()V])$
$Mark(c.Methods[run()V])$
$Mark(c.Methods[finalize()V])$

$$
\begin{array}{lll}
\mathcal{MM}(m): & Methods \rightarrow VOID & \\
& Mark(m) & \forall c' \in C, \forall m' \in c'.Methods : \\
& & c' \text{ extends } m.class \ \wedge \\
& & m.name == m'.name \ \wedge \\
& & m.descriptor == m'.descriptor \\
& \mathcal{MI}(i) & \forall i \in m.Instructions
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{MI}(i): & Instructions \rightarrow VOID & \\
& \text{i is fieldset} \\
& \text{i is fieldref} & \left\{ \begin{array}{l} Mark(i.class) \\ Mark(i.field) \end{array} \right. \\
& \text{i is methodref} & \left\{ \begin{array}{l} Mark(i.class) \\ Mark(i.method) \end{array} \right. \\
& \text{i is instanceof} \\
& \text{i is checkcast} & \left\{ \begin{array}{l} Mark(i.params[1]) \end{array} \right. \\
& \text{i is new} & \left\{ \begin{array}{l} Mark(i.params[0]) \end{array} \right.
\end{array}
$$

FIG. 4.9 – Algorithme conservateur d'élimination du code mort

L'algorithme suppose qu'initialiement tout le code est mort et par conséquent que l'ensemble de code vivant est vide. Par la suite, les classes nécessaires au fonctionnement de base de l'environnement virtuel sont ajoutées à l'ensemble de code vivant ainsi que la classe principale et sa méthode `main`. À ce stade, l'ensemble de code vivant est initialisé.

L'algorithme ajoute itérativement à l'ensemble de code vivant le code nécessaire pour résoudre les dépendendances de l'ensemble de code vivant. Il existe plusieurs sortes de dépendances et elles sont souvent conservatrices. L'algorithme s'arrête lorsqu'il n'a pu déterminer aucune dépendance manquante. Il fait ainsi converger l'ensemble de code vivant vers un ensemble consistant.

Après avoir déterminé, de manière conservartice, l'ensemble de code vivant, l'algorithme soustrait le complément de cet ensemble à l'ensemble global de code. Le code mort est ainsi éliminé.

Nos expériences sur des programmes typiques ont indiqué que cette étape élimine une grande partie du code inutilement chargé (près de 60%). Nous discutons des performances de l'analyse de code mort à la section 4.4 (voir tableau 4.3). Cette phase n'est pas obligatoire puisque le code mort sera de nouveau éliminé dans une future phase avec un algorithme plus précis. Son existence est justifiée par la complexité algorithmique des analyses qui suivent rendant le temps

d'analyse coûteux.

4.3.2 Analyse de type

Cette étape de compilation permet de déterminer les types et de borner le domaine des variables par exemple, qu'une variable de type référence ne peut être *null* ou bien qu'une variable entière contient seulement des positifs. Les informations fournies par l'analyse de type permettent au compilateur d'effectuer de meilleures optimisations puisqu'il peut tenir compte du contenu des variables manipulées. Le langage Java, étant dynamique et modulaire, ne permet pas de faire une analyse de type statique exacte puisque le système de type peut varier dynamiquement. Même le langage Java* sans ces propriétés dynamiques ne le permet pas puisqu'il est modulaire et possède des librairies externes dont l'information n'est pas disponible. Pour parvenir à notre but, nous avons ajouté comme entrée à l'analyse de type le langage NAT permettant d'exprimer les effets de bord des librairies externes. Cela rend disponible l'information nécessaire à l'analyse de type statique et globale plus précise.

Algorithme d'analyse

L'algorithme employé pour l'analyse de type est un algorithme à point fixe. Son fonctionnement consiste à prétendre initialement que les variables n'ont aucune valeur admissible puis agrandir itérativement l'ensemble des valeurs admises tant que toutes les valeurs possibles du type n'y sont pas représentées. La condition d'arrêt est atteinte lorsqu'aucun ensemble n'a été agrandi lors d'une itération ; le système est alors stable et consistant. Comme l'algorithme ne fait qu'agrandir les domaines, il n'y a jamais de pas en arrière ; il est garanti d'arrêter puisque les domaines que nous utilisons sont finis. La Figure 4.10 montre l'évolution des domaines selon les itérations. Nous constatons l'atteinte d'un état stable après quatres itérations. L'analyse a déterminé que les variables x, y et z peuvent contenir des références vers des instances des classes A et B.

```
Object x = new A;
Object y = new B;
Object z;
while (z != y) {
  z = x;
  x = y;
  y = z;
}
```

iter	x	y	z
0	{ }	{ }	{ }
1	{ A }	{ B }	{ }
2	{ A B }	{ B }	{ A }
3	{ A B }	{ A B }	{ A B }
4	{ A B }	{ A B }	{ A B }

FIG. 4.10 – Analyse de type par l'algorithme du point fixe

Domaine des références

Les algorithmes d'analyse de type cherchent à faire converger, de manière conservatrice, un ensemble représentant le domaine réel d'un type vers un ensemble voisin surensemble de ce domaine [ASU86, App99]. Généralement, les possibilités de type sont représentées par un treillis et l'ensemble représentant un type par un noeud du treillis. Dans [GHM00] le treillis est l'arbre d'héritage des classes. La Figure 4.11 représente un sous-ensemble du treillis pour Java. Par exemple, si une référence peut contenir un objet de type `java.lang.String` ou un objet de type `java.lang.Number` un ensemble représentatif du domaine serait `java.lang.Object`. L'ensemble obtenu est représentatif du domaine mais inclut aussi d'autres types tel un objet de type `java.lang.Error`.

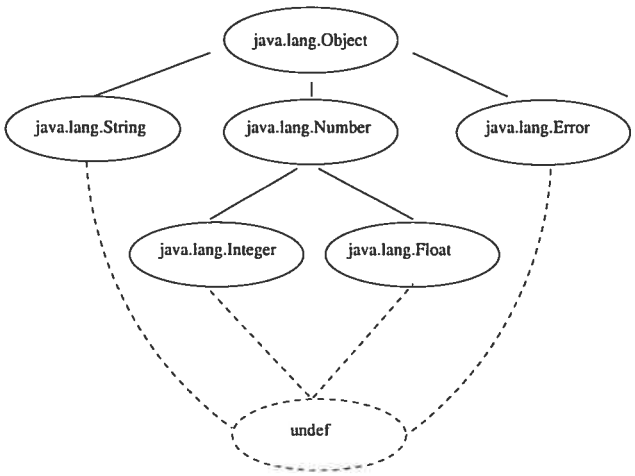

FIG. 4.11 – Treillis par hiérarchie de classe

Cette représentation du domaine revient à trouver la classe de base commune à toutes les instances potentielles.

Notre compilateur effectue une analyse de type plus précise qui n'élargit pas de manière conservatrice l'ensemble. Le treillis choisi est l'ensemble des sous-ensembles des classes. L'ensemble obtenu pour un type est la représentation exacte des données potentielles du domaine. Par exemple, si une référence peut contenir un objet de type `java.lang.String` ou un objet de type `java.lang.Number` un ensemble représentatif du domaine serait { `java.lang.String` ∪ `java.lang.Number` }.

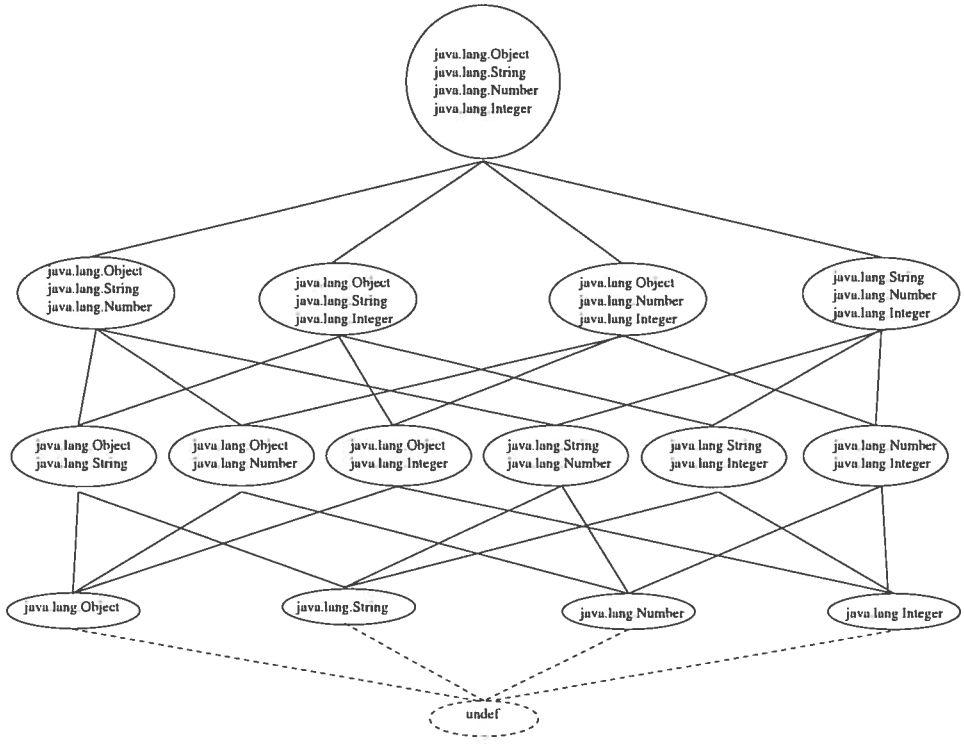

FIG. 4.12 – Treillis de sous-ensembles de classes

Cette représentation revient à énumérer toutes les classes potentiellement présentes dans le domaine d'un type.

Domaines numériques

Les types numériques sont connus statiquement lors de l'importation du *bytecode*. Pour des raisons de performance que nous discutons plus loin, notre compilateur prévoit une analyse de domaine des entiers. Les domaines numériques entiers possèdent des bornes *min* et *max* limitant les valeurs possibles du domaine. L'Annexe C fournit les règles d'analyse de type conservatrices selon les opérateurs arithmétiques et logiques rencontrés en respectant le standard des calculs de Java.

Les calculs en nombres flottants pourraient être sujets à cette analyse mais comme il est difficile de prédire les erreurs sur la précision des calculs flottants, nous avons préféré ne pas les effectuer et d'être ainsi très conservateur.

Quoique notre prototype effectue une analyse détaillée, certains algorithmes présentent des

améliorations offrant une meilleure précision. L'algorithme présenté par [Pat95] divise le domaine et conserve une probabilité pour chaque division. Ces probabilités sont utilisées lors de prédictions effectuées par le compilateur améliorant la qualité de l'information. L'analyse de type pourrait différencier les types par leurs sites de création. La version actuelle de notre compilateur n'associe pas de probabilités aux éléments de l'ensemble et ne différencie pas les sites de création.

Discussion sur l'analyse de type

L'analyse de type globale effectuée sur la représentation intermédiaire permet au compilateur d'estimer le comportement potentiel du programme. Nous pouvons observer le résultat de l'analyse de type en étudiant la représentation intermédiaire aussitôt après avoir effectué cette analyse et avant toute autre transformation.

L'analyse de type que nous avons implantée fournit une bonne quantité d'informations qu'un compilateur dynamique ne pourrait pas obtenir. Les domaines des références sont souvent bornés à un type effectif (voir la Figure 3.6). Par type effectif, nous entendons la connaissance exact du domaine. La précision des domaines des références permet de réaliser plusieurs optimisations dont nous discutons dans la Section 4.3.5.

Les domaines numériques entiers offrent une moins bonne précision que les domaines de référence qui sont exacts. L'imprécision vient de leur modèle d'analyse. En effet, l'analyse n'emploie que deux bornes pour représenter un domaine.

Les coûts de compilation associés à l'analyse numérique proviennent de la complexité de la tâche. Prenons pour exemple la boucle montrée à la Figure 4.13 et l'algorithme de convergence à point fixe avec les règles d'analyse décrites dans l'Annexe C. Dans ce cas, l'algorithme devrait ré-itérer 50 000 fois avant d'obtenir un état stable puisque le domaine de la variable s'incrémente à chaque itération.

```
int i = 1;
while(i < 50000) {
  [...]
  i = i++;
}
```

FIG. 4.13 – Code limitant la convergence d'un domaine numérique

Pour éviter ces cas, des amplificateurs de domaine sont ajoutés dans le processus d'itération pour accélérer leur convergence, exagérant toutefois le domaine évalué. L'amplificateur est initialement petit, et augmente exponentiellement par rapport au nombre d'itérations de l'ana-

lyse. Lorsqu'un domaine numérique est élargie selon les règles d'analyse, l'algorithme amplifie l'élargissement du domaine selon l'amplificateur. L'impact des amplificateurs est une diminution de la précision de l'analyse. Lorsqu'un domaine numérique dépasse la capacité de son type, le compilateur doit le borner par l'infini négatif et l'infini positif (*-inf,+inf*). Comme Java spécifie le résultat des calculs avec dépassement de capacité, dès qu'une limite (positive ou négative) est franchise, le compilateur ne peut plus borner le domaine avec simplement deux bornes et doit rester conservateur en le bornant à l'infini. Les domaines bornés à l'infini, n'offrant aucune information pertinente, montrent qu'il y a place à amélioration dans notre analyse.

Il est difficile d'obtenir une bonne précision sur un calcul sans l'effectuer. Même l'emploi de variables symboliques pour obtenir des contraintes sur les domaines est complexe et offre une analyse imprécise. La Figure 4.14 montre un exemple où les deux variables employées dans une boucle convergent vers l'ensemble indéfini. Même l'emploi de constantes symboliques et d'un système de résolution de contraintes ne pourrait résoudre ce problème sans simuler l'exécution car un débordement se produit lors du calcul avant que la condition de boucle soit vérifiée. L'emploi de constantes symboliques et de systèmes d'équations est aussi contraint par des limites de précision.

```
int i = 2;
int j = 5000;
while(i < j) {
  i = i * i;
  j = 2 * j;
}
```

FIG. 4.14 – Code limitant l'analyse par système de contraintes

Généralement, les compilateurs effectuent une analyse de domaine numérique restreinte qui se limite à la propagation des constantes et à l'évaluation des expressions constantes. Les gains de performance obtenus par une meilleure analyse de domaine ne sont pas significatifs lorsque l'architecture ciblée offre les mêmes contraintes d'évaluation que le langage compilé. Les compilateurs n'essaient pas d'aller chercher des gains avec ces analyses, car les coûts d'analyse ne justifient pas les gains de performance obtenus. Nous discutons de l'importance de cette analyse pour le compilateur J2S dans le chapitre 5.

Notre compilateur n'effectuant que la 0-CFA (Control Flow Analysis) [Muc97], l'entrée et la sortie d'une méthode sont des points communs d'analyse à tous les sites d'appel de cette même méthode. Il n'y a pas de différenciation de l'analyse selon le site d'appel.

```
int f(int n) {
  return n;
}

int test() {
  return f(2) + f(6);
}
```

FIG. 4.15 – Limite de la 0-CFA

La Figure 4.15 montre un exemple où la 0-CFA limite l'analyse de type. La méthode `test` effectue deux appels à la méthode `f` avec des valeurs distinctes. Comme l'algorithme ne tient pas compte de la provenance de l'appel à la méthode `f`, le domaine entier de la variable `n` et le domaine entier de la valeur de retour seront bornés entre 2 et 6. Si notre compilateur effectuait la 1-CFA, il aurait pu déterminer les domaines des deux appels de méthode. Les domaines des appels de méthode auraient été respectivement bornés à 2 et à 6.

Nous observons une pollution de l'analyse de type dans les librairies fortement employées telles les tables de dispersion (*hashtable*) de Java. Par exemple, s'il existe deux instances de tables de dispersion contenant deux types d'objets différents dans une même application, notre compilateur ne peut différencier le type de sortie de la méthode `get` de la table de dispersion.

4.3.3 Élimination précise du code mort

Les résultats précis de l'analyse de type sur les variables de référence permettent de nettoyer le code. Les types effectifs de toutes les variables sont connus et le compilateur peut procéder à une nouvelle passe d'élimination de code mort. Cette passe, contrairement à la première, est basée sur une analyse riche.

La Figure 4.16 montre les modifications apportées à l'algorithme de code mort conservateur de la Figure 4.9.

$$\begin{array}{ll} \mathcal{MM}(m): & Methods \rightarrow VOID \\ & \mathcal{MI}(i) \quad \forall i \in m.Instructions \end{array}$$

$$\mathcal{MI}(i): \quad Instructions \rightarrow VOID$$

i is fieldset / i is fieldref : $Mark(i.class)$, $Mark(i.field)$

i is methodref :
$$\begin{cases} Mark(c) & \forall c \in \mathcal{T}_{\text{i.this}} \\ Mark(m) & \forall c \in \mathcal{T}_{\text{i.this}}, \forall m \in c.Methods : \\ & i.method.name = m.name \end{cases}$$

i is instanceof / i is checkcast : $Mark(i.params[1])$

i is new : $Mark(i.params[0])$

FIG. 4.16 – Modifications à l'algorithme d'élimination du code mort

La différence majeure entre les deux algorithmes réside dans la précision de l'analyse des accès aux objets (méthode, champs et création). L'analyse de type fournit les types exacts et l'analyse de code mort n'a plus à être aussi conservatrice.

Cette deuxième phase d'élimination de code mort enlève, dans nos programmes tests, environ la moitié des classes et méthodes restantes. Nous discutons des performances de l'analyse de code mort à la section 4.4 (voir tableau 4.3).

4.3.4 Analyse de variables vivantes

Il est à noter que l'analyse de code mort décrit dans la section précédente n'est pas appliquée au corps des méthodes. L'algorithme a un impact global et pour limiter la complexité du traitement, l'analyse de code mort pour le corps des méthodes a été isolée.

Pour chaque variable et expression d'une méthode, l'algorithme à point fixe détermine si elles sont vivantes. Une expression est vivante si elle contient une donnée nécessaire à l'exécution du programme ou modifie l'état ou le comportement du programme.

L'algorithme est similaire à l'algorithme employé pour l'analyse de type et à l'algorithme de détection de code mort. Il suppose toutes les expressions mortes et itérativement marque les expressions vivantes jusqu'à ce qu'un état stable soit atteint. Nous ne décrivons pas l'algorithme car il est défini dans le chapitre 10 de [App99]. Après cette analyse, le compilateur connaît les variables et les expressions vivantes dans une méthode.

4.3.5 Transformation et substitution

Dans cette section nous décrivons les optimisations et transformations de programme effectuées par notre compilateur sur son langage intermédiaire. Nous étudions l'évolution du code de la Figure 4.17.

```
1. {
2.   let v#1 0
3.   jump BasicBlock#1 ( l#0 v#1 )
4. }
5.
6. BasicBlock#0( l#0 l#1 ) {
7.   let v#1 fieldref foo . CST0
8.   let v#2 fieldref foo . CST1
9.   let v#3 v#2
10.  let v#4 add v#1 v#3
11.  let v#5 l#0
12.  let v#6 v#5
13.  let v#7 checknull v#5
14.  let v#8 methodref foo.bar v#7 ( v#4 )
15.  let v#9 v#1
16.  let v#10 checknull v#6
17.  let v#11 methodref foo.bar v#10 ( v#9 )
18.  let v#12 add l#1 1
19.  let l#1 v#12
20.  jump BasicBlock#1( l#0 l#1 )
21. }
22.
23. BasicBlock#1( l#0 l#1 ) {
24.   if (lt l#1 10)
25.     jump BasicBlock#0( l#0 l#1 )
26.   else
27.     jump BasicBlock#2( l#0 l#1 )
28. }
29.
30. BasicBlock#2( l#0 l#1 ) {
31.   return l#1
32. }
```

FIG. 4.17 – Code d'exemple utilisé pour les substitutions

Il est à noter que le choix de cet exemple se justifie car il se prête aux optimisations que nous allons décrire et est facilement optimisable. Le but de cet exemple est de montrer un cas simple d'optimisation sans amener le lecteur dans les complications des cas d'exception.

Les optimisations décrites ci-dessous sont effectuées en boucle tant que des changements persistent. L'ordre d'exécution n'a pas d'impact sur le résultat mais il accélère généralement l'exécution globale de toutes les transformations.

Substitution des expressions constantes

Après l'analyse de type, les domaines des variables sont bornés et certains domaines sont bornés à une constante. Par exemple, une méthode qui prend toujours en paramètre un élément *null* verra le domaine de son paramètre borné à *null* et par conséquent est une expression constante. Le compilateur peut substituer une expression constante par sa valeur lorsqu'elle n'a aucun effet de bord.

