UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PREUVE DE VALIDITÉ DU VÉRIFICATEUR DE CODE OCTET JAVA

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

JAMAL LAZAAR

DÉCEMBRE 2008

REMERCIEMENTS

Je tiens à remercier les personnes qui, tout au long de mes années d'études et de recherche, m'ont permis d'atteindre les résultats qui sont présentés dans ce mémoire. Je remercie premièrement mes parents et tous les membres de ma famille pour leur soutien et leurs encouragements. Merci au directeur de mémoire, le professeur Étienne Gagnon, pour les orientations fournies, ce qui m'a permis de recadrer mon travail tout au long de ce mémoire me permettant d'aller de l'avant.

Je tiens à remercier ensuite les rapporteurs qui ont accepté de corriger ce mémoire et d'apporter un oeil nouveau et critique sur mon travail. Finalement, je remercie toutes les personnes que j'ai eu le plaisir de côtoyer durant mes années d'études à l'UQAM.

Table des matières

Liste des figures

LEXIQUE

MVJ :	Machine Virtuelle Java.
Fichier .class :	Fichier résultat de la compilation d'un fichier .java.
Offset :	Indice d'instructions.
Chargeur de classe :	Appelé aussi ClassLoader, c'est une classe de l'API Java dont le rôle est de charger les classes dynamiquement.
BootStrap :	Chargeur de classes qui fait partie intégrante de la MVJ et qui est responsable du chargement des classes de confiance (par exemple, la bibliothèque de classes Java).
Sous-routine :	Séquence d'instructions générées lors de la compilation d'un `try finally`.
Treillis :	Ensemble d'éléments munis d'une relation d'ordre.
Frame :	Structure de données composée d'une pile et de variables locales.
Frame secondaire :	Structure de données composée d'une pile et de variables locales dont le rôle est de mémoriser les types de l'instruction *jsr*.
API :	Interface de programmation (*Application Programming Interface*).

RÉSUMÉ

L'utilisation du langage Java dans plusieurs environnements (web, systèmes embarqués, systèmes mobiles, etc.) a élevé considérablement le niveau d'exigence envers ce langage, ce qui a amené les chercheurs et les développeurs à s'intéresser au système de sécurité de la Machine Virtuelle Java (MVJ) qui repose principalement sur le vérificateur du code octet.

Dans ce mémoire, nous expliquons le fonctionnement du vérificateur Java, son rôle, les différentes techniques proposées pour son implémentation et un algorithme que nous proposons comme alternative sérieuse aux autres vérificateurs qui existent déjà. Nous nous intéresserons plus particulièrement à l'effet des sous-routines sur le bon typage des instructions.

Nous présentons aussi une nouvelle approche de vérification de la synchronisation en nous basant sur l'analyse de flot de données et en identifiant les références qui pointent vers le même objet.

Mots clés : Machine Virtuelle Java, Code octet, Vérificateur, Synchronisation, Java, ClassLoader, Instructions, Treillis, Analyse de flot de données, Fonctions de transfert, Point fixe.

INTRODUCTION

Avec une importante utilisation du langage Java dans plusieurs systèmes informatiques, il est essentiel d'assurer le bon fonctionnement des systèmes qui utilisent cette technologie, leur intégrité, leur confidentialité et leur sécurité. Le langage Java est un langage orienté objet fortement typé, il se distingue par plusieurs caractéristiques comme le ramasseur de miettes (*Garbage collector*), les exceptions, l'implémentation de plusieurs interfaces et la possibilité de synchroniser des objets ou des méthodes. Comme indiqué dans la figure 0.1, un fichier Java est compilé par la commande `javac` qui génère le fichier `.class` qui peut être interprété par la machine virtuelle Java (MVJ) indépendamment de la plateforme utilisée.

