UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

# Transformation statique de binaires exécutables

Mémoire présenté en vue de l'obtention
du grade de Licencié en Informatique

Aris ADAMANTIADIS
Année académique 2008–2009

## Remerciements

Écrire un mémoire est un travail de longue haleine, et je n'aurais probablement pas réussi sans aide :

Je voudrais remercier Messieurs R. Devillers et Y. Roggeman d'avoir accepté de participer au jury de mémoire.

Je voudrais aussi remercier ma compagne Valérie, mes parents, mes amis ainsi que mes collègues de BELNET qui m'ont soutenu tout le long de ce périple. Enfin, je remercie tous les amis et collègues qui ont relu et commenté mon travail.

# Table des matières

**1 Introduction** **1**
1.1 Le reverse engineering . . . 1
1.2 Transformation de binaire . . . 3
1.3 État de l'art . . . 4
1.3.1 Implémentations existantes . . . 4
1.4 But du mémoire . . . 6
1.4.1 Front-end . . . 6
1.4.2 Image intermédiaire . . . 6
1.4.3 Back-end . . . 6
1.4.4 Documentation . . . 7

**2 L'architecture Sparc** **8**
2.1 Description . . . 8
2.1.1 Registres . . . 9
2.1.2 Procédures . . . 10
2.2 Instructions . . . 12
2.2.1 Syntaxe . . . 14
2.2.2 Types de données . . . 14
2.2.3 Sémantique . . . 14
2.3 Graphe de contrôle de flot . . . 18
2.3.1 Motivations . . . 18
2.3.2 Pipeline . . . 19
2.3.3 Enchaînement des instructions . . . 20
2.3.4 Graphe de flot naïf . . . 21
2.3.5 Problème du delay slot . . . 22

**3 Représentations intermédiaires** **25**
3.1 Rôle . . . 25
3.2 Types de formes intermédiaires . . . 25
3.2.1 Code à trois adresses . . . 26
3.2.2 Machine à pile . . . 26
3.2.3 Choix de la forme intermédiaire . . . 26
3.3 Formes intermédiaires particulières . . . 27
3.3.1 RTL . . . 27
3.3.2 Gnu RTL . . . 27
3.3.3 LLVM . . . 28

3.3.4 SSA . . . . . . . . . . 28
3.4 Une forme intermédiaire . . . . . . . . . . 28
3.4.1 Définition . . . . . . . . . . 29
3.4.2 Règles de transformation sémantique . . . . . . . . . . 30

**4 Analyse sémantique 40**
4.1 Basic Blocks . . . . . . . . . . 40
4.1.1 Définition . . . . . . . . . . 40
4.1.2 Algorithme . . . . . . . . . . 41
4.1.3 Liens entre Basic Blocks . . . . . . . . . . 42
4.2 Analyse de flots de données . . . . . . . . . . 42
4.2.1 Définitions . . . . . . . . . . 42
4.2.2 Dépendance entre instructions . . . . . . . . . . 43
4.2.3 Vie d'une variable . . . . . . . . . . 45
4.3 Autres informations sémantiques . . . . . . . . . . 50
4.3.1 Analyse de type et pointeurs . . . . . . . . . . 50
4.3.2 Optimisation des pointeurs . . . . . . . . . . 51
4.3.3 Analyse inter-procédures . . . . . . . . . . 54
4.4 Détection des structures particulières . . . . . . . . . . 55
4.4.1 Analyse des sauts indexés . . . . . . . . . . 56
4.4.2 Pointeurs de procédures et méthodes virtuelles . . . . . . 57

**5 Optimisation 59**
5.1 Nécessité de l'optimisation . . . . . . . . . . 59
5.2 Critères d'optimisation . . . . . . . . . . 60
5.3 Optimisation des variables . . . . . . . . . . 61
5.3.1 Propagation des constantes . . . . . . . . . . 61
5.3.2 Propagation des copies . . . . . . . . . . 61
5.4 Optimisation des instructions . . . . . . . . . . 64
5.4.1 Suppression de code mort . . . . . . . . . . 64
5.4.2 Évaluation statique . . . . . . . . . . 64
5.4.3 Factorisation d'expressions . . . . . . . . . . 65
5.4.4 Optimisation des boucles . . . . . . . . . . 65
5.5 Remarques . . . . . . . . . . 69
5.5.1 Exhaustivité . . . . . . . . . . 69
5.5.2 Ordre de passage . . . . . . . . . . 69

**6 Génération de code 71**
6.1 Rôle du Back-end . . . . . . . . . . 71
6.2 Architecture Java et JVM . . . . . . . . . . 71
6.2.1 Architecture de la JVM . . . . . . . . . . 72
6.2.2 Le langage de programmation Jasmin . . . . . . . . . . 74
6.2.3 Vérificateur . . . . . . . . . . 77
6.3 Traduction . . . . . . . . . . 77
6.3.1 Conventions . . . . . . . . . . 79
6.3.2 Traduction dirigée par la syntaxe . . . . . . . . . . 80
6.3.3 Traduction de la forme intermédiaire . . . . . . . . . . 80

6.3.4 Optimisations . . . . . 81
6.4 Librairie runtime . . . . . 84
6.4.1 Accès à la mémoire . . . . . 84
6.4.2 Pointeurs de fonction . . . . . 84
6.4.3 Interception des erreurs . . . . . 85
6.4.4 Émulation de l'environnement . . . . . 85

**7 Prototype** **87**
7.1 Architecture . . . . . 87
7.2 Utilisation . . . . . 88
7.2.1 Compilation . . . . . 88
7.2.2 Utilisation . . . . . 89
7.3 Capacités et limitations . . . . . 90

**8 Conclusion** **91**
8.1 Travail écrit . . . . . 91
8.2 Prototype . . . . . 92
8.3 Travaux futurs . . . . . 92

**Bibliographie** **94**

# Chapitre 1

# Introduction

## 1.1 Le reverse engineering

Le reverse engineering, ou ingénierie inverse, est un ensemble de techniques s'appliquant à des programmes informatiques sous leur forme compilée, ou sous une forme incomplète. Souvent, les programmes s'achètent ou se distribuent sous forme exécutable, c'est-à-dire une forme adaptée à l'exécution sur le microprocesseur d'un ordinateur.
Or, cette forme n'est pas adaptée à la lecture par les humains. Les humains comprennent plus facilement des codes sources, écrits dans un ou plusieurs langages de programmation. Les techniques d'ingénierie inverse consistent donc à récupérer de l'information sur le fonctionnement d'un programme sans en posséder le code source (fig. 1.1), ou à obtenir des informations sur le design d'une application sans posséder la totalité de la documentation.

Les raisons qui poussent les experts à pratiquer l'ingénierie inverse sont très variées et font appel à des outils et des méthodes très différentes selon les cas (sécurité, compréhension de code, interopérabilité, ...). On peut citer, de manière non exhaustive :

**Désassembleur** Programme qui fournit un programme en assembleur depuis un exécutable. Certains peuvent être interactifs (par exemple *IDA*[3]) et permettre à l'analyste d'ajouter des commentaires, nommer des entités ou modifier du code.

**Débugger** Programme qui analyse l'exécution d'un programme, généralement de manière interactive, pour pouvoir observer sa structure interne lors du fonctionnement (runtime). Il est donc nécessaire d'exécuter le programme pour l'analyser. En général, un debugger est toujours fourni avec un désassembleur sommaire.

**Profiler** Programme qui trace l'exécution à la manière d'un debugger, mais renvoie des informations sur les endroits forts visités dans le code. Il permet de trouver les points chauds (bottlenecks) et les morceaux de code à analyser.

**Traceur** Programme qui analyse les appels systèmes ou librairies effectués par le programme à analyser. Cela fournit des informations très utiles sur

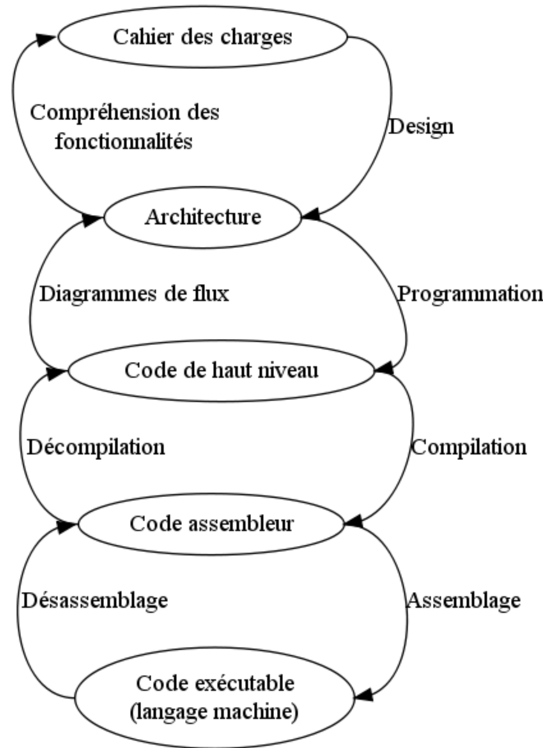


FIG. 1.1 – L'ingénierie et l'ingénierie inverse sont deux processus opposés

les entrées et les sorties du programme analysé. *strace* [10] et *truss* (Sun Solaris) listent les appels systèmes effectués par un programme pendant son exécution. *ltrace* [7] liste les appels aux fonctions des librairies.

**Décompilateur** Désassembleur très évolué : il transforme le code binaire du programme analysé en un langage de haut niveau (C, C++ ou autre). Ce genre d'outil est généralement statique (pas d'exécution nécessaire) mais certains peuvent s'aider de données de profiling, ou être interactifs (par exemple *Hex-Rays Decompiler* [4] intégré à *IDA*). L'intérêt de l'interactivité pour un tel outil est de permettre à l'analyste l'utilisant d'y laisser des notes, de nommer des variables ou même d'aider l'outil à trouver la traduction la plus adaptée.

**Vérificateur** Ce programme vérifie soit de manière statique, soit à l'exécution, qu'un programme ne comporte par certaines classes de bugs connus (overflows, double free, ...). Par exemple, *Valgrind* [11] (analyse runtime) et Chevarista [19] (analyse statique). Certains outils travaillent soit sur du code de haut niveau, soit sur du code compilé. Analyser du code compilé peut apporter l'avantage de découvrir des bugs dans le compilateur.

**Sniffer** Traceur réseau permettant d'enregistrer et d'analyser le trafic réseau. Cet outil est indispensable pour faire l'ingénierie inverse d'un protocole réseau.

**Émulateur** Programme simulant le processeur et l'environnement d'une machine afin d'y faire tourner des programmes conçus pour celle-ci. L'ému-

lateur et le simulateur interprètent ou compilent à la volée le code du programme.

**Transformateur** Programme qui transforme un binaire exécutable prévu pour une machine en un binaire exécutable sur une autre machine cible.

En règle générale, toutes les utilisations de ces techniques ne servent pas au même objectif. Par exemple, l'utilisateur voulant simuler le fonctionnement d'un programme n'a pas nécessairement besoin de comprendre comment le programme fonctionne, pourvu qu'il ait un simulateur. L'expert en sécurité voulant comprendre le fonctionnement d'un virus l'analysera sous toutes ses formes mais se gardera de l'exécuter sur son ordinateur de bureau !

Dans ce travail, nous nous focaliserons sur des techniques statiques de transformation de binaire. L'avantage de la transformation de binaire sur la simulation sont l'efficacité et le gain en performance. Aussi, la transformation de binaire permet d'appliquer des optimisations sur le code, optimisations qui pourraient avoir été omises lors de la compilation.

## 1.2 Transformation de binaire

La transformation de binaire et la simulation d'ordinateurs sont très courantes de nos jours. Certains langages de programmation (Java, Python, Perl, C#, ...) se basent même exclusivement sur la transformation et la simulation pour garantir l'interopérabilité.
Selon l'utilisation, certaines techniques se révèlent plus efficaces que d'autres.

**Simulation** La machine est entièrement simulée. Un programme simule le matériel de la machine, ainsi que le processeur, et interprète les instructions qui lui sont envoyées. Comme la machine entière est simulée, le système d'exploitation est aussi simulé dans cette machine virtuelle. Les défauts de cette technique sont sa lenteur (interprétation des instructions) et la lourdeur (nécessité de simuler un disque dur, une carte réseau, une carte graphique,...).
Cette méthode est très utilisée pour concevoir des systèmes embarqués, pour lesquels le matériel doit lui-même souvent être simulé.

**Émulation** L'émulation est un cas particulier de la simulation, dans lequel seule une partie de la machine est simulée. Typiquement, le processeur et le système d'exploitation sont simulés, à travers une couche spécifique pour les appels systèmes.

**Transformation statique** Le programme est recompilé une seule fois (approche statique) pour une plateforme cible. Le système d'exploitation est simulé au travers d'une couche d'abstraction.

**Simulation avec JIT** La simulation de la machine se dote d'un programme spécial, le *Just In Time compiler*, qui compile des morceaux de codes à la volée pendant l'exécution du programme cible. Les parties exécutées en boucle sont donc exécutées nativement et accélérées. Dans ce cas, on parle

de *Transformation dynamique*. Cette technique est un compromis entre la simulation par interprétation, ayant comme défaut le manque d'efficacité, et la transformation statique, qui manque de souplesse et impose une étape supplémentaire.

La transformation statique n'est donc pas l'unique méthode pour exécuter un logiciel "étranger". La transformation statique se comporte très bien lorsque le temps pris par l'étape de transformation n'est pas pénalisant : le transformateur statique peut appliquer des optimisations beaucoup plus poussées qu'un transformateur JIT, parce que l'analyse peut être plus profonde bien que plus longue. Un compilateur JIT peut partir sur certaines hypothèses sur la qualité du code à transformer et la présence d'optimisations préalables, pour gagner du temps à la transformation.

La transformation statique convient moins bien à l'implémentation d'un langage à machine virtuelle et à bytecode tel que Java : le langage lui-même est dynamique et permet de charger des modules en bytecode. Avec un transformateur statique, tous les modules devraient être transformés avant d'être utilisés, ce qui est peu commode.

Les programmes automodifiants tels que certains virus et empaqueteurs de logiciels sont par nature impossibles à transformer statiquement. Pour transformer statiquement un programme qui se modifie lui-même, il faudrait que le transformateur puisse accéder à toutes les formes possibles du programme (les modifications au cours du temps). Cela nécessiterait au transformateur de simuler tous les états possibles du programme, ce qui, dans un cas suffisamment complexe, est irréalisable. De tels programmes doivent être interprétés lorsque le code devient automodifiant, et pour de tels cas les interpréteurs JIT conviennent mieux.

La transformation statique ne convient pas du tout pour la simulation d'une machine complète : un système d'exploitation complet n'est rien d'autre qu'un programme automodifiant, qui charge et décharge des morceaux de programmes dans la mémoire, et les exécute.

## 1.3 État de l'art

### 1.3.1 Implémentations existantes

Un grand nombre d'outils impliquant la décompilation totale ou partielle de binaires ont été conçus depuis 1960. En effet, la transformation de binaires et la décompilation sont intimement liées. Cette partie est inspirée du *Decompilation Wiki* [1] et de la thèse de Cristina Cifuentes [13], où l'on peut retrouver un historique très complet.
Parmi les projets les plus remarquables, on peut citer :

**D-Neliac decompiler** D-Neliac decompiler a été développé par J.K.Donnelly et H.Englander au Navy Electronics Laboratory en 1960. Ce décompila-

teur transformait du binaire exécutable en un langage de type Algol.
Ce projet a une grande importance historique, car il a prouvé la possibilité d'implémenter un décompilateur utilisable en situation réelle.

**C.R. Hollander** En 1973, Hollander décrit dans sa thèse une nouvelle technique de décompilation, fondé sur un métalangage basé sur la syntaxe. Chaque étape de la décompilation passait d'une représentation à une autre par un ensemble de règles formelles et strictes (principalement du pattern-matching).
Le défaut de cette conception est d'être inefficace sur les patterns inconnus, notamment les compilateurs non supportés ou générant du code très optimisé.

**exe2c** exe2c a été développé par Austin Code Works en 1990. Ce décompilateur cible la plateforme 286 sous MS-DOS. L'outil comporte 3 modules séparés : e2a, prenant un fichier .exe en entrée et produisant un programme assembleur, a2aparse, transformant un code assembleur en représentation intermédiaire, et e2c transformant cette représentation en C.
Le décompilateur applique une analyse de flux d'instructions, mais aucune analyse du flux de données. Le code C produit est très rudimentaire, utilisant des variables globales pour gérer l'état des registres. Le code produit pourrait être qualifié d'assembleur de haut niveau.

**DCC** DCC est un programme développé par Cristina Cifuentes (Université de Queensland) de 1990 à 1994 dans le cadre de sa thèse de doctorat, "Reverse Compilation Techniques" [13]. DCC décompile des binaires 286 sous MS-DOS et produit des programmes en C.
DCC utilise des techniques d'analyse de flot de données, des transformations de graphes ainsi que de l'analyse de types, afin de produire le code C le plus clair et efficace possible, ce qui est un énorme progrès par rapport aux outils de l'époque.

**UQBT** UQBT (University of Queensland Binary Translation) est un framework de translation statique de binaires, développé par l'université de Queensland et Sun Microsystems de 1996 à 2001. L'équipe était composée de Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey et Brian Lewis. Le but du projet était d'expérimenter la translation automatique de binaires CISC vers la plateforme Sparc/Solaris, et des binaires Sparc vers des processeurs Sparc encore inexistants. UQBT a été conçu dès le départ pour gérer plusieurs architectures, tant en entrée qu'en sortie. En 2001, il accepte des fichiers Sparc32 et Pentium en entrée, et produit du code Sparc32, Pentium et Java.
UQBT se démarque des projets concurrents par le support de plusieurs "front-ends" et "back-ends", car jusqu'alors, la conception des décompilateurs et traducteurs était très dépendante des plateformes uniques choisies. UQBT a aussi introduit un langage de description sémantique pour les instructions du processeur. Nous y reviendrons plus tard dans le point 3.3.1.

**Boomerang** Boomerang [2] est un décompilateur libre développé depuis 2002 et basé sur UQBT. Il supporte des binaires IA32, Sparc et PowerPC en

entrée, et produit du code C. Il se démarque par l'utilisation d'une représentation interne basée sur SSA (Single Static Assignement), qui a commencé à être très utilisée dans les nouveaux compilateurs en raison de ses qualités. Nous étudierons brièvement la forme SSA dans le chapitre 3.

## 1.4 But du mémoire

L'objet du présent travail est l'étude théorique, la conception et la réalisation d'un prototype de transformateur statique de binaires.
Un transformateur de binaires est avant tout un compilateur spécial qui utilise un fichier exécutable comme fichier source, et génère du code exécutable natif pour une plateforme cible. Le transformateur est qualifié de statique, car il n'opère qu'une seule fois, avant l'exécution du programme traduit.

### 1.4.1 Front-end

Dans ce travail, nous avons choisi la plateforme source comme étant Sparc32/linux, c'est-à-dire les programmes en mode utilisateur compilés pour *sparcV8* (32 bits) sous l'environnement linux.
Les raisons de ce choix sont les suivantes :
- *sparcV8* est une architecture de moins en moins utilisée ;
- De nombreux programmes propriétaires ont été développés et n'ont plus de support de la part de l'éditeur. Une migration implique forcément de simuler le processeur ou de transformer le binaire ;
- *sparcV8* utilise un format d'instructions très simple à décoder et dont la description sémantique est réduite (philosophie Risc) ;
- *linux* est un environnement courant, facile à obtenir, et relativement semblable à Solaris (fourni historiquement sur les stations *Sun*).

### 1.4.2 Image intermédiaire

L'image intermédiaire que nous obtenons est un produit du front-end. Cette image intermédiaire convient à l'application d'algorithmes d'analyse de flots de données et d'optimisations, et sert d'interface au Back-end.

### 1.4.3 Back-end

Le choix de la plateforme cible est plus arbitraire. Le but du travail n'étant pas de simuler un environnement complet (travail fastidieux et répétitif) mais seulement un proof-of-concept pour réaliser les quelques tests, nous avons décidé de cibler la machine virtuelle *Java* et plus particulièrement le *bytecode Java*.
Ce choix comporte plusieurs avantages :
- La plateforme *Java* est bien définie [17], a plusieurs implantations, est très portable et assure aussi la portabilité du code généré ;
- La conception du prototype sera simplifiée, vu que le back-end sera unique dès la conception ;

– Le fait d'exécuter le programme final sur la JVM assure que certaines optimisations omises à la recompilation seront faites à l'exécution, grâce à la présence de compilateurs à la volée (JIT). De cette manière, le programme sera toujours exécuté efficacement sur toutes les plateformes gérant une machine Java JIT ;

### 1.4.4 Documentation

Dans ce travail écrit, nous documenterons les étapes de la conception et la réalisation du prototype, et leurs différences par rapport à un compilateur classique prenant du code source en entrée.
Le programme source, à l'origine en format binaire, passe à travers une série de phases qui appliqueront chacune une modification au programme. La première de ces phases est le front-end, dont le rôle est d'obtenir une représentation intermédiaire permettant d'appliquer la deuxième phase, l'optimisation, dont le rôle est de rendre le programme plus performant. Ensuite, la dernière phase est le back-end qui génère le bytecode java à partir de la représentation intermédiaire.
Nous documenterons aussi le code runtime nécessaire pour interfacer le bytecode généré avec les librairies java, en émulant l'environnement linux à l'intérieur de la JVM.
En outre, le travail s'introduira dans une démarche de documentation critique des solutions existantes, afin de fournir au lecteur une vue d'ensemble des différentes techniques utilisées actuellement.

Le Chapitre 2 expliquera la première partie du front-end, celle qui consiste à transformer un programme Sparc en un graphe de contrôle de flot utilisable par l'analyse sémantique. Le chapitre 3 décrira les différentes formes de représentations intermédiaires disponibles et leur pertinence par rapport au problème de la transformation de binaires, ainsi que la transformation d'un graphe de contrôle de flot spécialisé Sparc en une forme intermédiaire. Le chapitre 4 détaillera l'analyse sémantique, c'est-à-dire les analyses permettant de récupérer certaines informations perdues dans le programme lors de la compilation.
Le chapitre 5 décrira les différentes optimisations réalisables sur un programme afin de le rendre le plus efficient possible. Le chapitre 6 décrira la partie la plus importante du *back-end*, la génération de bytecode Java, ainsi que la description de la librairie runtime nécessaire au fonctionnement du programme transformé.
Le chapitre 7 présentera le prototype de transformateur réalisé, et le chapitre 8 conclura ce travail.

# Chapitre 2

# L'architecture Sparc

Nous allons dans ce chapitre introduire l'architecture Sparc. Le langage d'assemblage Sparc est en effet le langage source du transformateur, et il convient d'expliquer en détail son fonctionnement. Au terme de ce chapitre, nous obtiendrons les outils indispensables pour obtenir un CFG (Graphe de contrôle de flot), clef de voûte de l'analyse et la compilation de programmes.

## 2.1 Description

L'architecture Sparc, dans sa version 8, est une architecture de processeurs RISC 32 bits, conçus et produits par *Sun Microsystems* à partir de 1991. Ces processeurs et leurs successeurs ont équipé jusqu'à maintenant les ordinateurs produits par *Sun Microsystems*. Les spécifications logicielles, pour programmer ou fabriquer ce type de processeur, se trouvent dans "The SPARC Architecture Manual (Version 8)" [18].

L'architecture du processeur est basée sur le modèle RISC : le jeu d'instruction est réduit au minimum, et les instructions sont réduites à des tâches élémentaires (contrairement au modèle CISC). De ce fait, une des caractéristiques des programmes pour processeurs RISC est le nombre d'instructions nécessaires à la réalisation d'une opération. Les instructions RISC ont une sémantique par instruction très réduite, ce qui est très intéressant dans le cadre de notre travail.

Sparc base son fonctionnement sur des entiers en 32 bits, des flottants en 32, 64 et 128 bits, et des pointeurs en 32 bits. L'espace d'adressage est linéaire (absence de segmentation), c'est-à-dire que les programmes utilisateurs accèdent à la mémoire utilisateur via des adresses logiques, et ne semblent pas partager de mémoire avec les autres processus. En effet, Sparc est prévu pour fonctionner avec un système d'exploitation gérant la pagination. De plus, l'espace d'adressage des instructions étant le même que celui des données, le processeur Sparc suit le modèle des machines de Von Neumann.
Nous nous limiterons à l'étude de programmes en mode utilisateur, le mode noyau étant beaucoup plus compliqué, car il comporte de nombreux registres et instructions supplémentaires nécessaires au bon fonctionnement du système

d'exploitation.

### 2.1.1 Registres

Une implantation d'un processeur Sparc peut contenir de 40 à 520 registres généraux de 32 bits, nombre qui diffère en fonction du prix du processeur ainsi que de sa génération.
Ce nombre de registres correspond à 8 registres globaux (accessibles en permanence) ainsi que de 2 à 32 ensembles de 16 registres particuliers, nommés les registres de fenêtre. Bien que l'architecture admette un nombre variable de registres, seule une partie est accessible par le programme à chaque instant, comme nous le verrons dans la gestion des fenêtres.
Sparc possède deux types de registres : les registres généraux et les registres systèmes.

#### Registres généraux

Les registres généraux sont des registres assignables, permettant de faire des échanges avec la mémoire ou des calculs. Ils sont numérotés de 0 à 31, et adressés sous la forme d'un tableau par les noms r[0] ... r[31]. Tous ces registres peuvent être assignés ou utilisés dans une procédure. Ces registres ont des utilisations standards, et possèdent d'autres noms afin de souligner ces utilisations.
Les registres r[0] à r[7] sont les registres globaux, aussi nommés de g0 à g7. Ils sont accessibles tout au long du programme, et référencent toujours les mêmes registres physiques. r[0]=g0 est un registre dont le contenu reste indéfiniment à zéro, il est donc très pratique pour réinitialiser d'autres registres ou la mémoire.
Les registres r[8] à r[31] sont les registres de fenêtre, dont le but est d'optimiser les passages de paramètres entre procédures. Ces registres ne référencent pas en permanence les mêmes registres physiques : en fonction de la procédure en cours d'exécution, ils sont des références sur un sous-ensemble précis des registres de fenêtres, c'est-à-dire la partie visible des registres de fenêtre.
Ainsi, les registres r[8] à r[15] (également nommés o0 à o7) sont les registres d'output. Ces registres servent à stocker les paramètres sortant lors d'un appel de procédure.
Les registres r[16] à r[23] (également nommés l0 à l7) sont les registres locaux. Ils servent à stocker des variables locales à une procédure.
Les registres r[24] à r[31] (également nommés i0 à i7) sont les registres d'entrée. Ils contiennent les paramètres de la procédure.
Nous verrons plus loin que certains registres possèdent plusieurs alias, par exemple sp = o6 = r[14] et fp = i6 = r[30] et que certains registres ont une utilisation définie dans l'ABI, comme par exemple le registre contenant l'adresse de retour de procédure.
Un registre particulier, le registre Y, est un registre général de 32 bits accessible uniquement à travers les instructions de multiplication, de division entière et de lecture-écriture **rdy** et **wry**.
D'autres registres, définis par l'architecture, sont volontairement écartés pour simplifier l'implémentation du transformateur (par exemple les registres à vir-

gule flottante).