Supposons que l'analyse de type a borné le domaine des champs statiques `foo.CST0` et `foo.CST1` aux valeurs 0 et 1 respectivement. La transformation est effectuée aux lignes 7 et 8 et le code résultant est donné à la Figure 4.18.

```
1.  {
2.    let v#1 0
3.    jump BasicBlock#1 ( l#0 v#1 )
4.  }
5.
6.  BasicBlock#0( l#0 l#1 ) {
7.  •  let v#1 0
8.  •  let v#2 1
9.    let v#3 v#2
10.   let v#4 add v#1 v#3
11.   let v#5 l#0
12.   let v#6 v#5
13.   let v#7 checknull v#5
14.   let v#8 methodref foo.bar v#7 ( v#4 )
15.   let v#9 v#1
16.   let v#10 checknull v#6
17.   let v#11 methodref foo.bar v#10 ( v#9 )
18.   let v#12 add l#1 1
19.   let l#1 v#12
20.   jump BasicBlock#1( l#0 l#1 )
21. }
22.
23. BasicBlock#1( l#0 l#1 ) {
24.   if (lt l#1 10)
25.     jump BasicBlock#0( l#0 l#1 )
26.   else
27.     jump BasicBlock#2( l#0 l#1 )
28. }
29.
30. BasicBlock#2( l#0 l#1 ) {
31.   return l#1
32. }
```

FIG. 4.18 – Code après substitution des constantes

Propagation des constantes et des variables

La propagation des constantes et des variables se produit pour chaque bloc de base. L'algorithme consiste à parcourir le bloc de base du début à la fin en ajoutant dans un ensemble associatif les variables avec leur nouvelle valeur.

```
1.  {
2.    let v#1 0
3.  •  jump BasicBlock#1 ( l#0 0 )
4.  }
5.
6.  BasicBlock#0( l#0 l#1 ) {
7.    let v#1 0
8.    let v#2 1
9.  •  let v#3 1
10.   let v#4 add 0 1
11.   let v#5 l#0
12.   let v#6 v#5
13.   let v#7 checknull v#5
14.   let v#8 methodref foo.bar v#7 ( v#4 )
15. •  let v#9 0
16. •  let v#10 checknull v#5
17. •  let v#11 methodref foo.bar v#10 ( 0 )
18.   let v#12 add l#1 1
19.   let l#1 v#12
20.   jump BasicBlock#1( l#0 l#1 )
21. }
22.
23. BasicBlock#1( l#0 l#1 ) {
24.   if (lt l#1 10)
25.     jump BasicBlock#0( l#0 l#1 )
26.   else
27.     jump BasicBlock#2( l#0 l#1 )
28. }
29.
30. BasicBlock#2( l#0 l#1 ) {
31.   return l#1
32. }
```

FIG. 4.19 – Code après propagation des constantes et des variables

Dans notre exemple, l'expression constante 1 affectée à la variable v#2 à la ligne 8 est propagée à l'expression de la ligne 9. pour donner le code de la Figure 4.19. Seules les constantes et les variables temporaires sont sujettes à cette optimisation. Rappelons qu'une variable temporaire ne possède qu'un site d'affectation (voir SSA ci-dessous). Les variables locales et les variables mutables peuvent subir des modifications à plusieurs sites d'affectation ou devenir incohérentes suite à une exception.

Évaluation d'expressions constantes

Les expressions arithmétiques et logiques constantes sont pré-évaluées et sont remplacées par leur valeur.

```
1.  {
2.    let v#1 0
3.    jump BasicBlock#1 ( l#0 0 )
4.  }
5.
6.  BasicBlock#0( l#0 l#1 ) {
7.    let v#1 0
8.    let v#2 1
9.    let v#3 1
10. •  let v#4 1
11.   let v#5 l#0
12.   let v#6 v#5
13.   let v#7 checknull v#5
14.   let v#8 methodref foo.bar v#7 ( v#4 )
15.   let v#9 0
16.   let v#10 checknull v#5
17.   let v#11 methodref foo.bar v#10 ( 0 )
18.   let v#12 add l#1 1
19.   let l#1 v#12
20.   jump BasicBlock#1( l#0 l#1 )
21. }
22.
23. BasicBlock#1( l#0 l#1 ) {
24.   if (lt l#1 10)
25.     jump BasicBlock#0( l#0 l#1 )
26.   else
27.     jump BasicBlock#2( l#0 l#1 )
28. }
29.
30. BasicBlock#2( l#0 l#1 ) {
31.   return l#1
32. }
```

FIG. 4.20 – Code après l'évaluation d'expressions constantes

Nous observons que l'expression `add 0 1` à la ligne 10 a été remplacée par le résultat de son évaluation. Le compilateur doit s'assurer qu'aucun effet de bord n'est retiré ou modifié. Par exemple, la division d'une constante par zéro doit générer une exception.

Transformation en forme SSA

La forme SSA (Static Single Assignement) décrite dans le livre d'Appel [App99] transforme le programme de sorte qu'une variable ne possède qu'un site d'affectation pour plusieurs sites d'utilisation. Notre compilateur effectue partiellement cette transformation pour les variables locales car il n'effectue pas les transformations sur les variables mutables pour limiter la complexité des analyses. Les variables locales sont substituées à partir d'un site d'affectation par des variables temporaires, lorsque possible. Ainsi la transformation en forme SSA s'effectue automatiquement.

```
1.  {
2.    let v#1 0
3.    jump BasicBlock#1 ( l#0 0 )
4.  }
5.
6.  •BasicBlock#0( v#13 v#14 ) {
7.    let v#1 0
8.    let v#2 1
9.    let v#3 1
10.   let v#4 1
11. •  let v#5 v#13
12.   let v#6 v#5
13.   let v#7 checknull v#5
14.   let v#8 methodref foo.bar v#7 ( v#4 )
15.   let v#9 0
16.   let v#10 checknull v#5
17.   let v#11 methodref foo.bar v#10 ( 0 )
18. •  let v#12 add v#14 1
19. •  let v#15 v#12
20. •  jump BasicBlock#1( v#13 v#15 )
21.  }
22.
23. •BasicBlock#1( v#16 v#17 ) {
24. •  if (lt v#17 10)
25. •    jump BasicBlock#0( v#16 v#17 )
26. •  else
27. •    jump BasicBlock#2( v#16 v#17 )
28.  }
29.
30. •BasicBlock#2( v#18 v#19 ) {
31. •  return v#19
32.  }
```

FIG. 4.21 – Code après la transformation en forme SSA

L'algorithme employé est similaire à l'algorithme de propagation des constantes et des variables. En fait, notre compilateur l'effectue dans la même passe. Nous l'avons isolé dans ce document pour mettre en évidence la technique employée pour obtenir du code sous la forme SSA.

Dans notre exemple, la définition de la variable `l#1` de la ligne 6 est écrasée par la définition de la ligne 19 et par conséquent, le compilateur peut séparer le domaine où la variable est vivante en deux domaines indépendants. Cette division permettra des analyses plus précises et localisées.

Substitution d'expressions équivalentes

Les substitutions d'expressions équivalentes remplacent une expression par une autre qui lui est équivalente et plus efficace ou plus simple à analyser. Les expressions équivalentes n'affectent qu'une instruction à la fois, et ne demandent aucune analyse détailée.

```
1.  let v#2 rem v#1 16   ⇒   1.  let v#2 and v#1 15
```

FIG. 4.22 – Substitution d'une expression équivalente

La Figure 4.22 montre une substitution d'expression équivalente lorsque le domaine numérique de `v#1` est borné positivement.

Élimination d'expressions communes

La détection d'expressions communes demande une analyse de dépendances de données et une analyse du flux de contrôle. Les algorithmes pour éliminer les expressions communes sont décrits dans [App99, ASU86, Muc97].

Ces analyses sont importantes lors de la compilation de Java pour éliminer les tests de sûreté redondants.

```
1.  {
2.    let v#1 0
3.    jump BasicBlock#1 ( l#0 0 )
4.  }
5.
6.  BasicBlock#0( v#13 v#14 ) {
7.    let v#1 0
8.    let v#2 1
9.    let v#3 1
10.   let v#4 1
11.   let v#5 v#13
12.   let v#6 v#5
13.   let v#7 checknull v#5
14.   let v#8 methodref foo.bar v#7 ( v#4 )
15.   let v#9 0
16. •  let v#10 v#7
17.   let v#11 methodref foo.bar v#10 ( 0 )
18.   let v#12 add v#14 1
19.   let l#15 v#12
20.   jump BasicBlock#1( v#13 v#15 )
21. }
22.
23. BasicBlock#1( v#16 v#17 ) {
24.   if (lt v#17 10)
25.     jump BasicBlock#0( v#16 v#17 )
26.   else
27.     jump BasicBlock#2( v#16 v#17 )
28. }
29.
30. BasicBlock#2( v#18 v#19 ) {
31.   return v#19
32. }
```

FIG. 4.23 – Code après l'élimination d'expressions communes

La Figure 4.23 montre un exemple où nous pouvons éliminer un des deux `checknull`, celui à la ligne 16, puisqu'il est déjà effectué sur la même référence à la ligne 13. Les tests redondants sont fréquents en Java.

Substitution de groupes d'expressions

À ce stade, le compilateur possède un graphe de dépendance du programme *(PDG : Program Dependance Graph)* [BKM93] qui consiste en une analyse de dépendances de données et une analyse du flux de contrôle. Le compilateur effectue des transformations sur plusieurs expressions en respectant le PDG pour générer du code optimisé. Ces transformations sont fréquentes en Java*.

Par exemple, il est fréquent qu'un programme Java, en effectuant du traitement de chaînes

de caractères, copie ces caractères d'un tableau à un autre. L'extraction d'un caractère d'un tableau demande une expansion du type vers un entier 32 bits signé et la remise du caractère dans un autre tableau demande la retransformation de l'entier en caractère.

```
1. let v#1 checknull l#0                                    ⇒  1. let v#1 checknull l#0
2. let v#2 fieldref java.lang.<array> <elements> v#1           2. let v#2 fieldref java.lang.<array> <elements> v#1
3. let v#3 arrayref v#2 5                                      3. let v#3 arrayref v#2 5
4. let v#4 B2I v#3                                             4. let v#4 B2I v#3
5. let v#5 checknull l#1                                       5. let v#5 checknull l#1
6. let v#6 fieldref java.lang.<array> <elements> v#5           6. let v#6 fieldref java.lang.<array> <elements> v#5
7. let v#7 I2B v#4                                             7. let v#7 v#3
8. arrayset v#6 5 v#7                                          8. arrayset v#6 5 v#7
```

FIG. 4.24 – Substitution d'un groupe d'expressions

La Figure 4.24 montre une optimisation de substitution d'un groupe d'expressions. L'expression de conversion à la ligne 7 est remplacée et prend directement sa valeur à la source de l'instruction de conversion de la ligne 4 qui doit rester intacte au cas où d'autres expressions en seraient dépendantes.

En modifiant la source de l'instruction, les variables temporaires employées pour faire les transformations deviennent généralement des expressions mortes et sont nettoyées par l'algorithme d'élimination de code mort. L'effet de cette transformation est de ne plus effectuer les conversions inutiles de *byte* vers *int* suivi de *int* vers *byte* puisqu'aucune perte d'information n'est possible.

Élimination des expressions mortes

L'élimination des expressions mortes utilise l'analyse d'expressions vivantes pour éliminer ou remplacer les expressions mortes. Les expressions mortes possédant des effets de bord ne peuvent pas être éliminées simplement.

```
1.  {
2.    jump BasicBlock#1 ( l#0 0 )
3.  }
4.
5.  BasicBlock#0( v#13 v#14 ) {
6.    let v#4 1
7.    let v#5 v#13
8.    let v#7 checknull v#5
9.  •  methodref foo.bar v#7 ( v#4 )
10.   let v#10 v#7
11. •  methodref foo.bar v#10 ( 0 )
12.   let v#12 add v#14 1
13.   let l#15 v#12
14.   jump BasicBlock#1( v#13 v#15 )
15. }
16.
17. BasicBlock#1( v#16 v#17 ) {
18.   if (lt v#17 10)
19.     jump BasicBlock#0( v#16 v#17 )
20.   else
21.     jump BasicBlock#2( v#16 v#17 )
22. }
23.
24. BasicBlock#2( v#18 v#19 ) {
25.   return v#19
26. }
```

FIG. 4.25 – Code après l'élimination du code mort

Nous observons dans la Figure 4.25 que l'algorithme élimine les variables inutilisées et remplace certaines expressions avec effet de bord (`methodref`) par une expression équivalente ne possédant pas les variables mortes. La valeur de retour des appels de méthode étant morte, le compilateur peut éliminer la liaison mais doit conserver l'appel de méthode.

Il est à noter que le code présenté est encore à la première itération d'optimisation, et que sans une autre passe de propagation de constantes et de variables beaucoup de variables mortes ne sont pas détectées ou optimisées ; la variable `v#4` est morte et la variable `v#5` doit être substituée par la variable `v#13`.

Substitution de branchements

Les optimisations précédentes effectuent des changements sur les expressions sans modifier le flux de contrôle de l'application. La première structure de contrôle modifiée est les branchements. Le compilateur effectue plusieurs types d'optimisations de branchement. Par exemple, les branchements jamais atteints sont éliminés, les branchements communs avec ou sans paramètres distincts sont remplacés et les branchements uniques sont insérés sur leur site de branchement.

```
1. BasicBlock#0( l#0 v#1 v#2 ) {
2.   if( 0 )
3.     jump BasicBlock#1 ( l#0 0 )
4.   else
5.     jump BasicBlock#2 ( l#0 1 )
6. }
7.
8. BasicBlock#0( l#0 v#1 v#2 ) {
9.   if( lt v#1 v#2 )
10.    jump BasicBlock#1 ( l#0 0 )
11.  else
12.    jump BasicBlock#1 ( l#0 0 )
13. }
14.
15. BasicBlock#0( l#0 v#1 v#2 ) {
16.   let v#1 0
17.   jump BasicBlock#1 ( l#0 v#0 )
18. }
19.
```

⇒

```
1. BasicBlock#0( l#0 v#1 v#2 ) {
2.     jump BasicBlock#2 ( l#0 1 )
3. }
4.
5.
6.
7.
8. BasicBlock#0( l#0 v#1 v#2 ) {
9.     jump BasicBlock#1 ( l#0 0 )
10. }
11.
12.
13.
14.
15. BasicBlock#0( l#0 v#1 v#2 ) {
16.   let v#1 0
17.   return v#0  // inlining
18. }
19.
```

FIG. 4.26 – Substitutions de branchement

La Figure 4.26 montre des exemples d'optimisation de site de branchement.

L'insertion d'un bloc de base sur son site de branchement améliore la qualité des analyses et des optimisations possibles. Une conséquence de cette transformation est le déroulement des boucles internes.

Substitution des sites d'appel de méthode

La substitution de sites d'appel de méthode modifie les appels virtuels en appels directs lorsque l'analyse de type informe le compilateur que pour un site d'appel il n'y a qu'une méthode visée. Le gain réalisé par cette optimisation vient du fait que le code généré n'a pas à déterminer la méthode cible en passant par des tables de méthodes virtuelles. Par défaut, en Java, contrairement à C++, toutes les méthodes sont virtuelles. Par conséquent, les tables virtuelles ont souvent des méthodes ne nécessitant pas ce mécanisme. Cette optimisation offre un gain lors de l'exécution de l'application et permet de nettoyer les tables virtuelles.

Pour un appel de méthode virtuel, cette optimisation nécessite une analyse de type détaillée garantissant tous les types effectifs du domaine de l'objet référencé. Cette optimisation est difficilement réalisable sur les modèles de compilation séparé ou dynamique.

Une variante de cette optimisation modifie le site d'appel en une cascade de vérifications et d'appels de méthode. Cette optimisation n'est effectuée que lorsque le site d'appel possède peu de méthodes visées, que les méthodes visées semblent courtes et qu'elles semblent être de bonnes candidates pour l'optimisation d'insertion décrite ci-dessous. Cette variante peut être employée par d'autres modèles de compilation.

Insertion de corps de méthode

L'insertion du corps d'une méthode sur son site d'appel n'est valide que lorsque le site d'appel est un appel direct. L'insertion de corps de méthode consiste à insérer le code d'une méthode sur son site d'appel pour enlever le coût du mécanisme d'appel de méthode et pour rendre les analyses plus précises.

Notre compilateur supporte partiellement l'insertion des corps de méthode car l'algorithme complet n'a pu être terminé dans les limites de temps. Nous n'avons donc pas pu mesurer les gains de cette optimisation.

4.3.6 Analyses de finalisation et de synchronisation

Des analyses spécifiques à Java* déterminent des propriétés spécifiques au langage. Certains mécanismes comme la finalisation et la synchronisation sont coûteux dans les environnements d'exécution. Le compilateur détermine les classes nécessitant les mécanismes de synchronisation et de finalisation, éliminant ainsi les emplois inutiles de ces mécanismes et diminuant aussi le coût d'exécution.

4.3.7 Améliorations futures

Comme mentionné dans la Section 4.3.2, notre compilateur n'effectue que l'analyse 0-CFA. Pour une meilleure précision, une version effectuant des analyses N-CFA [ASU86, App99] sur les sites fréquents serait à envisager, car cette analyse est plus précise que la 0-CFA.

Notre compilateur ne supporte pas d'algorithmes de déroulement de boucles. Les boucles déroulées sont un effet de bord de l'optimisation d'insertion de branchement. Vu les gains de ces algorithmes, des algorithmes comme celui de [Sar00] sont à prévoir dans les prochaines versions.

Pour obtenir de bons résultats en calcul numérique, les transformations de programme proposées par [AGMM00] sont prometteuses pour notre compilateur. Elles introduisent la notion de *safe zone* et *unsafe zone* sélectionnées dynamiquement permettant au compilateur d'optimiser différentes zones sous différentes contraintes.

Notre compilateur n'effectue aucune prédiction statique de la fréquence d'exécution du code (profilage). Plusieurs algorithmes [BL93, Won99, WL94, CJM+96, Sar89] proposent des techniques pour évaluer statiquement la fréquence d'exécution et des optimisations potentielles basées sur celle-ci.

Une reconstruction de structure de contrôle de plus haut niveau comme dans [Hec77] permettrait de meilleurs analyses du flux de contrôle.

La liste des travaux futurs pour un compilateur est toujours longue, voir infinie. Il y aura toujours des optimisations et des analyses à améliorer. Faute de place, nous ne développerons pas ici d'autres idées d'analyses ou d'optimisations.

4.4 Analyse de performance

Dans cette section, nous analysons les performances obtenues par la partie frontale et l'optimiseur (commun à nos deux versions de compilateurs).

4.4.1 Élimination de code mort

Le Tableau 4.3 montre les résultats des différentes étapes d'élimination de code mort sur notre représentation intermédiaire : le nombre de méthodes et de champs vivants après chaque passe d'analyse de code mort.

	Chargement du *bytecode*		Élimination de code mort rapide		Élimination de code mort détaillée			
					Itération 1		Itération 2	
Test	Méth.	Ch.	Méth.	Ch.	Méth.	Ch.	Méth.	Ch.
Fib	827	247	332	196	167	167	156	165
jBYTEmark	1072	497	535	358	366	325	347	323
Parallele	881	265	386	215	229	185	219	183

TAB. 4.3 – Résultat de l'élimination de code mort

Au chargement de l'ensemble des classes, notre compilateur ne charge que les classes nommées dans les tables de constantes des fichiers chargés. Ainsi, il obtient un sous-ensemble des librairies de Java. L'élimination de code mort rapide élimine plus de la moitié de ce code sans analyse détaillée.

Une analyse de type globale, qui d'obtient de l'information précise, permet d'éliminer encore la moitié du code vivant. Nous observons peu de code mort éliminé entre la première itération et la deuxième d'élimination de code mort précis qui nécessite à chaque itération une analyse globale). Nous observons que le temps d'analyse, et de compilation, augmente pour moins de

gain. Nous observons le même comportement pour les itérations suivantes d'élimination de code mort globale.

Notre compilateur obtient ainsi un sous-ensemble très restreint des librairies Java nécessaire pour le bon fonctionnement de l'application.

4.4.2 Temps de compilation

Notre compilateur effectue une compilation globale qui lui permet d'obtenir l'information de toutes les classes à compiler. Ce type de compilation entraîne un coût de compilation élevé. Nous avons mesuré le temps de compilation de notre compilateur pour évaluer le coût associé à une compilation globale.

Test	Chargement du bytecode	Exportation vers JST	Chargement des fichiers NAT	Analyses et Optimisations	
				1-itération	2-itérations
	(ms)	(ms)	(ms)	(ms)	(ms)
Fib	534	1629	67	65729	76463
jBYTEmark	1287	4074	96	3062031	3699440
Parallele	637	1630	72	89739	105698

TAB. 4.4 – Temps de compilation

Le Tableau 4.4 donne le temps de compilation requis par les différentes phases de notre compilateur. Nous observons que les premières phases du compilateur s'effectue rapidement et que le temps d'analyse et d'optimisation globale est élevé. Par exemple, le compilateur nécessite une minute d'analyse et optimisation globale pour compiler une application comme Fib (23 lignes). Notre compilateur augmente son temps de compilation pour obtenir de meilleures performances.

Nous expliquons le temps élevé d'analyse et d'optimisation globale de *jBYTEmark* par sa consommation élevée d'espace mémoire pour ses analyses. La machine hôte n'a pas assez de mémoire pour contenir l'ensemble de l'analyse et, par conséquent, nécessite l'utilisation de mémoire virtuelle sur disque dur.

Chapitre 5

Compilation vers Scheme

La première version de notre compilateur, J2S, génère du code Scheme à partir de notre représentation intermédiaire. Nous discutons dans ce chapitre ce choix de langage cible, nos expériences et les résultats obtenus. Il est à noter qu'un compilateur générant du code Scheme a déjà été implanté pour le langage Erlang [FL98] pour démontrer les avantages d'un compilateur qui a également pour cible Gambit. Nous avons employé le même modèle de compilateur.