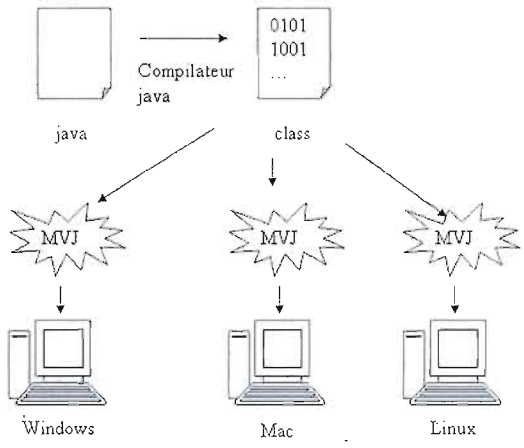


FIGURE 0.1 Portabilité du fichier .class

Cette caractéristique fait de Java un langage portable et idéal pour les systèmes mobiles et les applications web (applets, services web). Ceci conditionne ces environnements à exécuter le code octet en ignorant sa provenance et sa source ; par conséquent, ils ne peuvent ni savoir ce qu'il peut réaliser, ni ses vrais effets avant qu'il soit exécuté. À cette étape intervient le vérificateur du code octet Java qui a pour objectif d'assurer l'intégrité et la sécurité du système

qui exécute un fichier `.class`. Tel que montré dans la figure 0.2, après le chargement de la classe par la MVJ, le vérificateur s'assure que le fichier `.class` est bien valide.

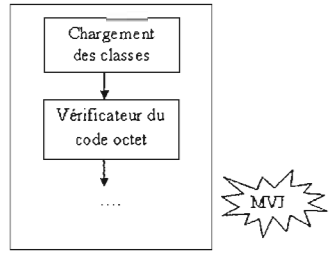


FIGURE 0.2 Portabilité du fichier .class

Le vérificateur est une composante importante dans l'architecture de sécurité de la MVJ. Son rôle est de simuler l'exécution du fichier `.class` et de prévoir toutes infractions des consignes de sécurité. Le vérificateur s'assure que le format du fichier `.class` est conforme à la syntaxe décrite dans les spécifications de la MVJ [LY99] et que le code octet de chaque méthode contenu dans le fichier `.class` ne met en péril ni l'intégrité de la MVJ, ni le système d'exécution.

De nombreuses recherches ont été effectuées sur le vérificateur du code octet Java. La plupart de ces travaux s'intéressent à la formalisation du système de vérification mais ils ne présentent ni preuve de validité en présence des sous-routines, ni modélisation complète des fonctions de transfert.

L'approche de Sun décrit d'une manière informelle les spécifications du vérificateur du code octet [LY99, Section 4.9.6] qui est formalisé dans plusieurs études comme celles de Stata [SA99], Qian [Qia99], Starck [RSB01] [SS03] et Freund [FM03]. Leurs approches se basent sur l'idée du calcul de l'ensemble des variables locales qui ne sont pas utilisées par la sous-routine. La difficulté à déterminer les instructions d'une sous-routine et la preuve de monotonie des fonctions de transfert a compliqué considérablement la présentation d'une formalisation complète.

Ce mémoire a pour but la réalisation d'un vérificateur du code octet efficace, muni d'une preuve formelle simple et valide en se basant sur une analyse de flot de données avec des fonctions de transfert monotones et distributives. Notre objectif consiste aussi à présenter une approche capable d'assurer d'une part le bon typage des instructions, d'autre part le bon fonctionnement de la synchronisation.

Les contributions apportées dans ce mémoire se résument comme suit :

- Présentation d'une preuve formelle complète du bon fonctionnement du vérificateur de type en se basant sur la théorie des ensembles et en traitant les sous-routines ;
- Conception d'une nouvelle méthode formelle de vérification de la synchronisation au niveau du code octet Java en présence des sous-routines ;
- Présentation d'une preuve mathématique simple basée sur la théorie des ensembles ;
- Implémentation des contraintes de chargement dans la machine virtuelle SableVM.

Nous signalons que les contributions suivantes sont le fruit d'un travail d'équipe avec M. Mathieu Corbeil :

- L'implémentation des contraintes de chargement.
- L'identification de la structure de données nécessaire («*frames* secondaires») comme piste de solution pour la vérification des types.