#### Registres systèmes

Parmi les registres systèmes importants, on remarquera *PC* et *nPC*, qui sont les deux compteurs d'instructions, ainsi que *CWP*, le compteur de fenêtres.

**PC** (Process Counter) est le registre contenant l'adresse logique de l'instruction en cours d'exécution. Il est très semblable aux pointeurs d'instruction que l'on retrouve dans la quasi-totalité des processeurs. Ce registre n'est pas accessible directement, il est mis à jour automatiquement par le processeur à la fin de chaque cycle.

**nPC** (Next Process Counter) est le registre contenant l'adresse de la prochaine instruction à être exécutée. Ce registre est modifié par toutes les instructions de transfert de contrôle. Nous détaillerons l'utilité de ce registre dans le point 2.3.

**CWP** (Current Window Pointer) est le registre contenant le numéro de la fenêtre en cours d'utilisation, c'est-à-dire des registres de fenêtre actuellement visibles. Ce registre ne peut être mis à jour que par les instructions d'entrée et de sortie de procédure, *save* et *restore*, et par les instructions privilégiées réservées au système d'exploitation. Nous verrons l'utilité de ce registre dans le point suivant.

#### Flags

Sparc possède 4 registres de un bit, mis à jour par des instructions suffixées -*cc* en fonction du résultat de l'opération. Ces flags sont ensuite réutilisés dans les instructions de contrôle de flot, ou d'autres instructions arithmétiques. Les flags sont :

**Z** Zero. Le résultat de l'opération est nul.

**N** Negative. Le résultat de l'opération est négatif.

**V** Overflow. Le résultat de l'opération ne peut pas être contenu dans un registre. Le résultat est tronqué.

**C** Carry. Un bit de report doit être comptabilisé dans la prochaine opération.

### 2.1.2 Procédures

Une *procédure* est une routine conçue pour être réutilisée et appelée depuis une autre routine en cas de besoin. Elle est caractérisée par un point d'entrée (adresse de la première instruction), un ensemble de paramètres ou arguments, des points de sortie, des variables locales, et éventuellement une valeur de retour. On parle communément dans ce cas d'une *fonction*.
Lors de l'appel à une procédure, l'état des variables locales de la procédure ou routine appelante est gelé jusqu'à ce que la procédure appelée se termine. Ce procédé permet par exemple de créer des procédures récursives, sans qu'aucune des instances de la même procédure n'interfère avec les variables locales d'autres instances.

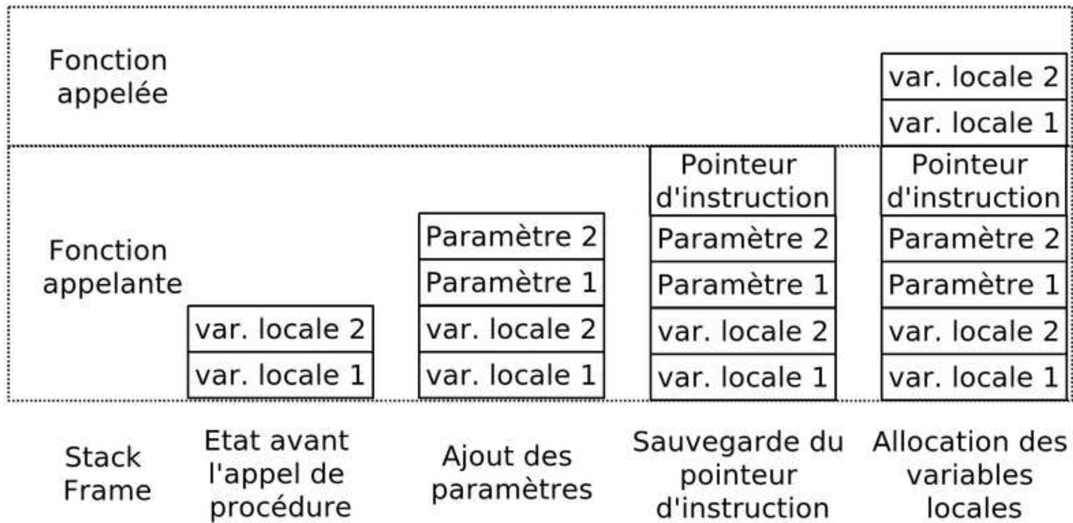


Fig. 2.1 – Évolution de la pile d'une procédure lors d'un appel de procédure sur un processeur ordinaire

La plupart des architectures implémentent ce concept avec une pile. L'état des variables locales, les paramètres et le pointeur d'instruction sont stockés sur une pile au moment d'un appel de procédure, afin de libérer les registres. La procédure appelée peut utiliser ses paramètres qui sont disponibles sur la pile. À la fin de cet appel, le contrôle reprend là où il avait été interrompu (grâce au pointeur d'instruction sur la pile) et les variables locales sont récupérées.
Généralement, l'espace pour les variables locales est réservé et utilisé dans la pile dès l'entrée dans la procédure pour éviter de devoir les sauvegarder lors de l'appel d'une autre procédure. Un exemple d'appel de procédure dans ce cas se trouve sur la fig. 2.1.

Sparc utilise un autre système pour passer les paramètres à la procédure appelée : les registres. Au lieu d'utiliser systématiquement la pile pour effectuer les passages de paramètres ou la sauvegarde des variables locales, l'architecture a déterminé les registres servant à stocker les variables locales, à rechercher les paramètres d'entrée et à stocker les paramètres de sortie, respectivement les registres **l0-l7**, **i0-i7** et **o0-o7**.
Comme dit précédemment, une fenêtre est une vue à un moment précis sur l'ensemble des registres de fenêtre. Le but des fenêtres est de permettre aux fonctions d'un programme de recevoir des paramètres, de contenir des variables locales, et d'appeler d'autres fonctions avec des paramètres, de façon transparente. En effet, les fenêtres sont coulissantes : lors de l'entrée dans une procédure, l'instruction **save** permet d'appliquer une rotation des registres ; le registre **CWP** est incrémenté ; les registres d'entrée et locaux de la procédure appelante ne sont plus accessibles ; le contenu des registres de sortie est maintenant référencé par les registres d'entrée, et les registres locaux et de sortie de la nouvelle procédure sont réinitialisés (fig. 2.2).

Ce système de fenêtre coulissante permet de minimiser de manière très importante les accès à la pile lors des appels de fonction, car dans la plupart des cas, seules des manipulations de registres sont appliquées.

En cas de débordement (le compteur de fenêtre dépasse le nombre de fenêtres disponibles), une exception est levée par le processeur. Le système d'exploitation doit reprendre la main pour sauver sur la pile l'une des fenêtres précédentes afin de laisser de la place à la nouvelle procédure.
Notre processeur idéalisé comportant une infinité de fenêtres, ce cas ne se présentera jamais, et on peut donc l'ignorer.
Malgré que la pile a une importance moindre en Sparc, il est important de noter qu'elle est toujours utilisée pour sauvegarder les registres de fenêtre en cas d'exception ou de Context Switch (le système d'exploitation décide d'exécuter un autre processus sur le processeur), et sert aussi à communiquer les paramètres entre procédures lorsque leur nombre dépasse six ou est variable. L'instruction **save** a trois responsabilités : incrémenter le compteur de fenêtres **CWP**, allouer de la mémoire sur la pile pour une sauvegarde éventuelle des registres de fenêtre, et allouer de la mémoire sur la pile pour les éventuelles variables locales qui ne rentreraient pas dans les registres locaux.
Pour pouvoir utiliser cette mémoire, deux registres sont utilisés :

**fp** Frame Pointer. Ce registre pointe sur le bas de la stack frame, c'est-à-dire sur la première adresse mémoire utilisée par une éventuelle variable locale.

**sp** Stack Pointer. Ce registre pointe sur le haut de la stack frame, à la première adresse immédiatement après la dernière variable locale. En cours d'exécution de la procédure, il peut être ajusté afin de rajouter une ou plusieurs variables locales.

L'instruction **restore** possède les responsabilités inverses de l'instruction **save** : décrémenter le compteur de fenêtre cwp afin de restaurer l'état antérieur de la fenêtre, et désallouer la mémoire de la pile.

## 2.2 Instructions

Il est important, lors de l'analyse d'instructions binaires, de distinguer l'aspect syntaxique et sémantique des instructions. En effet, certaines instructions ayant des sémantiques proches l'une de l'autre peuvent avoir une syntaxe très différente (par exemple si la source d'une instruction est un registre ou une valeur immédiate), et certaines instructions ayant une représentation syntaxique pratiquement identique ont une sémantique radicalement différente.
En outre, l'aspect syntaxique est nécessaire pour comprendre pourquoi certaines instructions ne peuvent faire appel qu'à des valeurs immédiates de taille très limitée.

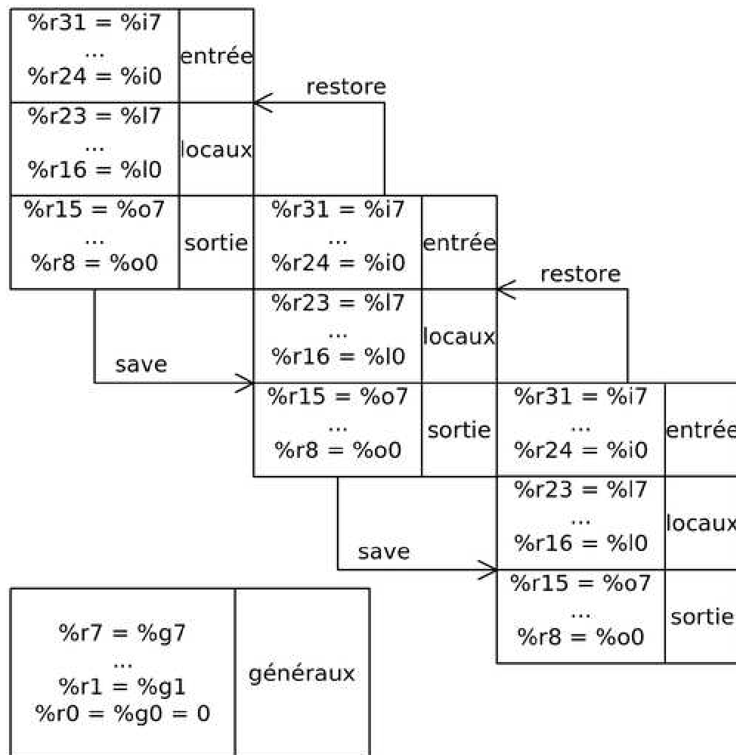


FIG. 2.2 – Registres généraux et fenêtres

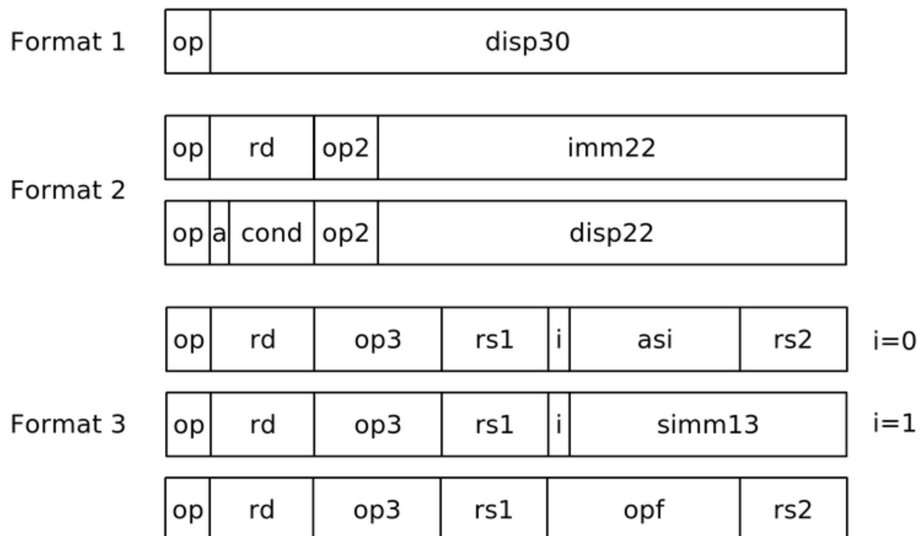


FIG. 2.3 – Les 3 formats d'instruction Sparc32

### 2.2.1 Syntaxe

Sparc étant une architecture RISC, toutes les instructions font la taille d'un mot, c'est-à-dire 32 bits. Les instructions Sparc32 possèdent 3 formats d'instructions (fig. 2.3), le champ *op1* permettant de déterminer le format utilisé :

**Format 1** Ce format d'instruction ne sert que pour l'instruction *call*.

**Format 2** Ce format d'instruction sert aux instructions de branchement, ainsi qu'à l'instruction sethi. le champ *op2* détermine l'instruction, le champ *rd* contient le numéro du registre de destination, Le champ *imm22* contient une valeur immédiate de 22 bits, le champ *a* contient le bit d'annulation, le champ *cond* contient les codes de condition, et le champ *disp22* contient une valeur immédiate signée.

**Format 3** Ce format sert aux instructions load/store, les instructions arithmétiques et logiques, ainsi que les autres instructions. le champ *op3* détermine l'instruction, le champ *rd* contient le numéro du registre de destination (la plupart du temps), le champ *i* contient un bit spécifiant si le paramètre est immédiat ou non. Les champs *rs1* et *rs2* contiennent les numéros des registres sources. Les champs *asi* et *opf* contiennent respectivement un identificateur d'espace alternatif et un codage d'instruction floating-point ou coprocesseur, et ne seront pas utilisés dans ce travail.

L'utilisation de tous ces champs dépend évidement de l'instruction en faisant usage, et leur sémantique peut dévier en fonction de l'instruction (par exemple l'instruction **st** (store) utilise *rd* comme numéro de registre source).
Certains champs utilisés pour des valeurs immédiates ont une taille limitée, car le modèle RISC n'autorise pas les instructions à taille variable.

### 2.2.2 Types de données

Les instructions admettant un paramètre immédiat utilisent une convention de nommage pour ces valeurs immédiates, qui précisent comment elles doivent être interprétées mathématiquement :

**disp30** Offset signé de 30 bits, utilisé pour le branchement. Cet offset est compté en nombre de mots de 32 bits, il est donc nécessaire de le multiplier par 4 avant de l'additionner au registre **PC**.

**disp22** Offset signé de 22 bits, utilisé pour le branchement. Cet offset est compté en nombre de mots de 32 bits, il est donc nécessaire de le multiplier par 4. Comme le résultat de cette multiplication est un entier signé de 24 bits, il est nécessaire d'étendre le bit de signe avant d'effectuer l'addition.

**imm22** Constante de 22 bits utilisée telle quelle dans l'instruction sethi.

**simm13** Valeur immédiate signée de 13 bits. Le bit de signe est étendu sur 32 bits avant utilisation par l'instruction.

### 2.2.3 Sémantique

Sparc possède trois types d'instructions importantes : les Load/Store, les instructions arithmétiques et logiques, et les instructions de transfert de contrôle.

Nous décrivons dans cette section la syntaxe de ces instructions et leur sémantique de manière informelle. Au chapitre suivant, dédié à la traduction de ces instructions, nous donnerons des règles de traduction sémantiques beaucoup plus complètes et précises.

### Load/Store

Les instructions Load/Store sont les seules qui accèdent à la mémoire du système. Elles prennent en paramètre un registre, et une donnée en mémoire à l'adresse pointée par $r[rs1] + r[rs2]$ , ou bien $r[s1] + sign_ext(r[simm13])$ (si $i = 1$).

| Instruction | Description |
|---|---|
| **ld** [address], r[rd] | Charge un word (32 bits) depuis la mémoire à l'emplacement [address] dans le registre de destination. |
| **ldsb** [address], r[rd] | Charge un byte signé (8 bits) depuis la mémoire à l'emplacement [address] dans le registre de destination. |
| **ldub** [address], r[rd] | Charge un byte non signé (8 bits) depuis la mémoire à l'emplacement [address] dans le registre de destination. |
| **ldsh** [address], r[rd] | Charge un half-word signé (16 bits) depuis la mémoire à l'emplacement [address] dans le registre de destination. |
| **lduh** [address], r[rd] | Charge un half-word non signé (16 bits) depuis la mémoire à l'emplacement [address] dans le registre de destination. |
| **ldd** [address], r[rd] | Charge un double-word (64 bits) depuis la mémoire à l'emplacement [address] dans le registre de destination r[rd & 0x1e] et son successeur r[rd & 0x1e + 1]. |
| **st** r[rd], [address] | Enregistre un word (32 bits) depuis un registre dans la mémoire. |
| **stb** r[rd], [address] | Enregistre un byte (8 bits) depuis les 8 bits de poids faible du registre dans la mémoire. |
| **sth** r[rd], [address] | Enregistre un half-word (16 bits) depuis les 16 bits de poids faible du registre dans la mémoire. |
| **std** r[rd], [address] | Enregistre un double-word (64 bits) depuis la paire de registres r[rd & 0x1e] et r[rd & 0x1e +1] dans la mémoire. |
| **swap** [address], r[rd] | Échange le contenu de r[rd] avec le contenu de la mémoire à l'emplacement [address]. |

### Arithmétiques et logiques

Les instructions arithmétiques et logiques englobent les instructions d'arithmétique entière simple (additions, soustractions, multiplications et divisions), de logique, de décalage, de copie de données et d'initialisation.
La plupart de ces instructions prennent 3 registres en paramètres : une destination et deux sources. Les instructions suffixées *-cc* modifient les Flags en fonction du résultat de l'opération.

| | |
|---|---|
| **add** r[rs1], reg_or_imm, r[rd]<br>**sub** r[rs1], reg_or_imm, r[rd] | Addition de r[rs1] et reg_or_imm.<br>Soustraction de r[rs1] et reg_or_imm. |
| **addcc** r[rs1], reg_or_imm, r[rd]<br>**subcc** r[rs1], reg_or_imm, r[rd] | Comportement de **add** avec écriture des flags.<br>Comportement de **subb** avec écriture des flags. |
| **addx** r[rs1], reg_or_imm, r[rd]<br>**subx** r[rs1], reg_or_imm, r[rd] | Comportement de **add** avec utilisation du Carry Flag.<br>Comportement de **sub** avec utilisation du Carry Flag. |
| **addxcc** r[rs1], reg_or_imm, r[rd]<br>**subxcc** r[rs1], reg_or_imm, r[rd] | Comportements associés de **addcc** et **addx**.<br>Comportements associés de **subcc** et **subx**. |

| | |
|---|---|
| **umul** r[rs1], reg_or_imm, r[rd]<br><br><br>**smul** r[rs1], reg_or_imm, r[rd] | Multiplication non signée de r[rs1] et reg_or_imm. Les 32 bits de résultat de poids faible sont assignés à r[rd], les 32 bits de poids fort sont assignés au registre **Y**.<br>Multiplication signée de r[rs1] et reg_or_imm. Les 32 bits de résultat de poids faible sont assignés à r[rd], les 32 bits de poids fort sont assignés au registre **Y**. |
| **umulcc** r[rs1], reg_or_imm, r[rd]<br>**smulcc** r[rs1], reg_or_imm, r[rd] | Même comportement que **umul** avec écriture des flags.<br>Même comportement que **smul** avec écriture des flags. |
| **mulscc** r[rs1], reg_or_imm, r[rd] | Multiplication par étape. Non utilisée dans ce travail. |
| **udiv** r[rs1], reg_or_imm, r[rd]<br><br><br>sdiv r[rs1], reg_or_imm, r[rd] | Division entière non signée de **Y** :r[rs1] par reg_or_imm. **Y** contient les 32 bits de poids fort et r[rs1] les 32 bits de poids faible. Le résultat 32 bits est assigné à r[rd].<br>Division entière signée de **Y** :r[rs1] par reg_or_imm. **Y** contient les 32 bits de poids fort et r[rs1] les 32 bits de poids faible. Le résultat 32 bits est assigné à r[rd]. |
| **udivcc** r[rs1], reg_or_imm, r[rd]<br>**sdivcc** r[rs1], reg_or_imm, r[rd] | Même comportement que **udiv** avec écriture des flags.<br>Même comportement que **sdiv** avec écriture des flags. |

| | |
|---|---|
| **and** r[rs1], reg_or_imm, r[rd]<br>**andn** r[rs1], reg_or_imm, r[rd] | r[rd] := r[s1] and reg_or_imm.<br>r[rd] := r[s1] and not reg_or_imm. |
| **or** r[rs1], reg_or_imm, r[rd]<br>**orn** r[rs1], reg_or_imm, r[rd] | r[rd] := r[s1] or reg_or_imm.<br>r[rd] := r[s1] or not reg_or_imm. |
| **xnor** r[rs1], reg_or_imm, r[rd] | r[rd] := r[s1] xor not reg_or_imm. |

| | |
|---|---|
| **sll** r[rs1], reg_or_imm, r[rd]<br>**srl** r[rs1], reg_or_imm, r[rd]<br>**sra** r[rs1], reg_or_imm, r[rd] | r[rd] := r[s1] shifté de reg_or_imm bits vers la gauche.<br>r[rd] := r[s1] shifté de reg_or_imm bits vers la droite.<br>r[rd] := r[s1] shifté de reg_or_imm bits vers la droite. La valeur des bits vacants est remplacée par la valeur du bit de poids fort de r[s1]. |

Cette classe d'instructions inclus aussi les instructions **save**, **restore**, **sethi**, **rdy** et **wry** :

| | |
|---|---|
| **sethi const22, r[rd]** | Réinitialise les 10 bits de poids faible de r[rd] et initialise les 22 bits de poids fort par la valeur immédiate const22. Cette instruction est la seule permettant d'initialiser les bits de poids fort d'un registre depuis une valeur immédiate. |
| **save r[rs1]**, reg_or_imm, r[rd] | Le rôle de cette instruction est d'ouvrir une nouvelle fenêtre, comme vu dans l'allocation des fenêtres. De plus, save est une instruction permettant de faire une addition comme l'instruction **add**, avec comme particularité que r[rs1] et reg_or_imm appartiennent à l'ancienne fenêtre, et r[rd] appartient à la nouvelle fenêtre. Ce mécanisme permet d'allouer automatiquement de l'espace sur la pile en initialisant le registre **sp** de la nouvelle fenêtre depuis la valeur du registre **sp** de l'ancienne fenêtre. |
| **restore r[rs1]**, reg_or_imm, r[rd] | Cette instruction est parfaitement symétrique à l'instruction **save**. Elle détruit la fenêtre actuelle et restaure la fenêtre précédente. Cette instruction est aussi une instruction d'addition. |
| **rdy** r[rd] | Écrit le contenu du registre **Y** dans r[rd]. |
| **wry** r[rd] | Écrit le contenu de r[rd] dans le registre **Y**. |

### Transfert de contrôle

Ces instructions permettent de modifier le registre **nPC**. Ces instructions sont **call**, **jmpl**, et les sauts conditionnels de type **Bicc** :

| | |
|---|---|
| **jmpl** address, r[rd] | Effectue un saut inconditionnel sur l'adresse, de type r[rs1] + r[rs2] ou r[rs1] + sign_extend(simm13), et sauvegarde le contenu de PC dans le registre r[rd]. Cette instruction est utilisée pour effectuer des **call** indirects. |
| **call** disp30 | Effectue un appel de procédure. La valeur de **PC** est sauvegardée dans le registre r[15] = **o7**. |
| **Bicc** address_offset | Les instructions Bicc sont une classe d'instructions de sauts conditionnels. Elles ont chacune un nom d'instruction différent, et un comportement différent en fonction de l'état des flags et de la condition demandée. Ces instructions peuvent être suffixées par ",a", qui précise que le *annul* bit est mis à 1. Une instruction Bicc avec le *annul* bit n'exécute jamais l'instruction dans le delay slot si la branche n'est pas prise comme nous le verrons dans le point 2.3. |

Les différents types d'instructions **Bicc** :

| | |
|---|---|
| **ba** | Branch Always (inconditionnel). |
| **bn** | Branch Never (inconditionnel). |
| **bne** | Branch on Not Equal. |
| **be** | Branch on Equal. |
| **bg** | Branch on Greater. |
| **ble** | Branch on Less or Equal. |
| **bge** | Branch on Greater or Equal. |
| **bl** | Branch on Less. |
| **bgu** | Branch on Greater Unsigned. |
| **bleu** | Branch on Less or Equal Unsigned. |
| **bcc** | Branch on Carry Clear. |
| **bcs** | Branch on Carry Set. |
| **bpos** | Branch on Positive. |
| **bneg** | Branch on Negative. |
| **bvc** | Branch on Overflow Clear. |
| **bvs** | Branch on Overflow Set. |

**Instructions synthétiques**

L'architecture Sparc définit des instructions synthétiques. Ces instructions sont reconnues par l'assembleur mais ne possèdent pas d'opcode ; elles sont implémentées par d'autres instructions :

| Instruction | Description | Implémentée par |
|---|---|---|
| **cmp** r[rs1], reg_or_imm | Comparaison de valeurs | **subcc** |
| **jmp** reg_or_imm | Saut inconditionnel | **jmpl**, **ba** |
| **call** reg_or_imm | Call indirect | **jmpl** |
| **ret** | Retour de procédure | **jmpl** |
| **set** r[rs1], imm | Initialisation de registre | **sethi**, **or** |
| **mov** r[rs1], reg | Copie de registre | **or** |
| **nop** | Instruction sans effet | **sethi** |

Dans certains cas, il peut être intéressant de reconnaître ces instructions synthétiques car elles ne sont utilisées que dans un cadre particulier. Par exemple, l'instruction synthétique **ret**, marque la fin d'une procédure. Les instructions **cmp**, **set**, **mov**, **nop**, etc. ne nécessitent pas de traitement ad hoc, car l'analyse sémantique et les optimisations (chapitres suivants) permettent de retrouver la forme originale de l'instruction.

## 2.3 Graphe de contrôle de flot

### 2.3.1 Motivations

Le CFG est la pierre angulaire de l'analyse d'algorithmes. Un graphe de contrôle de flot est un graphe dirigé dont les noeuds sont des instructions et les transitions sont des branchements possibles entre les instructions.

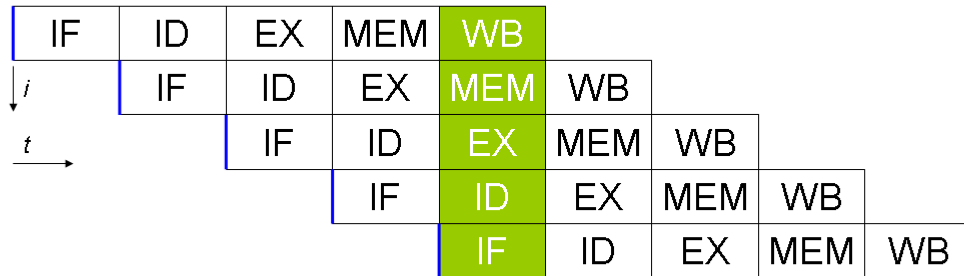


FIG. 2.4 – Schéma d'un pipeline de cinq unités. Source : Wikipédia [8]

Nous expliquerons ici comment se déroule l'exécution d'un programme sur un processeur Sparc. Ensuite, nous présentons deux modèles de CFG spécifiques à Sparc. Une fois ce graphe réalisé, il servira de base pour l'analyse sémantique et la transformation en représentation intermédiaire (voir au chapitre suivant).