5.1 Le langage

Scheme [CR91], une version épurée de Lisp, est un langage fonctionnel de haut niveau typé dynamiquement. Le langage Scheme permet différents styles de programmation : impératif, fonctionnel et orienté objet. Il contient une librairie de fonctions standards et possède un ramasse-miettes facilitant la gestion de la mémoire.

La tour des nombres offre des entiers de précision infinie, des nombres rationnels exacts, des nombres réels inexacts et des nombres complexes. Les nombres sont organisés en tour où chaque niveau est un sous ensemble du niveau précédent : $number \supseteq complex \supseteq real \supseteq rational \supseteq integer$. La représentation des nombres en Scheme est soit exact ou inexacte. Chaque ensemble de la tour des nombres est divisé en deux sous-ensembles qualifiant l'exactitude. Les nombres exacts représentent la valeur numérique exacte tandis que les nombres inexacts sont une approximation du nombre.

La Figure 5.1 montre des exemples du système numérique de Scheme.

```
(display (expt 2 64))  ;;- sortie : 18446744073709551616
(display (expt 2 128)) ;;- sortie : 340282366920938463463374607431768211456
(display (* 5 2/4))    ;;- sortie : 5/2
(display (* 5 0.5))    ;;- sortie : 2.5
(display (sqrt -2))    ;;- sortie : +1.4142135623730951i
```

FIG. 5.1 – Exemple de calculs en Scheme

Les variables ne possèdent pas de type. Ce sont plutôt les données qui ont un type. Les données sont des objets avec une identité et un type. Les fonctions sont des données, ce qui facilite le passage de fonction en paramètre, les fonctions de rappel (*callback*), la redéfinition de fonction et d'autres mécanismes complexes de gestion de fonction.

```
(define traiter-nombre
  (lambda (n)
    (display "Nombre :")
    (display n)
    (newline)))

(define traiter-texte
  (lambda (txt)
    (display "Texte :")
    (display txt)
    (newline)))

(define error
  (lambda (n)
    (display "error")
    (newline)))

(define x (read))                    ;;- Lecture de la donnée.

(define func
  (cond                              ;;- Détermine la fonction de traitement.
    ((number? x ) traiter-nombre)
    ((string? x ) traiter-texte)
    (else error)))

(func x)                             ;;- Appel de la fonction.

(map traiter-nombre '(3 5 4 2 3))  ;;- Appelle la fonction pour chaque nombre
```

FIG. 5.2 – Fonction d'ordre supérieur

Les fonctions possèdent une fermeture sur les variables libres *(true lexical scoping)*. Scheme permet l'encapsulation de données à l'aide du mécanisme de fermeture. Les fermetures permettent de créer des objets avec des données distinctes pour chaque instance. À l'aide de ce mécanisme, nous pouvons effectuer de la programmation orientée objet comme le montre la Figure 5.3.

```
(define make-person
  (lambda (name age gender)
    (lambda (msg)                    ;;- Retourne une fermeture
      (cond
       ((eq? msg 'name)     name  )
       ((eq? msg 'age)      age   )
       ((eq? msg 'gender)   gender)
       (else (error "oopss"))))))

(define p1 (make-person "paul" 25 'male))
(define p2 (make-person "jill" 33 'female))

(write (p1 'age))    ;;- sortie : 25
(write (p2 'gender)) ;;- sortie : female
```

FIG. 5.3 – Programmation orienté-objet en Scheme

Le langage Scheme est un langage sûr et sécuritaire. Il limite le programmeur en l'empêchant d'employer des styles de programmation non-standards, et souvent d'effectuer des abus de langage, source de problèmes.

L'ajout des tests de sécurité, du typage dynamique et des fonctions d'ordre supérieur font de Scheme un langage généralement plus lent que les langages ne possédant pas ces fonctionnalités.

5.2 Gambit

Gambit [Fee98] est un compilateur et un interprète de code Scheme écrit par Marc Feeley. C'est un système mature et stable existant depuis 1989. La première version générait du code natif pour M68000 pour l'ordinateur parallèle GP1000 (128 processeurs avec mémoire partagée). Gambit est compact et efficace ; les vitesses sont comparables à celles des langages de bas niveau tel que C. Gambit-C est une variante récente de Gambit génèrant du code C qui sera compilé à l'aide d'un compilateur C pour l'architecture voulue. Le langage C étant portable, les programmes compilés par Gambit-C le sont facilement. Gambit-C fonctionne sur Windows, Unix et Macintosh. Il est facilement portable vers de nouveaux systèmes d'exploitation car il possède peu de dépendance face au système d'exploitation.

Gambit contient des consignes de compilation *(Pragmas)* pour optimiser le code. Ces consignes permettent d'éviter des tests de type insérés pour détecter les erreurs de programmation.

Le compilateur Gambit-C a introduit une nouvelle forme permettant les appels de fonction vers le langage C et C++. Ce mécanisme permet de définir des fonctions dépendantes du système d'exploitation ou de l'architecture non implantées dans les librairies de Scheme.

Gambit-C implante la tour des nombres de Scheme. De plus, il est possible d'accéder à des sous-ensembles des ensembles de la tour. Ainsi, les nombres entiers sont divisés en *fixnum* et *bignum*. L'attrait de la représentation *fixnum* est qu'il n'est pas nécessaire d'allouer un objet dans le tas pour contenir la valeur. Les calculs sur les *fixnums* qui ne causent pas de débordement peuvent se faire rapidement avec une ou deux intructions machine. Les *fixnums* sont représentés sur 30 bits, tandis que les *bignums* ne possèdent pas de limite. Lorsqu'un calcul dépasse la capacité des *fixnums*, le résultat est converti en *bignum* de manière transparente pour l'utilisateur.

La compilation vers Scheme permet de supposer que le compilateur Gambit-C effectuera les optimisations locales aux méthodes. Notre compilateur peut se concentrer sur les optimisations que Gambit-C n'effectue pas. De plus, notre compilateur hérite des propriétés et des fonctionnalités du langage Scheme telle la gestion de la mémoire.

5.3 Motivations

Le langage Java et le langage Scheme étant des langages dynamiques, sûr et sécurisé partagent plusieurs propriétés et fonctionnalités communes. Cette similarité nous permet de croire que les deux langages sont aptes à être compilés avec le modèle de couche de compilateur.

5.4 Génération du code Scheme

Le code Scheme est généré à partir de la représentation intermédiaire JST. La représentation intermédiaire étant proche de la syntaxe de Scheme, la génération de code est relativement simple. Nous ne décrivons pas les détails de la génération de code. Nous donnons l'idée du code Scheme généré pour le code JST.

5.4.1 Mécanisme et fonctionnement

Instructions élémentaires

Les instructions élémentaires sont représentées par un *let* comme le montre l'exemple ci-dessous. Le *let* donne un nom au résultat de l'instruction à l'intérieur du corps du *let*.

```
(let ((result (instr param1 param2))) ; cas general
  ...
)


(let ((v1 (+ l1 l2)))
  (let ((v2 (* v1 2)))
    (let ((v3 (java/lang/String/<new><I>Ljava/lang/String- v2)))
      v3)))
```

FIG. 5.4 – Représentation des instructions élémentaires en Scheme

Le choix des instructions dépend du résultat des analyses du compilateur. Par exemple, si le compilateur ne peut pas borner les paramètres d'une instruction arithmétique, des tests de débordement dynamiques doivent être présents pour éviter d'obtenir un nombre représenté en *bignum* qui ne respecte pas les limites des types de Java et les contraintes de débordement des calculs.

Blocs de base

Les instructions élémentaires sont regroupées en blocs de base. Nous représentons chaque bloc de base par une fonction et un *letrec* donne un nom aux fonctions. Les blocs de base se terminent par un appel en position terminale ou par la dernière évaluation de la méthode (généré par l'instruction *return* d'une méthode). Les appels en position terminale ne créent pas de bloc d'activation. Comme Scheme garantit l'optimisation des appels en position terminale, le code obtenu pour un branchement est l'équivalent d'un `goto` et est efficace.

Pour permettre aux blocs de base de s'appeler entre eux, nous définissons les fermetures dans

un *letrec* définissant dans un même environnement tous les blocs de bases.

```
(letrec ((bb#0 (lambda ()
                 1))
         (bb#1 (lambda ()
                 (let ((v1 (+ 1 2)))
                   (let ((v2 (- v1 2)))
                     temp2))))
         (bb#2 (lambda ()
                 (if (< 0 1)
                     (bb#0)
                     (bb#1)))))
  (bb2)) ;;- Appel initial
```

FIG. 5.5 – Représentation des blocs de base en Scheme

Ces blocs de base peuvent s'appeler entre eux car ils sont au même niveau.

Variables locales

Les variables locales et les paramètres dynamiques non-consommés sont passés en paramètres aux fermetures des blocs de base, comme pour le langage JST.

```
(letrec ((bb#0 (lambda (l%1 l%2)
                 l%1))
         (bb#1 (lambda (l%1 l%2)
                 (let ((t%1 (+ l%1 l%2)))
                   (let ((t%2 (- t%1 l%2)))
                     t%2))))
         (bb#2 (lambda(l%1 l%2)
                 (let ((t%1 (+ l%1 l%1)))
                   (let ((l%2 t%1))     ;;- Affectation a une variable locale
                     (if (< l%1 l%2)
                         (bb#0 l%1 l%2)
                         (bb#1 l%1 l%2)))))))

  (bb#2 0 0)) ;;- Appel initial
```

FIG. 5.6 – Représentation des variables locales

Le mécanisme employé pour les variables locales n'est pas valide pour les variables vivantes après le lancement d'une exception. Le gestionnaire d'exceptions n'a pas accès aux variables locales.

```
(letrec ((bb#0 (lambda ( l%1 l%2 )
                 (let ((l%1 l%2)) ;- Affectation a une variable locale
                   (let ((foo (method-call ...))) ;- Une exception est lancer
                     l%1))))       ;- dans la methode.
         (bb#1 (lambda ()          ;- Gestionnaire d'exception
                 ..                ;- Il est impossible d'obtenir
                 )))               ;- les variables l%1 et l%2.
  (bb#0 0 0))
```

FIG. 5.7 – Problème lié aux variables d'exceptions

Pour résoudre ce problème, les variables vivantes après une exception n'emploient pas le même mécanisme. La même astuce que celle employée pour JST est utilisée : une fermeture englobant tous les blocs de base et les gestionnaires d'exception est créée pour les variables mutables. Ce mécanisme est plus coûteux et limite les optimisations de Gambit-C.

```
(let* ((l%1 0)                  ;- Creation d'un environnement
       (l%2 0))                 ;- pour les variables mutables.
  (letrec ((bb#0
            (lambda ()
              (let (foo (set! l%1 l%2)) ;- Affectation a une variable locale.
                (let ((foo (method-call))) ;- Une exception est lancer
                  l%1))))                  ;- dans la methode.
           (bb#1
            (lambda ()              ;- Gestionnaire d'exception
              ..                    ;- l%1 et l%2 sont disponibles.
              )))
    (bb#0)))
```

FIG. 5.8 – Fermeture englobant les blocs de base et les gestionnaires d'exceptions

Variables globales

Les variables globales, ou variables statiques, sont définies comme des variables globales en Scheme. Un préfixe est ajouté à leur nom pour éviter des conflits. L'accès et l'affectation aux variables globales emploient les mécanismes de Scheme.

```
(define %%gbl%java/lang/System/%out% (%%null))

(set! %%gbl%java/lang/System/%out% (%%null))
```

FIG. 5.9 – Représentation des variables globales en Scheme

Méthodes statiques et virtuelles

Les méthodes statiques et virtuelles sont définies comme étant des fermetures en Scheme. Pour éviter les conflits de nom, le compilateur ajoute le nom de la classe ainsi qu'un préfixe.

```
(define %%mtd%java/lang/System/%exit<I>V%
  (lambda ( l%0 )
  ...))

(%%mtd%java/lang/System/%exit<I>V% 0)
```

FIG. 5.10 – Représentation des méthodes statiques et virtuelles en Scheme

Tables des méthodes virtuelles et des types d'instances

Pour chaque classe, nous définissons globalement la table des méthodes virtuelles. La table est représentée par un vecteur Scheme contenant des pointeurs vers les fermetures représentant les méthodes de la classe. Le premier élément du vecteur est la table des types d'instances représentée par un vecteur de booléen et permet de vérifier dynamiquement le type d'un objet. Cette table sera consultée lors de l'exécution des instructions `instanceof` et `checkcast`.

```
(define %%vt%java/lang/Object%
  (vector
   (vector #t #f #f #f #t )
   #f
   #f
   %%gbl%java/lang/Object/%wait<>V%
   ...))
```

FIG. 5.11 – Représentation des Tables des méthodes virtuelles et des types d'instances en Scheme

Objets Java*

Un objet Java est représenté par un vecteur Scheme contenant les champs propres à l'instance. Le premier élément du vecteur est un pointeur vers la table des méthodes virtuelles de la classe. Le reste du vecteur contient les champs spécifiques à l'instance de la classe.

```
(define %%mtd%java/lang/System/%<new><>Ljava/lang/System-%
  (lambda ()
    (vector %%vt%java/lang/System% 0 0 0.0 (%%null))))
```

FIG. 5.12 – Représentation des Objet Java* en Scheme

Exceptions Java

Le mécanisme d'exceptions de Java* permet au programmeur de lancer une exception lorsque survient une situation anormale. Les exceptions explicites que le programmeur lance volontairement permettent une gestion simple puisque le site de lancement est connu statiquement. Les exceptions implicites et asynchrones rendent le mécanisme d'exception complexe et imprévisible et rendent ainsi les analyses impossibles à réaliser.

Pour gérer le mécanisme d'exceptions, nous avons décidé d'employer le système de continuations de Scheme. Lorsqu'une méthode contenant des gestionnaires d'exceptions est appelée, la continuation est capturée et ajoutée dans une pile de gestionnaires d'exceptions. Lorsqu'une exception doit être lancée, la première continuation de la pile est appelée avec l'exception en paramètre. Le gestionnaire décide s'il traite l'exception ou s'il la passe au gestionnaire suivant. Un gestionnaire global interceptant toutes les exceptions est installé au début du programme pour imprimer à l'écran les exceptions sans gestionnaire et termine l'exécution de l'application.

Il est à noter que chaque thread possède sa propre pile de gestionnaires d'exceptions.

Synchronisation

Java prévoit des mécanismes de synchronisation pour gérer le parallélisme [LY99b]. Notre compilateur introduit un champ `<synchronized>` dans la classe de base `java.lang.Objet` pour gérer ces mécanismes.

Le contenu du champ `<synchronized>` est créé dynamiquement pour les objets nécessitant un des mécanismes de synchronisation. Il contient un mutex, un compteur et une variable de condition. À l'aide de ces champs, nous pouvons simuler les moniteurs de Java* basés sur le concept de moniteur de Hoare [Hoa74].

Finalisation

Les objets Java peuvent définir une fonction `finalize` qui doit être appelée lorsque le ramasse-miettes libère l'objet. Nous avons employé le type de donnée *will* de Gambit-C qui permettent d'attribuer un testament aux objets (une action qui est exécutée à la mort de l'objet. Par raison d'efficacité, une analyse est faite pour ne créer des testaments qu'aux objets qui contiennent du code avec effets de bord dans leur méthode `finalize` allégeant ainsi la création et la collecte d'objets.

5.4.2 Vérification dynamique

Lorsque le compilateur détecte que les paramètres, le résultat et le calcul d'une instruction sont bornés par le domaine des *fixnums*, il génère une série d'instructions optimisée pour Gambit. La Figure 5.13 montre une addition bornée par le domaine des *fixnums*.

```
(##fixnum.+ x y)
```

FIG. 5.13 – Addition sous un domaine borné en *fixnum*

Malgré les analyses précises sur les domaines numériques entiers que notre compilateur effectue, il ne peut pas prouver statiquement tous les cas sujets à cette optimisation. Par contre, il peut introduire des tests dynamiques pour modifier le flux d'exécution permettant d'effectuer plus efficacement le calcul. La Figure 5.4.2 montre un exemple de test dynamique où le compilateur n'a pas borné le paramètre x.

```
(if (##fixnum? x)
    (##fixnum.+ x y)
    (generic-java.+ x y))
```

FIG. 5.14 – Vérification dynamique de domaine non borné en *fixnum*

Comme la vérification dynamique de l'appartenance au domaine des *fixnums* est moins coûteuse que l'appel de l'instruction générique, un gain de vitesse est obtenu lorsque x est un élément des *fixnums*. Par contre, une perte de vitesse se manifeste dans les autres cas. La majorité de nos tests dynamiques proviennent des travaux de Dominique Boucher [Bou99].

Le compilateur introduit des tests similaires pour la plupart des instructions où il n'a pas pu borner le domaine des variables. Par contre, certaines instructions comme SHL (décalage à gauche) ne permettent pas de vérifications dynamiques offrant des gains de performance et forcent l'appel de l'instruction générique de Scheme. Nous verrons plus loin l'impact de cette lacune sur les performances des applications générées par J2S.

5.5 Performance du compilateur

Dans cette section nous étudions les performances du code généré par J2S. Nous discutons de ses points forts et points faibles.

5.5.1 Autres compilateurs

Nous avons décidé de comparer notre compilateur avec d'autres compilateurs répandus et qui obtiennent de bonnes performances.

Hotspot

Hotspot est la machine virtuelle de Sun pour la plate-forme Java. Il donne de bonnes performances pour les applications Java. Il utilise des techniques de compilation avancées, des modèles de mémoire complexes et un ramasse-miettes adapté au langage Java. Il est écrit dans un langage orienté objet et possède les propriétés suivantes [Sun02] :

- modèle objet uniforme,
- blocs d'activation interprétés, compilés et natifs qui partagent la même pile d'exécution.
- parallélisme préemptif basé sur les threads natifs,
- ramasse-miettes générationnels et compactant précis,
- synchronisation rapide des threads,
- déoptimisation dynamique et optimisation agressive,
- interface vers le compilateur supportant la compilation parallèle,
- profilage dynamique qui concentre l'effort sur les points chauds.

Il utilise une représentation intermédiaire sous forme SSA pour effectuer ses optimisations. Son compilateur dynamique effectue les optimisations classiques :

- élimination de code mort,
- sortie d'invariant de boucle (*loop invariant hoisting*),
- déroulement de boucle,
- élimination d'expression commune,
- propagation de constante,
- *inlining* de méthode,
- élimination de *checknull*,
- élimination de vérification de borne.

Son allocateur de registre est basé sur la coloration de graphe.

Nous avons choisi de comparer notre compilateur avec la machine virtuelle de Sun parce qu'elle donne de très bonnes performances et qu'elle est la machine virtuelle Java la plus répandue.

version : *java version "1.4.0"*
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.0-b92)
Java HotSpot(TM) Client VM (build 1.4.0-b92, mixed mode)

Kaffe

Kaffee, une application gratuite sous la license GPL (*GNU Public License*), est une implantation d'une machine virtuelle Java qui fournit un environnement d'exécution.

Kaffe n'est pas la meilleure machine virtuelle Java pour le développement d'application car il possède plusieurs problèmes d'incompatibilité. Il est surtout employé dans les milieux de recherche et de développement.

Kaffe possède un compilateur dynamique moins évolué que le compilateur dynamique de *Hotspot*. Il n'effectue pas d'optimisation dynamique.

Vu son attachement au domaine scientifique et la disponibilité du code source, nous avons jugé bon de comparer les performances de ce compilateur avec celles de notre compilateur.

version : *Kaffe Virtual Machine*
Copyright (c) 1996-2000
Transvirtual Technologies, Inc. All rights reserved
Engine : Just-in-time v3 Version : 1.0.6 Java Version : 1.1

JDK-1.2

Nous avons décidé de comparer notre compilateur avec une ancienne version de la machine virtuelle de Sun. La machine virtuelle fournit avec le JDK-1.2 de Sun contient un compilateur dynamique.

Contrairement à *Hotspot*, le compilateur dynamique de JDK-1.2 n'effectue pas d'optimisation agressive et ne possède pas un environnement d'exécution aussi dévellopé. Il n'effectue qu'une seule compilation au chargement du code. Au chargement d'une classe, le compilateur effectue une compilation native du *bytecode* Java. Cette compilation cause une petite pause au chargement de chaque classe mais améliore la performance globale de l'application.

Nous avons choisi de comparer notre compilateur J2S avec la machine virtuelle de JDK-1.2 parce qu'elle effectue une compilation dynamique qui ne possède aucune information dynamique (contrairement à *Hotspot*). De plus, il est intéressant de comparer les performances de JDK-1.2 et *Hotspot* pour leurs différentes implantations de l'environnement d'exécution.

version : *java version "1.2.2"*
Classic VM (build 1.2.2_ 006, green threads, javacomp)

5.5.2 Résultats

Nous avons effectué plusieurs tests pour comparer l'efficacité des programmes générés par notre compilateur avec différentes machines virtuelles existantes. Dans cette section nous présentons certains résultats sélectionnés pour mettre en évidence les forces et faiblesses de notre compilateur.

Tous les tests ont été effectués sur un Althon 1.2 GHz avec 512 mégaoctets de mémoire sous le système d'exploitation Linux (RedHat 7.1).