Nous nous sommes basés sur ce travail préliminaire pour identifier les étapes nécessaires de l'algorithme de vérification des types et pour présenter une preuve formelle de validité du vérificateur de types en présence de sous-routines.

Le reste du mémoire est structuré de la façon suivante :

- Le premier chapitre présente les notions mathématiques fondamentales pour la compréhension des algorithmes et propriétés utilisés pour valider les preuves formelles. Nous présentons la notion de treillis et une introduction à l'analyse de flot de données ;

- Le deuxième chapitre expose une description de l'architecture de la machine virtuelle Java ainsi que la place qu'occupe le vérificateur dans cette architecture ;

- Le troisième chapitre se concentre sur le fonctionnement du chargeur de classes (`ClassLoader`) et le rôle qu'il joue pour renforcer la sécurité de la MVJ ;

- Le quatrième chapitre se consacre à la présentation de notre vérificateur du code octet. Nous expliquons la problématique des sous-routines, notre algorithme de vérification et la preuve de sa validité ;

- Le cinquième chapitre est dédié à la présentation de notre méthode de vérification de la synchronisation au niveau du code octet. Nous expliquons la problématique des vérificateurs actuels avec la synchronisation, ensuite nous présentons notre algorithme de vérification de la synchronisation et la preuve de validité ;

- Le sixième chapitre présente un certain nombre de travaux effectués autour du vérificateur du code octet et de la vérification de la synchronisation ;

- Nous terminons avec une conclusion.

Chapitre I

NOTIONS MATHÉMATIQUES

Ce chapitre est un rappel et une présentation des théories et propriétés concernant l'analyse du flot de données utilisées dans ce document. Dans la première section, nous présentons les notions de relation d'ordre partiel et de treillis. Dans la deuxième section, nous présentons la notion du graphe de contrôle et le théorème du point fixe. Toutes les définitions et théorèmes sont extraits des documents écrits par Pottier [Pot00] et Jensen [Jen06].

1.1 Ordre partiel et treillis

Définition 1 (Relation d'ordre). *Une relation d'ordre $\subseteq$ sur E est une relation d'ordre partiel si elle est :*

- *Réflexive :* $\forall x \in E : x \subseteq x$;

- *Anti-symétrique :* $\forall x, y \in E \; x \subseteq y$ *et* $y \subseteq x \Rightarrow x = y$;

- *Transitive :* $\forall x, y, z \in E \; x \subseteq y$ *et* $y \subseteq z \Rightarrow x \subseteq z$.

Définition 2 (Borne supérieure et borne inférieure). *La borne supérieure de deux éléments, a et b, d'un ordre partiel, notée $a \cup b$, est le plus petit élément tel que :*

$$a \subseteq a \cup b \text{ et } b \subseteq a \cup b$$

De la même façon, la borne inférieure, $a \cap b$, est le plus grand élément tel que :

$$a \cap b \subseteq a \text{ et } a \cap b \subseteq b$$

Définition 3 (Treillis). *Un treillis est un ensemble E muni d'une relation d'ordre partiel, généralement notée $\subseteq$, tel que :*

- *Il existe un plus petit élément noté $\bot$ et un plus grand élément noté $\top$*

- *Pour tout élément a et b dans E, $a \cap b$ et $a \cup b$ appartiennent à E.*

Lemme 4. *Le produit de deux treillis E_1 et E_2 est un treillis muni de la relation d'ordre suivante :*

$$\forall\ (a_1, a_2), (b_1, b_2) \in E_1 \times E_2, \quad (a_1, a_2) \subseteq (b_1, b_2) \Leftrightarrow a_1 \subseteq b_1 \text{ et } a_2 \subseteq b_2$$

1.2 Analyse de flot de données

L'analyse de flot de données est un environnement d'analyse de programmes qui permet d'associer une propriété à chaque point du code. Si l'ensemble de ces propriétés constitue un treillis complet, alors toute suite croissante est convergente.