### 2.3.2 Pipeline

L'apprentissage du fonctionnement d'un processeur Sparc est impossible sans une description de son pipeline.
Un pipeline, dans le contexte d'un microprocesseur, est une file d'instructions devant être exécutées. Le processeur remplit cette file parallèlement à sa tâche première (exécuter des instructions), dans le but de rester le moins possible en état oisif, en attente d'instructions à exécuter. Ce pipeline permet aussi au processeur d'utiliser au maximum toutes les unités de travail disponibles (décodage des instructions, exécution des instructions, sauvegarde des résultats, ...) en les faisant travailler en parallèle.
Pratiquement tous les processeurs modernes possèdent un pipeline plus ou moins profond, dans le but d'approcher au plus possible du ratio d'une instruction exécutée par cycle processeur. Le schéma de la figure 2.4 expose le pipeline à cinq étages d'un processeur.
Il existe deux catégories de pipelines :

#### Pipeline transparent

La plupart des processeurs, notamment la série des x86, utilisent un pipeline transparent. Le processeur possède un pipeline d'une taille indéterminée, et ce pipeline est invisible pour le programmeur. Si une instruction de branchement modifie totalement l'état du processeur (par exemple suite à un saut conditionnel), le processeur prend la responsabilité d'invalider tout le contenu du pipeline qui n'est plus pertinent. Le contenu du pipeline sera rechargé de façon transparente.
Certains processeurs tentent de "deviner" quels seront les états futurs du programme, ou quels seront les branchements qui seront faits, afin d'éviter de devoir vider le pipeline. Dans la majorité des cas, le programmeur n'a que peu

```
 0 subcc l1, g0, g0 ; l1 > 0 ?
 4 bneg 20 ; jump à 20 si < 0
 8 sub g0, l1, l2 ; l2 := 0 - l1
12 jmp 24
16 nop
20 or l2, l2, l1 ; l1 := l2
24 ret
28 nop
```

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Fetch | 0 | 4 | 8 | 20 | 24 | 28 | ? |
| Execute | ? | 0 | 4 | 8 | 20 | 24 | 28 |

FIG. 2.5 – Exemple d'utilisation du pipeline avec $l1 < 0$.

```
 0 subcc l1, g0, g0 ; l1 > 0 ?
 4 bneg, a 20 ; jump à 20 si < 0
 8 sub g0, l1, l2 ; l2 := 0 - l1
12 jmp 24
16 nop
20 or l2, l2, l1 ; l1 := l2
24 ret
28 nop
```

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Fetch | 0 | 4 | 8 | 12 | 16 | 24 | 28 | ? |
| Execute | ? | 0 | 4 | ~~8~~ | 12 | 16 | 24 | 28 |

FIG. 2.6 – Exemple d'utilisation du pipeline avec $l1 > 0$ et une instruction de contrôle avec le *annul* bit.

de contrôle sur ce processus[1].

**Pipeline apparent**

D'autre processeurs, construits dans une optique de simplicité et d'économie, prévoient d'exposer en partie le fonctionnement du pipeline au programmeur. Sparc fonctionne selon ce modèle.
Le pipeline d'un processeur Sparc possède deux étages apparents : le *fetch* et l'*execute*. Toute instruction exécutée passe d'abord dans l'étage fetch, qui récupère l'instruction en mémoire, et l'étage *execute*, qui effectue le calcul et met à jour les registres ou la mémoire.
Le programmeur doit donc prendre en compte l'instruction qui restera dans le pipeline lors d'un transfert de contrôle, car cette instruction sera exécutée dans la plupart des cas.

### 2.3.3 Enchaînement des instructions

À tout instant, le registre **PC** contient l'adresse de l'instruction en cours d'exécution, et **nPC** l'adresse de la prochaine instruction à être exécutée. Lorsqu'une instruction est en cours d'exécution, l'instruction pointée par **nPC** est récupérée (fetch) et analysée. Sparc possède un pipeline de deux instructions : il

[1] En pratique, certains processeurs comme le Pentium 4 permettent au programmeur d'indiquer les branchements les plus probables

y a toujours simultanément une instruction en cours d'exécution (celle pointée par **PC**) et une instruction en cours de fetch/analyse.
Lorsque l'instruction en cours d'exécution est une instruction de transfert de contrôle, cette instruction modifie (en cas de saut) le registre **nPC**. Cependant, l'instruction qui était pointée par l'ancien **nPC** se trouve déjà dans le pipeline. Au cycle suivant, l'instruction qui était pointée par l'ancien **nPC** est effectivement exécutée.

Ceci implique que l'instruction qui succède à une instruction de transfert de contrôle sera toujours exécutée (sauf exception). Cette instruction prisonnière du pipeline s'appelle la *delay* instruction, et s'exécute dans le *delay slot*.
Dans l'exemple de la figure 2.5, l'instruction 4 est une instruction de transfert de contrôle et l'instruction 8 est une delay instruction. Cette instruction s'exécute quand même, malgré le fait que l'instruction 4 ait fait un saut à l'instruction 20.

Il existe deux exceptions à cette règle : lorsqu'une des instructions **Bicc, a**, qui possèdent le *annul* bit, est exécutée, l'instruction du delay slot est éliminée si la branche n'est pas prise. Le processeur annule les modifications effectuées par cette instruction.
L'exemple de la figure 2.6 montre ce cas : l'instruction 4 **bneg, a** possède le bit *annul*. L'instruction du delay slot, l'instruction 8, est donc ignorée.
Les instructions **Bicc, a** non conditionnelles (**ba**,**bn**) ayant le annul bit éliminent toujours l'instruction du delay slot, même si la branche est effectivement prise.

### 2.3.4 Graphe de flot naïf

Quelques notions doivent être introduites :

**Noeud** Dans un graphe de flot, chaque instruction est représentée par un noeud contenant un successeur, une branche et l'instruction en elle-même.

**Successeur** Un noeud $n_2$ est successeur d'un noeud $n_1$ ssi l'instruction de $n_2$ est l'instruction suivant immédiatement l'instruction de $n_1$ dans le flux d'exécution. Il se peut qu'un noeud n'ait pas de successeur.

**Branche** Un noeud $n_2$ est branche d'un noeud $n_1$ ssi l'instruction de $n_1$ est une instruction de transfert de contrôle et ssi l'instruction de $n_2$ est l'instruction pointée par l'instruction de branchement.

**Accessibilité** Un noeud $n_2$ est accessible depuis le noeud $n_1$ ssi
- $n_2$ est successeur ou branche de $n_1$ ou
- $\exists n$ où $n$ est successeur ou branche de $n_1$ et $n_2$ est accessible depuis $n$.

Cette relation est donc transitive.

Lorsque le prototype ouvre un fichier exécutable, un graphe est construit depuis le point d'entrée. Pour chaque instruction accessible depuis le point d'entrée, un noeud est créé. Les procédures sont découvertes (grâce aux instructions **call**). Une procédure est caractérisée par un point d'entrée, et un ensemble de noeuds. Une instruction **ret** au moins est accessible depuis chaque noeud de la procédure. Le graphe de flot d'une procédure est donc un graphe dirigé dont les

éléments sont les noeuds de l'ensemble, dont le premier noeud est le noeud du point d'entrée de la procédure, et où une transition entre deux noeuds $n_1$ et $n_2$ est présente ssi $n_2$ est successeur ou branche de $n_1$.

### 2.3.5 Problème du delay slot

Le graphe de flot naïf ne peut pas directement servir à l'analyse du programme. En effet, il n'est pas représentatif de la manière dont les instructions vont réellement être exécutées, puisque les instructions de transfert de contrôle exécutent presque toujours l'instruction suivante avant de transférer l'exécution à la branche.

La solution à ce problème est de réorganiser le graphe de façon à réinsérer les instructions du delay slot à la sortie des instructions de flot.

Pour tout $n_1$, noeud du graphe de flot naïf, $i_1$ son instruction correspondante, $n_2$ son successeur et $i_2$ son instruction :

- si $i_1$ n'est pas une instruction de transfert de contrôle, aucune modification n'est faite ;
- si $i_1$ est une instruction **ret**, alors le successeur de $n_1$ reste $n_2$. $n_2$ n'a plus de successeur ;
- si $i_1$ est une instruction **Bicc** (fig. 2.8) dont le bit *annul* est 0, alors le successeur de $n_1$ reste $n_2$. Un nouveau noeud nommé $n'_2$ est créé et sera inséré dans la branche de $n_1$ : son instruction est l'instruction de $i_2$, le successeur de $n'_2$ est la branche de $n_1$ et la branche de $n_1$ devient $n'_2$ ;
- si $i_1$ est une instruction **Bicc, a** conditionnelle (fig. 2.9), le successeur de $n_1$ devient le successeur de $n_2$. Le successeur de $n_2$ devient la branche de $n_1$, et la branche de $n_1$ devient $n_2$ ;
- si $i_1$ est une instruction **Bicc, a** non conditionnelle (**ba**,**bn**), le successeur de $n_1$ devient le successeur de $n_2$ ;
- si $i_1$ est une instruction **call** (fig. 2.7), le successeur de $n_1$ devient le successeur de $n_2$. Le successeur de $n_2$ devient $n_1$. Les successeurs ou branches des ancêtres de $n_1$ sont mis à jour pour pointer sur $n_2$.

Cet algorithme est nettement suffisant pour traiter la plupart des binaires Sparc disponibles. Néanmoins, il traite de manière incorrecte les cas dans lesquels l'instruction du delay slot (qui est donc dupliquée et déplacée dans chaque branche) est elle-même une instruction de contrôle de flot. Dans ces cas extrêmes, le flot d'instruction ne suit plus exactement les mêmes règles de traduction. Il est possible de traiter correctement ces cas en simulant l'état des registres **PC** et **nPC** lors des branchements, et d'ordonner les instructions sur le graphe de sorte qu'elles s'exécutent dans le bon ordre.

Ce cas est suffisamment rare pour que notre prototype ne s'en occupe pas.

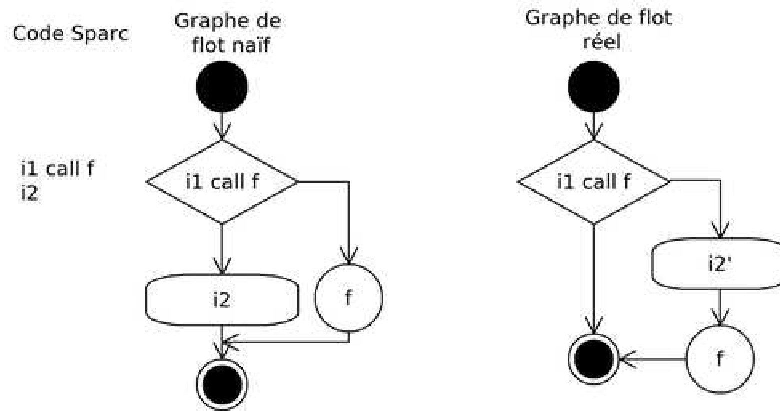


FIG. 2.7 – Transformation du graphe de flot d'un **call**

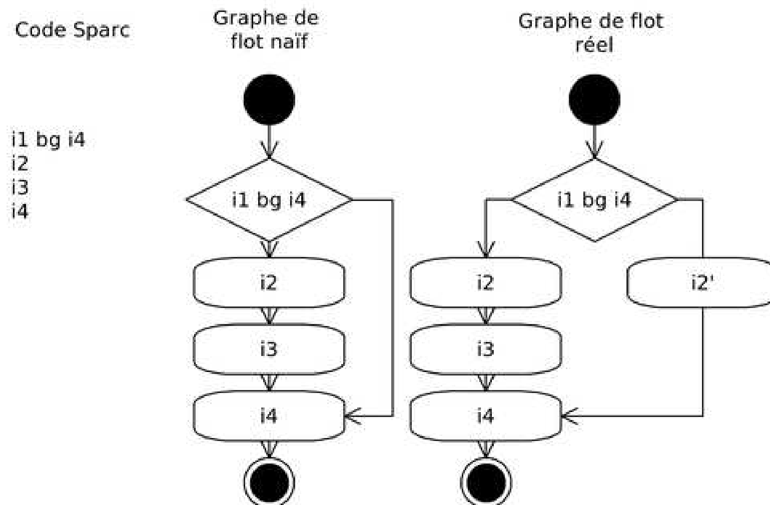


FIG. 2.8 – Transformation du graphe de flot d'une instruction de contrôle

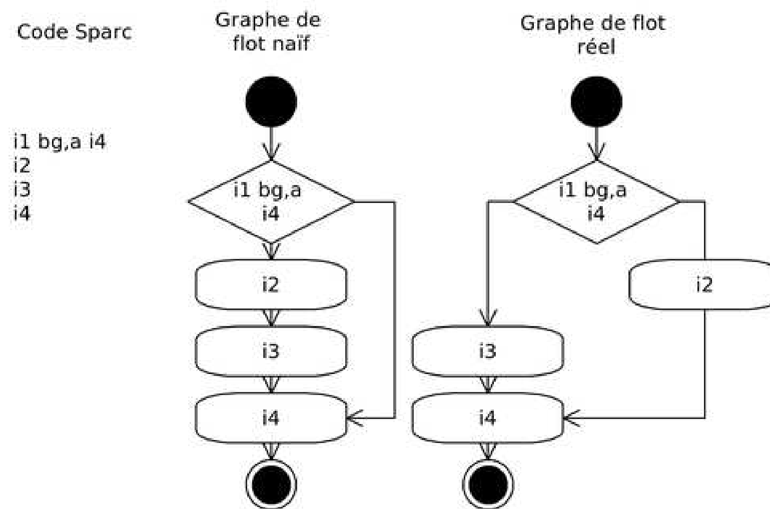


FIG. 2.9 – Transformation du graphe de flot d'une instruction de contrôle avec le *annul* bit

# Chapitre 3

# Représentations intermédiaires

## 3.1 Rôle

Dans le chapitre précédent, nous avons expliqué comment obtenir un graphe de contrôle de flot à partir du code Sparc. Nous avons obtenu un graphe dirigé qui décrit le fonctionnement du programme. Néanmoins, la forme de ce graphe ne nous permet pas d'effectuer facilement certaines opérations d'optimisation ou de transformation.

Plusieurs caractéristiques sont attendues d'une représentation intermédiaire :

- C'est un langage simple et complet permettant d'exprimer la sémantique d'un programme sans devoir se lier aux contraintes du langage source. Les effets de bord de certaines instructions du langage source compliqueraient inutilement l'étape d'optimisation.
- Chaque instruction de cette représentation intermédiaire ne doit avoir qu'un seul effet et ne modifier qu'une seule destination. En minimisant le nombre d'actions réalisées par une instruction, on simplifie l'implémentation de l'optimiseur et du back-end, car ils auront moins de cas particuliers à gérer.
- Idéalement, une représentation intermédiaire doit être suffisamment générique pour pouvoir servir à traduire des programmes de langages différents, par exemple en rajoutant des front-ends au compilateur.
- Le langage doit s'éloigner des spécificités de la machine source. La représentation intermédiaire ne possède pas de variables privilégiées, et donc pas de registres.

## 3.2 Types de formes intermédiaires

Il existe deux catégories de formes intermédiaires : les codes à trois adresses et les machines à pile, qui se différencient notamment par la forme et la sémantique des instructions.

### 3.2.1 Code à trois adresses

Un code à trois adresses, tel qu'il est décrit dans le Dragon Book [12], est un jeu d'instructions réduit suivant la grammaire suivante :
instr → call ($v_0$, $v_1$, ..., $v_n$)
instr → ret $v_s$
instr → $v_d := v_0$ op $v_1$
instr → if (cond) jump label

Dans ce jeu d'instructions, les opérations fondamentales sont l'assignation, l'appel et le retour de procédure, et le saut conditionnel.
Énormément de formes intermédiaires de compilateurs s'apparentent à un code à trois adresses. Des variantes possibles sont le rajout d'instructions, le rajout d'opérateurs unaires, la spécification d'opérateurs particuliers, des conventions de nommages des variables, ...
L'environnement de travail de cette forme intermédiaire ressemble à celui que l'on retrouve dans la plupart des systèmes informatiques : un espace contenant les instructions (le code), une zone mémoire statique (les variables globales) et une zone de mémoire dynamique, la pile. Cette pile n'est pas indispensable pour décrire le fonctionnement d'une procédure (on peut se contenter de dire que certaines variables sont dans la portée (scope) de la procédure en cours). Si le passage des paramètres se fait explicitement, il n'est pas nécessaire de gérer une pile pour le passage de ceux-ci.

### 3.2.2 Machine à pile

Une machine à pile est une machine dont les opérations ne communiquent que par la pile. Toutes les instructions obtiennent leurs arguments sur la pile, et renvoient leur résultat sur la pile. Tous les opérateurs se présentent sous la forme d'instruction, prenant les arguments sur la pile, pour en déposer le résultat.
Des exemples de machines à pile sont la P-Machine, définie dans le livre de Wilhelm et Maurer [21], la JVM, qui est précisément le modèle géré par notre back-end, la machine virtuelle Python (l'interpréteur Python [9] interprète du bytecode précompilé), ...
Le code pour la P-Machine est utilisé comme représentation intermédiaire par Wilhelm et Maurer pour leur compilateur Pascal. Ces formes relativement différentes des formes en codes à trois adresses sont très faciles à générer automatiquement, car elles évacuent certains problèmes compliqués de compilation, dont le plus connu est le problème de choix des registres. Néanmoins, effectuer des transformations et des optimisations sur ces formes demande un travail différent d'une forme à trois adresses.

### 3.2.3 Choix de la forme intermédiaire

Il a été nécessaire de faire un choix entre ces deux moyens de représenter les instructions. En faveur de la machine à pile, l'argument principal est la ressemblance entre la JVM et la P-Machine, ce qui aurait permis de rendre le

back-end très simple. En faveur du code à trois adresses, l'argument principal est l'abondance de documentation sur la manière de pratiquer des analyses et optimisations sur cette forme. La ressemblance avec Sparc est aussi un avantage, qui permet de simplifier la traduction en langage intermédiaire.

Tenter les deux solutions pour évaluer laquelle est la plus adaptée aurait nécessité beaucoup de travail, c'est pour cela que nous avons du choisir. L'abondance de documentation nous a poussé à utiliser une forme de code à trois adresses.

## 3.3 Formes intermédiaires particulières

### 3.3.1 RTL

L'équipe d'UQBT a décrit dans son rapport ([15]) les techniques mises en oeuvre pour transformer le programme binaire en RTL (Register Transfer List). RTL est un langage permettant de décrire les actions effectuées par le processeur. RTL décrit des assignations d'expressions à des variables (sans distinction entre les registres et les variables locales et globales), sous la forme $variable := expression$, où $expression$ est une formule mathématique impliquant d'autres variables ou constantes.
UQBT distingue plusieurs sortes de RTL : $M_s$-RTL et H-RTL.
$M_s$-RTL est une forme de RTL spécialisée pour la machine source. Chaque registre ou drapeau prend la forme d'une variable. Chaque instruction est traduite en $M_s$-RTL suivant des règles bien définies (langage SSL).
H-RTL (High level RTL) est une forme de RTL ne dépendant pas de la plateforme source. Le rôle de UQBT est de transformer successivement le programme source en $M_s$-RTL puis cette forme en H-RTL.
Pour c e faire, l'équipe de UQBT a développé un langage de description de transformation sémantique, nommé SSL, Semantic Specification Language ([14]). Une règle SSL décrit formellement, pour une instruction du langage source, le nouvel état de la machine après l'exécution de l'instruction, c'est-à-dire les registres ou les éléments mémoires modifiés.

### 3.3.2 Gnu RTL

Gnu RTL est la version de RTL utilisée par GCC, le compilateur de GNU. Gnu RTL n'est pas une forme intermédiaire formelle définie par une grammaire formelle, mais un ensemble de structures interconnectées au sein du compilateur. Il n'existe pas de forme écrite de Gnu RTL. Le compilateur ne permet pas d'écrire la forme intermédiaire sur le disque, celle-ci n'existe que sous la forme de structures de données dans le programme du compilateur.
Gnu a choisi cette voie principalement pour des raisons politiques : la structure intermédiaire du compilateur n'étant pas exposée à l'utilisateur, les compagnies utilisant la suite GCC seraient obligées d'y intégrer leurs modifications, et donc de les rendre libres.
Cette forme intermédiaire interne permet aussi à GCC d'évoluer sans devoir

mettre à jour les documentations du format Gnu RTL. Les seules spécifications de Gnu RTL sont donc les sources du compilateur GCC.

### 3.3.3 LLVM

LLVM [6] est une infrastructure pour la conception de compilateurs comprenant un langage de machine virtuelle bas niveau. Ce framework permet de générer facilement un compilateur statique ou dynamique à partir de la représentation intermédiaire de LLVM. Le framework s'occupe des optimisations et de la gestion d'une partie du back-end.
Le prototype gagnerait probablement en efficacité et en temps d'implémentation en utilisant ce framework, mais cela nous obligerait à documenter le fonctionnement complet de LLVM, ce qui sort du cadre de ce travail. En outre, LLVM utilise en interne beaucoup des techniques documentées dans ce présent travail, du moins dans les sections optimisations et back-end.
L'utilisation de LLVM ne nous dispenserait pas du travail de traduction vers le format LLVM, et les travaux d'analyse sémantique décrits dans le chapitre suivant.

### 3.3.4 SSA

SSA (Single Static Assignement) est une forme particulière de code à trois adresses dans lequel toute partie gauche d'une assignation n'est présente qu'une seule fois, c'est-à-dire que chaque variable n'est assignée qu'une seule fois. Un morceau de programme du type

$$x := x + 1$$

$$x := x * 5$$

se traduit par

$$x_2 := x_1 + 1$$

$$x_3 := x_2 * 5$$

La forme SSA est utilisée par de nombreux compilateurs, car elle permet de rendre beaucoup d'algorithmes de transformation de code plus simples à concevoir. Des exemples sont GCC, Boomerang (issus de UQBT), LLVM, le compilateur JIT du JDK Sun, ...

## 3.4 Une forme intermédiaire

Un code à trois adresses est une représentation intermédiaire dans laquelle la plupart des opérations sont sous la forme d'une instruction de la forme $dest := s_1 \; op \; s_2$, *op* étant un des opérateurs connus, *dest* étant une variable et $s_1$ et $s_2$ des variables ou des constantes. D'autres instructions peuvent être rajoutées pour obtenir un langage suffisamment complet pour simuler le programme d'origine, par exemple l'appel de procédure ou le déférencement de variables.

Nous utiliserons à l'avenir une version simpliste de ce code à trois adresses, qui a l'avantage d'être facile à générer depuis du code RISC. Notre travail ayant une portée plus limitée que d'autres gros projets comme UQBT ou LLVM, nous utiliserons une forme intermédiaire simplifiée et ayant des règles de traduction sémantiques les plus simples possible.

### 3.4.1 Définition

Les instructions disponibles sont l'assignation, l'appel de procédure, le retour de procédure et le saut conditionnel. La grammaire simplifiée de ces instructions est la suivante :

**Assignation** dest := $operand_1$ op $operand_2$
dest := *operand*
dest := op *operand*
dest := call *entrypoint*, $operand_1$, $operand_2$, ..., $operand_n$

**Retour de procédure** ret *operand*

**Saut conditionnel** if (*operand*) goto *entrypoint*

Les opérandes peuvent être des variables locales, des variables globales, et des espaces en mémoire (paramétrés par une variable locale ou globale), ou des valeurs immédiates (constantes). La destination d'une assignation peut être une variable locale ou globale, ou un espace en mémoire.
Ces opérandes se représentent de cette manière :

VAR[v] Variable locale de nom 'v'.

MEM[operand] Zone mémoire de 32 bits, pointée par operand. Operand peut être une constante ou une autre variable.

GLOBAL[v] Variable globale (accessible depuis toutes les procédures) de nom 'v'.

IMM[value] Valeur immédiate (constante) dont la valeur est 'value'.

MEM16[operand] Zone mémoire de 16 bits, pointée par operand.

MEM8[operand] Zone mémoire de 8 bits, pointée par operand.

Les opérateurs disponibles sont très nombreux. On compte les opérations arithmétiques de base, les opérateurs logiques, les opérateurs de comparaison (<,<=,...), et des opérateurs particuliers :

sign_extend(imm13) Extension d'un entier signé de 13 bits vers un entier signé de 32 bits.

sign_extend(operand, m) Extension de signe sur la valeur entière 'operand' de m bits vers un entier signé de 32 bits.

zero_extend(operand) Extension d'un entier non signé de valeur 'operand' vers un entier non signé de 32 bits.

overflow(operator, operand1, operand2 [,carry]) Prédicat renvoyant 1 si l'opération 'operator' appliquée sur les deux entiers signés 'operand1' et 'operand2' produit un résultat provoquant un dépassement. Un dépassement a lieu lorsque le résultat de l'opération n'est pas exprimable en un entier

signé de 32 bits (il faudrait par exemple un entier signé de 33 bits ou plus pour pouvoir exprimer le résultat). Si le paramètre 'carry' optionnel est présent, il intervient dans le calcul du dépassement.

carry(operator, operand1, operand2 [,carry]) Prédicat renvoyant 1 si l'opération 'operator' appliquée sur les deux entiers non signés 'operand1' et 'operand2' produit un bit de retenue. Si le paramètre 'carry' optionnel est présent, il intervient dans le calcul de la production de retenue.

VAR[v][0 :31] assigne les 32 bits de poids faibles de la variable 'v'.

VAR[v][32 :63] assigne les 32 bits de poids forts de la variable 'v'.

L'extension de signe est une opération binaire permettant d'étendre un entier signé de m bits vers un entier signé de n bits, lorsque n est plus grand que m.
Une grammaire formelle plus complète, en forme BNF étendue, est décrite à la figure 3.1. Cette grammaire sera revisitée et enrichie plus loin lors de la production de code au chapitre 6.

### 3.4.2 Règles de transformation sémantique

Des règles de transformation sémantique sont des règles permettant de transformer toute instruction en assembleur Sparc en instructions du langage intermédiaire. Nous ne suivrons pas l'approche de UQBT qui a développé un langage (SSL) dans le but unique de décrire des règles de traduction sémantique. Nos règles sont statiques et programmées dans le prototype.
Cette approche convient pour l'élaboration d'un petit prototype, car le nombre de règles à créer est relativement faible (UQBT supportait plusieurs front-ends et donc plusieurs ensembles de règles différentes).

Certaines règles de traductions produisent plusieurs instructions en langage intermédiaire depuis une unique instruction Sparc. C'est notamment le cas pour les instructions Sparc qui modifient plusieurs registres, ou utilisent un résultat intermédiaire.