Calculs numériques intenses

Le premier test de performance effectué mesure l'efficacité du calcul numérique intense et les accès aux tableaux. Le programme *jBYTEmark* [IBM01] est disponible sur Internet. Il effectue une série de tests représentatifs des applications numériques. Le Tableau 5.1 montre les résultats obtenus par différentes machines virtuelles et par notre compilateur.

Benchmark	Hotspot	Kaffe	JDK-1.2	J2S
	(iter/sec)	(iter/sec)	(iter/sec)	(iter/sec)
Numeric Sort	147,06	122,40	63,65	**20,71**
String Sort	47,62	43,01	33,33	**2,83**
Bitfield Operations	1,555E7	1,392E7	3,515E7	**3,41E5**
FP Emulation	25,00	12,20	11,70	**0,15**
Fourier	4071,43	6443,55	7537,74	**1442,03**
Assignment	3,85	2,55	1,95	**0,72**
IDEA Encryption	187,50	178,22	87,50	**3,09**
Huffman Comp.	181,82	174,76	173,91	**8,77**
Neural Net	14,81	4,75	8,00	**1,13**
LU Decomposition	119,05	21,28	47,62	**5,49**

TAB. 5.1 – jBYTEmark

Il est à noter que ce programme emploie peu les librairies de Java. Toutes les fonctionnalités requises par l'application sont implantées en Java et respecte les critères du langage Java*.

Les résultats obtenues sont en itérations par seconde. Une exécution plus rapide effectue plus d'itérations par seconde qu'une exécution plus lente. Nous observons dans le Tableau 5.1 que J2S n'obtient pas de bonnes performances sur le calcul numérique intense. Il est au moins 3 plus lent que les autres compilateurs et parfois 166 fois plus lent. Les performances de notre

compilateur sont en-dessous des performances de tous les autres compilateurs pour tous les tests de *jBYTEmark*.

Nous avons analysé les causes de ces mauvaises performances. L'application *jBYTEmark* effectue du calcul numérique intense et utilise beaucoup de tableaux. Le calcul numérique introduit des tests dynamiques et des ajustements pour le débordement des entiers. Les accès aux tableaux dans J2S nécessitent deux déréférencements de pointeurs contrairement aux autres implantations qui n'effectuent qu'un seul déréférencement. Nous avons vérifié d'où provient la perte de performance observée et nous en discutons dans notre analyse.

De plus, il est intéressant de voir l'évolution entre le compilateur dynamique de JDK-1.2 et *Hotspot*. *Hotspot* effectue plusieurs compilations dynamiques du même code pour l'adapter dynamiquement contrairement à JDK-1.2 qui n'effectue qu'une compilation au chargement de la classe.

Hotspot est généralement plus rapide que JDK-1.2 excepté pour deux tests : *Bitfield Operations* et *Fourier*.

Applications parallèles

Le second test de performance mesure l'efficacité du système de threads. L'application effectue le calcul de la suite de Fibonacci en parallèle. Cette application effectue peu de calculs numériques mais beaucoup de gestion de threads et de synchronisation.

Nb Threads	N	Hotspot	Kaffe	JDK-1.2	J2S
		(ms)	(ms)	(ms)	(ms)
41	7	820	10	6	**2**
67	8	1320	38	6	**2**
109	9	2181	53	12	**3**
177	10	3530	75	20	**9**
287	11	5741	126	38	**14**
465	12	crash	179	79	**26**
753	13	crash	310	134	**60**
1219	14	crash	550	309	**108**
1973	16	crash	947	1275	**245**
3193	17	crash	2292	3146	**585**
5167	18	crash	crash	11296	**1365**
8361	19	crash	crash	43960	**3289**
13529	20	crash	crash	103236	**8211**

TAB. 5.2 – Fibonacci (Threads)

Cette application crée deux threads pour évaluer de manière récursive les deux branches du calcul créant un nombre de threads exponentiel par rapport à N.

Pour ce test, notre compilateur obtient toujours de meilleures performances que les autres compilateurs. Nous observons de très bonnes performances dans les applications parallèles que nous justifions par l'emploi de Scheme et du compilateur Gambit-C comme cible. Gambit-C fournit une librairie de threads légers ayant un coût de gestion peu élevé. L'application calculant la suite de Fibonacci en parallèle profite de la rapidité de création de threads de Gambit. Les processus lourds basés sur les POSIX Threads qu'emploie *Hotspot* [Sun99] donnent de mauvaises performances lorsque la quantité de traitement par le processeur est petite et le temps de création du processus ne peut pas être absorbé par le temps de traitement. Nous observons que toutes les machines virtuelles employant des processus légers obtiennent de meilleures performances que *Hotspot*. Le système de threads de JDK-1.2 est basé sur les *green threads*, une librairie de threads légers. Contrairement à Gambit-C, les processus lourds permettent à *Hotspot* de profiter des architectures multi-processeurs.

Application parallèle et librairies

La troisième application représente une application parallèle typique effectuant beaucoup d'appels aux librairies Java. L'application recherche en parallèle, dans un répertoire et tous ses

sous-répertoires, le fichier Java possédant le plus grand nombre de lignes.

Niveau de concurrence	Hotspot (ms)	Kaffe (ms)	JDK-1.2 (ms)	J2S (ms)
1	10 744	7 960	7 898	**7 604**
2	5 520	3 983	4 014	**3 844**
3	3 746	2 690	2 716	**2 546**
4	3 093	2 082	2 144	**1 970**
5	3 085	1 691	1 774	**1 567**
6	3 086	1 441	1 474	**1 329**
7	2 976	1 271	1 294	**1 108**
8	2 997	1 145	1 153	**997**
9	3 016	1 000	1 043	**908**
10	2 993	940	1 024	**826**
11	3 047	869	854	**776**
12	2 966	869	854	**687**
13	2 986	767	854	**675**
14	2 967	791	824	**625**
15	2 963	700	764	**614**

TAB. 5.3 – Recherche de fichier (69 fichiers, 145 threads)

Le niveau de concurrence indique le nombre maximal de thread s'exécutant en parallèle. Il est à noter que des exécutions répétitives de ce programme test sont nécessaires pour absorber l'impact du phénomène de cache créé par le système d'exploitation et le matériel pour l'accès au système de fichiers.

Nous observons que notre compilateur obtient dans tous les cas un temps d'exécution plus rapide que les autres compilateurs.

La dernière application test nous permet d'observer une application parallèle dont le temps de création de threads est absorbé par la tâche à accomplir. Pour ce test, notre compilateur obtient de meilleures performances que les autres compilateurs. Cette application effectue beaucoup d'entrées/sorties. Pour obtenir de bonnes performances, l'environnement d'exécution doit effectuer du tamponnage *(buffering)* de données. Par défaut, les librairies Java n'effectuent pas de tamponnage. Nous expliquons les bonnes performances de J2S par les mécanismes d'entrées/sorties fournis par Gambit et par son faible coût de gestion des threads légers.

Des ce test, nous observons de meilleures performances pour les compilateurs employant un

système de threads légers. Par contre, le rapport des performances entre *Hotspot* et les autres compilateurs est moins élevé parce que le temps de création de thread est absorbé par la quantité de travail à effectuer par thread.

5.5.3 Analyse

Dans cette section, nous analysons et justifions les performances obtenues par notre compilateur. Nous avons séparé l'analyse des performances de notre compilateur en deux sections : l'analyse des applications à calculs numériques intenses et l'analyse des applications parallèles.

Calculs numériques intenses

Suite aux résultats obtenue par J2S sur le calcul numériques intenses, (voir Tableau 5.1), nous avons cherché la cause de ces mauvaises performances.

La Figure 5.15 montre le code généré par une addition sachant que tous les paramètres et le résultat sont bornés par le domaine des *fixnums*.

```
(##fixnum.+ l%1#11 l%31#11)
```

FIG. 5.15 – Calcul numérique borné en *fixnum*

La Figure 5.16 montre le code généré par une addition où le compilateur n'a pas pu borner le domaine des variables. Ce code contient les tests dynamiques ainsi que les ajustements de débordements.

```
(if (and (##fixnum? l%1#11) (##fixnum?l%31#11))
    (let ((r#13 (##fixnum.+ l%1#11 l%31#11)))
      (if (##fixnum.<
           (##fixnum.bitwise-and
            (##fixnum.bitwise-xor
             r#13
             l%31#11)
            -536870912)
           (##fixnum.bitwise-and
            (##fixnum.bitwise-xor
             l%1#11
             l%31#11)
            -536870912))
          (generic-int.+ l%1#11 l%31#11)
          r#13))
    (generic-int.+ l%1#11 l%31#11))
```

FIG. 5.16 – Calcul numérique non-borné en *fixnum*

Nous observons à la Figure 5.16 les tests dynamiques vérifiant l'appartenance au domaine des *fixnums*. Ces tests déterminent dynamiquement si le calcul suit les intructions optimisées pour le domaine des *fixnums* ou s'il emploie la fonction générique de calcul. Nous observons aussi l'ajustement de débordement de capacité.

Nous avons déduit que la perte de performance observée provenait des cas de calcul non borné. Pour vérifier la provenance de la perte de performance, nous avons écrit une application test où la proportion de variables non bornées est similaire à l'application *jBYTEmark*. Nous pouvons contrôler avec des paramètres passés au compilateur la génération des tests dynamiques. Grace à une analyse manuelle du programme, nous savons que tous les calculs effectués dans cette application s'effectuent dans le domaine des *fixnums*. Nous avons mesuré les performances de l'application lorsque nous la forçons à effectuer tous les calculs en *fixnum*, lorsque le compilateur détecte lui même les calculs possibles d'être effectués en *fixnum* et lorsqu'aucun calcul n'est en *fixnum*. Il est à noter que les trois exécutions effectuent la même tâche et obtiennent les mêmes résultats. Elles varient sur le nombre de vérifications dynamiques inutiles pour les *fixnums* et pour les débordements de domaine.

```
1. int x,y;
2. for (int i=0; i<N; i++)
3.   for (int j=0; j<M; j++ ) {
4.       x += i+j;
5.       y += j-i;
6.       [...]
7.       y -= j-i;
8.       x -= i+j;
9.   }
```

FIG. 5.17 – Exemple de programme test

Les trois tests effectuées consistent en deux boucles avec deux index distincts qui s'incrementent. Le corps des boucles varie selon l'application test. La Figure 5.17 montre le modèle employé pour les tests. L'exécution de la boucle n'effectue aucun changement global et effectue plusieurs opérations d'un certain type pour en évaluer son coût d'exécution. Le test *numeric 1* effectue plusieurs additions et soustractions selon les deux index tandis que le test *numeric 2* effectue des multiplications et divisions. Le test *shifting* effectue des décalages arithmétiques et logiques selon les deux index. Le test *cmp* effectue des comparaisons sur les deux index. Le corps des méthodes doit contenir assez d'opérations pour absorber le coût des boucles.

	J2S			Hotspot
	Borné Fixnum	Borné sous optimisation	Non borné	
	(ms)	(ms)	(ms)	(ms)
Numeric 1	934	4567	6555	1106
Numeric 2	9434	12394	15287	8141
Shifting	61	5701	5726	20
Cmp	3122	6701	12547	2025

TAB. 5.4 – Comparaison des performances des *fixnums* et *bignums*.

Le Tableau 5.4 compare les performances de J2S avec celles de *Hotspot* pour notre application test. Nous observons que l'ordre de grandeur des temps d'exécution des calculs bornés en *fixnum* est du même ordre de grandeur que ceux effectués par *Hotspot* contrairement aux autres cas partiellement bornés ou non bornés.

Comme nous le constatons, si nous pouvions borner toutes les variables, nous obtiendrions des performances similaires à celles obtenues par *Hotspot*. Le problème vient du fait que Gambit-C représente ses *fixnums* sur 30 bits et non 32 bits ce qui force le compilateur à générer des instructions de gestion de débordement au lieu de laisser le processeur (32 bits) s'en occuper. *Hotspot*, étant un compilateur dynamique proche de l'architecture, ne rencontre pas ce problème puisqu'il emploie des instructions du processeur supportant directement le type natif des entiers 32 bits.

Nous avons implanté l'analyse de domaines numériques entiers pour essayer d'optimiser plus de cas. L'analyse permet de borner certains domaines et opérations en dessous de la limite des 30 bits et permet au compilateur d'éviter de produire des vérifications et réajustements dynamiques inutiles. Cette analyse est nécessaire pour obtenir de bonnes performances avec J2S puisque son système numérique est son point faible.

Si les calculs numériques sont effectués sur un type numérique dont la limite est en dessous des 30 bits, les performances des compilateurs sont comparables. Par exemple, les calculs effectués sur les *short* donnent des performances similaires à celle de *Hotspot*.

Pour J2S, l'accès aux tableaux est problématique. À chaque accès à un élément du tableau, nous devons effectuer un test `checknull` et vérifier les bornes du tableaux. Le test `checknull` n'est pas effectué par *Hotspot* car ce dernier emploie les mécanismes de gestion de pages du processeur. J2S, étant un compilateur détaché de l'architecture, n'est pas en mesure de pouvoir employer ce mécanisme, et par conséquent, une vérification dynamique est effectuée à chaque accès à un tableau. Pour la vérification des bornes d'un tableau, il est possible de vérifier les

deux bornes à l'aide d'une seule comparaison. Les index des tableaux sont signés et peuvent être négatifs. Nous devons vérifier que l'index est supérieur à zéro et inférieur à la dimension du tableau. Si nous effectuons la comparaison en arithmétique non-signée, nous n'avons qu'à effectuer la comparaison de la borne supérieure puisqu'un nombre négatif en arithmétique non-signée est toujours un nombre plus élevé que l'index maximal d'un tableau. Le système numérique de Scheme ne prévoit pas de mécanisme pour effectuer des calculs et des comparaisons en arithmétique non-signée.

La Figure 5.5 nous montre les coûts associés aux vérifications dynamiques imposées par Java et que J2S ne peut pas enlever en employant Scheme comme langage cible.

	J2S			Hotspot
	Aucun test	Tests avec optimisation	Tous les tests	
	(ms)	(ms)	(ms)	(ms)
tableau	33155	58126	218513	19789

TAB. 5.5 – Comparaison de performances des tests dynamiques.

L'application test employée consiste à deux boucles qui copie partiellement un tableau vers un autre tableau selon les index des boucles. Nous pensons que la différence de performance restante entre *Hotspot* et J2S s'explique par le fait que *Hotspot* est en mesure d'appliquer des algorithmes d'optimisation locale comme le déroulement de boucles que J2S n'implante pas encore.

Applications parallèles

La performance des applications parallèles qu'obtient J2S s'explique par l'environnement d'exécution de Gambit-C. Gambit-C possède une implantation de threads légers efficace. Il fournit une librairie de threads légers ayant un coût de gestion peu élevé puisque les fonctionnalités de son système de threads nécessitent peu d'appels systèmes.

Le fonctionnement de la pile d'exécution de Gambit-C permet un partage de la pile d'exécution entre les threads. Cette fonctionnalité permet au système de threads de supporter un nombre élevé de threads. Dans les implantations typiques de système de threads, la pile d'exécution est un goulot d'étranglement limitant le nombre de threads puisque l'espace adressable de la pile d'exécution est limité et les piles d'exécution ne peuvent pas être déplacées. Par exemple, *Hotspot* ne supporte généralement pas plus de 1024 threads (sans ajustement des paramètres). Contrairement à *Hotspot*, J2S permet un nombre élevé de threads et donne un environnement d'exécution permettant un très haut niveau de concurrence.

5.6 Discussion

Après avoir comparé les résultats de J2S et de *jBYTEmark*, nous ne pensons pas que la compilation de Java vers Scheme (ciblant spécifiquement le compilateur Gambit-C) soit une bonne approche pour les applications de calculs intenses. Même avec des analyses de type complexes et précises permettant de mieux borner les domaines, il ne nous semble pas possible d'obtenir des performances similaires à *Hotspot*. Pour ce type d'application, le compilateur doit être de bas niveau et générer du code efficace pour l'architecture hôte. Nous pensons que la compilation de Java vers C ou un autre langage de bas niveau donnerait de meilleurs résultats pour les applications numériques.

Par contre, la compilation de Java vers un langage de plus bas niveau nécessite l'implantation d'un ramasse-miettes, du système de threads et des autres fonctionnalités fournies par Gambit-C. Les analyses et optimisations effectuées par Gambit-C et par le compilateur C doivent être implantées pour obtenir des performances comparables à celle d'un compilateur effectuant ces mêmes analyses. Un compilateur Java vers Scheme est plus simple qu'un compilateur natif et possède un temps de développement et un risque d'erreur plus faible.

D'après les performances obtenues avec les applications parallèles, un compilateur vers Scheme est très efficace et simple d'implantation pour les applications nécessitant plus de traitements parallèles et d'appels aux librairies. Un certain nombre d'applications pourraient tirer profit de ce type de compilation.

L'architecture par couches de compilateurs employée par J2S facilite le développement rapide d'un compilateur relativement efficace. Elle offre un système plus stable où les entités de compilation possèdent peu de dépendances et sont facilement analysables. Le choix des compilateurs pour chaque couche permet de sélectionner les propriétés et fonctionnalités désirées.

Les performances de la plupart des applications testées démontrent l'efficacité de ce modèle de développement. Malheureusement, certaines propriétés et fonctionnalités ne peuvent pas être abstraites sans coût et les incompatibilités des langages intermédiaires, comme l'évaluation numérique, peuvent entraîner des coûts non négligeables pour des applications nécessitant de hautes performances. Le choix des langages intermédiaires et des compilateurs qui leurs sont associés est le point crucial lors du développement d'un compilateur sous ce modèle de compilation.

Chapitre 6

Compilation vers Intel x86

Vu les performances obtenues par J2S dans l'application *jBYTEmark* [IBM01] (voir le Tableau 5.1), nous avons entrepris la compilation de Java* vers un langage cible de plus bas niveau pour obtenir de meilleures performances pour les applications de calculs numériques intenses. Nous avons choisi de générer de l'assembleur pour l'architecture Intel laissant à notre compilateur la liberté d'employer des mécanismes spécifiques à l'architecture.

Contrairement au compilateur J2S, JBCC intègre les phases de compilation dans le même compilateur. JBCC effectue seul toutes les phases de compilation, les analyses et les optimisations. De plus, il offre l'environnement d'exécution requis par le langage. Aucune propriété n'est fournie par un autre compilateur ou un environnement d'exécution.

Dans ce chapitre, nous décrivons le processus de génération de code vers l'architecture Intel. Nous étudions les analyses et optimisations effectuées sur les langages de bas niveau. Nous discutons ensuite des performances obtenues par cette version de notre compilateur.

6.1 Structure du compilateur

Prévoyant l'ajout de nouvelles architectures cibles, nous avons adopté pour notre compilateur une structure par couche comme les compilateurs conventionnels ont coutume de faire. Chaque couche effectue des analyses et optimisations sur le langage qu'elle reçoit et passe à la couche suivante le résultat de ses transformations dans le même langage ou dans un nouveau langage. Généralement, les couches supérieures sont dépendantes du langage source et les couches inférieures sont dépendantes de l'architecture.

Notre compilateur n'a pas une structure prévue pour compiler plusieurs langages sources et ses premières couches sont très dépendantes du langage Java*. Par contre, comme plusieurs architectures étaient visées, les analyses, les transformations et les optimisations de bas niveau doivent présenter certain un niveau d'abstraction.

Les premières phases de compilation de JBCC sont communes à J2S. Ainsi, le traitement du langage intermédiaire JST pour les deux compilateurs est commun. JBCC poursuit le traitement du langage JST en le convertissant vers un langage assembleur abstrait, JASM, limitant les dépendances d'une architecture. Les analyses et les optimisations communes à toutes les architectures seront appliquées sur ce langage.

Pour chaque architecture visée, un module donne des informations spécifiques aux architectures pour les différents algorithmes qui traitent le langage JASM. À l'aide de ce module, le compilateur peut générer du code assembleur spécifique à chaque architecture.

6.2 Architecture

Une classe abstraite `jbcc.out.Architecture` contient les informations de l'architecture cible. Pour cibler une nouvelle architecture, il suffit de décrire les informations dans une classe en respectant l'interface. Cette classe contient les informations sur les registres disponibles, le format des blocs d'activations, les définitions de tuiles (*patterns*) d'instructions utilisées lors de la sélection d'instructions et les optimisations spécifiques à l'architecture.

JBCC fournit l'implantation de la classe `jbcc.out.intel.Architecture` qui est spécifique à l'architecture Intel. Les algorithmes décrits ci-dessous sont indépendants de l'architecture ciblée mais nécessitent de l'information distincte.

6.2.1 Sélection d'instructions

La sélection d'instructions se fait à partir du langage JASM qui est proche de l'assembleur. Nous avons employé un algorithme générique pour toutes les architectures. L'algorithme utilisé, connu sous le nom de *Maximal Munch*, est dû à Appel [App99].

Une approche par programmation dynamique offrirait une meilleure sélection d'instructions. Toutefois, vu la complexité du problème de sélection d'instructions, nous avons préféré une approche simple qui donne généralement de bons résultats pour l'architecture Intel.

Notre compilateur ne supporte pas l'émission d'instructions provenant des jeux *3DNow*, *MMX*, *SSE* ou *SSE2*. L'utilisation de ces instructions SIMD (*Single Instruction Multiple Data*)

donnerait de meilleures performances mais rendrait la sélection d'instructions encore plus complexe.