Ce type d'analyse se base sur une représentation du code sous forme d'un graphe dont chaque noeud représente une instruction, ce que l'on nomme graphe de contrôle (*Control Flow Graph*, CFG). Chaque noeud peut avoir plusieurs propriétés à l'entrée, ce que l'on note souvent par *in*, et une seule à la sortie que l'on note *out*. Chaque instruction a une fonction associée que l'on nomme fonction de transfert et qui modélise l'effet de l'instruction sur le *in* pour déterminer le *out*.
Dans un CFG l'ensemble des successeurs immédiats d'un noeud n est noté par $succ(n)$, de la même façon, l'ensemble des prédécesseurs immédiats de n est noté par $pred(n)$.

Définition 5 (Ffonction monotone). *Une fonction f est monotone si et seulement si :*

$$a \subseteq b \Rightarrow f(a) \subseteq f(b)$$

Définition 6 (Fonction distributive). *Une fonction f est distributive si et seulement si :*

$$f(a \cap b) = f(a) \cap f(b)$$

Théorème 7 (Point fixe). *une fonction monotone f sur un treillis possède un plus petit point fixe, qui est la limite de la suite $f^k(\perp)$, où $\perp$ est le plut petit élément du treillis. Si f est distributive alors ce minimum est unique.*

La condition de monotonie signifie qu'une meilleure information à l'entrée d'une instruction doit donner une meilleure information à la sortie.

Exemple :

Dans cet exemple nous présentons l'exemple d'un treillis, et de la fonction constante avec sa preuve de monotonie et de distributivité.

Soit T un ensemble composé de trois éléments a, b et c :

$$T = \{a, b, c\}$$

Alors, l'ensemble E, ensemble des sous-ensembles de T muni des relations d'union, d'intersection, d'inclusion et de l'ensemble vide $\emptyset$ est un treillis :

$$E = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

Soit f la fonction constante définie sur E comme suit :

$$f(x) = x$$

La fonction f est une fonction monotone et distributive. En effet, soient x et y deux éléments de E tel que $x \subseteq y$.

Or : $$f(x) = x \text{ et } f(y) = y \quad (*)$$

Alors :

$$f(x) \subseteq f(y)$$

D'où la monotonie de la fonction f.

D'autre part, par définition de la fonction f on a : $f(x \cup y) = x \cup y$

En utilisant la même équation (*), nous avons :

$$f(x \cup y) = f(x) \cup f(y)$$

D'où la distributivité de la fonction f.

En plus, chaque élément de E est un point fixe de cette fonction. Son plus petit point fixe est l'ensemble vide $\emptyset$ et son plus grand point fixe est l'élément $\{a, b, c\}$.

1.3 L'algorithme d'analyse de flot de données

L'algorithme d'analyse de flot de données consiste à insérer le noeud d'entrée du graphe dans une liste de travail, et tant que cette liste n'est pas vide, on en retire un noeud n_c et on recalcule sa propriété de sortie, si celle-ci a cru, alors on ajoute tous les successeurs de n_c à la liste de travail en vérifiant qu'ils n'existent pas déjà.

```
L : Liste de travail
M : Ensemble pour marquer les noeuds visités
n_c : Noeud courant

f_{n_c} : La fonction de transfert associée au noeud n_c

Ajouter la première instruction dans L

Tant que L ≠ ∅
  n_c := défiler L
  in(n_c) = ∪_{p∈pred(n_c)} out(p)
  out(n_c) = f_{n_c}(in(n_c))
  ajouter n_c dans M
  Pour tout s successeur de n_c :
    si s ∉ M alors ajouter s dans L
Fin Tant que.
```

Pour plus de détails veuillez voir les documents de Pottier [Pot00] et Appel [App02].

Chapitre II

ARCHITECTURE DE LA MACHINE VIRTUELLE JAVA

Ce chapitre présente la machine virtuelle Java, son fonctionnement et son architecture de sécurité. Dans la première section nous élaborons une description du fichier *.class* et du code octet Java. Nous enchaînons ensuite avec une deuxième section qui explique les composantes principales de la machine virtuelle Java et nous terminons avec la description de son architecture et du rôle du vérificateur du code octet.