Les instructions load/store décrites dans la section 2.2.3 sont les seules instructions ayant accès à la mémoire du programme. Ces instructions ont deux modes d'adressage : l'adresse en mémoire peut être la somme du contenu de deux registres, ou la somme du contenu d'un registre avec un offset immédiat (constant) de 13 bits. Le tableau de la figure 3.2 décrit le principe de transformation de ces instructions. D'abord, une variable temporaire reçoit l'adresse mémoire calculée depuis les deux registres ou le registre et la valeur immédiate. Ensuite, la mémoire à cette adresse est assignée ou récupérée à l'aide de l'opérateur MEM[$x$].
Dans le cas de l'instruction **swap**, une valeur temporaire est nécessaire afin de faire l'échange de données entre le registre et l'emplacement mémoire.

| | | |
|---|---|---|
| *instr* | $\leftarrow$ | *dest* := *operand op operand* |
| | $\leftarrow$ | *dest* := *operand* |
| | $\leftarrow$ | *dest* := (*overflow* $\mid$ *carry*) (*op*, *operand*,*operand*) |
| | $\leftarrow$ | *dest* := call *entrypoint*, *operandlist* |
| | $\leftarrow$ | ret *operand* |
| | $\leftarrow$ | if (*operand*) goto *label* |
| *operand* | $\leftarrow$ | VAR[*name*] *range* |
| | $\leftarrow$ | GLOBAL[*name*] |
| | $\leftarrow$ | IMM[*integer*] |
| | $\leftarrow$ | MEM[*operand*] |
| | $\leftarrow$ | MEM8[*operand*] |
| | $\leftarrow$ | MEM16[*operand*] |
| *dest* | $\leftarrow$ | MEM[*operand*] |
| | $\leftarrow$ | MEM8[*operand*] |
| | $\leftarrow$ | MEM16[*operand*] |
| | $\leftarrow$ | VAR[*name*] |
| | $\leftarrow$ | GLOBAL[*name*] |
| *operandlist* | $\leftarrow$ | (*operand*,)* *operand* |
| | $\leftarrow$ | $\epsilon$ |
| *op* | $\leftarrow$ | $+ \mid - \mid * \mid / \mid$ or $\mid$ and $\mid$ xor $\mid$ nor $\mid$ nand $\mid < \mid = \mid > \mid \leq \mid \geq$ |
| *range* | $\leftarrow$ | [0 :31] |
| | $\leftarrow$ | [32 :63] |
| | $\leftarrow$ | $\epsilon$ |
| *integer* | $\leftarrow$ | **int** |
| | $\leftarrow$ | zero_extend(**int**) |
| | $\leftarrow$ | sign_extend(**int**) |
| *name* | $\leftarrow$ | **identifier** |

FIG. 3.1 – Grammaire E-BNF de la forme intermédiaire

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| ld r[rs1], r[rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[rd] := MEM[tmp] |
| ld r[rs1], imm13, r[rd] | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[rd] := MEM[tmp] |
| st r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>MEM[tmp] := VAR[rd] |
| st r[rd], r[rs1], imm13 | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>MEM[tmp] := VAR[rd] |
| swap r[rs1], r[rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[tmp2] := VAR[rd]<br>VAR[rd] := MEM[tmp]<br>MEM[tmp] := VAR[tmp2] |
| swap r[rd], r[rs1], imm13 | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[tmp2] := VAR[rd]<br>VAR[rd] := MEM[tmp]<br>MEM[tmp] := VAR[tmp2] |

FIG. 3.2 – Transformation des instructions load/store

Les instructions *load* (**ld**,**ldsb**,**ldub**,**ldsh**,**lduh**,**ldd**) et les instructions *store* (**st**,**stb**,**stj**,**std**) traitent les entiers de tailles différentes. Les instructions **ldsb** et **ldsh**, décrites dans le tableau de la figure 3.3 récupèrent une valeur de 8 bits ou de 16 bits depuis la mémoire pour la placer dans un registre, en pratiquant l'extension de signe.

Les instructions **ldub** et **lduh** récupèrent une valeur de 8 bits ou de 16 bits depuis la mémoire pour la placer dans un registre, mais en étendant les bits manquants par des zéros (en utilisant zero_extend()).
L'instruction **ldd** copie une valeur de 64 bits depuis la mémoire dans deux registres. Le premier de ces registres est spécifié dans l'instruction lui-même (ici noté VAR[rd]). Le second est statiquement associé au premier, et est choisi selon la formule décrite dans la section 2.2.3. Il est ici noté VAR[rd'].

Les instructions **stb**, **sth** et **std** (fig. 3.4) servent à copier le contenu d'un registre vers une zone mémoire de respectivement 8, 16 ou 64 bits. Les instructions **stb** et **sth** tronquent le contenu du registre à 8 bits et 16 bits. L'instruction **std** fonctionne selon le même principe que **ldd**, et le schéma de traduction est similaire.

L'instruction **sethi** initialise un registre depuis une valeur immédiate. La traduction faite en langage intermédiaire (fig. 3.5) multiplie la valeur immédiate par $2^{10}$, car l'instruction ne peut contenir que les 22 bits de poids forts. Les 10 bits de poids faibles sont fixés à 0.

Les instructions arithmétiques simples (**add**,**sub**) sont traduites très sim-

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| ldsb r[rs1], r[rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[rd] := sign_extend(MEM8[tmp],8) |
| ldsb r[rs1], imm13, r[rd] | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[rd] := sign_extend(MEM8[tmp],8) |
| ldsh r[rs1], r[rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[rd] := sign_extend(MEM16[tmp],16) |
| ldsh r[rs1], imm13, r[rd] | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[rd] := sign_extend(MEM16[tmp],16) |
| ldub r[rs1], r[rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[rd] := zero_extend(MEM8[tmp]) |
| ldub r[rs1], imm13, r[rd] | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[rd] := zero_extend(MEM8[tmp]) |
| lduh r[rs1], r[rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[rd] := zero_extend(MEM16[tmp]) |
| lduh r[rs1], imm13, r[rd] | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[rd] := zero_extend(MEM16[tmp]) |
| ldd r[rs1], r[rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[rd][0 :31] := MEM[tmp]<br>VAR[tmp] := VAR[tmp] + IMM[4]<br>VAR[rd'][32 :64] := MEM[tmp] |
| ldd r[rs1], imm13, r[rd] | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[rd][0 :31] := MEM[tmp]<br>VAR[tmp] := VAR[tmp] + IMM[4]<br>VAR[rd'][32 :64] := MEM[tmp] |

FIG. 3.3 – Transformation des instructions load signées et de tailles différentes

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| stb r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>MEM8[tmp] := VAR[rd] |
| stb r[rd], r[rs1], imm13 | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>MEM8[tmp] := VAR[rd] |
| sth r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>MEM16[tmp] := VAR[rd] |
| sth r[rd], r[rs1], imm13 | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>MEM16[tmp] := VAR[rd] |
| std r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>MEM[tmp] := VAR[rd]<br>VAR[tmp] := VAR[tmp] + IMM[4]<br>MEM[tmp] := VAR[rd'] |
| std r[rd], r[rs1], imm13 | VAR[tmp] := VAR[rs1] + IMM[sign_extend(imm13)]<br>MEM[tmp] := VAR[rd]<br>VAR[tmp] := VAR[tmp] + IMM[4]<br>MEM[tmp] := VAR[rd'] |

FIG. 3.4 – Transformation des instructions store de tailles différentes

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| sethi const22, r[rd] | VAR[rd] := IMM[const22 * $2^{10}$] |

Fig. 3.5 – Transformation de l'instruction **sethi**

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| add r[rd], r[rs1], r[rs2] | VAR[rd] := VAR[rs1] + VAR[rs2] |
| add r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] + IMM[sign_extend(imm13)] |
| sub r[rd], r[rs1], r[rs2] | VAR[rd] := VAR[rs1] - VAR[rs2] |
| sub r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] - IMM[sign_extend(imm13)] |

Fig. 3.6 – Transformation des instructions arithmétiques

plement en une seule instruction du langage intermédiaire (fig. 3.6). Les instructions arithmétiques avec modification des flags (fig. 3.7) sont plus complexes : les instructions **addcc** et **subcc** modifient les flags **N**,**Z**,**V** et **C**. Le flag **N** est écrit en examinant si le résultat est négatif. Le flag **Z** est mis à 1 si le résultat est égal à 0. Le flag **V** est mis à 1 si le résultat provoque un overflow. Le prédicat overflow(opérateur, opérande1, opérande2) calcule s'il y a overflow. Le flag **C** est mis à 1 si le résultat provoque une retenue, et est déterminé par le prédicat carry(opérateur, opérande1, opérande2).
Les instructions arithmétiques utilisant le bit de retenue (fig. 3.8) sont **addx** et **subx**. Ces instructions additionnent ou soustraient la valeur du bit de retenue (le flag **C**) au résultat.
Les instructions arithmétiques **addxcc** et **subxcc** (fig. 3.9) additionnent les propriétés des instructions ci-dessus : le bit de retenue est utilisé pour calculer le résultat, et en fonction du résultat, les différents bits sont mis à jour.

Les instructions de multiplication et de division (**umul**,**udiv**) sont présentées à la figure 3.10. L'instruction **umul** est traduite en plusieurs opérations : le résultat de la multiplication est affecté à une variable temporaire de 64 bits. Ensuite, la partie haute du résultat est affecté à la variable **Y**. La partie basse est affectée à la variable du résultat.
La division (**udiv**) est l'opération inverse : l'entier 64 bits formé par le contenu de la variable **Y** et de la variable source est divisé par le diviseur de 32 bits.
Les instructions **rdy** et **wrt** permettent respectivement de lire et écrire le contenu de la variable **Y**.

Les instructions **and** et **or** sont traduites simplement en une expression en langage intermédiaire, à l'aide de l'opérateur en question (fig. 3.11). Les opérateurs **nand**,**nor** et **xor**, omis de la liste, sont définis de manière totalement semblable.

Les instructions **save** et **restore** ont une sémantique plus compliquée, car elles effectuent des copies de registres vers la mémoire et des copies entre registres. La traduction de ces instructions (fig. 3.12) effectue l'allocation de nouvelles variables locales pour la sauvegarde des registres.

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| addcc r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(+, VAR[rs1], VAR[rs2])<br>C := carry(+, VAR[rs1], VAR[rs2])<br>VAR[rd] := VAR[tmp] |
| addcc r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] + IMM[sign_extend(imm13)]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(+, VAR[rs1], IMM[sign_extend(imm13)])<br>C := carry(+, VAR[rs1], IMM[sign_extend(imm13)])<br>VAR[rd] := VAR[tmp] |
| subcc r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] - VAR[rs2]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(-, VAR[rs1], VAR[rs2])<br>C := carry(-, VAR[rs1], VAR[rs2])<br>VAR[rd] := VAR[tmp] |
| subcc r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] - IMM[sign_extend(imm13)]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(-, VAR[rs1], IMM[sign_extend(imm13)])<br>C := carry(-, VAR[rs1], IMM[sign_extend(imm13)])<br>VAR[rd] := VAR[tmp] |

FIG. 3.7 – Transformation des instructions arithmétiques avec modification des flags

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| addx r[rd], r[rs1], r[rs2] | VAR[rd] := VAR[rs1] + VAR[rs2]<br>VAR[rd] := VAR[rd] + GLOBAL[C] |
| addx r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[rd] := VAR[rd] + GLOBAL[C] |
| subx r[rd], r[rs1], r[rs2] | VAR[rd] := VAR[rs1] - VAR[rs2]<br>VAR[rd] := VAR[rd] - GLOBAL[C] |
| subx r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] - IMM[sign_extend(imm13)]<br>VAR[rd] := VAR[rd] - GLOBAL[C] |

FIG. 3.8 – Transformation des instructions arithmétiques avec utilisation du report

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| addxcc r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[tmp] := VAR[tmp] + GLOBAL[C]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(+, VAR[rs1], VAR[rs2], GLOBAL[C])<br>C := carry(+, VAR[rs1], VAR[rs2], GLOBAL[C])<br>VAR[rd] := VAR[tmp] |
| addxcc r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] + IMM[sign_extend(imm13)]<br>VAR[tmp] := VAR[tmp] + GLOBAL[C]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(+, VAR[rs1], IMM[sign_extend(imm13)], GLOBAL[C])<br>C := carry(+, VAR[rs1], IMM[sign_extend(imm13)], GLOBAL[C])<br>VAR[rd] := VAR[tmp] |
| subxcc r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] - VAR[rs2]<br>VAR[tmp] := VAR[tmp] - GLOBAL[C]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(-, VAR[rs1], VAR[rs2], GLOBAL[C])<br>C := carry(-, VAR[rs1], VAR[rs2], GLOBAL[C])<br>VAR[rd] := VAR[tmp] |
| subxcc r[rd], r[rs2], imm13 | VAR[rd] := VAR[rs1] - IMM[sign_extend(imm13)]<br>VAR[tmp] := VAR[tmp] - GLOBAL[C]<br>GLOBAL[N] := VAR[tmp] < 0<br>GLOBAL[Z] := VAR[tmp] = 0<br>GLOBAL[V] := overflow(-, VAR[rs1], IMM[sign_extend(imm13)], GLOBAL[C])<br>C := carry(-, VAR[rs1], IMM[sign_extend(imm13)], GLOBAL[C])<br>VAR[rd] := VAR[tmp] |

Fig. 3.9 – Transformation des instructions arithmétiques avec modification des flags et utilisation du report

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| umul r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[rs1] * VAR[rs2]<br>VAR[rd] := VAR[tmp][0 :31]<br>VAR[Y] := VAR[tmp][32 :63] |
| umul r[rd], r[rs2], imm13 | VAR[tmp] := VAR[rs1] * IMM[sign_extend(imm13)]<br>VAR[rd] := VAR[tmp][0 :31]<br>VAR[Y] := VAR[tmp][32 :63] |
| udiv r[rd], r[rs1], r[rs2] | VAR[tmp] := VAR[Y] * $2^{32}$ + VAR[rs1]<br>VAR[rd] := VAR[tmp] / VAR[rs2] |
| udiv r[rd], r[rs2], imm13 | VAR[tmp] := VAR[Y] * $2^{32}$ + VAR[rs1]<br>VAR[rd] := VAR[tmp] / IMM[sign_extend(imm13)] |
| rdy r[rd] | VAR[rd] := VAR[Y] |
| wry r[rd] | VAR[Y] := VAR[rd] |

FIG. 3.10 – Transformation des instructions de multiplication et division

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| or r[rd], r[rs1], r[rs2] | VAR[rd] := VAR[rs1] or VAR[rs2] |
| or r[rd], r[rs1], imm13 | VAR[rd] := VAR[rs1] or IMM[sign_extend(imm13)] |
| and r[rd], r[rs1], r[rs2] | VAR[rd] := VAR[rs1] and VAR[rs2] |
| and r[rd], r[rs1], imm13 | VAR[rd] := VAR[rs1] and IMM[sign_extend(imm13)] |

FIG. 3.11 – Transformation des instructions logiques

La traduction des instructions de transfert de contrôle est présentée dans la figure 3.13. Les instructions **call** et **ret** sont traduites immédiatement par leurs équivalents en langage intermédiaire. L'instruction **call** traduite utilise par défaut 8 paramètres, mais le nombre exact de paramètres est raffiné pendant l'analyse sémantique (voir section 4.3.3).
Les instructions de sauts inconditionnels sont traduites par un **If**, dont la condition est fixée à 1. Ces sauts seront éliminés lors des optimisations, comme nous le verrons dans la section 5.4.4. Toutes les instructions de type **Bicc** (fig. 3.14) (**ba**,**bn**,**bne**,...) sont traduites en plusieurs étapes : premièrement, la condition du saut est évaluée et stockée dans une variable temporaire. Ensuite, le saut est effectué si le contenu de cette variable est différent de 0.

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| save r[rs1], r[rs2], r[rd] | VAR[save0] := VAR[i0]<br>VAR[save1] := VAR[i1]<br>...<br>VAR[save7] := VAR[i7]<br>VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[i0] := VAR[o0]<br>VAR[i1] := VAR[o1]<br>...<br>VAR[i7] := VAR[o7]<br>VAR[rd] := VAR[tmp] |
| restore [rs1], [rs2], r[rd] | VAR[tmp] := VAR[rs1] + VAR[rs2]<br>VAR[o0] := VAR[i0]<br>VAR[o1] := VAR[i1]<br>...<br>VAR[o7] := VAR[i7]<br>VAR[i0] := VAR[save0]<br>VAR[i1] := VAR[save1]<br>...<br>VAR[i7] := VAR[save7]<br>VAR[rd] := tmp |

FIG. 3.12 – Transformation des instructions **save** et **restore**

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| call disp30 | VAR[o0] := call disp30, VAR[o0], VAR[o1],...,VAR[o7] |
| ret | ret VAR[i0] |
| jmpl r[rs1], r[rs2], r[rd] | VAR[tmp] := r[rs1] + r[rs2]<br>r[rd] := PC<br>if(1) goto VAR[tmp] |
| jmpl r[rs1], imm13, r[rd] | VAR[tmp] := r[rs1] + IMM[sign_extend(imm13)]<br>r[rd] := PC<br>if(1) goto VAR[tmp] |

FIG. 3.13 – Transformation des instructions de transfert de contrôle

| Instruction Sparc source | Traduction en forme intermédiaire |
|---|---|
| ba address | if(1) goto address |
| bn address | if(0) goto address |
| bne address | VAR[tmp] := not VAR[Z]<br>if(VAR[tmp]) goto address |
| be address | VAR[tmp] := VAR[Z]<br>if(VAR[tmp]) goto address |
| bg address | VAR[tmp] := VAR[N] xor VAR[V]<br>VAR[tmp] := VAR[Z] or VAR[tmp]<br>VAR[tmp] := not VAR[tmp]<br>if(VAR[tmp]) goto address |
| ble address | VAR[tmp] := VAR[N] xor VAR[V]<br>VAR[tmp] := VAR[Z] or VAR[tmp]<br>if(VAR[tmp]) goto address |
| bge address | VAR[tmp] := VAR[N] xor VAR[V]<br>VAR[tmp] := not VAR[tmp]<br>if(VAR[tmp]) goto address |
| bl address | VAR[tmp] := VAR[N] xor VAR[V]<br>if(VAR[tmp]) goto address |
| bgu address | VAR[tmp] := VAR[C] nor VAR[Z]<br>if(VAR[tmp]) goto address |
| bleu address | VAR[tmp] := VAR[C] or VAR[Z]<br>if(VAR[tmp]) goto address |
| bcc address | VAR[tmp] := not VAR[C]<br>if(VAR[tmp]) goto address |
| bcs address | VAR[tmp] := VAR[C]<br>if(VAR[tmp]) goto address |
| bpos address | VAR[tmp] := not VAR[N]<br>if(VAR[tmp]) goto address |
| bneg address | VAR[tmp] := VAR[N]<br>if(VAR[tmp]) goto address |
| bvc address | VAR[tmp] := not VAR[V]<br>if(VAR[tmp]) goto address |
| bvs address | VAR[tmp] := VAR[V]<br>if(VAR[tmp]) goto address |

FIG. 3.14 – Transformation des instructions **Bicc**

# Chapitre 4

# Analyse sémantique

Dans le chapitre précédent, nous avons exposé la forme intermédiaire choisie pour notre implémentation, ainsi que des règles de traduction sémantique de Sparc vers la forme intermédiaire. Cette traduction ne permet cependant pas d'obtenir toute l'information nécessaire pour pouvoir optimiser le programme : il nous manque des informations importantes sur l'interdépendance entre variables et instructions.
Le rôle des techniques que nous allons décrire dans ce chapitre est de récupérer un supplément d'informations sur le programme analysé afin de rendre possible certaines optimisations, comme l'élimination des codes morts, la propagation des constantes, ...

Le premier outil est le Basic Block, qui permet de découper le programme en sections indépendantes prêtes à être optimisées. Le second outil est l'analyse de flot de données, qui permet d'obtenir les interdépendances entre variables à tout moment dans le programme.

## 4.1 Basic Blocks

### 4.1.1 Définition

Le Basic Block est une notion très connue et utilisée dans le domaine de la compilation. Cette notion est définie dans le Dragon Book [12] et dans beaucoup d'autres références en compilation.
Un Basic Block est un ensemble d'instructions consécutives, qui respectent les conditions suivantes :

- Seule la dernière instruction peut être une instruction de transfert de contrôle,
- Seule la première instruction peut avoir plus d'une instruction comme prédécesseur,
- Aucune instruction ne peut appartenir à plusieurs Basic Blocks.

Tout programme accepte une découpe de son ensemble d'instruction en Basic Blocks. La taille d'un Basic Block est le nombre d'instructions qu'il contient. Un Basic Block est sous forme canonique lorsqu'il n'existe pas de Basic Block

contenant les mêmes instructions et de taille supérieure.

L'utilisation de Basic Blocks permet de simplifier les opérations sur les instructions qui en font partie, car l'ordre de ces instructions est linéaire. Un Basic Block est aussi un outil utile pour l'analyse visuelle du résultat d'un compilateur, car les relations entre blocs résument efficacement le flot de contrôle du programme, et permettent d'obtenir un résultat visuellement agréable.

### 4.1.2 Algorithme

Quelques prédicats doivent d'abord être définis avant de pouvoir décrire un algorithme :
$Branche(i) \equiv \{j \in instructions \mid i$ est une instruction *if* ou *call*, et $j$ est l'addresse de branchement.$\}$

$Successeur(i) \equiv \{j \in instructions \mid i$ est une instruction et $j$ est l'instruction naturellement exécutée après i, si aucun transfert de contrôle n'a lieu.$\}$

$Predecesseurs(i) \equiv \{j \in instructions \mid i \in Successeur(j) \vee i \in Branche(j)\}$
Les prédécesseurs d'une instruction $i$ sont les instructions dont le successeur ou la branche sont $i$.

$nPredecesseurs(i) \equiv$ nombre d'instructions $j \in\ Predecesseurs(i)$

$NewBlock(i) \equiv (nPredecesseurs(i) = 0) \vee (nPredecesseurs(i) > 1) \vee [nPredecesseur(i) = 1 \wedge \forall j \in Predecesseurs(i) : EndBlock(j)]$
Une instruction $i$ délimite un nouveau Basic Block si elle n'a pas de prédécesseur (instruction racine), si elle a plusieurs prédécesseurs, ou bien si son unique prédécesseur est un délimiteur de fin de bloc.

$EndBlock(i) \equiv (i$ a une branche $) \vee (Successeur(i) = \emptyset) \vee (\exists j \in Successeur(i) : nPredecesseurs(j) > 1)$
Une instruction $i$ délimite une fin de bloc si elle possède une branche ou si elle ne possède pas de successeur (i.e. fin de procédure), ou si son successeur possède plusieurs prédécesseurs.

L'ensemble des instructions appartenant au même Basic Block, commençant par une instruction $i$, est défini comme ceci :
$BB_i = \{j \in instructions \mid NewBlock(i)\ \wedge [j = i \vee j = Successeur(i) \vee \exists k \in BB_i \mid (j = Successeur(k) \wedge \neg EndBlock(k))]\}$

L'algorithme nécessaire pour la division d'une procédure en Basic Blocks est donc assez simple. Les opérations sont les suivantes :

1. Appliquer les prédicats NewBlock et EndBlock sur toutes les instructions de la procédure,
2. Chaque instruction pour laquelle NewBlock($i$) est vrai commence un nouveau Basic Block,

3. Chacun de ces Basic Block peut être rempli en ajoutant les instructions successives à l'instruction $i$, jusqu'à ce qu'on arrive sur une instruction $j$ pour laquelle EndBlock($j$) est vrai.

### 4.1.3 Liens entre Basic Blocks

Les Basic Blocks schématisant le graphe de contrôle de flot, ils héritent des propriétés des instructions : chaque Basic Block possède des prédicats Successeur(), Branche(), Prédécesseurs(), qui sont définis de la même manière que pour les instructions.

## 4.2 Analyse de flots de données

Dans la section 2.3, nous avons introduit la notion de graphe de contrôle de flot. Ce graphe dirigé nous renseigne sur les différents chemins d'exécution possibles dans un programme, et sur certaines relations entre instructions (successeur, ancêtre, prédécesseur,...). Cependant, ce graphe ne donne absolument aucune information sur les états possibles lors de l'exécution du programme : on ignore quelles sont les valeurs possibles pour les variables, quelles sont les relations entre ces variables, les interdépendances et l'utilisation de ces variables.

L'analyse du flot de données est un outil permettant de répondre à ces questions.

### 4.2.1 Définitions

**Dépendance entre instructions** Une instruction $i_1$ dépend d'une instruction $i_2$ lorsque $i_1$ dépend directement[1] du résultat de $i_2$ pour pouvoir opérer.

**Dépendance entre instruction et variable** Une variable $v$ dépend d'une instruction $i$ lorsque le contenu de la variable $v$ dépend du résultat de cette instruction $i$. Cette relation n'a de sens que dans un contexte, c'est-à-dire l'endroit sur le graphe de contrôle de flot où on l'évalue.

**Dépendances d'une instruction** Ensemble de toutes les instructions desquelles dépend directement une instruction $i$.

**Dépendances d'une variable** Ensemble de toutes les instructions desquelles dépend directement le contenu de la variable $v$. Cet ensemble peut être différent selon l'endroit où on l'évalue sur le graphe de contrôle de flot.

**Vie d'une variable** Espace ou ensemble d'instruction dans le graphe de contrôle de flot dans lequel une variable $v$ assignée à l'instruction $i$ n'est plus modifiée, et dont la valeur peut être considérée constante. Cet espace est différent en fonction de l'instruction de départ $i$. Une variable $v$ peut donc avoir plusieurs vies dans un même programme.

[1] À opposer à une dépendance indirecte, transitive, dans laquelle toute instruction ayant contribué, au cours du programme, à $i_2$, serait incluse.

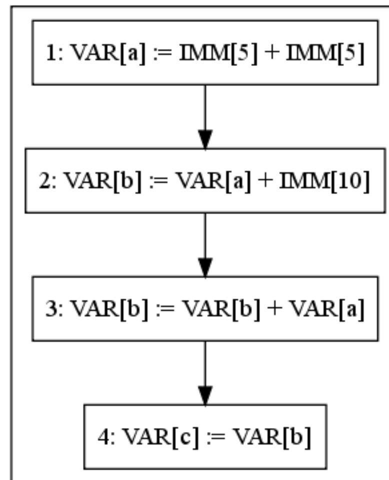


FIG. 4.1 – Exemple de dépendances simple.

La relation de dépendance directe n'est pas transitive : les dépendances d'une variable $v$ à l'instruction $i$ sont les derniers ancêtres de $i$ qui ont affecté la variable $v$.

### 4.2.2 Dépendance entre instructions

#### Exemples

Dans l'exemple de la figure 4.1, $i_1$ n'a pas de dépendance : elle ne dépend d'aucune instruction. $i_2$ dépend directement du résultat de $i_1$ : il y a donc une dépendance entre $i_2$ et $i_1$. $i3$ dépend de $i_2$ et de $i_1$. $i_4$ dépend de $i_3$, car la dernière affectation de $b$ se trouve dans $i_3$.
La variable $a$ n'a aucune dépendance en 1. En 2, elle dépend de $i_1$. En 3, elle dépend toujours de $i_1$. La variable $b$ n'a aucune dépendance en 1 et en 2. En 3, elle dépend de $i_2$. En 4, elle dépend de $i_3$.

Dans l'exemple de la figure 4.2, la relation de dépendance est plus compliquée à cause de la boucle. Les dépendances de $i_2$ sont $i_1$ et $i_2$, car $i_2$ est un ancêtre possible de lui-même. Il est donc nécessaire, lors de la recherche de dépendance, d'analyser intelligemment tous les chemins possibles pour éviter de manquer une dépendance.