6.2.2 Allocation de registres

L'allocation de registres se fait par une coloration de graphe comme décrit par l'algorithme de Yorktown de Chaitin [CAC+81, Cha82]. Certaines améliorations de l'algorithme de Yorktown présentées par Briggs [Bri92] ont été implantées.

L'algorithme de coloration de graphe appliqué à l'allocation de registres offre des performances plus intéressantes que les algorithmes simples qui allouent les registres à la volée. Par contre, notre algorithme d'allocation, étant local aux méthodes, n'effectue pas l'allocation inter-procédurale. Une approche par fusion de graphe de coloration [LGAT96] permettrait d'allouer plus efficacement les registres pour les variables locales, globales et inter-procédurales. Vu l'information des structures de contrôle de notre compilateur, une approche d'allocation basée sur le graphe de dépendance (PDG) [NP94] serait à étudier.

Les informations de l'architecture fournit à l'algorithme un ensemble de registres possédant des propriétés telle leur taille. L'algorithme essaie d'associer ces registres aux registres virtuels du langage JASM importé en respectant les contraintes spécifiées.

La gestion de la convention d'appel (*caller/callee safe*) est réalisée lors du processus de coloration en introduisant des contraintes sur les registres coloriés. Pour les registres sauvegardés par l'appelé, nous introduisons des instructions *def* et *use* à l'entrées et aux sorties de la méthode pour spécifier le code où ces registres sont vivants. Lorsqu'un de ces registres est requis, pendant la coloration de graphe, le code nécessaire pour la sauvegarde et le rechargement de la valeur (*Spill Code*) est automatiquement généré par les mécanismes. Pour les registres sauvegardés par l'appelant, une contrainte sur les registres vivants lors d'un appel de fonction est ajoutée et limite le choix des registres.

Groupes de registres

L'allocateur de Yorktown ne permet pas de définir des contraintes d'équivalence sur les registres systèmes. Par exemple, sur l'architecture Intel, les registres `%al`, `%ah`, `%ax` et `%eax` partagent un espace commun et par conséquent possèdent une contrainte d'équivalence. Dans notre implantation, un seul registre est créé et donné à l'algorithme pour l'ensemble de ces registres et une propriété de taille est ajoutée aux registres. Après l'allocation, une étape de finalisation renomme les registres selon leur taille. Par exemple, le registre système `A` possède comme propriété de taille 8, 16 et 32 bits et, selon la taille du registre virtuel remplacé, porte

le nom `%al`, `%ax` ou `%eax` respectivement. Le registre `%ah` n'existe pas dans par notre allocation de registre et ne sera jamais alloué.

Registres à 64 bits

L'architecture Intel ciblée ne possède pas de registres entiers à 64 bits. Les registres 64 bits sont alloués en paire comme `%edx:%eax`. L'allocateur de Yorktown ne permet pas d'allouer des paires de registres. Briggs [Bri92] propose des solutions au problème d'allocation de paires de registre mais nous avons employé un mécanisme plus simple fonctionnant sous l'architecture Intel. Les registres 64 bits entiers sont séparés en deux registres à 32 bits coloriés individuellement. Il est donc possible d'avoir la moitié d'une valeur 64 bits sauvegardée et l'autre moitié vivante dans un registre système. Sous l'architecture Intel, les registres flottants sont plus larges que 64 bits et ne causent donc pas de problèmes.

Registres flottants

Historiquement, l'architecture x86 possédait un co-processeur mathématique, le 80x87, qui fût intégré au processeur avec les années. Notre compilateur emploie les instructions du 80x87 pour le calcul numérique flottant. Les calculs du 80x87 s'effectuent au moyen d'une pile. L'allocateur de Yorktown ne supporte pas l'allocation de registres pour une pile de registres contenant les données à traiter.

Le 80x87 fournit des instructions pour charger, décharger et échanger des registres de la pile flottante. Pour parvenir à colorier les registres de la pile flottante, les instructions de gestion de la pile ont été qualifées de trois états (indiquant le nombre de valeurs temporaires au dessus de la pile) et les huits registres systèmes de la pile flottante ont été divisés en registres virtuels et registres temporaires. La Figure 6.1 montre les registres et les états.

%st(0)	Registre f1
%st(1)	Registre f2
%st(2)	Registre f3
%st(3)	Registre f4
%st(4)	Registre f5
%st(5)	libre
%st(6)	libre
%st(7)	libre

État 0

%st(0)	Temporaire 1
%st(1)	Registre f1
%st(2)	Registre f2
%st(3)	Registre f3
%st(4)	Registre f4
%st(5)	Registre f5
%st(6)	libre
%st(7)	libre

État 1

%st(0)	Temporaire 2
%st(1)	Temporaire 1
%st(2)	Registre f1
%st(3)	Registre f2
%st(4)	Registre f3
%st(5)	Registre f4
%st(6)	Registre f5
%st(7)	libre

État 2

FIG. 6.1 – Registres et états de la pile 8087

Lors de la génération d'instructions, toutes les tuiles (*pattern*) prennent et laissent la pile dans l'état 0. Un état de pile est ajouté aux instructions de gestion de pile. Par exemple, l'instruction `fld.0` charge une donnée dans la variable temporaire 1 lorsque la pile est dans l'état 0. L'instruction suivante recevra la pile dans l'état 1. Une tuile flottante typique charge ses paramètres dans les registres temporaires, effectue son calcul et met le résultat dans un registre flottant. Cette tuile prend et laisse la pile flottante à l'état 0. Ainsi les tuiles peuvent se succéder sans problème. La Figure 6.2 montre un exemple de code colorié avec la technique décrite.

Avant coloration	Après coloration	Finalisation
`fld.0 %f32.r1`	`fld.0 %f1`	`fld %st(0)`
`fld.1 %f32.r2`	`fld.1 %f4`	`fld %st(4)`
`faddp`	`faddp`	`faddp`
`fstp.1 %f32.r3`	`fstp.1 %f3`	`fstp %st(3)`

FIG. 6.2 – Allocation des registres flottants

Pour respecter la convention d'appel de C, toute la pile flottante doit être sauvegardée lors d'un appel de fonction. JBCC supporte une convention d'appel interne plus efficace mais lors d'appels à des méthodes natives, tous les registres flottants doivent être sauvegardés pour permettre l'intégration de modules écrits en C.

6.2.3 Optimisations

Nous détaillons dans cette section les optimisations sur le langage de bas niveau de notre compilateur. JBCC inclut peu d'analyses et d'optimisations de bas niveau.

Substitutions localisées

Des optimisations de substitutions localisées d'instructions de bas niveau, connues sous le nom de *peephole*, corrigent certaines faiblesses de la génération de code et de l'allocation de registres. L'algorithme générique applique en boucle les optimisations fournies par les spécifications de l'architecture tant qu'une itération est différente de la précédente. La Figure 6.3 montre des exemples de transformations de code. Plusieurs définitions de substitution localisée (*peephole pattern*) proviennent du manuel d'optimisation de AMD [AMD01] ou de livres de compilation classiques tels [ASU86] et [App99].

Avant optimisation		Après optimisation
`movl $0, %eax`	→	`xorl %eax, %eax`
`imull $2, %eax`	→	`addl %eax, %eax`

`imull $7, %eax`	→	`movl %eax, %ebx`
		`shll $3, %eax`
		`subl %ebx, %eax`

`jmp l1`
`jmp foo`
`l1 :`
	→	`l1 :`

FIG. 6.3 – Exemples de substitutions localisées

Il est important d'inclure des définitions de substitution localisée pour optimiser le code redondant généré par la sauvegarde et la restauration des registres (*spill*) lors de la coloration de graphe. Beaucoup d'instructions de gestion de pile inutiles sont générées par l'allocation des registres flottants tel que décrit dans la Section 6.2.2.

Avant optimisation	Après optimisation
`fld %st(4)`	`fld %st(4)`
`fld %st(3)`	`fld %st(3)`
`faddp`	`faddp`
`fstp %st(1)`	`fchs`
`fld %st(0)`	`fstp %st(1)`
`fchs`	
`fstp %st(1)`	

FIG. 6.4 – Substitution d'instructions avec registres flottants

Réordonnancement d'instructions

JBCC n'effectue aucun réordonnancement d'instructions. Sur l'architecture Intel ciblée, le réordonnancement d'instructions a peu d'effet sur les performances puisque les processeurs ef-

fectuent du réordonnancement dynamique d'instructions. Des algorithmes comme *ScoreBoard* ou *Tamasulo* [HP96] sont implantés en matériel.

Accès à la cache

JBCC n'effectue aucune optimisation d'accès à la cache. Ces optimisations auraient un impact positif sur les performances du code généré. Les optimisations de cache restructurent des boucles, des accès aux tableaux et des accès à la mémoire pour changer l'ordre d'accès ou le temps d'accès à la mémoire de façon à améliorer la probabilité de retrouver les données en mémoire cache.

6.3 Librairies externes

L'interface vers les librairies externes respecte les conventions d'appel de C, facilitant l'implantation des fonctions natives requises par l'API de Java. Des librairies compatibles avec les systèmes POSIX sont fournies avec la dernière version de notre compilateur. JBCC fournit certaines librairies standards de Java telles que les accès aux fichiers, le ramasse-miettes, les *sockets* et le système de threads. Une version future du compilateur devrait respecter l'interface JNI [Lia97] pour les librairies natives pour faciliter l'importation des librairies natives de Java.

Il est possible de modifier les librairies externes employées par le compilateur en modifiant les spécifications de librairies natives fournies au compilateur via les fichiers NAT et en liant les nouvelles librairies au programme généré. Les librairies fournies par le programmeur doivent respecter la convention d'appel de C et doivent être liées statiquement au programme.

Il n'est pas facile pour un utilisateur d'intégrer une nouvelle librairie sans connaître les impacts qu'ont les spécifications natives sur les analyses et, par conséquent, le fonctionnement interne du compilateur. Une interface plus abstraite et générique devrait être un prochain sujet d'étude.

6.3.1 Entrées/sorties

JBCC fournit les fonctions natives pour effectuer les entrées/sorties via les librairies Java. Deux versions sont disponibles. La première emploie les fonctions d'entrées/sorties (read/write) de UNIX et la seconde emploie le standard d'entrées/sorties ANSI, soit les routines de `stdio.h`.

Dans une future version nous espérons voir réalisée l'implantation d'un module d'entrées/sorties tamponisées.

6.3.2 Système de threads

Les librairies natives que notre compilateur fournit par défaut, emploient une interface vers les librairies POSIX threads. Ce système de threads emploie des processus lourds et fonctionne sur une architecture multi-processeurs.

Dans une future version nous espérons une intégration d'un système de threads légers ou mixtes aux librairies du compilateur.

6.3.3 Ramasse-miettes

JBCC effectue la gestion de sa mémoire en employant le modèle de mémoire du langage C. Cela permet l'utilisation de librairies de gestion de mémoire comme le ramasse-miettes pour le langage C de Boehm [HC92]. L'interface vers le ramasse-miettes de Boehm est simple. Pour l'allocation de la mémoire, il suffit d'appeler la fonction `GC_malloc` au lieu de `malloc`.

Pour supporter le parallélisme, certaines fonctions de la librairie des *pthreads* doivent être modifiées. Le ramasse-miettes de Boehm fournit un fichier d'en-tête C effectuant, à l'aide de macros, les modifications requises.

Pour le bon fonctionnement du ramasse-miettes, il est important de respecter les conventions d'appel et de gestion de mémoire du langage C. Par exemple, tous les pointeurs doivent être alignés sur des frontières de 32 bits, autant sur la pile d'exécution que sur le tas. La structure des blocs d'activation doit respecter le standard d'appel de C et aucun abus de langage avec les pointeurs n'est permis. Heureusement, le langage Java, étant strict sur la convention d'appel et sur la notion de pointeur, ne permet pas au programmeur de contrevenir à ces prérequis et de rendre le système instable.

6.4 Performance

Dans cette section, nous discutons des performances du compilateur JBCC. Nous comparons ses performances avec celle de J2S et d'autres machines virtuelles.

6.4.1 Résultats

Tous les tests ont été effectués sur un Althon 1.2GHz avec 512 megaoctets de mémoire exécutant le système d'exploitation Linux (RedHat 7.1). Nous avons employé la machine virtuelle

Hotspot fournie avec le SDK 1.4 de Sun. Nous avons aussi employé la machine virtuelle fournie avec le SDK 1.2 de Sun. Nous avons employé la version 1.0.6 de Kaffe (JIT v3).

Calculs numériques intenses

Le programme *jBYTEmark* [IBM01] est disponible sur Internet. Il effectue une série de tests représentatifs des applications numériques. Ces tests mesurent l'efficacité de calculs numériques intenses et d'accès aux tableaux. Le Tableau 6.1 montre les résultats obtenus par différentes machines virtuelles, J2S et JBCC.

Test	Hotspot	Kaffe	JDK-1.2	J2S	JBCC
	(iter/sec)	(iter/sec)	(iter/sec)	(iter/sec)	(iter/sec)
Numeric Sort	147,06	122,40	63,65	20,71	**74,96**
String Sort	47,62	43,01	33,33	2,83	**36,03**
Bitfield Operations	1,55E7	1,39E7	3,51E7	3,41E6	**1,74E7**
FP Emulation	25,00	12,20	11,70	0,15	**16,26**
Fourier	4071,43	6443,55	7537,74	1442,03	**8094,59**
Assignment	3,85	2,55	1,95	0,72	**1,31**
IDEA Encryption	187,50	178,22	87,50	3,09	**333,33**
Huffman Comp.	181,82	174,76	173,91	8,77	**149,53**
Neural Net	14,81	4,75	8,00	1,13	**11,11**
LU Decomposition	119,05	21,28	47,62	5,49	**45,04**

TAB. 6.1 – jBYTEmark

Comme prévu, les performances de notre compilateur sont du même ordre de grandeur que les performances des autres compilateurs. Le Tableau 6.1 démontre que JBCC ne possède pas les lacunes de J2S pour les calculs numériques. Les performances de JBCC sont du même ordre de grandeur que les autres compilateurs, contrairement à J2S. Cela s'explique par le fait que les calculs sont effectués directement par le processeur sans ajustements ou vérifications dynamiques.

Il est à noter que l'analyse de type sur les domaines entiers perd son utilité lorsque le code généré sur l'architecture ciblée possède les mêmes contraintes d'évaluation numérique. Cette analyse s'avère redondante sous l'architecture Intel.

Le calcul de la suite de Fibonacci n'utilise aucune fonctionnalité spécifique aux librairies Java. Nous avons mesuré l'efficacité du calcul de la suite de Fibonacci pour déterminer le coût associé à la convention d'appel. L'application effectue peu de calculs numériques et beaucoup d'appels de méthode récursifs. Le Tableau 6.2 montre les résultats obtenus par le compilateur

JBCC.

N	fib(N)	Hotspot (ms)	JBCC (ms)
30	832040	33	**41**
31	1346269	52	**62**
32	2178309	79	**98**
33	3524578	125	**151**
34	5702887	190	**240**
35	9227465	307	**385**
36	14930352	499	**619**
37	24157817	821	**997**
38	39088169	1310	**1606**
39	63245986	2120	**2590**

TAB. 6.2 – Fibonacci

Nous observons que le coût des appels de méthode est du même ordre de grandeur que celui de *Hotspot*. Dans ce test, *Hotspot* est environ 25% plus performant que notre compilateur.

Le problème N-Queens permet de démontrer les performances d'une exécution d'application effectuant des calculs numériques divers, les accès aux tableaux et la convention d'appel. Cette application démontre les performances de notre compilateur sur le langage Java* isolé de la JVM et ses librairies natives. Elle permet de juger la performance d'une application ne possédant pas les propriétés dynamiques de Java.

N	Nb solutions	Hotspot (ms)	JBCC (ms)
10	724	65	**84**
11	2680	321	**411**
12	14200	1903	**2529**
13	73712	11965	**17355**
14	365596	82696	**108395**
15	2279184	587580	**806637**

TAB. 6.3 – N-Queens

Pour le test des N-Queens, *Hotspot* est environ 30% plus efficace que notre compilateur.

Les Tableaux 6.2 et 6.3, qui illustrent deux applications hautement récursives, montrent les

performances de la convention d'appel de méthode et de gestion de la pile d'exécution. JBCC n'effectue pas d'allocation de registres inter-procédurale et n'effectue pas d'*inlining* de fonctions. Les tests de débordement de pile sont effectués par des mécanismes matériels pour la majorité des appels de fonction. Seuls les blocs d'activation dont la taille dépasse quatre kilo-octets, taille d'une page mémoire, doivent être vérifiés explicitement, ce qui n'arrive jamais dans nos tests.

Application parallèle et librairies

Comme *Hotspot* emploie le même modèle de système de threads, nous nous attendions à ce que les performances du système de threads de JBCC soient similaires à celle de *Hotspot*. Pour vérifier ce phénomène, et pour comparer l'implantation du système de threads de J2S à celle de JBCC, nous avons effectué des tests sur les performances des applications parallèles générées par notre compilateur. L'application recherche en parallèle, dans un répertoire et tous ses sous-répertoires, le fichier Java possédant le plus grand nombre de lignes.

N Thread	Hotspot (ms)	J2S (ms)	JBCC (ms)
1	10 744	7 604	**13 867**
2	5 520	3 844	**7 107**
3	3 746	2 546	**4 869**
4	3 093	1 970	**4 212**
5	3 085	1 567	**3 927**
6	3 086	1 329	**3 715**
7	2 976	1 108	**3 709**
8	2 997	997	**3 720**
9	3 016	908	**3 697**
10	2 993	826	**3 711**
11	3 047	776	**3 709**
12	2 966	687	**3 702**
13	2 986	675	**3 712**
14	2 967	625	**3 704**
15	2 963	614	**3 696**

TAB. 6.4 – Recherche de fichier (69 fichiers, 145 threads)

Le niveau de concurrence indique le nombre maximal de thread s'exécutant en parallèle. Il est à noter que des exécutions répétitives de ce programme test sont nécessaires pour absorber l'impact du phénomène de cache créé par le système d'exploitation et le matériel pour l'accès

au système de fichiers.

Nous pouvons observer le comportement du système de threads de JBCC au Tableau 6.4. Comme JBCC emploie les processus lourds du système d'exploitation, son comportement est similaire à celui de *Hotspot*. Nous pouvons comparer les performances d'un système de threads lourds et d'un système de threads légers en comparant les performances de JBCC et J2S sur l'application de recherche de fichiers en parallèle.

6.4.2 Analyse

Dans cette section nous analysons les performances obtenues par le compilateur JBCC pour les applications tests présentées dans la section précédente.

Calculs numériques intenses

Les calculs numériques de JBCC ne nécessitent pas d'ajustements et de tests dynamiques comme J2S car il emploie les mécanismes du processeur pour effectuer le calcul. Ces mécanimes respectent la sémantique du langage Java*. JBCC obtient des performances similaires à *Hotspot* contrairement à J2S.

Par contre, notre compilateur JBCC n'obtient pas des performances supérieures à *Hotspot*. Nous expliquons les bonnes performances de *Hotspot* par ses nombreuses optimisations locales. Notre compilateur JBCC a axé ses analyses et optimisations sur une précision et amélioration globale du programme qui améliore les aspects globaux, ce qu'un compilateur dynamique comme *Hotspot* ne peut pas effectuer dans un temps de compilation limité. Pour obtenir des performances sur les calculs numériques intenses, notre compilateur doit améliorer ses analyses et optimisations locales.

Applications parallèles

Les résultats de JBCC pour les applications parallèles étaient prévisibles car il emploie la même librairie de thread que *Hotspot*. Une architecture de threads possédant les mêmes propriétés que J2S serait intéressante pour un compilateur comme JBCC.

Pile d'exécution

L'emploi des mécanismes de l'architecture permet au compilateur d'introduire des mécanismes peu coûteux pour la détection de débordement de pile. Ces optimisations permettent à JBCC d'améliorer ses performances générales d'exécution et les performances des méthodes récursives.

L'emploi d'une pile d'exécution simulée qui n'utilise pas les registres standards de l'architecture donne de mauvaises performances et peut doubler le temps d'exécution. Les architectures modernes sont optimisées pour le cas général qui utilise les registres standards de l'architecture.

Mémoire cache

La mémoire cache est devenue, avec les processeurs modernes, un point chaud d'optimisation. Malheureusement notre compilateur n'effectue aucune analyse ou optimisation reliée à la cache.

Des tests que nous avons effectués pour copier des données d'un tableau à un autre montrent un potentiel d'accélération promettant de doubler la performance de certaines applications. L'utilisation d'instructions SSE comme `prefetch`, qui charge en cache une donnée avant son utilisation, améliorerait les performances considérablement si l'application effectue beaucoup de traitement en mémoire.

Caractéristiques de la compilation

Nous observons dans le Tableau 6.5 la taille des différents fichiers générés par les compilateurs.

Test	Taille (debug) (octets)	Taille (sans debug) (octets)	*bytecode* (octets)
Fib	146317	60672	873
jBYTEmark	408944	206720	79155
Parallele	171492	72256	4838

TAB. 6.5 – Dimensions des exécutables

Nous observons que les programmes en *bytecode* sont très compacts. Cette propriété, prévu par les concepteurs de Java, permet de transférer rapidement une application sur le réseau. Par contre, la machine hôte doit posséder une machine virtuelle et son environnement d'exécution.