2.1 Le fichier *.class*

Le fichier *.class* est un code binaire appelé code octet (bytecode) qui est souvent produit par le compilateur lors de l'utilisation de la commande *javac* ou édité par des éditeurs spéciaux comme *jasmine* [MB97].

Ce code n'est exécuté sur aucune machine, mais plutôt interprété par une machine virtuelle Java. Chaque instruction du code octet est lue, décodée, puis traduite en langage machine. Pour cette raison, l'exécutable Java est compilé une fois et exécuté sur différentes plateformes (PC, Linux, Mac), ce qui assure la portabilité du fichier *.class* (*compile once, run anywhere*).

Nous présentons dans ce qui suit la figure 2.1 qui décrit le format du fichier *.class*, où $u2$ représente deux octets et $u4$ quatre octets [LY99, Section 4.1].

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

FIGURE 2.1 Structure d'un fichier *.class*

Nous nous intéressons plus particulièrement au tableau `methods` qui contient toutes les méthodes de la classe, qui sont enregistrées dans une structure du type `method_info`, dont la description est dans la figure 2.2 :

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

FIGURE 2.2 Structure de `method_info`

2.2 Architecture de sécurité

La compilation d'un fichier Java peut produire plusieurs fichiers *.class*. Une application Java peut être exécutée soit sur un navigateur Web à travers des applets, soit dans des JDK (*Java Development Kit*), soit dans un environnement d'exécution comme JRE (*Java Runtime Environnement*).

La MVJ débute son exécution par le chargement (*loading*) de la classe contenant la méthode `main`, cela en utilisant le chargeur de classes (*bootstrap Loader*), ou bien un chargeur de classes défini par l'utilisateur. Ensuite, elle aborde la deuxième étape qui consiste à faire la liaison des liens qui se déroule elle-même en quatre étapes :

- **La vérification** : Elle consiste en la vérification des contraintes statiques et structurelles et la validation du format du fichier *.class*. Dans cette étape, nous proposons une analyse de flot de données pour vérifier les contraintes structurelles. On trouvera dans la section qui suit une description plus détaillée de ces contraintes.

- **La préparation** : Elle consiste en la création des champs statiques de la classe, en initialisant leurs valeurs par défaut.

- **La résolution** : Elle permet de transformer les références symboliques qui sont stockées dans le tableau des constantes (*Constantpool*) et qui peuvent causer d'autres chargements de classes,

- **Le contrôle d'accès** : Il consiste en la vérification des permissions et des droits d'accès à d'autres champs, méthodes ou classes.

Une fois que la liaison des liens s'est bien déroulée, l'initialisation prend place dans le processus d'exécution, qui consiste en l'initialisation des champs statiques, et en l'invocation des initialiseurs statiques (séquence d'instructions suivant la déclaration des champs, délimitée par des accolades et précédée par le mot `static`).

2.3 Contraintes de vérification

La machine virtuelle Java impose certaines règles pour s'assurer que le fichier *.class* est bien valide. Ces règles sont sous forme de contraintes statiques pour valider le format du fichier *.class* et sous forme de contraintes structurelles pour valider la syntaxe du code octet ([LY99, Section 4.8]).

2.3.1 Contraintes statiques

À part le fait que le fichier *.class* doit respecter le format défini par la figure 2.1, d'autres contraintes statiques sont imposées sur le code octet de la méthode :

- La longueur du tableau de code doit être comprise entre 1 et 65563 ;

- Le code (opcode) de la première instruction doit commencer à l'entrée 0 du tableau ;

- Tout code d'instruction (opcode) non cité dans les spécifications entraîne l'échec de la vérification ;

- L'index de chaque instruction, à l'exception de la dernière, est égal à la somme de l'index de l'instruction précédant et la longueur de ses opérandes ;

- L'index du dernier octet de la dernière instruction doit être égal à la longueur du code moins 1.