L'exemple de la figure 4.3 est une implémentation en langage intermédiaire de la fonction récursive Fibonacci. Les flèches bleues et rouges montrent les relations de dépendance entre variables et instructions. Deux types de relations sont présentes dans cet exemple :

1. La flèche rouge indique que la variable VAR[i0], définie au début du programme, est utilisée ensuite 3 fois. À chaque utilisation de la variable VAR[i0], il n'y a qu'une seule dépendance, la définition unique de VAR[i0] au début de la procédure.

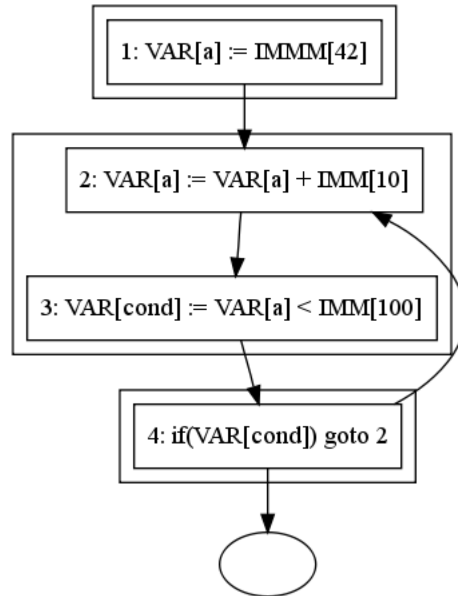


FIG. 4.2 – Exemple de dépendances en présence de boucle.

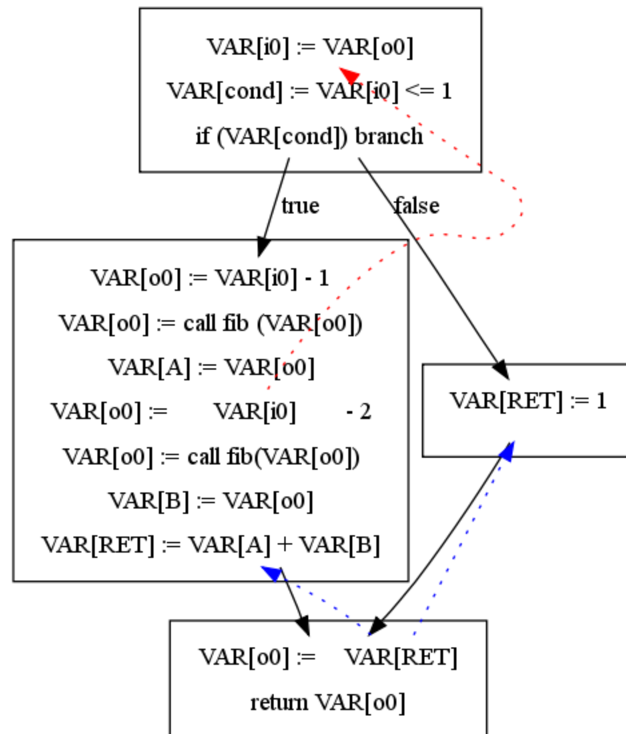


FIG. 4.3 – Exemple de découpe en Basic Blocks et d'analyse de dépendances de données sur 2 variables. En rouge, la variable analysée est VAR[i0], et en bleu VAR[RET].

2. La flèche bleue indique que la variable VAR[RET] est utilisée une seule fois, mais définie à deux endroits différents dans la procédure. Cela signifie que l'unique utilisation de VAR[RET] dépend de deux instructions.

**Algorithmes**

Nous décrivons plusieurs algorithmes pour calculer les dépendances entre variables et instructions. L'algorithme Modifie($i$,$v$) (alg. 1) retourne **true** si l'instruction $i$ modifie la variable $v$. L'algorithme VariableDependantes($i$) (alg. 2 retourne la liste des variables dont dépend l'instruction $i$.
L'algorithme Dependances ($v$,$i$) (alg. 3) calcule l'ensemble des instructions dont dépend l'instruction $i$ par l'intermédiaire de la variable $v$. Cet algorithme visite les instructions du graphe de contrôle de flot à l'envers, en partant de l'instruction $i$, jusqu'à ce qu'il arrive sur une instruction déjà visitée (marquée) ou sur une instruction qui modifie la variable $v$. L'algorithme retourne l'ensemble de ces instructions qui modifient $v$.
L'algorithme existeDependances ($i$) (alg. 4) détermine si l'exécution de l'instruction $i$ est nécessaire pour l'exécution d'autres instructions dans le programme. Cet algorithme visite le graphe de contrôle de flot depuis l'instruction $i$, jusqu'à ce qu'une instruction qui utilise la variable $v$ modifiée par $i$ soit trouvée. L'analyse ne continue pas dans les branches d'exécution qui modifient $v$, car elles ne dépendent plus de l'instruction $i$. Cet algorithme sert entres autres à repérer les instructions mortes, qui modifient des variables plus jamais utilisées.

Tous ces algorithmes sont écrits pour ne fonctionner que sur des variables locales. Les variables globables, pouvant être lues et réécrites dans les appels de procédures, ne peuvent pas être optimisées de la même manière. Certaines analyses comme nous le verront au point 4.3.3 permettent cependant d'améliorer ces algorithmes. Entre-temps, la condition pour pouvoir appliquer ces algorithmes aux variables globales est de considérer que les instructions call dépendent de toutes les variables globales et peuvent toutes les modifier. L'hypothèse est aussi faite que le programme n'est pas multi-thread.

**Algorithme 1** Calcul de Modifie($i$,$v$)

**Entrées:** instruction $i$ et une variable $v$
**Sorties:** retourne **true** si l'instruction $i$ modifie la variable $v$

```
si assignation(i) alors
  si destination(i = v) alors
    retourner true
  finsi
finsi
retourner true
```

### 4.2.3 Vie d'une variable

La notion de vie de variables est indispensable pour résoudre certains problèmes, comme la propagation des copies, que nous verrons au chapitre suivant.

---

**Algorithme 2** Calcul de VariablesDependantes($i$)

---

**Entrées:** $i$ instruction
**Sorties:** Renvoie une liste de variables dont dépend l'exécution de $i$

```
si i Assignation alors
  si i.expr() Call alors
    retourner  Liste de paramètres Call
  finsi
  si i.expr() Expression alors
    retourner  Liste des variables dans l'expression
  finsi
finsi
si i Ret alors
  retourner  Variable de retour
finsi
si i If alors
  retourner  Variable de condition
finsi
retourner  ∅
```

---

**Algorithme 3** Calcul de Dependances ($v$,$i$)

---

**Entrées:** $v$ variable, référencée dans l'instruction $i$
**Sorties:** $dSet$ contient les instructions dont dépend ($v$,$i$)

```
iSet ⇐ Predecesseurs(i)
dSet ⇐ ∅
tantque iSet ≠ ∅ faire
  j ⇐ élément de iSet
  iSet ⇐ iSet \ {j}
  si Marked(j) alors
    continuer
  sinon
    Mark(j)
    si Modifie(v,j) alors
      dSet ⇐ dSet + {j}
    sinon
      iSet ⇐ iSet + Predecesseurs(j)
    finsi
  finsi
fin tantque
retourner  dSet
```

---

**Algorithme 4** Calcul de existeDependances ($i$)

**Entrées:** $i$ une instructions
**Sorties:** renvoie **true** si cette instruction a des dépendances inverses.

```
si i est un Call, Return, If alors
  retourner true
finsi
v ⇐ i.dest
iSet ⇐ Successeurs(i)
tantque iSet ≠ ∅ faire
  j ⇐ élément de iSet
  iSet ⇐ iSet \ {j}
  si Marked(j) alors
    continuer
  sinon
    Mark(j)
    si v ∈ VariablesDependances(j) alors
      retourner true
    finsi
    si Modifie(j, v alors
      continuer
    sinon
      iSet ⇐ iSet + Successeurs(j)
    finsi
  finsi
fin tantque
retourner false
```

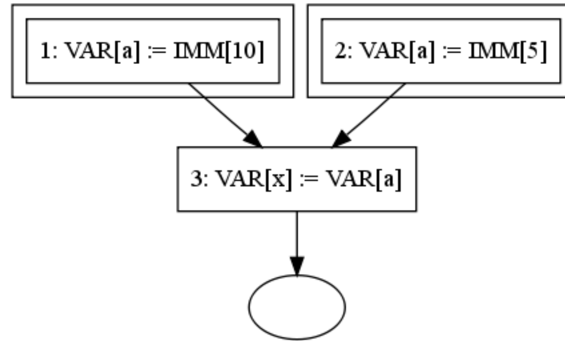


FIG. 4.4 – Exemple de vie de la variable VAR[a]

Les algorithmes et définitions qui suivent permettent de répondre à cette question : l'instruction $i_{cible}$ appartient-elle à l'espace de vie de la variable $v$ définie à l'instruction $i_{assign}$ ?

**Définition** Un chemin d'exécution $\sigma$ est une succession finie ou non d'étapes $\sigma_i$, lesquelles correspondent chacune à une instruction de la procédure. La suite des étapes $\sigma_1, \sigma_2, ..., \sigma_n$ doit suivre l'ordre d'exécution des instructions dans la procédure, en suivant les transitions autorisées, c'est-à-dire les branchements possibles. La propriété $\forall \sigma, i \in \mathbb{N} : \sigma_{i+1} = Branche(\sigma_i) \vee \sigma_{i+1} = Successeur(\sigma_i) \vee \sigma_{i+1} = \varnothing$ doit être vérifiée.

**Définition** L'instruction $i_{cible} \in live(i_{assign}, v)$ ssi il n'existe pas de chemin d'exécution entre $i_{assign}$ et $i_{cible}$ tel que la variable $v$ soit assignée.

De manière un peu plus formelle, $i_{cible} \in live(i_{assign}, v)$ ssi $\nexists \sigma$ :
$\exists i, j, k \in \mathbb{N} : i < j \leq k \wedge$
$\sigma_i = i_{assign} \wedge \sigma_k = i_{cible} \wedge$ Modifie$(\sigma_j, v)$
où $\sigma$ est une trace ou chemin d'exécution dans la procédure, et $\sigma_i$ l'instruction exécutée au temps $i$.

### Exemples

L'exemple de la figure 4.4 présente un programme dans lequel deux Basic Blocks du CFG se rejoignent. Dans chacun de ces deux Basic Blocks, la variable VAR[a] est assignée. La variable VAR[a] définie en $i_1$ est vivante en $i_3$, car il n'existe aucun chemin allant de $i_1$ à $i_3$ passant par $i_2$ (seule autre instruction modifiant VAR[a]).

### Algorithmes

L'algorithme 5 calcule l'appartenance d'une instruction $i_{cible}$ à l'espace de vie d'une variable $v$ assignée en $i_{assign}$. Cet algorithme fonctionne en parcourant le CFG à l'envers depuis l'instruction $i_{cible}$. Les instructions ne faisant pas partie de chemins d'exécution entre $i_{assign}$ et $i_{cible}$ sont ignorées, à l'aide de l'algorithme 6. Ce dernier fonctionne lui aussi en parcourant le CFG en sens inverse.

**Algorithme 5** Calcul de $i_{cible} \in \text{live}(i_{assign}, v)$

**Entrées:** $v$ variable assignée dans l'instruction $i_{assign}$
**Entrées:** $i_{cible}$ instruction dont il faut déterminer l'appartenance à l'espace de vie.
**Sorties:** retourne **true** si $i_{cible}$ appartient à l'espace de vie, sinon **false**

```
iSet ⇐ {i_cible}
tantque iSet ≠ ∅ faire
  j ⇐ élément de iSet
  iSet ⇐ iSet \ {j}
  si Marked(j) alors
    continuer
  sinon
    Mark(j)
  finsi
  si j = i_assign alors
    continuer
  finsi
  si ¬ existPath(i_assign,j) alors
    continuer
  finsi
  si Modifie(v,j) alors
    retourner false
  finsi
  iSet ⇐ iSet + Predecesseurs(j)
fin tantque
retourner true
```

**Algorithme 6** Calcul de existPath($from$, $to$)

**Entrées:** $from$ et $to$ deux instructions
**Sorties:** retourne **true** si il existe un chemin d'execution entre $from$ et $to$

```
iSet ⇐ {to}
tantque iSet ≠ ∅ faire
  j ⇐ élément de iSet
  iSet ⇐ iSet \ {j}
  si Marked(j) alors
    continuer
  sinon
    Mark(j)
  finsi
  si j = from alors
    retourner true
  finsi
  iSet ⇐ iSet + Predecesseurs(j)
fin tantque
retourner false
```

## 4.3 Autres informations sémantiques

Dans la plupart des langages de programmation, le compilateur reçoit de la part du programmeur énormément d'informations sur la manière dont doit être compilé le code, et sur la manière d'interpréter le programme. Par exemple, les langages fortement typés nécessitent que toute variable utilisée dans le programme soit préalablement déclarée, et qu'un type de donnée y soit associé. Certains langages possèdent plusieurs constructions pour effectuer des opérations avec des variables de types différents, et ont des règles de transtypage implicites et explicites. La plupart des langages ont des constructions spécifiques pour effectuer des boucles, pour des déclarations de fonctions, etc.

Dans un programme compilé, une grande partie de ces informations disparait. Il n'y a pas de déclaration des variables pour indiquer de quel type elles sont. Les procédures ne sont pas déclarées, et dans la plupart des cas, aucune signature ne permet de déterminer combien d'arguments elles utilisent, ni le type de ces arguments.

Cependant, ce n'est pas parce que ces informations ne sont pas présentes de manière explicite qu'elles sont totalement absentes. Plusieurs techniques permettent de récupérer des informations sémantiques afin de mieux comprendre le programme et de générer du code plus efficace.

### 4.3.1 Analyse de type et pointeurs

Deux catégories de langages de programmation existent : les langages aux variables typées et les langages aux variables non typés. Tout programme dans un langage aux variables typées doit contenir une description des variables référencées, avant usage, afin de pouvoir contrôler la validité des opérations, ainsi que la manière d'interpréter et de traduire ces opérations. De tels langages sont C, C++, Java, Fortran, Ada, Pascal, ...
Les langages aux variables non typées sont moins restrictifs. Ces langages ne nécessitent pas la présence de typage dans la déclaration des variables. Certains n'imposent pas non plus la déclaration d'une variable avant usage. La plupart de ces langages sont historiquement interprétés (Basic, Perl, Python, Smalltalk, ...) mais pour certains, il existe des compilateurs.
À l'exécution de programmes dans ces langages aux variables non typées, lorsqu'une variable est assignée, une méta-donnée est jointe à la valeur allant dans la variable, afin que la machine virtuelle ou interpréteur puisse déterminer comment effectuer les opérations et calculs sur les valeurs de cette variable. L'analyse de types est donc dans ces cas-là principalement un problème Runtime (à l'exécution).
Au contraire, lors de la traduction d'un programme aux variables typées, le compilateur possède suffisamment d'informations à la compilation pour déterminer comment le programme doit être traduit, car il connait le type de toutes les variables. Lors de la comparaison de deux entiers, il choisira la comparaison signée si par exemple l'un des deux arguments est un entier signé, et produira

du code correspondant à la comparaison signée. Si une mauvaise utilisation des variables est détectée (par exemple la division d'un pointeur, ce qui n'a pas de sens), le compilateur refusera de compiler ce code.

Un programme assembleur est à la croisée de ces deux types de langages : les langages d'assemblage sont le plus souvent non typés. Avant l'utilisation d'une variable, aucune déclaration n'indique de quel type elle est. Lors de l'assignation de la variable, aucune méta-donnée n'y est ajoutée pour informer le processeur du type de la variable. Aucun opérateur de transtypage n'est utilisé pour transformer le type d'une variable, si la représentation binaire de la variable est la même dans les deux types (par exemple, le nombre 0 a la même représentation binaire dans un entier signé 32 bits et un entier non signé 32 bits).
Le type d'une variable ne se révèle que lorsque sa valeur est utilisée. C'est donc en analysant la manière dont sont utilisées les variables que nous déterminerons de quels types elles sont.

#### Types de données

Les langages de programmation typés possèdent un nombre conséquent de types. Un langage comme le C définit des types primitifs : les entiers de 8,16,32, 64 bits, signés et non signés, et les pointeurs. D'autres types composés existent, comme les vecteurs et les structures. D'autres langages rajoutent à cela des objets, structures comportant des pointeurs de fonctions virtuelles.
Dans notre application, la transformation de binaires, les seuls types de données réellement pertinents sont les entiers et les pointeurs, sans distinction entre les entiers signés et non signés, ainsi que les pointeurs sur des types différents.

#### Distinction entre pointeur et entier

Dans un programme assembleur, les pointeurs et entiers ont le même format et sont représentés de la même manière. L'assignation d'une valeur dans un registre ne permet pas de déterminer si cette valeur est une valeur numérique arbitraire ou un pointeur sur des données existantes en mémoire. Il existe cependant plusieurs manières de determiner si une variable contient un pointeur ou un entier :

- Si une variable est utilisée dans un opérateur MEM[$x$], alors elle contient un pointeur
- Si une variable est utilisée comme paramètre $p_i$ d'une procédure dont l'argument $p_i$ est un pointeur, alors cette variable est un pointeur.
- Si une variable est utilisée comme paramètre à une instruction **Call** ou **If** en tant qu'adresse de destination, alors cette variable est un pointeur.

### 4.3.2 Optimisation des pointeurs

Les pointeurs peuvent être utilisés dans le programme final, sous leur forme d'origine, c'est à dire des entiers de 32 bits. Les routines runtime, écrites en Java, sont responsables de l'insertion de données en mémoire et la récupération des

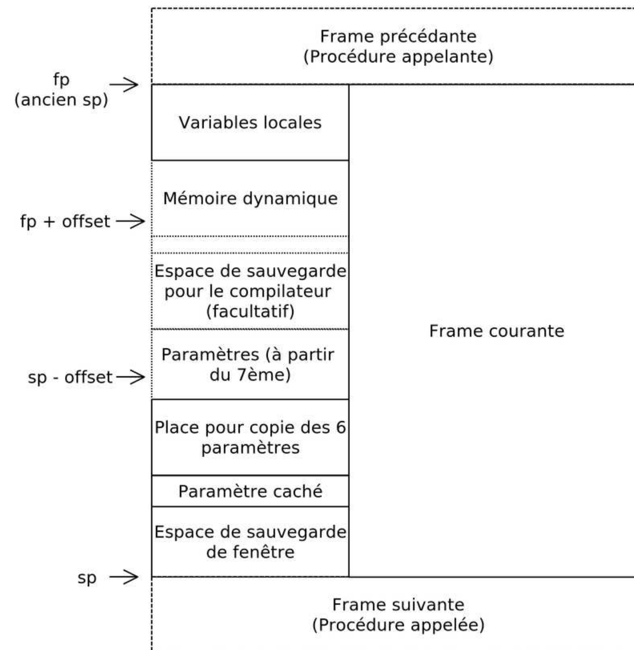


FIG. 4.5 – Organisation de la pile dans l'architecture Sparc. Inspirée du manuel de l'architecture Sparc [18], figure D-2.

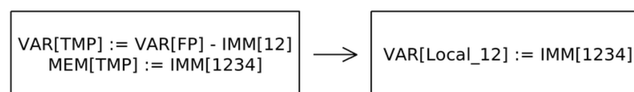


FIG. 4.6 – Optimisation d'une utilisation de pointeur de variable locale.

données, c'est à dire de l'implémentation de l'opérateur MEM[$x$]. Cependant, cette manière de procéder n'est pas optimale. En effet, l'utilisation de l'opérateur MEM[$x$] implique un couteux appel de procédure dans le programme traduit. Dans certains cas, il peut être plus efficace de considérer un pointeur non pas comme un descripteur sur une variable, mais une variable. Par exemple, les variables locales d'un programme Sparc sont soit contenues dans les registres locaux, ou dans la pile. Les éléments de la pile sont référencés en fonction des deux pointeurs de piles : le registre **sp** (stack pointer) et le registre **fp** (frame pointer). Le contenu de ces deux registres est transmis automatiquement de procédure en procédure, à l'aide des instructions **save** et **restore**. Ils apparaissent donc dans la forme intermédiaire sous la forme de variable, et se retrouvent dans les paramètres de procédure en forme intermédiaire. Une description de l'organisation de la pile dans un programme Sparc se trouve à la figure 4.5.

L'optimisation consiste donc à transformer toutes les références à ces variables locales (à l'aide d'un pointeur) et leur déréférencement (opérateur MEM[$x$]) en variables normales, sous plusieurs conditions :

- La variable référencée doit être une variable locale. La variable ne peut pas être initialisée dans une autre procédure. Autrement dit, les variables éligibles pour cette optimisation sont, sur la figure 4.5, les variables locales, mémoire dynamique, et l'espace de sauvegarde des fenêtres. Les autres espaces mémoires sont susceptibles d'avoir été initialisés dans une autre procédure, et de contenir des données.
- Le pointeur pointant sur cette variable doit être constant. Si la valeur d'un pointeur est susceptible de changer et d'être utilisée pour lire ou écrire dans d'autres variables locales, ces variables ne peuvent pas être optimisées.
- Le pointeur ne peut pas être paramètre d'une procédure appelée. Si le pointeur est "donné" à une autre procédure, le contenu pointé par celui-ci est susceptible de changer. Pour cette raison, il faut éviter qu'une variable puisse être présente en même temps sur la pile et dans les variables locales de la procédure (définies symboliquement), à moins qu'une analyse inter-procédure prouve que c'est sans danger pour la cohérence des variables (voir point 4.3.3).

Dans l'exemple de la figure 4.6, à la variable pointée par **fp**-12 (troisième variable locale assignée dans la pile) est assignée la constante 1234. L'optimisation consiste à créer une nouvelle variable symbolique, et à lui assigner cette constante. Les autres apparitions de ce même pointeur sont évidemment transformées elles aussi.

Les variables globales, dont l'adresse en mémoire est constante, peuvent aussi profiter de cette optimisation. Une table des variables globales doit donc être créée, contenant la liste des variables globales trouvées, ainsi que leur adresse virtuelle et leur nom symbolique (utilisable dans toutes les procédures). Cependant, ceci n'est possible que si ces variables ne sont pas susceptibles d'être réécrites à partir d'un autre pointeur.

### 4.3.3 Analyse inter-procédures

Jusqu'à présent, nous avons considéré que chaque procédure transmettait six paramètres, en plus des deux pointeurs de piles (*fp* et *sp*), même si ces paramètres ne sont pas utilisés, car l'architecture Sparc ne nécessite pas de préciser le nombre exact d'arguments pour chaque procédure (c'est au compilateur de s'assurer qu'aucun appel invalide n'est généré). En ce qui concerne les appels de procédures externes, nous considérons que ces procédures ont été implémentées dans la librairie standard du back-end, et que pour chaque procédure y sont associés le nombre de paramètres attendus ainsi que leur type (entier, pointeur).

Le rôle de l'analyse inter-procédure est de faire remonter des informations sur le nombre de paramètres et leurs types tout le long du graphique d'appel de procédures, afin de décorer chaque déclaration de procédure de ces informations. La figure 4.7 représente un exemple de graphique d'appel de procédures décoré. Il existe deux manières de décorer ces déclarations de procédures :

- Commencer par analyser la procédure racine, et ensuite redescendre vers les procédures appelées. Les informations pouvant être apportées sont les types des arguments des procédures appelées. Par exemple, si une variable de type pointeur est passée en paramètre $i$ à une procédure, la signature ou description de cette procédure peut être enrichie du type de l'argument $i$.
- Commencer par analyser les feuilles, et redescendre les informations jusqu'à la racine. Les informations pouvant être obtenues sont le nombre d'arguments réellement utilisés (Il suffit de scanner la procédure complète pour voir si ces variables sont utilisées), et parfois leur type (utilisation d'une variable dans l'opérateur MEM[$x$], passage de variable vers une procédure dont le type des arguments sont connus, ...).

Cette dernière méthode peut évidemment s'aider des informations statiques connues (par exemple les déclarations des librairies C). Les deux méthodes sont complémentaires, et peuvent être appliquées jusqu'à ce que plus aucune nouvelle information n'est découverte. D'autres analyses peuvent être faites au sujet de la communication entre procédures, par exemple l'analyse de l'utilisation des pointeurs donnés en paramètres (les valeurs pointées sont-elles modifiées ? Le pointeur est-il utilisé comme base dans un vecteur ?). Ces analyses sont faites par certains compilateurs afin d'augmenter les performances (par exemple les passages de paramètres par référence dans un compilateur C++, où il n'est pas toujours indispensable de communiquer un pointeur). Ces analyses sont beaucoup plus compliquées dans un transformateur de binaire en raison du faible nombre de renseignements disponibles sur la sémantique du programme.

#### Variables globales

L'analyse inter-procédures peut aussi recenser les variables globales lues et modifiées dans chaque procédure. Le but de ce recensement est de déterminer si une instruction d'écriture dans une variable globale avant l'exécution d'une pro-

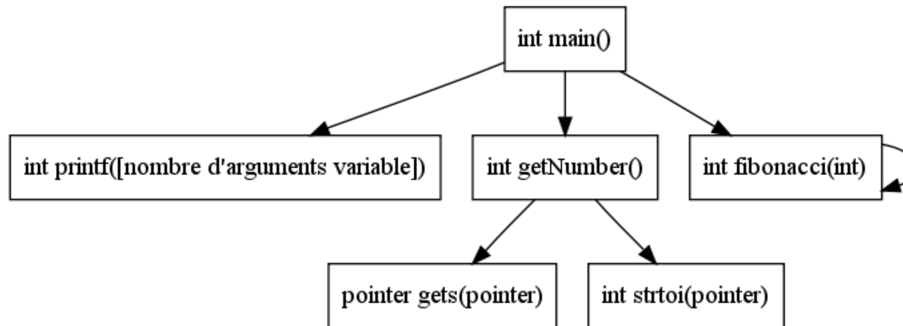


FIG. 4.7 – Exemple de graphique d'appel de procédures d'un programme lisant un nombre à la console et affichant sa valeur de fibonacci.

cédure peut être éliminée ou déplacée. On aurait donc pour chaque procédure, associée à chaque variable globale, une série de propriétés :

**Est lue** La variable globale est lue dans la procédure.

**Peut être écrite** La variable globale pourrait être écrasée par une nouvelle valeur, mais pas par tous les chemins possibles.

**Est écrite** Tous les chemins d'exécution de la procédure écrivent une nouvelle valeur.

**Inconnu** On ignore si la procédure réécrit ou lit la variable, par exemple parce que l'analyseur ne peut pas analyser toutes les sous-routines de la procédure (appel de fonction virtuelle, voir section 4.4.2).

Le but de cette analyse est de permettre le même type d'optimisations sur les variables globales que sur les variables locales, comme nous le verrons dans le chapitre suivant. Une amélioration des algorithmes de la section 4.2.2 permettrait de prendre en compte les informations obtenues par l'analyse inter-procédures, et de considérer les appels de procédure comme des dépendances supplémentaires sur les variables globales (pour éviter de supprimer par mégarde des modifications de variables globales qui sont effectivement utilisées dans d'autres procédures).