Pour *Hotspot* la machine virtuelle fait plus de 5 megaoctets (21252 octets *java* sans ses librairies et 5003433 octets *client/libjvm.so*) et son environnement d'exécution fait au moins 20 megaoctets (23556160 octets *rt.jar*). Nous observons que le code produit par notre compilateur est plus grand que le *bytecode*. Par contre, comparé à l'ensemble des librairies requises (même en *bytecode*), notre compilateur produit un petit exécutable possédant sa propre copie nettoyée et optimisée de l'environnement d'exécution.

6.5 Discussion

JBCC est un prototype et plusieurs améliorations pourraient y être apportées. Le compilateur effectue de bonnes analyses et optimisations globales mais les optimisations offrant des gains de performance spécifiques à l'architecture n'ont pas été intégrées à notre compilateur. La qualité de code généré pour l'architecture Intel est comparable à celle d'un compilateur simple. Avec ses optimisations globales, le compilateur JBCC obtient des performances comparables aux autres compilateurs Java (parfois plus rapide, lorsque les gains d'analyses et optimisations globales sont rentables, et parfois plus lente, lorsque les analyses et optimisations spécifiques à l'architecture effectués par les autres compilateurs donnent de bons gains).

La compilation vers un langage de plus bas niveau laisse plus de flexibilité au compilateur mais demande de prévoir plusieurs analyses dépendantes de l'architecture pour obtenir de bonnes performances. Les mécanismes matériels offrant des possibilités d'optimisation améliorent la performance des applications et un compilateur ne peut pas les ignorer sans perte de vitesse d'exécution.

Les performances de JBCC sont similaires à celles d'autres compilateurs et machines virtuelles répendus. L'amélioration de la partie dorsale de JBCC pourrait l'amener parmis les compilateurs Java les plus efficaces. Ses performances montrent l'impact des propriétés dynamiques d'un langage sur les analyses et optimisation d'un compilateur et par conséquent, la performance globale d'une application.

La taille de l'application générée par notre compilateur est relativement petite et ne contient pas de dépendance sur un environnement d'exécution présent sur la machine hôte.

L'emploi d'une architecture de bas niveau évoluant rapidement limite la portabilité des analyses. Par exemple, une analyse optimisant les accès mémoire en cache à l'aide de l'instruction `prefetch` dépend de la vitesse d'accès à la mémoire. La distance de préchargement peut doubler si le temps d'accès à la mémoire double. De plus, la taille de la mémoire cache, le nombre de niveaux de mémoire cache et les algorithmes de remplacement et de cohérence peuvent modifier les temps et la méthode d'accès à une donnée.

Le système de threads basé sur la même implantation que *Hotspot* lui donne les mêmes propriétés. JBCC est plus lent que J2S sur les applications parallèles. Ces performances parallèles étaient prévisibles vu l'implantation du système de threads employée. J2S et JBCC permettent d'observer l'impact de l'implantation des librairies fournies avec le langage. Les deux compilateurs effectuent les mêmes analyses et optimisations globales. Malgré ces analyses, J2S obtient de meilleures performances que JBCC sur les applications parallèles. Les performances d'un langage fournissant de nombreuses librairies reposent sur une bonne implantation de ces librairies. Nous discutons de nos observations de l'impact des librairies dans le Chapitre 7.

La séparation d'un compilateur en couche communiquant par langage intermédiaire permet d'isoler les analyses et optimisations dépendantes de l'architecture et celles dépendantes du langage. Pour suivre l'évolution d'une architecture ou d'un langage, il suffit de modifier la couche affectée.

L'implantation d'un compilateur et de toutes ses couches est une tâche laborieuse. La taille du code rend complexe le développement de ce type d'application. Ce modèle de compilateur sacrifie la portabilité, la stabilité et le temps de développement pour des gains de performance.

Chapitre 7

Environnement d'exécution

Après avoir observé nos deux versions de compilateur et comparé leurs performances pour différentes applications, nous avons remarqué l'importance qu'a l'implantation de l'environnement d'exécution sur les performances obtenues par les applications. Par exemple, les performances des applications parallèles observées au Tableau 6.4 montrent l'impact qu'a l'implantation du système de threads et des entrées/sorties fournie par l'environnement virtuel.

Pour obtenir des applications performantes, le compilateur doit fournir un environnement d'exécution et des librairies efficaces. Dans ce chapitre nous discutons des modèles d'environnement d'exécution employés par notre compilateur et par les différentes machines virtuelles ainsi que leur impact sur les performances des applications Java.

7.1 Système de threads

Étant donné que la concurrence est une notion importante dans les applications Java, le système de threads devient un point chaud pour l'environnement d'exécution. Dans cette section nous discutons des modèles de système de threads, de certains de leurs aspects et de leurs impacts sur les performances d'applications Java.

7.1.1 Processus lourds et processus légers

Nous pouvons diviser les systèmes de threads actuels en deux catégories : les systèmes à processus lourds et les systèmes à processus légers.

Les systèmes à processus lourds emploient les processus du système d'exploitation. Les coûts de gestion et de synchronisation sont dépendants du coût d'appel aux services du système d'exploitation et sont généralement élevés.

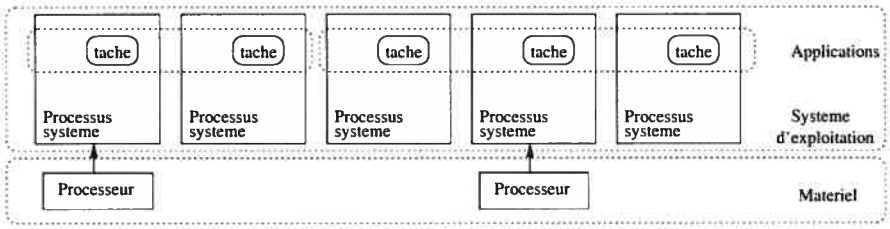

FIG. 7.1 – Système de threads à processus lourds

La Figure 7.1 montre un exemple de système à processus lourds. Les deux applications, contenant respectivement deux et trois tâches, s'exécutent sur deux processeurs de façon parallèle. Notons qu'il est possible d'obtenir du parallélisme entre les applications et entre les tâches de la même application.

Les systèmes à processus légers gèrent dans un processus lourd plusieurs processus parallèles légers. Les mécanismes de gestion et de synchronisation ne passent pas par les mécanismes coûteux des appels systèmes.

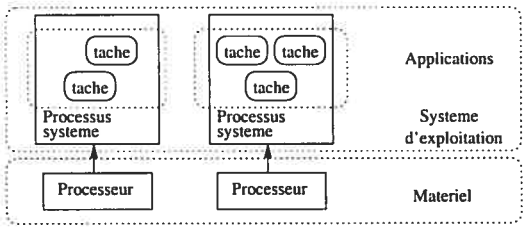

FIG. 7.2 – Système de threads à processus légers

La Figure 7.2 montre un exemple de système à processus légers. Il est possible d'obtenir du parallélisme entre les applications, mais il est impossible d'en obtenir entre les tâches d'une même application.

Pour les applications créant des processus parallèles pour simplifier la conception et l'architecture de l'application plutôt que dans le but d'accélérer leur exécution, les processus légers offrent des propriétés intéressantes. La notion de programmation par tâches communicantes est proche des concepts de la programmation orientée objet.

Le type de système de threads employé dans un environnement d'exécution influence la

conception de toutes les autres composantes de l'environnement d'exécution. Par exemple, un environnement d'exécution utilisant des processus légers sous Linux ne peut pas employer les librairies d'entrées et sorties bloquantes sans introduire une couche d'interface vers le système d'exploitation. Le système d'exploitation n'ayant aucune notion des processus légers, l'un d'eux pourrait bloquer le système de threads sur la lecture d'un *socket*, ce faisant empêchant un autre processus léger d'écrire sur ce *socket*. Une étreinte fatale (*deadlock*) en résulte.

Les deux systèmes possèdent des avantages et des inconvénients. Un système à processus lourds permet du vrai parallélisme (multi-processeurs) tandis qu'un système à processus légers se voit limité par les politiques de gestion de processus du système d'exploitation. Par contre, le choix d'un système à processus légers diminue les coûts de gestion et offre de meilleures performances. Le choix du type de système repose sur le type d'applications exécutées, le matériel, l'architecture et le système d'exploitation.

Un modèle de système de thread MxN est un juste milieu entre les deux modèles qui donne des performances intéressantes. Le système crée M processus lourds, généralement dépendant du nombre de processeurs, pour N processus légers, dépendant des besoins de l'application.

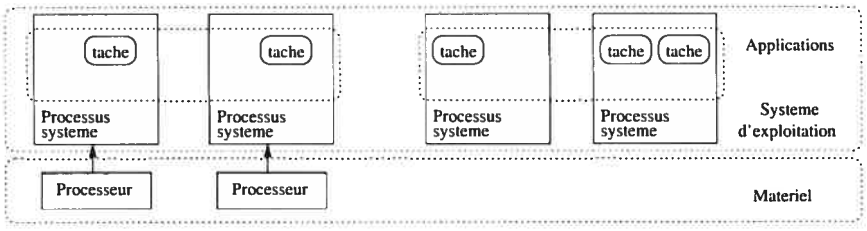

FIG. 7.3 – Système de threads mixte

La Figure 7.3 montre deux applications employant un système de threads mixte. Il est possible d'obtenir du parallélisme entre les deux applications et entre les tâches d'une même application.

Certaines librairies, tels LWT de Solaris et le système de threads de Windows, permettent la cohabitation de processus légers et de processus lourds. Elles fournissent l'implantation d'un système de threads mixte pour un environnement d'exécution.

7.1.2 Modèle de mémoire

Les propriétés parallèles fournies par la JVM [LY99b] offrent une grande liberté d'implantation pour les concepteurs d'environnements d'exécution. L'architecture du système de threads de Java supporte des modèles de mémoire relaxés [GLL+90], les DSM (*Distributed Shared Me-*

mory) [KCZ92] et les architectures multi-processeurs avec mémoire commune ou distribuée. Beaucoup de programmeurs supposent des propriétés fausses et emploient de manière erronée les propriétés parallèles de Java.

Chaque processus peut posséder une copie de la mémoire qui ne sera mise à jour que lors de la rencontre d'une barrière de synchronisation. Un processus peut conserver une copie locale d'un objet, d'un tableau ou d'une certaine partie de tableau et ne soumettre les changements qu'à la rencontre d'une barrière de synchronisation. Peu d'applications parallèles Java respectent ces contraintes.

Le non respect du modèle de mémoire de Java et le non respect des contraintes ont forcé les développeurs des JVM à supporter des modèles de mémoire plus communs et moins relaxés.

7.1.3 Implantation

Le système de threads de Gambit-C emploie des processus légers ce qui donne un système supportant un grand nombre de processus et un faible coût de gestion. Par contre, ce système n'est pas multi-processeurs. Les mécanismes de la pile d'exécution de Gambit-C l'adaptent dynamiquement aux besoins des threads. J2S emploie directement le système de threads de Gambit-C. L'interface du système de threads de Gambit-C est décrite dans [Fee98].

Hotspot et JBCC emploient le système de threads de POSIX (*pthread*) [ISO96, UNI]. Ce système de threads sous Linux emploie des processus lourds et possède de grands coûts de gestion. De plus, les contraintes de taille de la pile d'exécution limitent le nombre de threads.

Le Tableau 5.2 et le Tableau 5.3 montrent les performances du système de threads légers de Gambit-C pour une application hautement parallèle. Les systèmes de threads lourds ne peuvent pas supporter le nombre élevé de processus et ne terminent pas les calculs.

Le compilateur GCJ [GCC] supporte une implantation d'un système de threads mixte.

7.1.4 Système de threads spécifique pour Java

Java possède des propriétés pour la définition d'un système de threads évolué. Un sujet d'étude intéressant serait l'implantation d'un système de threads spécifique au langage qui suppose les contraintes du langage et de son modèle de mémoire relaxé pour améliorer son efficacité.

La notion de pointeur étant limitée aux références, la gestion de la mémoire est facilitée. La gestion de la pile d'exécution, handicap de plusieurs systèmes de threads, se voit simplifiée puisqu'il est possible d'obtenir une pile sans pointeurs entrants ce qui facilite le déplacement des

blocs d'activation sur la pile et enlève les contraintes sur la modification de la pile d'exécution rencontrées par les systèmes actuels.

Le modèle de mémoire de Java lui permet une implantation distribuée. En théorie, il serait possible d'excécuter un processus pendant qu'un autre collecte des éléments morts de sa mémoire tant qu'une barrière de synchronisation n'est pas rencontrée.

7.2 Ramasse-miettes

Le langage Java* n'autorise pas la gestion explicite de la mémoire. La mémoire est gérée par une librairie externe hors du contrôle du programmeur. De plus, le ramasse-miettes limite les erreurs d'utilisation de la mémoire. Il facilite le développement d'applications en éliminant la tâche laborieuse qu'est la gestion manuelle de la mémoire. L'emploi d'un ramasse-miettes se justifie par le faible coût qui lui est associé comparé aux avantages qu'il offre.

Java demande une bonne implantation du ramasse-miettes car le style de programmation fortement orientée objet qu'il promeut crée beaucoup d'objets dont la majorité ont un temps de vie très court. Nos applications de tests emploient peu de mémoire et ne permettent pas d'évaluer le coût réel associé à cette librairie. Les applications développées en Java sont généralement de petite taille et utilisent peu de mémoire. Par contre, la fréquence d'exécution du ramasse-miettes doit être élevée pour libérer le grand nombre de petits objets morts rapidement.

Hotspot utilise un algorithme de gestion de mémoire évolué connu sous l'algorithme par train [SG95]. Malgré quelques problèmes rares de temps de nettoyage causés par une configuration spéciale d'objets en mémoire, il offre de bonnes performances. L'algorithme permet des collectes partielles et parallèles. Il est très bien adapté au langage Java.

J2S emploie le ramasse-miettes copiant de Gambit-C. Ce type de ramasse-miettes offre de bonnes performances pour Java puisqu'en moyenne, la majorité des objets sont morts entre les passes de collecte, ce qui élimine la plupart des déplacements. De plus, la collecte d'objets compacte la mémoire et améliore la localité des données. La version du ramasse-miettes de Gambit-C ne fonctionne pas en parallèle. Son algorithme est général et efficace pour la majorité des langages, offrant ainsi une librairie pratique pour les compilateurs employant comme langage cible Scheme [FL98].

JBCC emploie une librairie de gestion de mémoire écrite par Hans-J. Boehm [HC92]. Ce ramasse-miettes est de type marque-et-collecte (*mark and sweep*). Son algorithme conservateur a été développé pour les langages C et C++. Vu la similarité des langages, il s'adapte bien pour Java*. Étant conservateur et ne tirant pas profit du contrôle serré de la mémoire de Java, il offre

de moins bonnes performances que les ramasse-miettes utilisés par *Hotspot* et J2S.

Plusieurs autres systèmes de gestion de mémoire offrant des propriétés différentes sont disponibles [BB02].

7.3 Entrées/sorties

Les mécanismes d'entrées/sorties sont dépendants du système d'exploitation. Par raison de portabilité, les concepteurs de Java ont fait une couche d'abstraction simplifiant l'implantation de ces mécanismes sur plusieurs systèmes d'exploitation. La couche d'abstraction possède un coût élevé puisque la majorité des entrées/sorties s'effectue caractère par caractère. Il existe des primitives d'accès par bloc mais les méthodes des librairies de Java ne peuvent pas les employer puisque le programmeur peut dériver une classe d'entrées/sorties pour obtenir un nouveau canal d'entrée/sortie spécifique à ses besoins.

Java 2 propose des nouveaux mécanismes d'entrées/sorties corrigeant plusieurs problèmes des anciennes versions. Il introduit la notion de *channel*, de multiplexage et des entrées/sorties non bloquantes pour permettre l'implantation de serveurs web évolutifs (*scalable*) et d'autres applications nécessitant de bonnes performances pour les entrées/sorties. Les nouveaux services se retrouvent dans le paquetage `java.nio`. Ils ne sont pas implantés ni par J2S ni par JBCC.

7.4 Conclusion

Les deux versions de notre compilateur nous permettent de comparer différentes implantations de librairies. Nous ne pouvons pas conclure qu'il existe pour chaque librairie une implantation plus efficace que les autres. L'efficacité des implantations varie selon les besoins de l'application. Par exemple, le choix d'implantation du système de threads pour un environnement virtuel varie selon l'application parallèle exécuté.

Nous avons remarqué une plus grande amélioration des performances dans le choix d'implantation des librairies que dans l'ajout d'analyses et optimisations globales. Les choix d'implantations améliorent les performances spécifiques au module visé tandis que les analyses et optimisations globales améliorent les performances générales des applications.

Chapitre 8

Conclusion

Dans cette section nous discutons de notre vision de la compilation pour un langage comme Java* et des formes de compilation qui lui permettent d'obtenir de bonnes performances.

L'évolution des concepts des langages de programmation, du style de programmation et du design des applications va modifier les techniques employées par les compilateurs. Les architectes de compilateurs doivent retourner à leurs tables à dessin car les modèles de compilateur actuels ne pourront plus subvenir aux besoins des nouvelles applications.

8.1 Domaines d'application

Au cours de notre recherche, nous avons étudié différents types de compilateur. Chaque type de compilateur possède ses avantages et ses inconvénients. Les concepteurs de compilateur définissent l'architecture du compilateur selon le domaine d'application ciblé.

Les compilateurs dynamiques comme *Hotspot* et *Kaffe* ciblent des applications qui emploient des mécanismes dynamiques de Java. Par exemple, les applications distribuées nécessitent des mécanismes dynamiques. Java* ne possède pas ces propriétés. Nos compilateurs ne ciblent pas les applications dynamiques. L'absence des propriétés dynamiques permet à J2S et JBCC d'effectuer des analyses plus précises.

L'idée d'une compilation statique globale permet au compilateur de supposer qu'il peut obtenir l'ensemble complet du code source. De nombreuses applications Java n'utilisent aucunes propriétés dynamiques et se prêtent bien à la compilation statique globale. Un compilateur comme JBCC qui implanterait plus d'optimisations locales pourraient obtenir de meilleures

performances que *Hotspot*.

Les applications embarqués, qui possèdent un espace d'exécution restreint, ne peuvent pas contenir l'ensemble des définitions de classes de l'environnement d'exécution de Java. Nos compilateurs, à l'aide d'analyses précises, nettoient la majorité du code inutile et diminue ainsi l'espace requis. Ainsi, l'idée d'une compilation globale de Java* se prête bien aux systèmes embarqués.

Le choix du compilateur cible de J2S est pour profiter de l'environnement d'exécution de Gambit-C soit un système de threads légers et une librairie d'entrées/sorties tamponnisées. Les applications ciblées par J2S sont les applications multi-threadées effectuant beaucoup d'entrées/sorties. Les serveurs web sont des applications parallèles ne nécessitant pas de propriétés dynamiques et qui effectuent beaucoup d'entrées/sorties.

La compilation dynamique introduit un coût de compilation lors de l'exécution du programme. Ce coût est négligeable pour la majorité des applications mais ne l'est pas pour une application où le temps de réponse est important. Une application compilée statiquement n'engendre pas de coût supplémentaire à l'exécution et est par conséquent mieux adaptée pour un environnement temps réel.

8.2 Contributions

Dans le cadre de notre recherche, nos contributions sont l'analyse de la faisabilité d'un compilateur par couche pour le langage Java vers le langage Scheme et les solutions aux problèmes reliés à ce type de compilation. Pour contourner les problèmes des performances numériques de J2S nous avons implanté des analyses statiques globales. Nous avons aussi comparé les différents modèles de compilateur et implanté un compilateur ayant une structure classique (JBCC).

8.3 Analyses globales

Nous avons vu que les analyses statiques globales offrent une richesse de renseignements. Par contre, leurs coûts élevés rendent leurs utilisations pour le développement d'applications très difficiles.

Les applications tendent vers des modèles basés sur les modules et orientés objets pour faciliter leur conception, leur implantation et leur gestion. Le développement modulaire produit un code plus aisément compréhensible pour un humain et facilite le développement en parallèle. L'abstraction et la séparation en modules ajoutent un coût global sur les performances d'une application. Un compilateur effectuant de la compilation séparée peut réaliser ses analyses dans

un temps raisonnable, contrairement aux compilateurs effectuant des analyses globales coûteuses. Il est fréquent de rencontrer du code redondant dans plusieurs modules ou du code de vérification dont les instructions sont mortes. Les analyses globales statiques peuvent résoudre ces problèmes en effectuant les analyses complexes requises.

Le développement rapide des applications et les nouvelles techniques de génération automatique de code rendent la tâche du compilateur plus ardue. La dimension des programmes augmente rapidement et la complexité des analyses globales précises devient trop coûteuse pour être réalisable en un temps raisonnable.

Alors que nous observons une tendance des compilateurs à rapprocher leurs analyses au niveau de l'architecture, nous observons un besoin d'analyse globale pour éviter une pollution des gros codes difficilement gérables par des êtres humains. Peut-être que l'avenir des analyses globales n'est plus au niveau des compilateurs mais comme assistants de développement effectuant les tâches complexes d'analyse, de recherche, de documentation et de vérification que les programmeurs n'effectuent pas.