Citons dans ce qui suit quelques exemples de contraintes statiques concernant les opérandes des instructions :

- La destination de chaque instruction de branchement, comme `if` ou `goto`, doit être dans le tableau de code et doit être différente du opcode utilisé par l'instruction `wide` ;

- L'opérande de chaque instruction `getfield`, `putfield`, `getstatic`, et `putstatic` doit être une entrée valide dans le tableau des constantes ;

- L'instruction `invokespecial` est supposée invoquer seulement les méthodes d'initialisation `<cinit>` ;

- La dimension du tableau créée par l'instruction `anewarray` ne doit pas dépasser 255 ;

- L'opérande de chaque instruction `ilaod`, `flaod`, `aload`, `istore`, `fstore`, `astore`, `iinc` et `ret` doit être un entier positif inférieur au nombre maximum de variables locales utilisées par la méthode.

2.3.2 Contraintes structurelles

Présentons d'abord dans la figure 2.3 l'attribut code ([LY99, Section 4.7.3]) qui se trouve dans les attributs de la structure `method_info` et qui contient les instructions du code octet dans le tableau `code[code_length]`.

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

FIGURE 2.3 Structure de l'attribut code

- Chaque instruction doit comporter le bon nombre d'opérandes dans la pile. Par exemple, l'instruction *iadd* impose que la pile doit contenir au moins deux entiers au sommet ;

- Si une instruction est atteinte à travers plusieurs chemins avant son exécution, la pile doit avoir la même hauteur indépendamment du chemin d'exécution ;

- Aucune variable locale ne peut être accessible avant qu'elle ait une valeur assignée ;

- La hauteur de la pile d'opérande ne doit pas dépasser la hauteur maximale indiquée dans l'attribut code ;

- À n'importe quel point d'exécution, on ne peut pas dépiler plus de valeur que la pile contient ;

- Toute instruction `invokespecial` doit invoquer une méthode d'initialisation (le constructeur) de la classe courante ou bien de sa classe mère ;

- Lors de l'invocation d'une méthode d'initialisation, la pile doit contenir une instance

non initialisée à un endroit bien précis, souvent le sommet de la pile. De plus, cette méthode d'initialisation ne doit pas être invoquée sur une instance déjà initialisée ;

- On ne peut pas invoquer une instance de variable ou une méthode d'une instance qui n'est pas encore initialisée ;

- Lors d'un branchement arrière, ni la pile, ni les variables locales ne peuvent contenir une instance non initialisée. En plus, les variables locales ne peuvent pas contenir une instance non initialisée pour un code protégé par une exception ;

- Les arguments de chaque méthode invoquée doivent respecter la signature de la méthode invoquée ;

- L'instruction `athrow` doit lancer seulement une instance de la classe `Throwable` ou bien l'une de ses sous classes ;

- Le type `ReturnAdress` généré par les sous-routines et qui mémorise l'adresse de retour ne peut pas être chargé depuis une variable locale ;

- L'instruction qui suit `jsr` ou `jsr_w` doit être retournée par une seule instruction `ret` ;

- Les instructions `jsr` et `jsr_w` ne peuvent pas être utilisées pour un appel récursif d'une sous-routine ;

- Chaque instance du type `returnAdress` ne peut être utilisée qu'une et une seule fois.

2.4 Conclusion

Dans ce chapitre, nous avons présenté la structure du fichier *.class* et l'architecture de sécurité de la MVJ. En fait, la MVJ est un processeur virtuel qui a un jeu d'instructions propre. Afin qu'un fichier *.class* soit exécuté par la MVJ, il doit satisfaire deux types de contraintes : les contraintes statiques qui assurent la validité du format du fichier *.class* et les contraintes structurelles, dont le but est de vérifier la sémantique des instructions.