## 4.4 Détection des structures particulières

Jusqu'à présent, nous n'avons pas étudié une certaine catégorie d'instructions : les sauts indexés et les appels de fonctions virtuelles. Cette catégorie d'instructions se distingue des autres, car il n'est pas toujours possible de prévoir quelles seront les prochaines instructions exécutées. Les sauts indexés sont généralement produits par des structures de langages, telles que la construction "switch" en C. Les appels de fonctions virtuelles se retrouvent très souvent en C++ (c'est d'ailleurs une des caractéristiques de tous les langages objets), et parfois en C (dans lequel elles sont plus souvent nommées "pointeurs de fonctions").

### 4.4.1 Analyse des sauts indexés

L'analyse des sauts indexés, ou récupération des tables de sauts, est un problème compliqué. Les compilateurs traduisant des constructions du type "switch(x)" produisent du code ressemblant au code suivant :

```
VAR[TMP] := VAR[X] < IMM[0]
  if(VAR[TMP]) goto end
VAR[TMP] := VAR[X] > IMM[10]
  if(VAR[TMP]) goto end
VAR[ADDR] := IMM[jump_table] + VAR[X]
VAR[JMP] := MEM[ADDR]
  if(1) goto VAR[JMP]
label1:
  ...
label10:
  ...
end:
  ...
```

Premièrement, la valeur de X est comparée aux différentes limites pour éviter de faire des lectures en dehors de la table de sauts. Ensuite, un pointeur est extrait de la table à l'offset X. L'exécution continue à ce pointeur.
La méthode décrite par les créateurs d'UQBT dans leur rapport [15] est la suivante :

1. Dès qu'un saut indexé est découvert, la partie pertinente pour l'analyse est découpée et isolée.
2. La propagation des copies (voir 5.3.1) est effectuée sur le code, afin d'obtenir la version la plus simplifiée possible.
3. Un algorithme de pattern-matching est appliqué sur le résultat, afin de comparer avec les différentes formes de sauts indexés connus (4 types ont été documentés).
4. À l'aide des nouvelles informations obtenues par pattern matching, les pointeurs sont extraits de la table de sauts et sont ensuite utilisés pour construire une nouvelle table, cette fois-ci contenant des pointeurs vers des étiquettes du programme en forme intermédiaire.

Ces techniques peuvent s'appliquer dans notre cas. Il est probable qu'après avoir reconnu les pointeurs de la table de saut, le transformateur doive appliquer des techniques de reprise arrière afin de compléter le graphe de contrôle de flot. Les techniques de reprise arrière sont des techniques courantes dans les compilateurs, visant à réappliquer un algorithme sur le programme après avoir appris une nouvelle information au sujet du programme. L'instruction de saut indexé en langage intermédiaire est ensuite décorée de ses différentes destinations possibles, afin de permettre aux différents algorithmes de calcul de dépendances de fonctionner.

### 4.4.2 Pointeurs de procédures et méthodes virtuelles

Les pointeurs de procédure sont des pointeurs qui ne pointent pas vers des données, mais vers du code exécutable, et plus précisément des procédures. En forme intermédiaire, nous aurions donc l'instruction $VAR[out] := call\ VAR[f](op_1, ..., op_n)$.
La première stratégie possible pour la transformation d'une telle instruction est la génération d'un code spécial, qui fait une recherche dans une table des procédures disponible en fonction de l'adresse Sparc de la procédure virtuelle. Ce code spécial, programmé en Java, trouverait l'adresse de la procédure dans la machine virtuelle, et exécuterait cette procédure.

Une autre possibilité complémentaire serait d'effectuer une analyse des pointeurs de procédures pouvant être appelées. La difficulté de cette analyse réside dans le fait que la liste des fonctions susceptibles d'être appelées depuis cette instruction n'est pas disponible immédiatement : il se peut que la valeur du pointeur de fonction se situe dans une autre procédure, ou même dans le segment de données.
Dans le cas du C++, le pointeur de fonction d'une méthode virtuelle se trouve dans un espace mémoire dynamique, dans lequel l'objet est alloué. Tenter de repérer les méthodes virtuelles associées aux différentes classes d'objets est un travail très compliqué, nécessitant d'effectuer des analyses de types très poussées pour pouvoir déterminer l'arborescence complète du diagramme de classes. Ensuite, il est nécessaire d'associer pour chaque type d'objet les méthodes virtuelles pouvant être appliquées à cet objet (par exemple en analysant le constructeur d'objet, qui initialise les pointeurs de méthodes virtuelles au sein de l'objet). Le but d'une telle analyse serait de remplacer par exemple l'initialisation du pointeur avec l'adresse de la procédure dans la machine virtuelle Java, afin d'épargner une recherche dans la table des procédures à l'exécution. Cependant, une telle méthode ne pourrait être complète, et une recherche dans la table des procédures devrait toujours être possible à l'exécution (il se peut que le pointeur de procédure ne soit pas du tout disponible pendant la transformation).

Une autre stratégie possible est inspirée de certains compilateurs assistés par profiling (ou dynamic optimizing compilers). Ces compilateurs nécessitent deux étapes pour produire du code. Lors de la première étape, un programme de test est généré. Ce programme n'est généralement pas efficace, mais possède des routines destinées à tracer l'exécution du programme, les chemins souvent empruntés, ainsi que les données les plus utilisées. Ensuite, le compilateur peut s'aider de ces informations pour produire du code plus efficace (par exemple en plaçant les zones mémoires les plus souvent utilisées dans un minimum de pages mémoires, afin de minimiser le temps de chargement et le swapping).
Dans le cas actuel, le transformateur génèrerait des informations à propos des procédures les plus souvent demandées pour chaque instruction d'appel de procédure virtuelle. Le transformateur s'aiderait de ces informations pour créer une table par appel de procédure virtuelle, de taille plus modeste que la table globale des procédures. Cette table serait utilisée lors de chaque appel, avant d'utiliser la table globale, si la procédure n'est pas trouvée dans la première table.

# Chapitre 5

# Optimisation

## 5.1 Nécessité de l'optimisation

L'objectif principal de la transformation de binaires est la possibilité d'exécuter d'anciens programmes sans recompilation des sources. L'avantage de la transformation sur l'interprétation ou la simulation est principalement le gain de performances. Le transformateur ayant moins de contraintes de temps qu'un compilateur JIT, il peut procéder à une optimisation plus agressive du programme source, afin de tirer parti au maximum des capacités de l'ordinateur qui le lancera.

Les techniques d'optimisation applicables dans un transformateur de binaires sont assez semblables à celles applicables dans un compilateur classique, à l'exception que certaines informations (par exemple le typage) ne sont pas disponibles de façon complète.

Il y a deux motivations pour l'optimisation d'un programme de sortie d'un transformateur de binaires :

- Optimisation d'un programme "mal" compilé : le programme a été à l'origine compilé avec un compilateur pas très efficace, ou a été compilé sans optimisations.
- Optimisation et amélioration du code généré par le front-end : la transformation d'une simple instruction Sparc peut produire un certain nombre d'instructions en forme intermédiaire, et dans de nombreux cas des techniques simples permettent de simplifier et améliorer le résultat.

Par exemple, le morceau de programme Sparc de la fig.5.1 réalise l'initialisation du registre **l1** à la valeur hexadécimale 0xffffffff. Dans ce cas précis, le compilateur génère deux instructions : la première (**sethi**) fixe les 22 bits de poids forts du registre, et la seconde (**or**), fixe les 10 derniers bits.
Le code intermédiaire qui en résulte contient donc deux instructions. Une optimisation de *propagation de constante*, telle qu'on le verra dans la section 5.3.1, combinée à une évaluation statique (section 5.4.2), permettent de facilement réduire ces deux instructions en une seule.

Soit le programme Sparc d'origine :

```
sethi 0x3fffff, %l1
or 0x3ff, %l1
```

Ce programme est traduit en forme intermédiaire :

```
VAR[l1] := IMM[0xfffffc00]
VAR[l1] := VAR[l1] or IMM[0x3ff]
```

Après la propagation de copies et la suppression de code mort :

```
VAR[l1] := IMM[0xfffffc00] or IMM[0x3ff]
```

Enfin, après l'évaluation statique de l'expression :

```
VAR[l1] := IMM[0xffffffff]
```

FIG. 5.1 – Exemple d'optimisation de morceau de programme Sparc.

## 5.2 Critères d'optimisation

Il est accepté que le but de l'optimisation d'un programme est de le rendre plus performant. Mais dans quelles circonstances ? Selon quels critères peut-on affirmer qu'un programme est optimisé ou ne l'est pas ?

L'optimiseur parfait n'existe pas. En effet, considérons qu'un programme parfaitement optimisé est le programme le plus rapide : il est très facile d'optimiser un programme pour certaines entrées précises.
En effet, on peut s'arranger pour précalculer certaines réponses en fonction de certaines entrées, et faire en sorte que le programme soit très rapide pour celles-ci. Par exemple, on peut optimiser la version récursive ou itérative d'une fonction de fibonacci pour répondre x lorsque l'entrée est y. Bien que ce programme soit optimal pour l'entrée y, sa version générale n'est pas plus optimisée. On pourrait alors dans ce cas optimiser la fonction en précalculant les 100 premières valeurs. L'appel de fonction sur une des 100 premières valeurs serait alors optimisé, mais le programme serait plus lent encore pour répondre lorsque l'entrée est supérieure à 100.
Il n'existe donc pas d'optimiseur qui puisse rendre le programme le plus rapide possible pour toutes les entrées possibles, car il existe forcément un optimiseur qui produira un programme plus rapide pour une entrée donnée précise.

Si nous considérons qu'un programme parfaitement optimisé est le programme le plus petit qui produise le même résultat que le programme original, et que nous tentons d'optimiser un programme qui cycle, le résultat en sera le plus petit programme qui cycle : l'unique instruction qui se branche sur elle-même, c'est-à-dire dans la plupart des langages d'assemblage "label : goto label".

Si un optimiseur parfait existe et est toujours capable d'obtenir ce résultat, alors cet optimiseur est capable de résoudre le halting problem (la preuve complète est disponible dans *Modern compiler implementation in Java*[20], p387).

Puisqu'il n'est pas possible de créer un optimiseur parfait, nous tenterons simplement d'améliorer les performances du programme. Nous utiliserons comme critère le nombre d'instructions en langage intermédiaire. Les différents algorithmes décrits ci-dessous ont pour but d'éliminer un maximum d'instructions inutiles et de simplifier celles qui peuvent l'être.

## 5.3 Optimisation des variables

Les optimisations sur les variables sont des optimisations qui ne considèrent pas la sémantique des instructions modifiées, mais uniquement les variables qu'elles utilisent et modifient. Ces optimisations servent à diminuer le nombre de références à des variables utilisées dans le programme, dans le but de pouvoir économiser de la mémoire, et les instructions de transfert entre variables.
En d'autres termes, ces optimisations permettent d'éviter que plusieurs copies de la même donnée soient simultanément présentes en mémoire.

### 5.3.1 Propagation des constantes

Les programmes en assembleur Sparc contiennent une quantité d'instructions initialisant des registres par des constantes (parfois même en deux étapes, comme dans l'exemple de la figure 5.1), pour ensuite réaliser une opération mathématique avec une autre constante ou variable. Le langage intermédiaire que nous avons défini n'a pas de restrictions sur les opérandes acceptés par les opérateurs, contrairement à Sparc, et permet de représenter ce type de calculs plus efficacement, sans devoir utiliser de variables temporaires.

La propagation des constantes est une technique de base intégrée à la plupart des compilateurs : elle permet de propager les constantes utilisées dans le programme directement dans les instructions qui en font usage. De cette façon, il n'est pas nécessaire de consommer une variable pour contenir une constante, qui peut directement être placée là ou elle est nécessaire.

### 5.3.2 Propagation des copies

La propagation des copies est une généralisation de la propagation des constantes : elle vise à minimiser le nombre de copies existantes simultanées d'une variable. On nomme ces copies simultanées des *alias*. $v_1$ est un alias de $v_2$ à l'instruction $i$ si $v_1$ est toujours égale à $v_2$ en $i$.
L'idée principale de la propagation des copies est de toujours utiliser le plus vieil alias possible. Dans l'exemple de la figure 5.2, la variable V est un alias pour la variable X. La variable X ayant été définie plus tôt dans le programme, elle est l'alias le plus ancien, et est donc la variable qui sera utilisée dans la dernière assignation. Dans l'exemple de la figure 5.3, l'instruction à modifier

| Forme originale | Forme optimisée |
|---|---|
| `VAR[V] := VAR[X]` | `VAR[V] := VAR[X]` |
| `...` | `...` |
| `VAR[Y] := VAR[V]` | `VAR[Y] := VAR[X]` |

FIG. 5.2 – Exemple d'optimisation de copies.

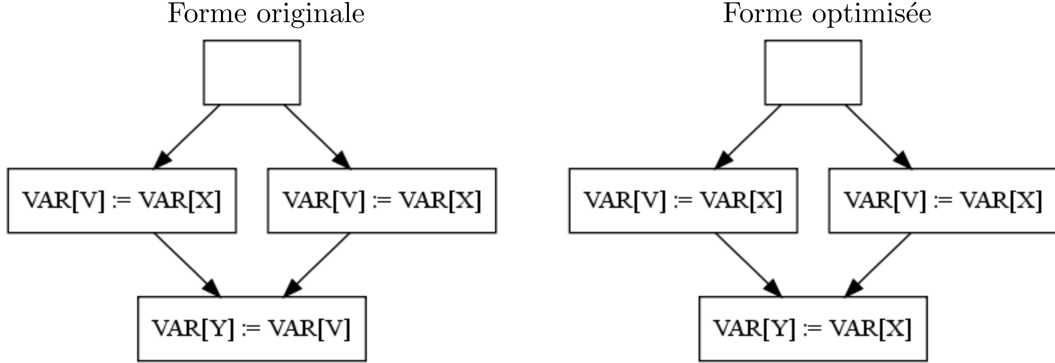


FIG. 5.3 – Exemple d'optimisation de copies.

a plusieurs dépendances pour la variable X. Cependant, ces deux dépendances sont deux assignations (de type $v := x_i$), dont les deux variables sources sont identiques ($x_1 = x_2 =$ X). Ensuite, dans aucun des chemins la variable X n'est modifiée.
En effet, l'exemple de la figure 5.4 présente le même programme, avec les mêmes assignations que dans l'exemple précédent, à la différence que dans un des chemins d'exécution, la variable X est modifiée. Dans ce cas précis, il n'est pas possible d'optimiser la dernière instruction en substituant X à V, car la variable X pourrait contenir une valeur différente de celle de V. X n'est plus un alias pour V dans ce cas. En d'autres termes, l'instructions $i_5 \notin live(i_3, X)$.

**Définition** Dans l'instruction $i_{cible}$ utilisant la variable $v$, celle-ci peut être remplacée par la variable $x$ ssi :

- $\forall i_{source_i}$ de type $v := x_i : x_i = x$
- $\forall i_{source_i} : i_{source_i} \in dependances(v, i_{cible})$
- $\forall i_{source_i} : i_{cible} \in live(i_{source_i}, x)$

Autrement dit, une variable $v$ peut être substituée à la variable $x$ dans l'instruction $i_{cible}$ ssi toutes les instructions dont dépendent $(v, i_{cible})$ sont effectivement des assignations de $x$ dans $v$, et ssi la variable $x$ est bien vivante en $i_{cible}$, le long de tous les chemins partant des différentes instructions sources $i_{source_i}$.

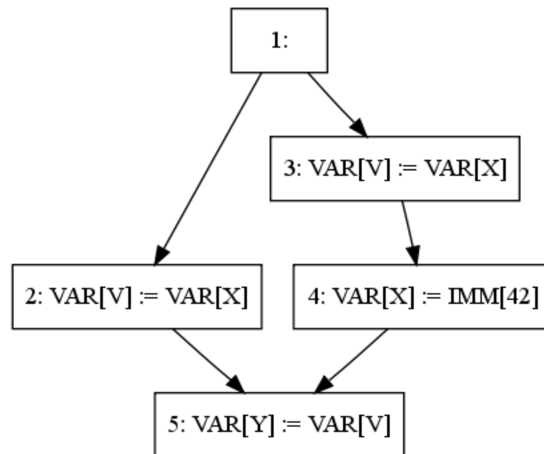


FIG. 5.4 – Exemple de copies non transférables.

**Algorithme 7** Optimisation des copies de l'instruction $i$

**Entrées:** $i$ instruction à optimiser.
**Entrées:** $v$ variable utilisée dans l'instruction $i$.
**Sorties:** $i$ utilise un alias de $v$ plus ancien, si possible.
**Sorties:** Retourne 1 si une modification a eu lieu.

  $deps \Leftarrow dependances(v, i)$
  **si** $deps = \emptyset$ **alors**
    **retourner** 0
  **finsi**
  $x \Leftarrow$ null
  **pour tout** $source \in deps$ **faire**
    **si** $source$ n'est pas une assignation $v := x$ **alors**
      **retourner** 0
    **finsi**
    **si** $x =$ null **alors**
      $x \Leftarrow source$.operande
    **sinon**
      **si** $source$.operand $\neq x$ **alors**
        **retourner** 0
      **finsi**
    **finsi**
  **fin pour**
  **pour tout** $source \in deps$ **faire**
    **si** $\neg i \in live(source, x)$ **alors**
      **retourner** 0
    **finsi**
  **fin pour**
  Remplacer($i$, $v$, $x$)
  **retourner** 1

**Remarques**

Cet algorithme (alg. 7) ne supprime aucune instruction, mais réduit par contre les dépendances entres variables, permettant par la suite la suppression d'instructions inutiles.
Comme expliqué à la section 4.2.2, cet algorithme ne peut être appliqué que sur des variables locales. Pour pouvoir être utilisable sur les variables globales, des analyses supplémentaires doivent être effectuées (section 4.3.3) pour déterminer si certains appels de procédure dépendent ou non de variables globales, et les algorithmes décrits au chapitre précédent doivent être adaptés.

## 5.4 Optimisation des instructions

### 5.4.1 Suppression de code mort

Un code est dit "mort" lorsqu'il n'est jamais exécuté, ou que le résultat de son exécution n'est jamais utilisé. L'algorithme permettant de découvrir ces instructions inutiles est l'algorithme existeDependances($i$) (alg. 4). Si une instruction n'a pas de dépendances, elle peut être éliminée sans risques de perturber le fonctionnement du programme. D'autres instructions peuvent être supprimées : les instructions qui n'ont aucun effet, comme l'instruction suivante.

```
VAR[X] := VAR[X]
```

### 5.4.2 Évaluation statique

La propagation des copies, décrite plus haute, permet en outre de propager les constantes. Il peut alors arriver que deux constantes soient présentes en même temps dans la même expression, et il est alors possible de précalculer le résultat d'une opération, et de l'insérer directement dans l'instruction.
Il existe trois cas de figure possibles :

**Deux opérandes constants** Deux opérandes constants sont paramètres d'une opération logique ou arithmétique. Il est alors facile d'effectuer l'opération et de remplacer l'expression d'origine par l'expression simplifiée.

**Opérande absorbant** Un opérande est absorbant si il détermine à lui seul le résultat de l'opération. Pour la multiplication, 0 est un absorbant : quel que soit le deuxième opérande, le résultat sera toujours 0. Les opérations logiques **and**, **nand**, et arithmétiques ***** acceptent 0 comme absorbant. D'autres opérateurs acceptent 0 ou 1 comme absorbant selon des règles plus précises.

**Opérande neutre** Un opérande est neutre lorsqu'il n'influence pas le résultat d'une opération, quelle que soit l'autre opérande. Les opérateurs logiques **or** et **nor**, ainsi que l'opérateur arithmétique + acceptent 0 comme neutre. D'autres opérateurs acceptent 0 ou 1 comme neutre selon des règles plus précises.

Après avoir effectué cette optimisation, si elle est possible, l'instruction se transforme dans le premier cas (deux opérandes constants) sous la forme :

```
VAR[X] := IMM[1234]
```

Dans le deuxième et troisième cas (opérande absorbant ou neutre), l'instruction devient sous la forme :

```
VAR[X] := VAR[Y]
```

L'instruction résultante sera très probablement réutilisée lors d'une propagation de copies ou de constantes, pour être ensuite éliminée par l'optimisation et suppression des codes morts.

### 5.4.3 Factorisation d'expressions

La factorisation d'expression est une technique d'optimisation visant à éliminer des calculs de sous-expressionq redondantes. Elle s'applique sur un groupe d'instructions faisant partie d'un basic block, c'est-à-dire sur un groupe d'instructions séquentielles. Voici un exemple d'un tel bloc :

```
VAR[X] := VAR[A] + VAR[B]
VAR[Y] := VAR[B] + VAR[C]
VAR[Z] := VAR[A] + VAR[B]
VAR[W] := VAR[X] + VAR[Y]
VAR[W] := VAR[W] + VAR[Z]
```

L'analyseur peut transformer cette série d'opérations en un arbre, dont les noeuds sont les opérateurs et les feuilles les variables (fig. 5.5). L'analyseur va ensuite trouver les sous-expressions communes et les simplifier afin de réduire le nombre de noeuds (fig. 5.6). En navigant dans cet arbre, l'analyseur peut trouver des simplifications algébriques (par exemple opérandes qui s'annulent), ou une réorganisation qui réduit le nombre d'opérations nécessaires, ou le nombre de variables temporaires. Cet arbre est ensuite retraduit en langage intermédiaire.

### 5.4.4 Optimisation des boucles

Une grande partie du temps passé dans les programmes est très souvent consommée dans les boucles, qui répètent des instructions un certain nombre de fois. C'est donc à l'intérieur de ces boucles que se situent les instructions qui sont exécutées le plus souvent, et qui ralentissent le programme. Plusieurs techniques existent pour réduire le temps passé dans les boucles.

#### Mise en évidence d'invariants

Tous les bons programmeurs le savent : il faut laisser le moins souvent possible d'expressions invariantes dans le corps d'une boucle. En effet, même si la valeur d'une expression ne change jamais au cours de la boucle, elle est néanmoins évaluée à chaque tour de boucle et ralenti inutilement le programme.
La mise en évidence d'invariants est une technique destinée à déplacer les parties invariantes d'une boucle en dehors de celle-ci. Pour pouvoir déplacer une instruction $i$ d'une boucle, plusieurs conditions doivent être réunies :

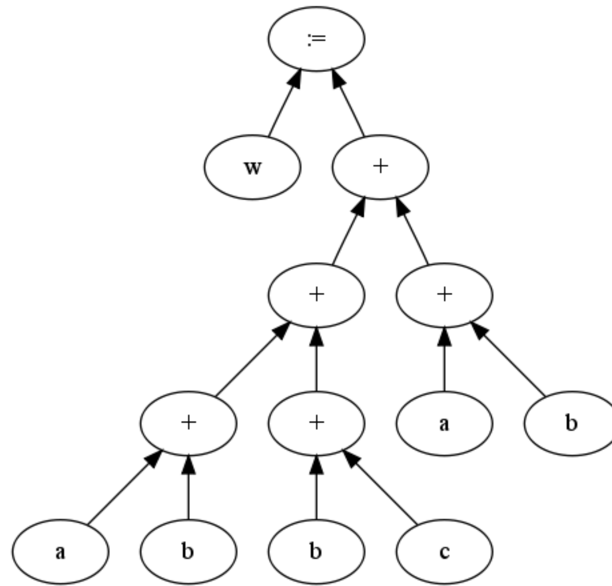


FIG. 5.5 – Vue en arbre d'une expression

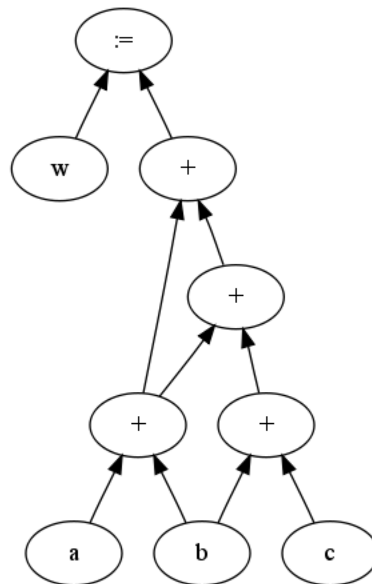


FIG. 5.6 – Vue en arbre d'une expression optimisée

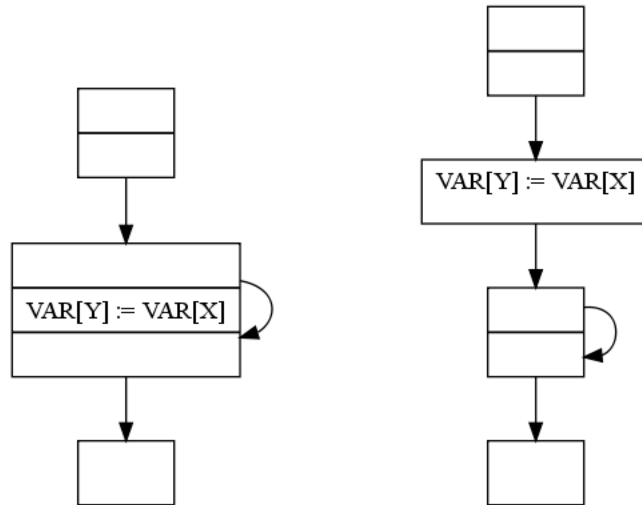


FIG. 5.7 – Mise en évidence d'une assignation.

- Les dépendances de la boucle doivent être à l'extérieur de celle-ci, pour éviter que ces dépendances puissent être modifiées.
- Le résultat de l'instruction (s'il s'agit d'une assignation) ne doit pas être modifié dans l'entièreté de la boucle, à l'exception évidement de l'instruction $i$.
- Les instructions Call, Ret et If ne peuvent pas être déplacées, car elles ont d'autres effets de bord.

Soient *loop* l'ensemble des instructions d'une boucle, $i \in loop$ l'instruction d'assignation à tenter de déplacer, et $l_1 \in loop$ la première instruction de la boucle :
$i$ peut être mis en évidence si $\forall x_i \in VariablesDependantes(i) : Dependances(x_i, i) \cap loop = \emptyset \wedge live(i.\text{dest}, i) \supseteq loop \setminus i$.

Si ces conditions sont réunies, la mise en évidence se réalise comme ceci : l'instruction $i$ est déplacée en tête de bloc, devant $l_1$. Les basic blocks sont recalculés, car le déplacement de l'instruction $i$ peut avoir créé un nouveau basic block. En effet, $i$ ne peut plus appartenir au même basic block que $l_1$, car elle précède $l_1$ et $l_1$ est une tête de boucle. Par la définition des basic blocks, $l_1$ est un début de basic block, et donc $i$ ne peut plus faire partie du même basic block. L'exemple de la figure 5.7 est une application d'une mise en évidence d'instruction dans un cas simple.