Les analyses globales statiques offrent des avantages à ne pas négliger, mais des mécanismes pour les rendre plus performantes devraient être étudiés.

8.4 Environnement d'exécution

La conception de l'environnement d'exécution influence grandement les performances des applications puisque ces applications reposent sur les services fournis par le langage et l'environnement d'exécution. Le concepteur d'un compilateur doit construire son modèle selon les types d'applications visées. Un modèle spécifique au langage et au type d'application visée offre de meilleures performances mais possède un coût de développement à ne pas négliger. L'étape de conception des librairies et de l'environnement doit se faire avant la conception du compilateur et de ses analyses et optimisations ; l'architecture et le fonctionnement interne de l'environnement sont extrêmement liés aux performances des applications.

8.5 Modèles de compilation

Nous avons discuté dans la section 2.5 de la différence entre la compilation statique et la compilation dynamique. Nous ne pensons pas qu'une approche soit définitivement meilleure que l'autre. Nous discutons dans cette section du modèle de compilation qui nous semble prometteur.

8.5.1 Compilation statique

La compilation statique permet au compilateur d'employer des algorithmes avec une complexité d'analyse élevée puisque le coût de compilation n'est pas ajouté au coût d'exécution. Elle se prête bien aux analyses globales coûteuses et dépendantes du langage.

Les analyses classiques ne subissent pas une évolution liée au matériel mais plutôt liée au langage et ses concepts. Des analyses statiques datant des premiers compilateurs restent valides et sont fortement employées de nos jours.

Malgré le fait que la compilation statique ait subi une perte de popularité face à la compilation dynamique, elle possède toujours ses avantages à ne pas négliger.

8.5.2 Compilation dynamique

La compilation dynamique, la nouvelle mode dans la compilation des langages, offre des performances surprenantes. Les gains obtenus par ces compilateurs se justifient par l'évolution du matériel et des gains très élevés associés aux transformations de code spécifiques à l'architecture.

La compilation dynamique se prête bien aux analyses locales fortement dépendantes des données dynamiques qui peuvent varier d'une exécution à l'autre ou d'une architecture à l'autre. Elle se prête bien aux analyses comme le préchargement de données en mémoire ou le déroulement de boucles car ces optimisations sont très dépendantes de l'environnement d'exécution et de l'architecture. Par exemple, les optimisations liées à la mémoire varient d'un ordinateur à l'autre puisqu'ils ne possèdent ni le même type ni la même quantité de mémoire.

Un compilateur dynamique effectue aussi des transformations de programme spécifiques au comportement de l'application. Deux exécutions de la même application peuvent effectuer des tâches différentes demandant des optimisations spécifiques. Ainsi, un algorithme de tri pourrait être optimisé selon les données qu'il reçoit. Cette propriété permet au code de s'adapter dynamiquement.

8.5.3 Modèle hybride

Nous pensons que l'avenir de la compilation réside dans un hybride de ces techniques. Les optimisations dépendantes du langage devrait se faire statiquement et les optimisations liées au matériel devrait se faire dynamiquement.

Les compilateurs statiques devraient générer un langage de plus haut niveau que l'assembleur

traditionnel et laisser un compilateur dynamique adapter le code et générer des instructions spécifiques à l'architecture.

Le modèle de compilation employé par *Hotspot* emploie déjà cette idée. *Javac* transforme le code source en *bytecode* pour être dynamiquement compilé. Le langage intermédiaire devrait contenir des informations comme les structures de contrôle pour éviter aux compilateurs dynamiques d'effectuer une analyse de flux de contrôle. Il pourrait même inclure un PDG ou d'autres informations pertinentes aux optimisations.

8.5.4 Compilation par couches

Nous pensons que l'utilisation d'un langage intermédiaire, comme Scheme, offre une couche intéressante pour l'abstraction de l'architecture. Le compilateur J2S offre un exemple de plusieurs couches de compilateurs séparant les niveaux de compilation et séparant les services fournis. La première couche, J2S, effectue les analyses globales dépendantes du langage. La seconde couche effectue la compilation du langage intermédiaire vers le langage C en effectuant des optimisations locales. La troisième couche, un compilateur C, effectue la compilation vers l'assembleur dépendante de l'architecture.

Ce type de compilation correspond bien aux besoins actuels. Il offre une portabilité par l'abstraction des langages intermédiaires indépendants d'une architecture spécifique. Chaque niveau de langage possède ses propriétés et ses services. De plus, il divise la tâche complexe qu'est l'implantation d'un compilateur. On retrouve cette idée de couche dans la majorité des compilateurs modernes mais les couches ne sont pas isolées en compilateurs indépendants et généralement sont dépendantes l'une de l'autre pour raison d'efficacité.

Ces abstractions ont leurs coûts comme nous l'avons observé avec J2S et Gambit-C. Le langage Scheme n'offre pas les mêmes contraintes d'évaluation de calculs numériques que Java. Par conséquent, des mécanismes d'adaptation doivent modifier l'évaluation d'une expression pour respecter la sémantique du langage source. Il est difficile de prévoir un langage intermédiaire de moyen ou de bas niveau offrant les services optimisés pour tout langage de haut niveau. Une simple différence entre une couche et une de ses couches inférieures peut engendrer des coûts non négligeables. Le compilateur J2S serait très performant si l'évaluation des calculs respectait les mêmes contraintes. Or Scheme possède un système d'évaluation numérique différent de celui de Java et le coût de conversion est non négligeable pour des applications nécessitant de bonnes performances.

Si la portabilité, la stabilité et le temps de développement sont des points plus importants que l'efficacité du code généré, l'approche J2S a fait ses preuves. Le compilateur J2S a été développé

en moins de 6 mois contrairement au compilateur JBCC qui nécessita 2 ans.

Nous conseillons l'implantation d'un compilateur sous le modèle de J2S. Vu l'évolution du matériel et des concepts des langages, il est mieux d'isoler le langage de l'architecture pour faciliter l'évolution du compilateur.

8.5.5 Futurs modèles

Plusieurs idées modernes promettent de modifier les techniques de compilation pour régler certains problèmes rencontrés avec l'évolution des programmes. Par exemple, pour diminuer le coût des vérifications dynamiques, l'idée d'un compilateur certifiant offrant un code authentifié et non malicieux pourrait être intéressante. Un compilateur avec système de preuves permettrait d'obtenir un code avec les garanties d'un langage sûr sans effectuer des vérifications dynamiques. Un compilateur distribué pourrait adapter génétiquement le code pour faire évoluer ses performances.

On doit s'attendre à des changements majeurs dans les techniques de compilation et les modèles de compilateur. Il reste à voir sur quels modèles précis ces derniers seront basés.

Bibliographie

[AGMM00] Pedro V. Artigas, Manish Gupta, Samuel P. Midkijj, and José E. Moreira. Automatic Loop Transformation and Parallelization for Java. *Parallel Processing Letters*, 2000.

[AMD01] AMD. *AMD Athlon Processors : x86 optimisation guide.* AMD, 2001.

[App99] Andrew W. Appel. *Modern compiler implementation in Java.* 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques and tools.* 1986.

[BB02] Richard Brooksby and Nicholas Barnes. The Memory Pool System. 2002.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : A transparent Dynamic Optimisation System. Technical report, Hewlett-Packard Labs, 2000.

[BDBV94] S. J. Bushell, Alan Dearle, Alfred L. Brown, and Francis Vaughan. Using C as a Compiler Target Language for Native Code Generation in Persistent Systems. In *POS*, pages 164–183, 1994.

[Ber97] Peter Berteksen. Semantic of Java Byte Code. Technical report, april 1997.

[BKM93] Robert A. Ballance, Ksheeerabdhi Krishna, and Arthur B. Maccabe. Program Dependence Graphs for the rest of us. 1993.

[BL93] Thomas Ball and James R. Larus. Branch Prediction For Free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.

[Bou99] Dominique Boucher. *Analyse et optimisation globales de modules compilés séparés.* PhD thesis, Université de Montréal, 1999.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[BS99] E. Borger and W. Schulte. Modular design for the Java Virtual Machine architecture, 1999.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Language*, 1981.

[Cha82] Gregory J. Chaitin. Register Allocation and Spilling via Graph Coloring. *ACM SIGPLAN*, 1982.

[CJM+96] Chandra Chekuri, Richard Johnson, Rajeev Motwani, B. Natarajan, B. Ramakrishna Rau, and Michael S. Schlansker. Profile-driven Instruction Level Parallel Scheduling with Application to Super Blocks. In *International Symposium on Microarchitecture*, pages 58–67, 1996.

[CR91] Wiliam Clinger and Jonathan Rees. *Revised 4 Report on the Algorithmic Language Scheme*, November 1991.

[Die97] Stephan Diehl. A Formal Introduction to the Compilation of Java. Technical report, jul 1997.

[Fee98] Marc Feeley. *Gambit-C, Version 3.0*, May 1998.

[FL98] Marc Feeley and Martin Larose. Compiling Erlang to Scheme. *Lecture Notes in Computer Science*, 1490 :300–317, 1998.

[GCC] GCC Compiler. http://gcc.gnu.org/.

[GHM00] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient Inference of Static Types for Java Bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[GJJB00] James Gosling, Bill Joy, Guy L. Steele Jr, and Gilad Bracha. *The Java[TM] Language Specification, Second Edition.* Sun Microsystems, 2000.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[Gos95] James Gosling. Java : an Overview. Technical report, Sun Microsystems, 1995.

[HC92] Boehm H. and D. Chase. A Proposal for Garbage-Collector-Safe C Compilation. *Journal of C Language Translation 4*, pages 126–141, December 1992.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* The computer science libray, 1977.

[HGmWH] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen mei W. Hwu. Java Bytecode to Native Code Translation : The Caffeine Prototype and Preliminary Results.

[HJU00] Aaron Hertzmann, Suresh Jagannathan, and Christian Ungureanu. The ILL Intermediate Language for Trail. Technical report, April 2000.

[Hoa74] Charles Antony Richard Hoare. Monitors : an Operating System Structuring Concept. *Communications of the ACM, 17(10)*, pages 549–57, 1974.

[HP96] John Hennessy and David A. Patterson. *Computer Architecture : A quantitative approach.* Morgan Kaufmaan, second edition, 1996.

[IBM01] IBM.
http://www-124.ibm.com/developerworks/oss/cvs/
jikesrvm/rvm/regression/tests/jbytemark, 2001.

[ISO96] ISO/IEC. Portable Operating System Interface (POSIX) – Part 1 : System Application : Program Interface (API) [C Language], 1996. ISO/IEC 9945-1.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Kra98] Andreas Krall. Efficient JavaVM Just-in-Time Compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, 1998. North-Holland.

[LGAT96] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Global Register Allocation Based on Graph Fusion. In *Languages and Compilers for Parallel Computing*, pages 246–265, 1996.

[Lia97] Sheng Liang. Java Native Interface Specification. Technical report, May 1997.

[LY99a] Tim Lindholm and Frank Yellin. *The Java[tm] Virtual Machine Specification.* Sun Microsystems, 1999.

[LY99b] Tim Lindholm and Frank Yellin. *The Java[tm] Virtual Machine Specification*, chapter 8, page 496. Sun Microsystems, 1999.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, second edition, 1997.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[NP94] Cindy Norris and Lori L. Pollock. Register Allocation over the Program Dependence Graph. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 266–277, 1994.

[Pat95] Jason R. C. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 67–78, jun 1995.

[Sar89] Vivek Sarkar. Determining Average Program Execution Times and their Variance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, 1989.

[Sar97] Vijay Saraswat. Java is not type-safe. Aug 1997.

[Sar00] Vivek Sarkar. Optimized Unrolling Nested Loops. *IBM*, 2000.

[SG95] Jacob Seligmann and Steffen Grarup. Incremental Mature Garbage Collection Using the Train Algorithm. *Ninth European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, 952 :235–252, 95.

[SSB] Robert Stark, Joachim Schmid, and Egon Börger. Java and the Java Virtual Machine - Definition, Verification, Validation.

[ST98] Peter F. Sweeney and Frank Tip. A Study of Dead Data Members in C++ Applications. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–332, 1998.

[Sun99] Sun. The Java Hotspot Performance Engine Architecture. Technical report, April 1999.

[Sun02] Sun. The Java HotSpotTM Virtual Machine, v1.4.1. Technical report, September 2002.

[UNI] UNIX The Open Group. http://www.unix-systems.org/.

[WL94] Youfeng Wu and James R. Larus. Static Branch Frequency and Program Profile Analysis. Technical Report CS-TR-1994-1248, 1994.

[Won99] W. F. Wong. Source Level Static Branch Prediction. *The Computer Journal*, 42(2) :142–149, 1999.

[Yel96] Frank Yellin. Java Native Code API. Technical report, Jul 1996.

Annexe A

Définition du langage JST

A.1 Grammaire du langage JST

$$
\begin{array}{rcl}
\textit{JST} & \Rightarrow & \{\textit{class} \in \textit{ClassDef}\}^* \ \{\textit{static-init-method} \in \textit{MethodRef}\}^* \\
\textit{ClassDef} & \Rightarrow & \textbf{CLASS } \textit{className} \in \textit{String} \\
 & & \quad \textbf{extends } \textit{className} \in \textit{String} \\
 & & \quad [\textbf{implements } \textit{className} \in \textit{String} \ \{,\textit{className} \in \textit{String}\}^*] \\
 & & \quad \{\textit{field} \in \textit{FieldDef}\}^* \\
 & & \quad \{\textit{method} \in \textit{MethodDef}\}^* \\
\textit{FieldDef} & \Rightarrow & \textbf{FIELD } \textit{fieldName} \in \textit{String} \ : \ \textit{type} \in \textit{Type} \\
\textit{MethodDef} & \Rightarrow & \textbf{METHOD } \textit{methodName} \in \textit{String} \ \ \textit{descriptor} \in \textit{String} \\
 & & \quad \textit{this} \in \textit{VarNode} \ : \ \textit{ret} \in \textit{VarNode} \\
 & & \quad (\{\textit{param} \in \textit{VarNode}\}^*) \\
 & & \quad [\textit{exceptions} \in \textit{ExceptionHandler}] \ [\textit{code} \in \textit{CodeNode}] \\
\textit{ExceptionHandler} & \Rightarrow & \textbf{ExceptionHandlers : } \textit{basicblockName} \in \textit{String} \ \{\textit{CatchClause}\}^* \\
\textit{CatchClause} & \Rightarrow & (v \in \textit{ExceptionVarNode}, \ t \in \textit{Type}, \ \textit{start} \in \textit{Int}, \ \textit{end} \in \textit{Int}) : \\
 & & \quad \textit{basicblockName} \in \textit{String} \\
\textit{CodeNode} & \Rightarrow & \textit{entry} \in \textit{Instruction} \ \{\textit{basicblock} \in \textit{BasicBlock}\}^* \\
 & | & \textbf{NativeCodeNode} \\
\textit{Block} & \Rightarrow & \{\{\textit{instruction} \in \textit{Instruction}\}^*\} \\
\textit{BasicBlock} & \Rightarrow & \textit{basicblockName} \in \textit{String} \\
 & & \quad (\{\textit{param} \in \textit{VarNode}\}^*) \\
 & & \quad \{\{\textit{instruction} \in \textit{Instruction}\}^*\} \\
\textit{Instruction} & \Rightarrow & \textit{L-Expr} \ | \ \textit{Block}
\end{array}
$$

L-expr ⇒ *R-Expr*
| **let** *var* ∈ *VarNode* *value* ∈ *R-expr*
| **if** *condition* ∈ *R-expr*
trueNode ∈ *Instruction*
else
falseNode ∈ *Instruction*
| **jump** *basicblockName* ∈ *String* ({*param* ∈ *R-expr*}*)

R-expr ⇒ *Type* | *Constant* | *VarNode* | *OpNode*
| *FieldRef* | *FieldSet* | *MethodRef*

Constant ⇒ *NumericConstant* | *StringConstant*
| *AddrNode* | *NullNode* | *VoidNode*

NumericConstant ⇒ *FixNumericConstant* | *FloNumericConstant*

VarNode ⇒ *LocalVarNode* | *TmpVarNode* | *ExceptionVarNode*

OpNode ⇒ *UnOpNode* | *BinOpNode* | *TriOpNode*

UnOpNode ⇒ *instr* ∈ *UnOpInstr* *type* ∈ *Type* *p1* ∈ *R-expr*

BiOpNode ⇒ *instr* ∈ *BiOpInstr* *type* ∈ *Type* *p1* ∈ *R-expr* *p2* ∈ *R-expr*

TriOpNode ⇒ *instr* ∈ *TriOpInstr* *type* ∈ *Type*
p1 ∈ *R-expr* *p2* ∈ *R-expr* *p3* ∈ *R-expr*

UnOpInstr ⇒ **not** | **neg** | **new** | **throw**
| **newarray** | **lock** | **unlock**
| **null** | **checknull** | **return**
| **d2f** | **d2i** | **d2l** | **f2d** | **f2i** | **f2l**
| **i2b** | **i2c** | **i2d** | **i2f** | **i2l** | **i2s** | **i2z**
| **l2f** | **l2i** | **l2d**
| **c2i** | **s2i** | **b2i** | **z2i**

BiOpInstr ⇒ **add** | **sub** | **mul** | **div** | **rem**
⇒ **and** | **or** | **xor**
⇒ **cmp** | **cmpg** | **cmpl**
⇒ **eqref** | **eq** | **lt** | **ge** | **gt** | **le**
⇒ **shr** | **shl** | **ushr**
⇒ **instanceof** | **checkcast**
⇒ **checkbound** | **arrayref**

TriOpInstr ⇒ **arrayset**

FieldRef ⇒ **fielref** *className* ∈ *String* . *fieldName* ∈ *String* [*this* ∈ *VarNode*]

FieldSet ⇒ **fieldset** *className* ∈ *String* . *fieldName* ∈ *String*
[*this* ∈ *VarNode*] *v* ∈ *VarNode*

MethodRef ⇒ **methodref** *className* ∈ *String* . *methodName* ∈ *String*
[*this* ∈ *VarNode*]
({*param* ∈ *R-expr*}*)

A.2 Système de type du langage JST

$$
\begin{array}{lcl}
\mathit{Type} & = & \mathit{PrimType} \cup \mathit{RefType} \cup \mathit{VoidType} \\
\mathit{PrimType} & = & \mathit{FixNumericType} \cup \mathit{FloNumericType} \\
\mathit{FixNumericType} & = & \mathit{LogicType} \cup \mathit{ByteType} \cup \mathit{CharType} \\
 & & \cup\ \mathit{ShortType} \cup \mathit{IntType} \cup \mathit{LongType} \\
\mathit{FloNumericType} & = & \mathit{FloatType} \cup \mathit{DoubleType} \\
\mathit{RefType} & = & \mathit{ArrayType} \cup \mathit{ClassRefType} \cup \mathit{NullType} \cup \mathit{AddrType}
\end{array}
$$

Annexe B

Règles d'importation du *bytecode* vers JST

Cette annexe décrit les règles d'importation des instructions du *bytecode* et le code intermédiaire en résultant. La liste des règles d'importation n'est pas exhaustive, mais les règles manquantes sont similaires à celles présentées.

La notation employée est décrite par la formule suivante :

$\mathcal{C}[\![\mathrm{I}]\!]_{\kappa}\mathrm{S}{\rightarrow}\mathrm{S'} \Rightarrow \mathrm{C}$

où I est l'instruction importée
κ : Continuation (facultative)
S : la pile d'exécution avant l'interprétation
S' : la pile d'exécution après l'interprétation
C : le code JST généré

Les fonctions et variables employées sont :

ω_n	Paramètre statique n
υ_n	Paramètre dynamique n
ρ_n	Variable locale n
$\mathcal{T}$	Type statique de l'instruction
$\mathcal{T}_n$	Type statique du paramètre n
σn	Fonction de traitement de la table des constantes
$\uparrow$t	z2i t = logic b2i t = byte s2i t = short c2i t = char sinon
$\downarrow$t	i2z t = logic i2b t = byte i2s t = short i2c t = char sinon
$SMASK$(t)	0x1f t = int 0x3f t = long
$\mathcal{J}$	Ensemble d'instruction JSR de la méthode.