Chapitre III

RÈGLES POUR LE CHARGEMENT DE CLASSES

Le langage Java permet de charger dynamiquement des composants logiciels dans une machine virtuelle. Ce chargement est réalisé par des chargeurs de classes qui jouent un rôle fondamental dans l'architecture de sécurité de la machine virtuelle Java. En effet, il faut s'assurer de maintenir une cohérence des types chargés dans la machine virtuelle dans un contexte de chargement dynamique avec plusieurs chargeurs de classes pouvant être implémentés par des tiers dont la confiance n'est pas établie.

Ce chapitre explique le fonctionnement de la classe `ClassLoader` et notre implémentation de la vérification des contraintes de chargement décri dans la spécification de la machine virtuelle Java (section 5.3.4) et aussi par. Liang et Bracha [LB98]. La première section est dédiée à la présentation du problème du chargement des classes. La deuxième section présente notre solution. Dans la troisième section, nous présentons un certain nombre de travaux reliés. Finalement, nous clôturons avec une conclusion.

3.1 Fonctionnement du chargeur de classes

Le langage Java est remarquable par son processus de chargement de classes et supporte d'importantes fonctionnalités, comme par exemple le chargement paresseux. Il ne charge que les classes nécessaires, ce qui réduit l'utilisation de la mémoire. Un autre aspect important est l'extensibilité du chargement. En créant des classes qui héritent de la classe `java.lang.ClassLoader`, il implémente une politique personnalisée de chargement.

Un chargeur de classe est une instance d'une sous-classe de la classe abstraite `java.lang.ClassLoader`. Il charge le code octet d'une classe à partir d'un fichier *.class* et rend cette classe accessible à d'autres classes. Parmi ses méthodes les plus importantes, mentionons `loadClass()`

dont le principe de fonctionnement est schématisé dans la figure 3.1 et décrit comme suit :

1- Il fait appel à la méthode `findLoadedClass()` pour vérifier si la classe est déjà chargée ;
2- Il demande de charger la classe par un chargeur parent (`getParent()`) ;
3- S'il ne trouve pas un `ClassLoader` parent, il demande de charger la classe par le `Bootstrap ClassLoader` ;
4- En cas d'échec du chargement, il appelle la méthode `findClass()` pour charger la classe localement ou dans le réseau ;
5- En cas d'un nouvel échec, il lance une exception `ClassNotFoundException`.