### Déroulage de boucle

Certaines boucles peuvent parfois être très petites (quelques instructions), et il se peut que le programme passe plus de temps à incrémenter un compteur qu'à effectuer l'instruction. Le déroulage de boucle est une technique permettant de remplacer la boucle par la série d'instructions qui auraient été exécutées par cette boucle. Par exemple, le programme suivant possède une boucle de 4 instructions, dont une seule sert réellement à faire le calcul.

```
VAR[i] := IMM[0]
VAR[s] := IMM[0]
loop: VAR[s] := VAR[s] + VAR[i]
VAR[i] := VAR[i] + 1
VAR[fin] := VAR[i] >= 8
if(VAR[fin]) goto loop
```

Le déroulage de boucle permet de transformer le programme comme ceci :

```
VAR[i] := IMM[0]
VAR[s] := IMM[0]
VAR[s] := VAR[s] + IMM[0]
VAR[s] := VAR[s] + IMM[1]
VAR[s] := VAR[s] + IMM[2]
VAR[s] := VAR[s] + IMM[3]
VAR[s] := VAR[s] + IMM[4]
VAR[s] := VAR[s] + IMM[5]
VAR[s] := VAR[s] + IMM[6]
VAR[s] := VAR[s] + IMM[7]
VAR[i] := IMM[8]
VAR[fin] := IMM[1]
```

Dans ce cas précis, les optimisations de propagation des variables et d'évaluations statiques permettront même de ne pas exécuter la boucle, car le résultat est calculable pendant l'optimisation.
Le déroulage de boucle n'est pas toujours une amélioration du programme, car il se peut que le nombre d'instructions à ajouter soit très grand et nuise à la taille du programme. On utilise dans ce cas un déroulage de boucle partiel : la structure de boucle est gardée, mais le nombre d'instances de la boucle réellement déroulées reste faible (la plupart des compilateurs déroulent au maximum 8 instances de boucles).

Nous ne détaillerons pas l'algorithme du déroulage de boucles, ni les différents algorithmes et heuristiques pour déterminer la pertinence d'un déroulage de boucle. Des explications plus complètes sont disponibles dans *Modern Compiler Implementation in Java*[20], p433.

### Optimisation du contrôle de flux

L'optimisation du contrôle de flux est une optimisation simple visant à supprimer les branches qui ne sont pas utilisées. Si l'optimiseur parvient à prouver qu'une condition de branchement n'est jamais satisfaite, le branchement conditionnel peut être supprimé et transformé en branchement simple. Cette optimisation a pour effet de simplifier le graphe de contrôle de flot. Il peut arriver que des dépendances imposées par un branchement disparaissent en même temps que celui-ci, et que cette disparition permette des optimisations plus poussées. Par exemple, la figure 5.8 présente les différentes étapes d'une optimisation de flux. La condition 6=5 étant fausse, le branchement "true" est supprimé. Comme le basic block n'a plus d'ancêtre, il est supprimé (code mort). De ce

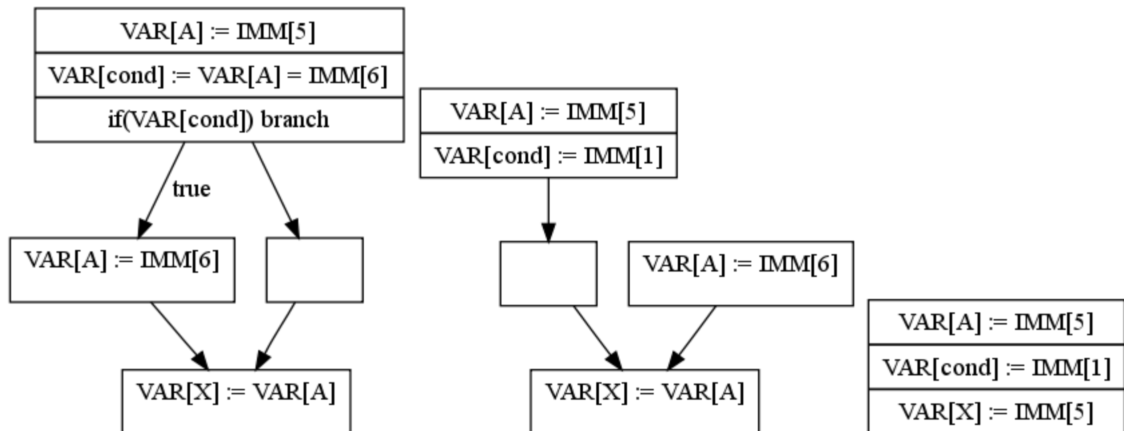


Fig. 5.8 – Les différentes étapes d'une optimisation de flot

fait, la dépendance sur la variable VAR[A] tombe, et donc la propagation des copies peut avoir lieu.

## 5.5 Remarques

### 5.5.1 Exhaustivité

Les optimisations décrites dans ce chapitre sont les optimisations les plus courantes dans la plupart des compilateurs. Ces optimisations ne sont qu'une fraction des optimisations possibles et réalisées sur certains compilateurs, car il n'est pas possible d'être exhaustif dans un travail de cette envergure (certains compilateurs intègrent des centaines d'optimisations disponibles). D'autres optimisations ne sont réalisables que dans certains types de machines (par exemple les machines à pile, que nous verrons au chapitre suivant). D'autres optimisations sont totalement dépendantes de l'architecture du processeur pour lequel le compilateur est conçu. Il y a donc autant de techniques qu'il y a de processeurs.

### 5.5.2 Ordre de passage

Nous avons décrit une série d'optimisations dans ce chapitre, mais rien n'a été dit sur la manière de faire fonctionner ces optimisations. Plusieurs stratégies s'offrent à nous :

**Stratégie simple** Toutes les optimisations sont exécutées en boucle sur toutes les instructions tant que la procédure optimisée change. Une fois qu'un point fixe est trouvé, la phase d'optimisation s'arrête. Cette méthode n'est pas performante, car elle lance de nombreuses tentatives d'optimisation sur des instructions qui ne sont pas optimisables (l'optimisation est sans objet sur certaines optimisations). Cependant, cette méthode fonctionne, car les optimisations décrites dans ce chapitre ne sont jamais destructives (elles gardent la sémantique du programme). Cette méthode ne cycle pas,

car aucunes des optimisations décritres dans ce chapitre ne sont antagonistes.

**Stratégie par dépendances** Une notion de dépendance entre optimisations est introduite : certaines optimisations, lorsqu'elles sont utilisées avec succès, déclenchent d'autres optimisations dépendantes de ces premières. Par exemple, lorsqu'une propagation de copie a pu être effectuée, une suppression de code mort peut être tentée. Cette stratégie permet de ne pas réessayer des optimisations qui sont vouées à l'echec.

**Stratégie par appel explicite** Lorsqu'une optimisation a été réalisée avec succès, le module effectuant cette optimisation marque les instructions devant être revisitées par les autres modules. De cette manière, on évite en plus de devoir revisiter des instructions qui ne peuvent plus être optimisées.

# Chapitre 6

# Génération de code

## 6.1 Rôle du Back-end

Le Back-end est la dernière partie d'un compilateur. Son rôle est de transformer une forme intermédiaire, compréhensible uniquement par le compilateur et ses outils associés, en code machine exécutable par un processeur d'ordinateur, ou dans un autre langage spécifique. Le résultat ne doit pas forcément être un programme, par exemple les outils LaTeX et *graphviz* sont des compilateurs générant des documents textes et des images.

Le Back-end décrit dans ce chapitre est un générateur de code pour la machine virtuelle Java, et s'interface sur la forme intermédiaire décrite auparavant dans ce travail. La JVM n'est pas la seule forme de sortie possible : on peut très bien, avec un peu de travail, greffer différents back-ends au compilateur pour générer du code dans d'autres architectures (comme la plupart des compilateurs), ou même un générateur de graphiques.
Dans ce chapitre, nous allons introduire le lecteur à l'architecture de la JVM, expliquer comment la programmer et comment l'utiliser comme cible pour la traduction. Dans ce chapitre, il ne sera pas question de donner des explications complètes sur la JVM ; les explications de ce chapitre sont le strict minimum pour pouvoir comprendre comment fonctionne le générateur de code du transformateur. Pour de plus amples informations, les ouvrages de référence "The Java Virtual Machine Specification" [17] et "Programming for the Java Virtual Machine" [16] sont idéaux.
Ensuite, nous décrirons les différentes routines devant être disponibles sous la forme de librairies afin de permettre au programme traduit de s'accomoder des différences d'architecture entre la JVM et un programme Sparc.

## 6.2 Architecture Java et JVM

Le langage Java est un langage de haut niveau orienté objet. Similaire en syntaxe au langage C++, il possède cependant une quantité de différences. Parmis celles-ci, on peut citer :

- Toute instance d'un objet est accessible depuis une référence sur cet objet (sorte de pointeur), et c'est l'unique moyen d'accéder à un objet.
- L'allocation et la désallocation de mémoire sont automatiques. La mémoire nécessaire pour un objet est automatiquement allouée lors d'un **new**, et est retournée au système grace au *garbage collector*. Le GC (garbage collector) est une routine exécutée régulièrement ou lors de certaines occasions, qui parcourt les structures de données en mémoire à la recherche d'objets inutilisés, prêts à être recyclés.
- Du fait que toutes les opérations sont accessibles depuis les objets, il n'existe pas de pointeurs ni d'espace d'adressage à proprement parler en Java. L'arithmétique de pointeurs n'existe pas en Java.
- Une librairie complète de classes permettant d'accéder aux fichiers, au réseau, aux interfaces graphiques, structures de stockage de données, etc. est inclue dans les spécifications du langage.

### 6.2.1 Architecture de la JVM

La Java Virtual Machine est la machine décrite dans les spécifications Java. Elle sert d'interface entre le programme Java, compilé en bytecode, et la machine réelle et son système d'exploitation. Comme dit précédemment, la machine JVM est une machine à pile, c'est-à-dire que les opérations s'aident d'une pile. Nous y reviendrons plus tard.
La JVM n'a pas connaissance du langage Java. Elle est seulement capable d'exécuter des fichiers *.class*, qui contiennent des classes Java compilées.

Cependant, la JVM hérite de nombreuses caractéristiques du langage Java, qu'il faut prendre en compte lors de l'écriture de bytecode Java :

- Les references dans la JVM sont des références vers des objets existants ou des nouveaux objets. Il n'est pas possible d'insérer une valeur invalide dans une référence.
- Le programme est divisé en classes, elles-mêmes divisées en membres (méthodes, variables d'objet). Tous ces membres doivent être déclarés explicitement dans les fichiers *.class*, et l'accès aux membres est scrupuleusement vérifié par la JVM. De ce fait, les membres déclarés *private* ne sont effectivement pas accessibles depuis d'autres classes.
- Les notions d'héritage et de surcharge sont exposées à la machine virtuelle et sont vérifiées avant chaque utilisation.
- Les types fondamentaux de la JVM sont les entiers, les nombres flottants et les références. Bien que les variables locales ne soient pas fortement typées (il est possible d'insérer successivement des entités de types différents dans la même variable locale), les instructions le sont. Il n'est par exemple pas possible d'additionner une référence avec un entier, tout comme il n'est pas possible d'appeler une méthode avec des paramètres de types incorrects.

### Entités de la JVM

Plusieurs concepts doivent être introduits :

**Thread** Un thread est un fil d'exécution parallèle d'un programme. Un programme ayant plusieurs threads peut virtuellement effectuer plusieurs opérations en même temps grâce au partage du temps processeur (multitasking). Chaque programme Java possède au minimum un thread principal, d'autres pouvant être créés par la suite si nécessaire.

**Frame** Une frame est l'entité qui décrit une instance d'exécution d'une méthode. Une nouvelle frame est créée lors de l'entrée dans une méthode et est détruite à la sortie de celle-ci. Elle contient toutes les informations nécessaires à l'exécution de la méthode courante, incluant la table des variables locales, ainsi que la pile des opérandes.

**Table des variables locales** Lors de l'entrée dans une frame, une nouvelle table des variables locales est créee. Cette table a une taille fixée à la compilation, et est indexable à l'aide de numéros de variables. Cette table contient aussi les paramètres d'appel de la méthode. Chaque frame possède une table des variables locales dédiée.
Cette table reprend donc en partie le rôle de la pile des architectures processeurs classiques.

**Heap** Le heap est la zone en mémoire où sont alloués les objets créés dynamiquement et statiquement. Cette zone est globale à tous les threads d'un programme Java. Tous les objets sont alloués dans le heap, et ne peuvent être supprimés explicitement.

**Pile des opérandes** Pile locale, différente pour chaque frame, servant à la communication entre instructions, ainsi qu'au passage de paramètres entre méthodes.

**Champ** Un champ est la terminologie JVM pour nommer une variable d'objet.

### Types de données

Comme dit précédemment, la JVM reconnait plusieurs types de données. Ces types sont **boolean**, **byte**, **char**, **short**, **int**, **long**, **float**, **double** et **reference**. Les types **boolean**, **byte**, **short**, **int**, **long** sont des entiers signés de respectivement 1, 8, 16, 32 et 64 bits. Les types numériques dont la taille est plus petite que 32 bits n'ont que quelques instructions dédiées ; en pratique, ils sont considérés comme des **int**, et la majorité des instructions s'appliquant sur des **int** sont valables sur ces types, contrairement au langage Java, dans lequel il n'est pas possible d'effectuer d'opérations logiques sur des entiers, ni de faire d'additions incluant des booléens.

Lors du chargement d'une référence, il est parfois nécessaire de préciser de quelle classe elle fait partie. La JVM exige une manière précise de nommer les classes : le chemin complet de la classe doit être précisé, en utilisant le caractère "/" entre les différents éléments. Par exemple, la classe InputStream du package java.io est nommé "java/io/InputStream".

```
ldc 1
ldc 10
iadd
istore 5
```

Fig. 6.1 – Exemple de programme d'une machine à pile

**Machine à pile**

La JVM est une machine à pile. Cela veut dire que l'exécution d'une opération nécessite toujours plusieurs étapes :

1. *Push* des paramètres sur la pile,
2. Exécution de l'instruction,
3. Sauvegarde du résultat (facultatif).

L'exemple de la figure 6.1 expose un morceau de programe simple calculant le résultat de 1 + 10, et l'enregistrant dans la variable locale 5.
De nombreuses instructions sont présentes dans la JVM. Le tableau de la figure 6.2 expose les instructions les plus courantes. Les instructions arithmétiques et logiques, par exemple, sont très simples : elles prélèvent deux arguments sur la pile des opérandes et y placent le résultat. La JVM possède aussi des instructions pour manipuler la pile (**pop**, **swap**, **dup**), pour charger et sauvegarder des valeurs dans les variables locales (**iload**, **istore**), une instruction de chargement de constantes (**ldc**), une instruction de saut inconditionnel (**goto**), et des instructions de retour de procédure (incluant **ireturn**, que nous utiliserons par la suite).
La JVM possède aussi une série d'instructions de sauts conditionnels, exposés dans la figure 6.3. Ces instructions testent la valeur de l'entier au sommet de la pile et effectuent un saut si la condition est respectée.

La JVM inclut aussi l'instruction **getfield**, dont le but est de charger sur la pile le *champ* d'un objet. L'instruction **invokevirtual** permet d'effectuer un appel de méthode virtuelle sur un objet (appel de procédure classique en Java). Nous reviendrons sur la syntaxe de ces deux instructions plus loin dans ce chapitre.

### 6.2.2 Le langage de programmation Jasmin

Jasmin [5] est un assembleur pour la machine virtuelle Java. Il traduit des programmes en format texte (assembleur Java) en fichiers ".class" interprétables directement par la JVM. La syntaxe utilisée par cet outil est pratiquement la même que celle du langage *Oolong* décrit dans l'ouvrage "Programming for the Java Virtual Machine" [16], bien que les différences ne soient pas documentées[1]. Nous considérerons par la suite que les deux langages sont équivalents.

[1] Il semblerait que le "Jasmine Oolong" soit une variété de thé, tout comme le Java est une variété de café.

| Instruction | Entrées | Sorties | Sémantique |
|---|---|---|---|
| **iadd** | 2 | 1 | Addition des deux entiers au sommet de la pile |
| **isub** | 2 | 1 | Soustraction des deux entiers au sommet de la pile |
| **idiv** | 2 | 1 | Division des deux entiers au sommet de la pile |
| **imul** | 2 | 1 | Multiplication des deux entiers au sommet de la pile |
| **iand** | 2 | 1 | Opération logique *and* bit à bit des deux entiers au sommet de la pile |
| **ior** | 2 | 1 | Opération logique *or* bit à bit des deux entiers au sommet de la pile |
| **swap** | 2 | 2 | Interversion des deux éléments au sommet de la pile |
| **pop** | 1 | 0 | Suppression de l'élément au sommet de la pile |
| **dup** | 1 | 2 | Duplication de l'élément au sommet de la pile |
| **iload** $i$ | 0 | 1 | Push sur la pile de la variable $i$ |
| **istore** $i$ | 1 | 0 | Sauvegarde de l'entier au sommet de la pile dans la variable $i$ |
| **ldc** *const* | 0 | 1 | Push sur la pile de la constante *const* |
| **goto** *label* | 0 | 0 | Transfert de contrôle inconditionnel vers l'étiquette *label* |
| **ireturn** | 1 | N/A | Fin de méthode et retour de l'entier au sommet de la pile à la méthode appelante |

FIG. 6.2 – Instructions de base de la JVM

| Instruction | Sémantique |
|---|---|
| **ifeq** *label* | Saut à *label* si la valeur en sommet de la pile est égale à 0 |
| **ifge** *label* | Saut à *label* si la valeur en sommet de la pile est plus grande ou égale à 0 |
| **ifgt** *label* | Saut à *label* si la valeur en sommet de la pile est plus grande que 0 |
| **ifle** *label* | Saut à *label* si la valeur en sommet de la pile est plus petite ou égale à 0 |
| **iflt** *label* | Saut à *label* si la valeur en sommet de la pile est plus petite que 0 |
| **ifne** *label* | Saut à *label* si la valeur en sommet de la pile est différente de 0 |

FIG. 6.3 – Instructions de sauts conditionnels

```
.class public MaClasse
.super java/lang/Object
.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V
  return
.end method
```

Fig. 6.4 – Classe Jasmin minimale

#### Programme Jasmin

Un programme Jasmin décrit une classe. Il commence donc par une description de la classe : son nom et protection, son héritage (superclasse et interfaces), la liste de ses champs (et protections), ainsi que la liste des méthodes et leurs codes sources. Une classe Jasmin minimale est décrite à la figure 6.4. Cette classe hérite de *java.lang.Object*, et ne contient qu'une seule méthode, le constructeur. Le constructeur appelle ensuite le constructeur de la classe *Object*.

#### Déclaration de méthodes

Une méthode doit être déclarée de cette manière :
`.method` *`protection nom(paramètres)type de retour`*
Où *protection* est *public*, *private*, *protected*, *nom* est le nom donné à la méthode, *paramètres* est une liste de types de paramètres, et *type de retour* est le type de retour de la méthode. Les types sont données sous une forme compacte : **I** pour un entier, **F** pour un flottant, **V** pour le type void (abscence de valeur de retour), et **Lclasse ;** pour les références d'objet. La méthode publique MaMethode prennant un entier, un flottant, une référence vers un objet de type java.lang.Object, et ne retournant pas de valeur serait déclarée comme ceci :

```
.method public MaMethode(IFLjava/lang/Object;)V
.end method
```

La déclaration de méthode doit aussi inclure le nombre maximum de variables locales, ainsi que la taille maximale de la pile des opérandes. L'exemple d'en haut peut être réécrit comme ceci :

```
.method public MaMethode(IFLjava/lang/Object;)V
.limit locals 2
.limit stack 10
.end method
```

#### Appel de procédure

L'appel de procédure, dans un assembleur orienté objet, est un peu plus complexe. Afin d'appeler une méthode, il est nécessaire de faire ces actions :

1. Pousser sur la pile une référence de l'objet sur lequel porte la méthode,
2. Pousser la liste des arguments sur la pile,

3. Effectuer l'appel de méthode. La valeur de retour de la méthode, si elle existe, est mise dans la pile.

La syntaxe de l'appel de méthode virtuelle est la suivante :
`invokevirtual` *`classe/méthode`*`(`*`paramètres`*`)`*`type de retour`*
Où *classe* est le nom de la classe, *méthode* le nom de la méthode, *paramètres* la liste des types des paramètres en forme compacte, et *type de retour* le type de retour de la méthode. La référence de l'objet peut être une référence statique, par exemple *System.out* (récupérable avec l'instruction **getstatic**), ou tout autre objet. La référence sur l'objet en cours (*this* en langage Java) se trouve toujours dans la liste des variables locales, à l'offset 0.
Le programme de la figure 6.5 est un exemple commenté de programme incluant ces notions.

### 6.2.3 Vérificateur

Tout programme bien formé (pouvant être assemblé par Jasmin) n'est pas forcément acceptable par la JVM. En effet, avant le chargement de toute classe Java, la JVM procède à plusieurs vérifications statiques afin d'empêcher le chargement de programmes incorrects. La première vérification porte sur l'intégrité du fichier .class. Ce fichier doit être valide. Ensuite, la JVM vérifie que toutes les constantes pointées par cette classe existent bien et sont valides. La vérification suivante porte sur les instructions : toutes les instructions doivent posséder un opcode et une sémantique valide (il est par exemple interdit d'effectuer un saut en dehors d'une méthode).
Enfin, le vérificateur détermine si le programme peut se trouver dans un état dans lequel une variable peut être lue sans avoir été assignée, ou si l'état de la pile peut être incohérent à certains moments (par exemple une instruction se retrouverait avec trop peu d'éléments sur la pile, ou la pile contiendrait encore des éléments à la fin de la méthode).

S'il existe une possibilité que le programme soit incorrect, ce programme est rejeté. En principe, il n'est pas possible de produire ce type de programmes en utilisant un compilateur Java, et notre transformateur ne devrait pas non plus générer de programmes invalides. Cette étape de vérification arrive généralement à découvrir une grande partie des bugs de la génération de code, entre autres en ce qui concerne les problèmes de pile.

## 6.3 Traduction

La traduction d'un langage vers un autre est une opération délicate. Bien que notre forme intermédiaire soit très simple, la machine virtuelle Java est une architecture de machine radicalement différente, à laquelle il faut s'adapter. Il est donc nécessaire de fixer des conventions sur la manière dont sont appelées les procédures, la manière de passer les paramètres, d'appeler les procédures externes, etc.

```
.class public Fibonacci
.super java/lang/Object
.method public <init>()V
    aload_0                 ; pointeur this
    invokespecial java/lang/Object/<init>()V
    return
.end method
.method public fib(I)I      ; méthode fib(arg)
    .limit stack 5          ; maximum 5 étages dans la pile
    .limit locals 5         ; maximum 5 variables locales
    iload 1                 ; charger le premier paramètre sur la pile
    ldc 1
    isub
    ifgt bigger             ; saut à bigger si arg est plus grand que 1
    ldc 1
    ireturn                 ; retourner 1

bigger:
    iload 1                 ; charger arg
    ldc 1
    isub
    aload 0                 ; pointeur this
    swap                    ; placer les arguments dans le bon ordre
    invokevirtual Fib/fib(I)I ; lancer Fib(arg -1)
    istore 2                ; sauvegarde du retour dans la variable
    iload 1                 ;                              locale 2
    ldc 2
    isub                    ; calcul de arg - 2
    aload 0
    swap
    invokevirtual Fib/fib(I)I
    ;call
    iload 2
    iadd                    ; addition des deux retours
    ireturn                 ; retour de la réponse
.end method
```

FIG. 6.5 – Implémentation récursive de la fonction fibonacci

Ensuite, nous établirons les règles de transformation sémantique nécessaires à la compréhension du dernier étage du compilateur. Nous expliquerons ensuite comment encore optimiser le résultat produit, afin de produire le code le plus comptact possible.
Enfin, nous donnerons quelques pistes pour réduire la surcharge de travail causée par la différence entre la machine Sparc et la machine Java.

### 6.3.1 Conventions

Comme dit plus haut, des conventions sont nécessaires afin de déterminer un contexte dans lequel les programmes vont s'exécuter.

#### Appel du programme

Deux possibilités s'offrent à nous en ce qui concerne l'appel du programme traduit. Le programme traduit peut être directement exécutable (la méthode statique main() lance immédiatement le programme à son point d'entrée), ou bien peut être lancé par un chargeur, écrit en Java pur. Nous préfererons cette dernière méthode, car elle permet de créer facilement plusieurs instances du programme (via des threads séparés), ou même de lancer plusieurs programmes différents en même temps, dans la même machine virtuelle Java. Cette méthode permet aussi une récupération des erreurs plus efficace, car le chargeur peut analyser plus en détail les circonstances du plantage et peut aussi charger un debugger.

Cette solution est aussi la plus flexible au niveau de la maintenance et de l'intégration. Un programmeur peut très facilement intégrer un programme traduit à une solution existante en Java, et introduire une interface graphique pour charger les programmes, voire même proposer une alternative aux routines d'entrées-sorties que nous décrirons à la section 6.4.4.

#### Variables locales

Les variables locales sont accessibles à partir des instructions **iload** et **istore**. La variable locale 0 devrait toujours contenir le pointeur *this*, et donc ne jamais être touchée. Ensuite, à partir de la variable locale 1, se situent les paramètres de procédure. Le premier paramètre est stocké dans la variable 1, le second dans la variable 2, et ainsi de suite.
Les variables suivantes sont disponibles pour le programme. Toutes les variables locales de chaque procédure sont numérotées afin de suivre ces conventions.

#### Variables globales

Les variables globales, présentes de manière symboliques dans la table des variables globales, sont chacunes nommées différement (par exemple par une règle statique, telle que "global_*address*"). Ces variables globales sont ensuite transformées en champs dans la classe du programme traduit, et peuvent ensuite être lues par l'instruction **getfield**.

| Production | | | Règle sémantique |
|---|---|---|---|
| $PROG$ | $\leftarrow$ | $E$ | print("Résultat :" + $E$.val) |
| $E$ | $\leftarrow$ | $E_1 + E$ | \$\$.val := $E_1$.val + $E$.val |
| | $\leftarrow$ | $E_1$ | \$\$.val := $E_1$.val |
| $E_1$ | $\leftarrow$ | $E_2 * E_1$ | \$\$.val := $E_2$.val * $E_1$.val |
| | $\leftarrow$ | $E_2$ | \$\$.val := $E_2$.val |
| $E_2$ | $\leftarrow$ | $(E)$ | \$\$.val := $E$.val |
| | $\leftarrow$ | entier | \$\$ := entier.toUnsignedInt() |

FIG. 6.6 – Exemple de traduction dirigée par la syntaxe d'une calculatrice

**Appels de procédure**

Les paramètres fournis aux appels de procédures internes (traduites) sont toujours des entiers, même lorsque l'analyse sémantique a pu déterminer qu'il s'agissait de pointeurs. Lors de l'appel de ces procédures, la variable 0 (pointeur *this*) est d'abord mise sur la pile. Ensuite, les paramètres sont placés sur la pile en commençant par le premier.

Les appels de procédure externes se passent presque de la même manière. À l'aide de la référence sur le runtime (voir section 6.4.4), la méthode voulue est appelée.

### 6.3.2 Traduction dirigée par la syntaxe

Afin de traduire la forme intermédiaire vers du code Jasmin valide, nous allons partir de la grammaire de la forme intermédiaire (fig. 3.1 page 31), que nous allons décorer à l'aide d'une traduction dirigée par la syntaxe.