Voici la liste des règles d'importations :

$\mathcal{C}[\![\text{POP}]\!]_{\upsilon_1,S\rightarrow S}$	$\Rightarrow$	<vide>
$\mathcal{C}[\![\text{DUP}]\!]_{\upsilon_1,S\rightarrow \upsilon_1,\upsilon_1,S}$	$\Rightarrow$	<vide>
$\mathcal{C}[\![\text{SWAP}]\!]_{\upsilon_1,\upsilon_2,S\rightarrow \upsilon_2,\upsilon_1,S}$	$\Rightarrow$	<vide>
$\mathcal{C}[\![\text{BREAKPOINT}]\!]_{S\rightarrow S}$	$\Rightarrow$	<vide>
$\mathcal{C}[\![\text{NOP}]\!]_{S\rightarrow S}$	$\Rightarrow$	<vide>
$\mathcal{C}[\![\text{ACONST\_NULL}]\!]_{S\rightarrow \text{nil},S}$	$\Rightarrow$	<vide>
$\mathcal{C}[\![\text{ICONST\_0}]\!]_{S\rightarrow 0,S}$	$\Rightarrow$	<vide>
$\mathcal{C}[\![\text{BIPUSH } \omega_1]\!]_{S\rightarrow \upsilon_1,S}$	$\Rightarrow$	let υ_1 $\uparrow\mathcal{T}$ ω_1
$\mathcal{C}[\![\text{BLOAD } \omega_1]\!]_{S\rightarrow \upsilon_2,S}$	$\Rightarrow$	let υ_1 ρ_{ω_1} let υ_2 $\uparrow\mathcal{T}$ υ_1

$\mathcal{C}[\![\text{BSTORE } \omega_1]\!]_{v_1,\text{S}\to\text{S}}$ $\Rightarrow$ let $v_2 \downarrow\mathcal{T}\ v_1$
let $\rho_{\omega_1}\ v_2$

$\mathcal{C}[\![\text{IADD}]\!]_{v_2,v_1,\text{S}\to\ v_3,\text{S}}$ $\Rightarrow$ let v_3 add $v_1\ v_2$

$\mathcal{C}[\![\text{SHL}]\!]_{v_2,v_1,\text{S}\to\ v_4,\text{S}}$ $\Rightarrow$ let v_3 and $SMASK(\mathcal{T})\ v_1$
let v_4 shl $v_2\ v_3$

$\mathcal{C}[\![\text{IINC } \omega_1\ \omega_2]\!]_{\text{S}\to\text{S}}$ $\Rightarrow$ $\mathcal{C}[\![\text{ILOAD } \omega_1\ ]\!]_{\text{S}\to v_1,\text{S}}$
let v_2 add $v_1\ \omega_2$
$\mathcal{C}[\![\text{ISTORE } \omega_1\ ]\!]_{v_2,\text{S}\to\text{S}}$

$\mathcal{C}[\![\text{GOTO } \omega_1]\!]_{\text{S}\to\text{S}}$ $\Rightarrow$ jump $\langle\lambda_{pc+\omega_1}\rangle$ (S)

$\mathcal{C}[\![\text{IFEQ } \omega_1]\!]_{v_1,\text{S}\to\text{S}}$ $\Rightarrow$ if eq v_1 0
$\mathcal{C}[\![\text{GOTO } \omega_1]\!]_{\text{S}\to\text{S}}$
$\mathcal{C}[\![\text{GOTO } 1]\!]_{\text{S}\to\text{S}}$

$\mathcal{C}[\![\text{IF\_CMPLT } \omega_1]\!]_{v_2,v_1,\text{S}\to\text{S}}$ $\Rightarrow$ if lt $v_1\ v_2$
$\mathcal{C}[\![\text{GOTO } \omega_1]\!]_{\text{S}\to\text{S}}$
$\mathcal{C}[\![\text{GOTO } 1]\!]_{\text{S}\to\text{S}}$

$\mathcal{C}[\![\text{TABLESWITCH } \omega_\text{d}\ \omega_\text{l}\ \omega_\text{h}\ \omega_0 \ldots \omega_{\omega_\text{h}-\omega_\text{l}}\ ]\!]_{v_1,\text{S}\to\text{S}}$
$\Rightarrow$ if eq v_1 add ω_l 0
$\mathcal{C}[\![\text{GOTO } \omega_0]\!]_{\text{S}\to\text{S}}$
if eq v_1 add ω_l 1
$\mathcal{C}[\![\text{GOTO } \omega_1]\!]_{\text{S}\to\text{S}}$
...
if eq v_1 add $\omega_{\omega_\text{h}-\omega_\text{l}}\ \omega_\text{h}-\omega_\text{l}$
$\mathcal{C}[\![\text{GOTO } \omega_{\omega_\text{h}-\omega_\text{l}}]\!]_{\text{S}\to\text{S}}$
$\mathcal{C}[\![\text{GOTO } \omega_\text{d}]\!]_{\text{S}\to\text{S}}$

$\mathcal{C}[\![\text{NEW } \omega_1]\!]_{\text{S}\to v_1,\text{S}}$ $\Rightarrow$ let v_1 new $\sigma\omega_1$

$\mathcal{C}[\![\text{GETFIELD } \omega_1]\!]_{v_1,\text{S}\to v_4,\text{S}}$ $\Rightarrow$ let v_2 checknull v_1
let v_3 fieldref $\sigma\omega_1\ v_2$
let $v_4 \uparrow\mathcal{T}\ v_3$

$\mathcal{C}[\![\text{GETSTATIC } \omega_1]\!]_{\text{S}\to v_2,\text{S}}$ $\Rightarrow$ let v_1 fieldref $\sigma\omega_1$
let $v_2 \uparrow\mathcal{T}\ v_1$

$\mathcal{C}[\![\text{PUTFIELD } \omega_1]\!]_{v_1,v_2,\text{S}\to\text{S}}$ $\Rightarrow$ let v_3 checknull v_2
let $v_4 \downarrow\mathcal{T}\ v_1$
fieldset $\sigma\omega_1\ v_3\ v_4$

$\mathcal{C}[\![\text{PUTSTATIC } \omega_1]\!]_{v_1,\text{S}\to\text{S}}$ $\Rightarrow$ let $v_2 \downarrow\mathcal{T}\ v_1$
fieldset $\sigma\omega_1\ v_2$

$\mathcal{C}[\![\text{INVOKEVIRTUAL } \omega_1]\!]_{v_{1\to n},S\to v_{3''},S}$
$\Rightarrow$ let v_1, checknull v_1
let v_2, $\downarrow\mathcal{T}_2\ v_2$
let $v_{2''}$ $\uparrow\mathcal{T}_2\ v_2$,
...
let v_n, $\downarrow\mathcal{T}_n\ v_n$
let $v_{n''}$ $\uparrow\mathcal{T}_n\ v_n$,
let v_3 methodref $\sigma\omega_1\ v_1\ (v_{2''\to n''})$
let v_3, $\downarrow\mathcal{T}\ v_3$
let $v_{3''}$ $\uparrow\mathcal{T}\ v_3$,

$\mathcal{C}[\![\text{INVOKEINTERFACE } \omega_1]\!]_{v_{1\to n},S\to v_{3''},S}$
$\Rightarrow$ $\mathcal{C}[\![\text{INVOKEVIRTUAL } \omega_1]\!]_{v_{1\to n},S\to v_{3''},S}$

$\mathcal{C}[\![\text{INVOKESTATIC } \omega_1]\!]_{v_{1\to n},S\to v_{3''},S}$
$\Rightarrow$ let v_1, $\downarrow\mathcal{T}_1\ v_1$
let $v_{1''}$ $\uparrow\mathcal{T}_1\ v_1$,
...
let v_n, $\downarrow\mathcal{T}_n\ v_n$
let $v_{n''}$ $\uparrow\mathcal{T}_n\ v_n$,
let v_3 methodref $\sigma\omega_1\ (v_{1''\to n''})$
let v_3, $\downarrow\mathcal{T}\ v_3$
let $v_{3''}$ $\uparrow\mathcal{T}\ v_3$,

$\mathcal{C}[\![\text{INVOKESPECIAL } \omega_1]\!]_{v_{1\to n},S\to v_{3''},S}$
$\Rightarrow$ $\mathcal{C}[\![\text{INVOKESTATIC } \omega_1]\!]_{v_{1\to n},S\to v_{3''},S}$

$\mathcal{C}[\![\text{IRETURN}]\!]_{v_1,S\to\ldots}$ $\Rightarrow$ return v_1

$\mathcal{C}[\![\text{BALOAD}]\!]_{v_1,v_2,S\to v_7,S}$
$\Rightarrow$ let v_3 checknull v_2
$\mathcal{C}[\![\text{GETFIELD } \sigma^{-1}\text{java/lang/<array><length>}]\!]_{v_3,S\to v_4,S}$
checkbound $v_4\ v_1$
$\mathcal{C}[\![\text{GETFIELD } \sigma^{-1}\text{java/lang/<array><elements>}]\!]_{v_3,S\to v_5,S}$
let v_6 arrayref $v_5\ v_1$
let v_7 $\uparrow\mathcal{T}\ v_6$

$\mathcal{C}[\![\text{BASTORE}]\!]_{v_1,v_2,v_3,S\to S}$
$\Rightarrow$ let v_4 checknull v_3
$\mathcal{C}[\![\text{GETFIELD } \sigma^{-1}\text{java/lang/<array><length>}]\!]_{v_4,S\to v_5,S}$
checkbound $v_5\ v_2$
$\mathcal{C}[\![\text{GETFIELD } \sigma^{-1}\text{java/lang/<array><elements>}]\!]_{v_4,S\to v_6,S}$
let v_7 $\downarrow\mathcal{T}\ v_1$
arrayset $v_6\ v_2\ v_7$

$\mathcal{C}[\![$AASTORE$]\!]_{v_1,v_2,v_3,S\rightarrow S}$
$\Rightarrow$ let v_4 checknull v_3
$\mathcal{C}[\![$GETFIELD σ^{-1}java/lang/<array><length>$]\!]_{v_4,S\rightarrow v_5,S}$
checkbound v_5 v_2
$\mathcal{C}[\![$GETFIELD σ^{-1}java/lang/<array><type>$]\!]_{v_4,S\rightarrow v_6,S}$
checkcast v_1 v_6
$\mathcal{C}[\![$GETFIELD σ^{-1}java/lang/<array><elements>$]\!]_{v_4,S\rightarrow v_7,S}$
let $v_8 \downarrow_{\mathcal{T}} v_1$
arrayset v_7 v_2 v_8

$\mathcal{C}[\![$ARRAYLENGTH$]\!]_{v_1,S\rightarrow v_3,S}$
$\Rightarrow$ let v_2 checknull v_1
$\mathcal{C}[\![$GETFIELD σ^{-1}java/lang/<array><length>$]\!]_{v_2,S\rightarrow v_3,S}$

$\mathcal{C}[\![$NEWARRAY $\omega_1]\!]_{\langle\kappa\rangle} v_1,S\rightarrow v_3,S$
$\Rightarrow$ if ge v_1 0
{
let v_2 newarray $\sigma\omega_1$ v_1
$\mathcal{C}[\![$NEW σ^{-1}java/lang/<array>$]\!]_{S\rightarrow v_3,S}$
$\mathcal{C}[\![$PUTFIELD σ^{-1}java/lang/<array><type>$]\!]_{\omega_1,v_3,S\rightarrow S}$
$\mathcal{C}[\![$PUTFIELD σ^{-1}java/lang/<array><length>$]\!]_{v_1,v_3,S\rightarrow S}$
$\mathcal{C}[\![$PUTFIELD σ^{-1}java/lang/<array><elements>$]\!]_{v_2,v_3,S\rightarrow S}$
jump $\kappa(\ v_3,\ S\ )$
}
{
$\mathcal{C}[\![$NEW σ^{-1}java/lang/NegativeArraySizeException$]\!]_{S\rightarrow v_4,S}$
$\mathcal{C}[\![$INVOKESPECIAL σ^{-1}java/lang/NegativeArraySizeException <init> ()V$]\!]_{S\rightarrow S}$
$\mathcal{C}[\![$THROW$]\!]_{v_4,S\rightarrow\ldots}$
}

$\mathcal{C}[\![\text{MULTINEWARRAY } \omega_1\ \omega_2]\!]_{\langle\kappa\rangle} v_{1\to n},S\to v_{a1},S$

si $\omega_2 = 1$ $\left\{\right.$ $\mathcal{C}[\![\text{ NEWARRAY } \omega_1]\!]_{\langle\kappa\rangle} v_1,S\to v_{a1},S$

$\Rightarrow$

si $\omega_2 > 1$ $\left\{\right.$

$\mathcal{C}[\![\text{NEWARRAY } \sigma^{-1}\text{java/lang/<array>}]\!]_{\langle\lambda_1\rangle} v_1,v_{1\to n},S \to S$

$\lambda_1(\ v_{a1},v_{c1},v_{2\to n},S\ )\ \{$
$\mathcal{C}[\![\text{MULTINEWARRAY } \omega_1\ \omega_2{-}1]\!]_{\langle\lambda 2\rangle}\ v_{2\to n},v_{2\to n},v_{c1},v_{a1},S \to v_{a2},v_{2\to n},v_{c1},v_{a1},S$
$\}$

$\lambda_2(\ v_{a2},v_{2\to n},v_{c1},v_{a1},S\ )\ \{$
let v_{c1}' sub v_{c1} 1
$\mathcal{C}[\![\text{AASTORE}]\!]_{v_{a2},v_{c1}',v_{a1},S\to S}$
if gt v_{c1}' 0
jump $\lambda_1(\ v_{a1},v_{c1},v_{2\to n},S\ )$
jump $\kappa(\ v_{a1},S\ )$
$\}$

$\mathcal{C}[\![\text{JSR } \omega_1]\!]_{S\to S}$ $\Rightarrow$ let v_1 *pc*
$\mathcal{C}[\![\text{GOTO } \omega_1]\!]_{v_1,S\to v_1,S}$

$\mathcal{C}[\![\text{RET}]\!]_{v_1,S\to S}$ $\Rightarrow$ if eq v_1 $\mathcal{J}[1]$
jump $\langle\lambda_{\mathcal{J}[1]+1}\rangle$ (S)
if eq v_2 $\mathcal{J}[2]$
jump $\langle\lambda_{\mathcal{J}[2]+1}\rangle$ (S)
...
jump $\langle\lambda_{\mathcal{J}[n]+1}\rangle$ (S)

Annexe C

Analyse de type pour domaines numériques entiers

Cette annexe décrit les règles d'analyse de type pour domaines numériques entiers que le compilateur emploie.

```
package jst;

public class FixRange {

    public long range_min, range_max;
    public long min, max;
    public boolean bignum = false;
    public boolean initialized = false;

    public static long FIXMAX = IntType.MAX;
    public static long FIXMIN = IntType.MIN;

    public FixRange( long range_min, long range_max ) {
        initialized = false;
        this.range_min = range_min;
        this.range_max = range_max;
    }

    public boolean neg( FixRange p1, long gpatch, long lpatch ) {
        boolean res = false;
        if( !p1.initialized ) return false;
        res |= set( -p1.max );
        res |= set( -p1.min );
        return res;
    }

    public boolean add( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
        boolean res = false;
        if( !p1.initialized ) return false;
        if( !p2.initialized ) return false;
        res |= set( p1.max + p2.max );
        res |= set( p1.min + p2.min );
        if( res ) patch( gpatch, lpatch );
        return res;
    }

    public boolean sub( FixRange p1, FixRange p2, long gpatch, long lpatch  ) {
        boolean res = false;
        if( !p1.initialized ) return false;
        if( !p2.initialized ) return false;
        res |= set( p1 );
        res |= set( p2 );
        res |= set( p1.max - p2.min );
        res |= set( p1.min - p2.max );
        if( res ) patch( gpatch, lpatch );
        return res;
    }

    public boolean mul( FixRange p1, FixRange p2, long gpatch, long lpatch  ) {
```

```
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    res |= set( p1.max * p2.max );
    res |= set( p1.max * p2.min );
    res |= set( p1.min * p2.max );
    res |= set( p1.min * p2.min );
    if( res ) patch( gpatch, lpatch );
    return res;
}

public boolean div( FixRange p1, FixRange p2, long gpatch, long lpatch  ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    res |= set( p1.max );
    res |= set( p1.min );
    res |= set( -p1.max );
    res |= set( -p1.min );
    if( res ) patch( gpatch, lpatch );
    return res;
}


public boolean rem( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    res |= set( p2.max );
    res |= set( p2.min );
    res |= set( -p2.max );
    res |= set( -p2.min );
    if( res ) patch( gpatch, lpatch );
    return res;
}

public boolean not( FixRange p1, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    res |= set( - (p1.max +1) );
    res |= set( - (p1.min +1) );
    return res;
}

public boolean and( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    if( p1.min >= 0 && p2.min >= 0) {
        res |= set( ((p1.max<p2.max)?p1.max:p2.max) );
        res |= set( 0 );
    }
    else if( p1.min >= 0 ) {
        res |= set( p1.max );
        res |= set( 0 );
    }
    else if( p2.min >= 0 ) {
        res |= set( p2.max );
        res |= set( 0 );
    }
    else return setBignum();
    return res;
}

public boolean or( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    if( p1.min >= 0 && p2.min >= 0 ) {
        res |= set( maskAllBits( p1.max ) );
        res |= set( maskAllBits( p2.max ) );
    }
    else res |= setBignum();
    return res;
}

public boolean xor( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    if( p1.min >= 0 && p2.min >= 0 ) {
        res |= set( maskAllBits( p1.max ) );
        res |= set( maskAllBits( p2.max ) );
    }
    else res |= setBignum();
    return res;
}

public boolean shl( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;

    if( p2.max >= 32 || p2.min <= 0 )
      setBignum();
    else {
      if( p1.min >= 0 )
        res |= set( p1.max << p2.max );
```

```
      else
        setBignum();
      if( res ) patch( gpatch, lpatch );
    }
    return res;
}

public boolean shr( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    if( p2.max >= 32 || p2.min <= 0 )
      setBignum();
    else {
      res |= set( p1.max >> p2.min );
      res |= set( p1.min >> p2.min );
      if( res ) patch( gpatch, lpatch );
    }
    return res;
}

public boolean ushr( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    if( p2.max >= 32 || p2.min <= 0 )
      setBignum();
    else {
      if( p1.min >= 0 )
        return set( p1 );
    }
    return setBignum();
}

public boolean cmp( FixRange p1, FixRange p2, long gpatch, long lpatch ) {
    boolean res = false;
    if( !p1.initialized ) return false;
    if( !p2.initialized ) return false;
    if( p1.max >= p2.min ) {
        res |= set( 1 );
        res |= set( 0 );
    }
    if( p1.min <= p2.max ) {
        res |= set( -1 );
        res |= set( 0 );
    }
    return res;
}

public void patch( long gpatch, long lpatch ) {
    long mmax = max;
    long mmin = min;
    if( mmax != 0 ) {
        set( mmax + gpatch );
        set( mmax + lpatch );
    }
    if( mmin != 0 ) {
        set( mmin - gpatch );
        set( mmin - lpatch );
    }
}

public boolean setBignum() {
    boolean res = false;
    if( bignum ) return false;

    if( isBignum( range_min ) || isBignum( range_max ) ) {
        initialized = true;
        min = range_min;
        max = range_max;
        bignum = true;
        return true;
    }
    else {
        res |= (min != range_min);
        res |= (max != range_max);

        min = range_min;
        max = range_max;
        initialized = true;
        return res;
    }
}

public boolean isBignum( long n ) {
    return ((n < FIXMIN) || (n > FIXMAX));
}


public boolean set( FixRange f ) {
    if( !f.initialized ) return false;
    if( bignum ) return false;
    boolean res = false;
    res |= set( f.max );
    res |= set( f.min );
    return res;
}

public boolean set( long m ) {
```

```
    boolean res = false;
    if( bignum ) return false;

    if( m < range_min ||
        m > range_max )
        return setBignum();

    if( isBignum( m ) )
        return setBignum();

    if( m < min || !initialized ) {
        min = m;
        res = true;
    }

    if( m > max || !initialized ) {
        max = m;
        res = true;
    }

    initialized = true;

    return res;
}


public boolean intersectEquals( FixRange r ) {
    boolean res = false;

    if( r.min > min ) {
        min = r.min;
        res = true;
    }

    if( r.max < max ) {
        max = r.max;
        res = true;
    }

    set( min );
    set( max );

    return res;
}

public boolean intersectLowerThan( FixRange r ) {
    boolean res = false;

    if( max > r.max) {
        max = r.max;
        res = true;
    }

    set( min );
    set( max );

    return res;
}

public boolean intersectHigherThan( FixRange r ) {
    boolean res = false;

    if( min < r.min) {
        min = r.min;
        res = true;
    }

    set( min );
    set( max );
    return res;

}



public void reset( boolean v ) {
    if( v ) {
        setBignum();
    }
    else {
        initialized = false;
        min = 0;
        max = 0;
        bignum = false;
    }
}

public boolean canBeLowerThan( FixRange r ) {
    return ( !initialized || bignum || r.bignum ||  min <= r.max );
}

public boolean canBeBiggerThan( FixRange r ) {
    return ( !initialized || bignum || r.bignum ||  max >= r.min );
}

public boolean canBeEquals( FixRange r ) {
    return ( !initialized || bignum || r.bignum ||
             (max >= r.min && min <= r.max) );
```

```
    }


    public boolean cantBeEquals( FixRange r ) {
        return ( !initialized || !bignum || !r.bignum ||
                 min >= r.max || max <= r.min );
    }

    public long getConstant() {
        return min;
    }

    public boolean isConstant() {
        return( !bignum && (min == max) );
    }

    public boolean isConstant( int x ) {
        return( !bignum && (min == max) && min == x );
    }

    public String toString() {
        return "[FixRange " + min + " " + max + " " +
                ((bignum)?"%t":"%f") + "]";
    }

    long maskAllBits( long n ) {
        long res = n;
        while( n != 0 ) {
            res = res | n;
            n>>=1;
        }
        return res;
    }

    public boolean canBe( long n ) {
        if( bignum ) return true;
        return( min <= n && max >= n );
    }

    public boolean isInRange( long nmin, long nmax ) {
        if( bignum ) return false;
        return( min >= nmin && max <= nmax );
    }
}
```