La traduction dirigée par la syntaxe est une méthode de compilation se basant sur la grammaire du langage de programmation. Dans cette méthode, chaque production de la grammaire est considérée comme un noeud dans un graphe. Chaque noeud possède un ou plusieurs attributs, qui peuvent contenir des données de n'importe quel type.
La valeur de ces attributs est fixée par des règles sémantiques, qui sont spécifiées pour chaque noeud ou production. Nous n'utiliserons ici que des attributs synthétisés : ces attributs sont définis uniquement en fonction des attributs des noeuds fils du noeud courant, et ne dépendent pas de la position du noeud courant dans un arbre. Dans l'exemple de calculatrice de la figure 6.6, tout attribut de noeud n'est fonction que des attributs de ses fils.

De plus amples informations sur la traduction dirigée par la syntaxe peuvent être trouvées dans le Dragon Book [12], chapitre 5.

### 6.3.3 Traduction de la forme intermédiaire

Nous allons décrire la transformation de la forme intermédiaire en bytecode à l'aide de la traduction dirigée par la syntaxe. Chaque règle sémantique dans

chaque production est une règle indiquant quel code doit être produit. En partant de la racine de l'arbre de la grammaire (au niveau d'une instruction), les règles de chaque branche sont exécutées, afin de fournir en fin de compte la traduction complète de chaque instruction.

Chaque production possède un attribut *code*, contenant la traduction en Jasmin de cette production. La notation $$ est un raccourci (emprunté à la syntaxe Yacc) pour indiquer la production en cours d'évaluation. Ainsi, la production annotée suivante :
*instr* ← *dest* := *expr* { $$.code := *expr*.code + *dest*.code }
signifie que le code équivalent à une instruction d'assignation d'une expression dans une variable se traduit par la concaténation du code de cette expression et du code de la destination.

La grammaire annotée des figures 6.7 et 6.8 décrit le procédé de traduction. Parmis ces traductions, il est intéressant de noter que les opérateurs $>$, *beq*, $<$, *leq*, *neq* sont implémentés à l'aide de sauts conditionnels, car ces opérateurs n'existent pas sous la forme d'instruction simple. Les opérateurs overflow() et carry() ne sont pas mentionnés. Ils sont implémentables à l'aide d'additions de nombres 64 bits et d'opérations logiques. Il est évident qu'un compilateur, ne devant pas être écrit sous la forme de traduction dirigée par la syntaxe, peut utiliser des techniques de traduction plus compliquées et plus complètes.

### 6.3.4 Optimisations

Une fois la traduction effectuée, de nouvelles optimisations sont réalisables. Nous n'entrerons pas dans les détails sur la réalisation d'un optimiseur pour machine à pile, car cela dépasse l'envergure de ce travail.
Cependant une optimisation peut facilement être introduite : la suppression de certaines variables temporaires. En effet, notre forme intermédiaire à tendance à utiliser des variables temporaires, qui ne sont utilisées qu'une seule fois. Le code suivant par exemple calcule le résultat de A + B + C :

```
VAR[TMP] := VAR[A] + VAR[B]
VAR[DEST] := VAR[TMP] + VAR[C]
```

Ce programme est traduit comme ceci en Jasmin :

```
iload 1  ; VAR[A]
iload 2  ; VAR[B]
iadd
istore 4 ; VAR[TMP]
iload 4  ; VAR[TMP]
iload 3  ; VAR[C]
iadd
istore 5 ; VAR[DEST]
```

Dans ce programme-ci, une variable locale a été utilisée pour stocker une valeur temporaire ne servant plus. Lorsque l'optimiseur détecte une sauvegarde

| | | Production | Règle sémantique |
|---|---|---|---|
| *instr* | ← | *dest* := *expr* | $$.code := *expr*.code + *dest*.code |
| | ← | *dest* := call *entrypoint*, *operandlist* | $$.code := "**aload** 0" + *operandlist*.code + "invokevirtual package" + *entrypoint*.name + "("+*operandlist*.types + ")I" |
| | ← | ret *operand* | $$.code := *operand*.code + "**ireturn**" |
| | ← | if (*operand*) goto *label* | $$.code := *operand*.code + "**ifne**" + *label* |
| *expr* | ← | $$.code := $operand_1$ *op* $operand_2$ | *dest*.code=$operand_1$.code + $operand_2$.code + *op*.code |
| | ← | *operand* | $$.code := *operand*.code |
| | ← | *overflow* (*op*, *operand*, *operand*) | * |
| | ← | *carry* (*op*, *operand*, *operand*) | * |
| *operand* | ← | VAR[*name*] | $$.code := "**iload**" + name.varnumber |
| | ← | VAR[*name*] *range* | $$.code := "**lload**" + name.varnumber + *range*.code |
| | ← | GLOBAL[*name*] | $$.code := "**aload** 0" + "**getfield** " + classname + "/" + name.globalvarnumber + "I" |
| | ← | IMM[*integer*] | $$.code := "**ldc**" + integer.intvalue |
| | ← | MEM[*operand*] | $$.code := "**aload** 0" + *operand*.code + "**invokevirtual** memoryLookup(I)I" |
| | ← | MEM8[*operand*] | $$.code := "**aload** 0" + *operand*.code + "**invokevirtual** memoryLookup8(I)I" |
| | ← | MEM16[*operand*] | $$.code := "**aload** 0" + *operand*.code + "**invokevirtual** memoryLookup16(I)I" |
| *dest* | ← | MEM[*operand*] | $$.code := "**aload** 0" + "**swap**" + *operand*.code + "**invokevirtual** memorySave(II)V" |
| | ← | MEM8[*operand*] | $$.code := "**aload** 0" + "**swap**" + *operand*.code + "**invokevirtual** memorySave8(II)V" |
| | ← | MEM16[*operand*] | $$.code := "**aload** 0" + "**swap**" + *operand*.code + "**invokevirtual** memorySave16(II)V" |
| | ← | VAR[*name*] | $$.code := "**istore**" + name.varnumber |
| | ← | GLOBAL[*name*] | $$.code := "**aload** 0" + "**swap**" + **putfield**" + classname + "/" + name.globalvarnumber + "I" |

FIG. 6.7 – Traduction dirigée par la syntaxe de la forme intermédiaire (partie 1)

| | | Production | Règle sémantique |
|---|---|---|---|
| *operandlist* | ← | (*operand*$_i$,)* | $$.code := *operand*$_1$.code + ... + *operand*$_n$.code |
| | ← | $\epsilon$ | |
| | | | |
| *op* | ← | + | $$.code := "**iadd**" |
| | ← | - | $$.code := "**isub**" |
| | ← | * | $$.code := "**imul**" |
| | ← | / | $$.code := "**idiv**" |
| | ← | (or \| and \| ...) | $$.code := ("**ior**" \| "**iand**" \| ...) |
| | ← | < | $$.code := "**ifle** tmp_label" + "**ldc** 0" + "**goto** end_label" + "tmp_label : **ldc** 1" + "end_label" |
| | ← | (≤ \| > \| ≥ \| ≠) | voir production de < |
| | ← | = | $$.code = "" |
| | | | |
| *range* | ← | [0 :31] | $$.code := "**l2i**" |
| | ← | [32 :63] | $$.code := "**ldc** 32" + "**lshr** " + "**l2i**" |
| | | | |
| *integer* | ← | **int** | $$.intvalue = Integer(**int**) |
| | ← | zero_extend(**int**) | $$.intvalue = Integer(**int**) |
| | ← | sign_extend(**int**) | $$.intvalue = sign_extend(13,Integer(**int**)) |
| | | | |
| *name* | ← | **identifier** | $$.varnumber=localVarNumber(**identifier**) |

FIG. 6.8 – Traduction dirigée par la syntaxe de la forme intermédiaire (partie 2)

dans une variable suivi d'une restoration de cette variable, et que cette variable n'est plus jamais utilisée, l'optimiseur peut détruire cette sauvegarde. De même, des techniques d'optimisation à base de factorisation d'expression (voir section 5.4.3 page 65) peuvent être employées afin de générer du code efficace et n'employant plus de variables temporaires.

## 6.4 Librairie runtime

La librairie runtime est une librairie développée indépendamment du compilateur et offrant des services aux programmes traduits. De tels services sont l'implémentation de l'opérateur MEM[$x$], les appels de pointeurs de fonctions, et toutes les procédures de la librairie C auxquelles le programme original aurait accès. Toutes ces fonctionnalités sont fournies par une classe Java, programmée en Java pur.

### 6.4.1 Accès à la mémoire

L'accès à la mémoire se fait grâce à l'opérateur MEM[$x$]. Cet opérateur possède plusieurs points d'entrées :

- `public int memoryLookup(int address)`
- `public int memoryLookup8(int address)`
- `public int memoryLookup16(int address)`
- `public void memorySave(int address, int value)`
- `public void memorySave8(int address, int value)`
- `public void memorySave16(int address, int value)`

Ces fonctions permettent de lire et d'écrire dans la mémoire, par mots de 8, 16 ou 32 bits. L'adressage mémoire n'existant pas en Java, il est nécessaire de gérer la mémoire à la manière d'un système d'exploitation, par des allocations de mémoire dans certaines zones. Le mécanisme de gestion de la mémoire est aussi responsable de régler l'état initial de la mémoire en fonction des pages définies dans le programme source.

### 6.4.2 Pointeurs de fonction

Les appels de fonctions virtuelles sont des appels de fonctions dans lesquels l'adresse de branchement n'est connue qu'au moment de l'exécution. Nous avons vu dans le chapitre précédant quelques pistes pour tenter de deviner quels sont les branchements possibles, mais cette méthode est incomplète. Le runtime Java doit donc posséder une liste indexée de toutes les procédures transformées en fonction de leur adresse virtuelle.
Lors d'un appel de procédure virtuelle, le code transformé appelle une méthode du runtime, en précisant l'adresse virtuelle de la procédure, ainsi que les arguments. Le runtime se charge, à travers les mécanismes d'introversion de Java, de rechercher la procédure à appeler et de lancer cette procédure.

### 6.4.3 Interception des erreurs

Il se peut qu'un bug du compilateur ait pour conséquence que le code généré ne soit pas l'équivalent de l'exécutable source. Une violation d'accès peut se produire, et le runtime se doit de donner des explications précises sur le contexte qui a généré l'erreur, afin que le compilateur puisse être débuggé et corrigé. Les erreurs pouvant se produire sont des erreurs de segmentation (un appel à MEM[$x$] illégal), une procédure virtuelle non trouvée, une procédure externe non trouvée, ...

Le runtime doit provoquer une exception lorsque cela est nécessaire. Ensuite, la partie du runtime exécutant le programme transformé (loader) doit réunir les informations nécessaires pour trouver le bug, qu'il soit dans le programme lui-même ou dans le transformateur. Ces informations peuvent être une stacktrace (liste des appels de procédures ayant abouti à l'exception), un désassemblage local de l'endroit où l'erreur s'est produite ou même l'adresse virtuelle de l'endroit correspondant dans le code Sparc.

### 6.4.4 Émulation de l'environnement

Lorsque nous avons transformé le programme, nous avons considéré que toutes les procédures externes ne devaient pas être transformées. Ces procédures externes sont pour la plupart des routines de la librairie C. Le code runtime doit donc être en mesure de fournir ces routines au programme. Plusieurs techniques, utilisables en même temps, sont applicables : l'implémentation manuelle des routines et l'interfaçage automatique.

#### Implémentation manuelle des routines

Dans cette méthode, chaque routine est réécrite en Java. Cela inclut les routines d'allocation de mémoire, les routines de manipulation de texte, d'entrée-sortie. Théoriquement, toutes les routines peuvent être traduites, mais cela nécessite beaucoup de temps et de patience.

Lorsqu'une nouvelle routine a été implémentée, elle est rajoutée dans la liste des routines externes disponibles. Cette liste contient, pour chaque routine, le nom de celle-ci, le nombre et le type des paramètres, et le type de la valeur de retour. Ainsi, cette liste est disponible pour que le compilateur puisse effectuer l'analyse inter-procédures dont nous avons parlée au point 4.3.3.

### Interfaçage automatique

Lorsqu'une routine ne peut pas être implémentée en Java, il existe une méthode alternative : l'interfaçage automatique sur des routines natives. Cette méthode exploite la capacité de Java à appeler du code natif (c'est-à-dire exécuté directement par le système d'exploitation). Lorsqu'une routine est nécessaire et ne possède pas d'implémentation manuelle, un interfaçage automatique à la routine du système peut être tentée. Cependant, cette méthode ne peut pas fonctionner pour toutes les routines, car les routines natives n'ont pas accès aux structures de la machine virtuelle. Tenter d'appeller nativement la routine d'allocation mémoire *malloc*() n'aurait aucun sens, car la mémoire allouée ne pourrait pas être exploitée par l'opérateur MEM[$x$].

# Chapitre 7

# Prototype

Le prototype conçu pour ce travail est une preuve de concept. Il est programmé entièrement en Java, et n'est capable de travailler que sur un nombre très réduits de programmes sources. Cependant, il permet de tester sans trop de difficulté les concepts énoncés dans ce travail. En effet, vu sa taille, son architecture est assez facile à comprendre et à modifier.

Le code source du prototype étant trop important pour être inclu dans les annexes, il est disponible sur le CD-Rom joint à ce travail.

## 7.1 Architecture

Le prototype contient 70 classes pour un total de 4500 lignes de code. Il est divisé en plusieurs packages :

**sparc2jvm** Programme principal, interconnectant les différents modules et interpretant la ligne de commande.

**elf** Analyseur ELF, le format de fichier exécutables de certains Unix (dont Solaris et Linux). Cette implémentation est minimale mais suffisante pour les besoins du transformateur.

**sparc** Ce package contient un parseur d'instruction Sparc, ainsi que le modèle objet des instructions Sparc.

**flow** Le package flow implémente l'algorithme pour réordonnancer les instructions et retirer les instructions du delay slot.

**semantic** Ce package implémente les algorithmes de transformation vers la forme intermédiaire, ainsi que les algorithmes d'analyse de flot de données.

**optimization** Le package optimization implémente les diverses optimisations disponibles

**back-end** Le package back-end contient les routines générant le code Jasmin.

L'analyse d'un programme se fait comme ceci :
Dans un premier temps, le programme est chargé par l'interface ELF. Cette interface analyse le binaire et extrait différentes informations : adressage mémoire utilisé (code exécutable, données statiques), nom des symboles et point

d'entrée.
Ensuite, toutes les instructions accessibles depuis le point d'entrée sont analysées et transformées en objets. Ces objets sont reliés entre eux pour former un graphe de contrôle de flot naïf. Ce graphe peut être dessiné avec l'outil *graphviz*.
Ensuite, l'algorithme de réordonnancement des instructions est exécuté sur le graphe de contrôle de flot. Il en résulte un nouveau graphe, qui peut lui aussi être dessiné.
Après cette étape, les procédures sont divisées en Basic Blocks. Ces Basic Blocks sont traduits en langage intermédiaire à l'aide de règles de traduction statiques. La traduction ne se fait que dans une seule classe, à l'aide du design pattern *visiteur*. Toutes les procédures sont ensuite optimisées en fonction des options choisies au lancement du programme. La main passe alors à l'un des deux disponibles. Le premier génère du code Jasmin, le second génère du code *graphviz*.

## 7.2 Utilisation

### 7.2.1 Compilation

Afin de compiler et utiliser le prototype, un environnement *JDK* est nécessaire. les outils *ant* et *graphviz* doivent être installés, ainsi que l'assembleur *Jasmin*. Tous ces outils sont disponibles sous Ubuntu Linux 8.10, qui a été utilisé comme plateforme de référence pour le développement. Il est probable que le prototype compile et fonctionne partiellement sous MS Windows, mais les appels aux programmes graphviz et jasmin fonctionneront probablement pas et devront être faits manuellement.

Après avoir décompressé le dossier *prototype*, contenant les fichiers sources du projet, se rendre dans ce dossier et exécuter la commande

```
$ ant compile
```

Si un message tel que celui-ci apparait, vérifiez que la variable JAVA_HOME pointe bien vers votre JDK : `Unable to locate tools.jar. Expected to find it in /usr/lib/jvm/java-6-openjdk/lib/tools.jar`.

### 7.2.2 Utilisation

La commande `./sparc2jvm.sh` permet de voir les différentes options disponibles :

```
$ ./sparc2jvm.sh
Sparc2JVM binary translation tool
Missing input file argument
Usage : Sparc2jvm [options] <sparc binary file>
 --backend <jvm,dot,cfg>          : Choose a back-end between JVM, dot intermedi
                                    ate, dot CFG
 --disable-nicenames              : Use original variable and register names for
                                     vars
 --entry <Symbol name or address> : Entrypoint of the main function
 --help                           : Print this message
 --optimize-copies                : Optimize copies propagation
 --optimize-deadcode              : Removes dead code
 --optimize-expr                  : Optimize expressions with constant parameters
 --optimize-flow                  : Remove redundant control flow
```

Le back-end jvm est le back-end produisant du code pour Jasmin. Le back-end cfg produit les graphiques du graphe de contrôle de flot naïf et modifié de chaque procédure. Le back-end dot produit un graphique de chaque procédure en forme intermédiaire, après les éventuelles optimisations. Ce back-end produit aussi toute une série de graphes de dépendances de variables, et dans chacun de ces graphes, les dépendances d'une seule variable sont dessinées.

Pour nettoyer le dossier courant de tous les fichiers produits par sparc2jvm, la commande `make clean` peut être entrée.
Les différentes options `-optimize-` servent à activer une des quatres optimisations disponibles : la propagation des copies, la suppression de code mort, l'optimisation de expressions constantes et l'amélioration du contrôle de flot.

Le dossier `testbed/` contient l'exemple de programme à analyser, ainsi que son code source et son desassemblage.

Afin de lancer le test, exécutez la commande `make`. Cette commande compilera la classe Testfib1.java, nécessaire à l'appel de la fonction transformée par sparc2jvm. Ce programme teste la fonction fibonacci traduite sur quelques valeurs, en chargeant la classe générée.

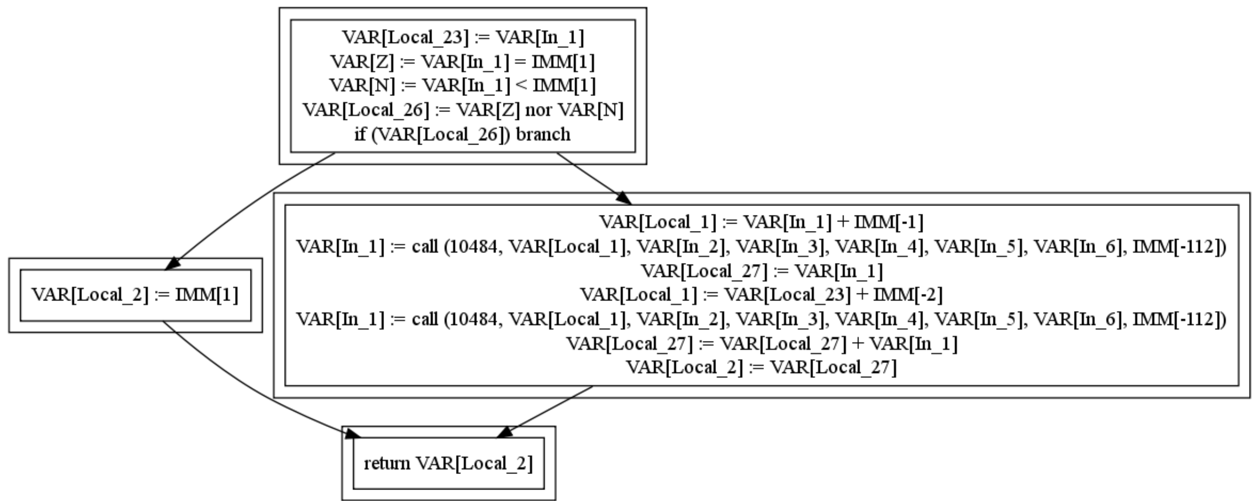


FIG. 7.1 – Résultat de la transformation du programme de test

## 7.3 Capacités et limitations

La figure 7.1 présente le résultat de la transformation du programme de test avec toutes les optimisations. Cependant, le défaut principal du prototype à l'heure actuelle est de ne pas supporter les appels de procédure externe, ni l'opérateur MEM[$x$]. De ce fait, la seule façon de vérifier le fonctionnement du transformateur est d'appeler, via une autre classe, directement la procédure traduite, et de vérifier la valeur de retour.

Il est évident qu'un tel comportement n'est pas acceptable pour un transformateur de binaire complet, mais dans le cadre du développement d'une preuve de concept, cela peut être suffisant pour démontrer que l'outil est réalisable.

# Chapitre 8

# Conclusion

## 8.1 Travail écrit

Le but premier de ce travail était de présenter les outils théoriques à l'élaboration d'un transformateur de binaires. Dans les chapitres qui précèdent, nous avons présenté l'architecture Sparc du point de vue du reverse engineering, c'est-à-dire dans l'idée de comprendre et de transformer le programme tout en gardant sa sémantique. Nous avons introduit un algorithme pour réordonner les instructions du delay slot afin de créer un graphe de contrôle de flot valide.

Nous avons présenté et documenté diverses formes intermédiaires, et défini une forme intermédiaire servant de référence pour notre implémentation. Nous avons créé des règles de traduction sémantiques pour traduire les instructions Sparc vers cette forme intermédiaire.

Nous avons ensuite énoncé et défini des notions classiques de compilateurs : les Basic Blocks, l'analyse de dépendances de données et l'analyse de vie des variables. Nous avons créé des algorithmes, certes peu performants, mais faciles à comprendre, concernant l'analyse de dépendances et de vie de variables, et montré comment ces algorithmes peuvent être appliqués à un code décompilé. Nous sommes ensuite allés plus loin en proposant des techniques pour effectuer de l'analyse et inférence de types de données sur base d'un programme sous forme intermédiaire, sans information de types de données. Nous avons énoncé une méthode pour récupérer les tableaux de sauts indexés, et nous avons proposé des méthodes pour optimiser l'analyse inter-procédures, ainsi que les appels de procédures virtuelles.

Au chapitre suivant, nous avons présenté différentes techniques d'optimisations pour améliorer la qualité du code transformé. Nous avons présenté des algorithmes pour la propagation de constantes, la suppression de code mort et l'évaluation statique, et avons présenté intuitivement la factorisation d'expressions ainsi que les optimisations des boucles.

Par la suite, nous avons présenté brièvement la machine virtuelle Java du

point de vue du développeur de compilateurs, énoncé les conventions de compilation afin de générer des programmes en Jasmin, l'assembleur pour la JVM, et donné des règles de traductions entre la forme intermédiaire et le langage Jasmin. Nous avons présenté brièvement la manière de concevoir des librairies runtime permettant au programme transformé de fonctionner parfaitement dans l'environnement de la machine virtuelle.

## 8.2 Prototype

Enfin, le deuxième objectif de ce travail était la réalisation pratique d'un prototype de transformateur de binaire. Nous avons réussi à produire un outil intéressant et instructif, capable de traduire de petits programmes, de visualiser graphiquement le résultat de la traduction, et surtout d'effectuer diverses optimisations sur la forme intermédiaire. L'objectif n'était pas de créer un outil professionnel et complet, capable de transformer des applications entières, car il aurait été difficilement possible de faire mieux que les outils existants comme UQBT, développé pendant plusieurs années par des experts dans le domaine.

## 8.3 Travaux futurs

Par souci de clarté et de brièveté, beaucoup d'éléments du travail ont été simplifiés. De nombreuses instructions Sparc sont omises, les appels systèmes n'ont pas été mentionnés, et les relations entre le binaire analysé et les librairies dynamiques n'ont pas été établies. Ces éléments seraient à prendre en compte dans d'éventuels travaux futurs, afin de rendre cette partie plus complète.

La forme intermédiaire choisie a été volontairement simplifiée. Il est possible de s'inspirer des formes intermédiaires utilisées dans des compilateurs de qualité pour définir des ajouts à la forme intermédiaire, et ainsi permettre une plus grande nuance dans la manière de traduire le code Sparc, ce qui peut permettre des analyses ou des optimisations plus poussées.
La forme intermédiaire étant relativement universelle, il serait possible de décrire des front-ends pour d'autres architectures, en voie de disparition ou non. En ce qui concerne l'analyse des pointeurs, beaucoup d'améliorations sont possibles, par exemple pour améliorer le mieux possible leur traduction sans passer par l'opérateur MEM[$x$].

Bien que les optimisations les plus courantes aient été présentées, de nombreuses possibilités s'offrent à nous pour faire des améliorations. Cependant, il y a plus de place dans le back-end pour les optimisations, car celui-ci, offrant une autre manière de représenter le programme, ouvre de nouvelles pistes d'optimisations.

En ce qui concerne le back-end, de nombreuses améliorations sont possibles. La traduction en Jasmin n'est pas toujours satisfaisante, par exemple en ce

qui concerne les opérateurs de comparaison et les opérations arithmétiques plus complexes, et la distinction entre les nombres signés, non signés, et de différentes tailles.
De plus amples recherches restent à faire au niveau des librairies runtime, afin de proposer des solutions automatiques plus efficaces et moins contraignantes.

Enfin, le prototype en lui-même mérite de recevoir du travail à tous les niveaux, afin d'accepter plus de programmes en entrée, et sa librairie runtime nécessite d'être développée afin de faire tourner de vrais programmes. Cependant, l'architecture interne du prototype est conçue de manière à rendre ce travail aisé, et donc le prototype pourrait devenir la base d'un travail ultérieur.

Pour conclure, nous sommes satisfaits des résultats obtenus, tant au niveau théorique qu'au niveau pratique, car ils prouvent que la réalisation d'un transformateur de binaires est possible, et ce avec le même ordre de grandeur d'efforts que ceux nécessaires à la réalisation d'un compilateur traditionnel.

# Bibliographie

[1] History of decompilation. http://www.program-transformation.org/Transform/HistoryOfDecompilation1, 2001.

[2] Boomerang decompiler. http://boomerang.sourceforge.net/, 2009.

[3] Datarescue interactive disassembler. http://www.datarescue.com/, 2009.

[4] Hex-rays decompiler. http://www.hex-rays.com/decompiler.shtml, 2009.

[5] The jasmin home page. http://jasmin.sourceforge.net/, 2009.

[6] Llvm compiler infrastructure. http://llvm.org/, 2009.

[7] Ltrace library call tracer. http://alioth.debian.org/projects/ltrace/, 2009.

[8] Pipeline (informatique). http://fr.wikipedia.org/wiki/Pipeline_(informatique), 2009.

[9] The python programming language. http://www.python.org/, 2009.

[10] Strace system call tracer. http://www.sourceforge.net/projects/strace/, 2009.

[11] Valgrind runtime analysis tools. http://ww.valgrind.com/, 2009.

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers : principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] Cristina Cifuentes. Reverse compilation techniques, November 19 1994.

[14] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Specifying the semantics of machine instructions. Technical report.

[15] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. a retargetable static binary translation framework. Technical report, April 05 2002.

[16] Joshua Engel. Programming for the Java Virtual Machine. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[17] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.

[18] S. Microsystems. The SPARC Architecture Manual (Version 8), 1990.

[19] Chevarista Team. Automated vulnerability auditing in machine code. http://www.phrack.org/issues.html?issue=64&id=8#article, 2007.

[20] Andrew w. Appel. Modern compiler implementation in Java. Cambridge University Press, Cambridge, United Kingdom, 1998.

[21] Renhard Wilhelm and Dieter Maurer. Compiler Design. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.