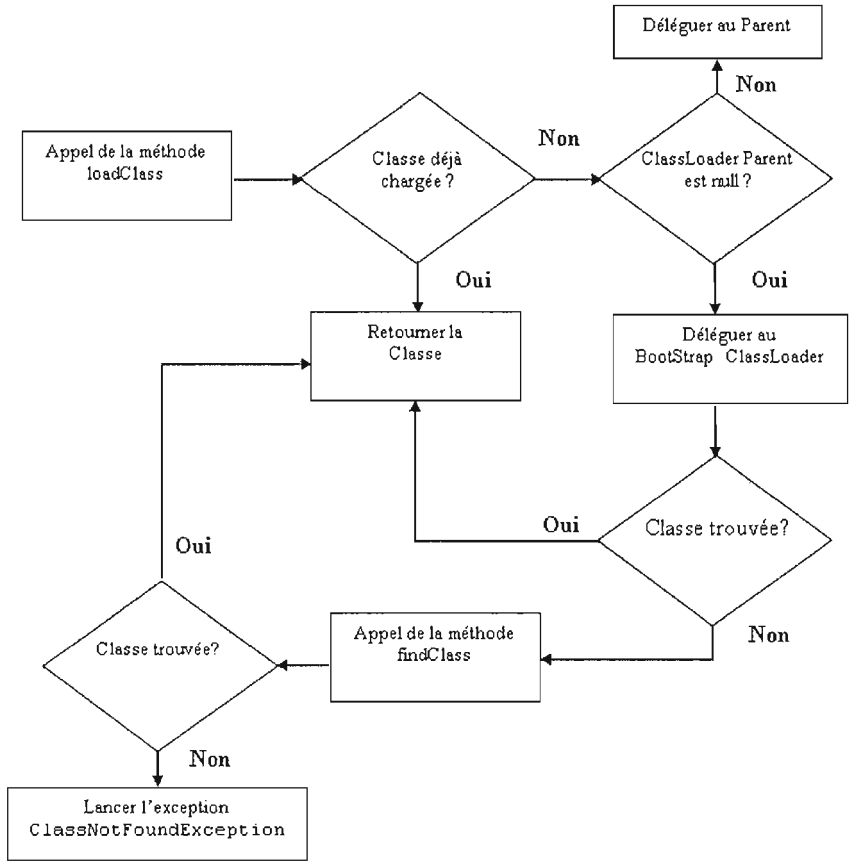


FIGURE 3.1 Principe de la méthode `loadClass`

Grâce aux chargeurs de classes, la MVJ peut gérer de multiples espaces de noms afin d'isoler les composants logiciels ; ceux-ci peuvent contenir des classes portant le même nom mais considérées comme des types distincts par la MVJ.

Les compilateurs produisent une représentation binaire du fichier source, le code octet, dans le format *.class*, normalement enregistré dans un fichier mais qui pourrait aussi être gardé soit en mémoire tampon, soit acheminé sur un réseau en utilisant les Applets, Servlets ou JavaBeans. La MVJ interprète le code octet, soit directement, soit en utilisant des techniques d'optimisation afin d'exécuter le code des méthodes des classes.

Le langage Java est basé sur des APIs qui sont des classes systèmes situées dans le `classpath` et chargées lors du démarrage de la MVJ par le chargeur de classes `Bootstrap`. Pour préserver les classes systèmes, les chargeurs de classes (dérivés de `java.lang.ClassLoader`) utilisent le mécanisme de délégation. Quand une classe D demande le chargement d'une autre classe C, le chargeur de classes de D vérifie qu'il n'a pas déjà chargé C. Si ce n'est pas le cas, il délègue le chargement à son parent. Si cette requête est un échec, alors L tente un chargement direct (par l'appel de la méthode `findClass()` comme nous verrons par la suite). Comme le mécanisme de délégation est récursif, toute requête de chargement d'une classe système sera transmise jusqu'au chargeur de classes `Bootstrap`.

3.2 Problématique

Le problème de sécurité du chargement de classes peut avoir de graves conséquences, telles que l'interruption de la MVJ, l'accès à des données encapsulées ou privées, ou même la prise de contrôle de l'ordinateur hôte par un ordinateur distant.

L'utilisation de plusieurs chargeurs de classes permet d'isoler les programmes qui s'exécutent concurremment. Ceci permet aussi d'éviter certaines attaques. En effet, pour des raisons d'efficacité, les classes sont chargées par la MVJ de façon paresseuse, c'est-à-dire qu'elles seront chargées seulement lorsqu'on en a besoin. De plus, un chargeur de classes ne recharge jamais une classe qu'il connaît déjà.

Nous présentons maintenant une faille classique dans le processus de sécurité du système de chargement [LB98]. Une attaque peut être effectuée de la façon suivante :
On s'arrange pour faire charger par la MVJ le programme d'attaque avant le programme attaqué. Dans le programme d'attaque, on inclut des classes qui portent le même nom que les classes

sensibles du programme attaqué, mais en ajoutant des portes dérobées. Par exemple, on peut rendre une variable qui est sensible publique. Quand le programme attaqué est chargé, il utilisera les classes déjà chargées. Le programme d'attaque peut alors utiliser ses portes dérobées pour obtenir des informations sensibles, modifier le comportement du programme attaqué, etc.

La figure 3.2 illustre la manière avec laquelle une incohérence de types peut être conceptualisée. Une ligne pleine indique une référence symbolique, un rectangle pointillé indique un espace de nom (Système, chargeur L_1 et chargeur L_2) et la ligne double indique une incohérence de type.

Imaginons un scénario où le `ClassLoader` L_2 est parent du `ClassLoader` L_1. On force l'application à utiliser le `ClassLoader` L_1 pour charger une version trafiquée de la classe `Spoofed` avec un attribut public, ensuite on crée une instance c de la classe `Spoofed`.

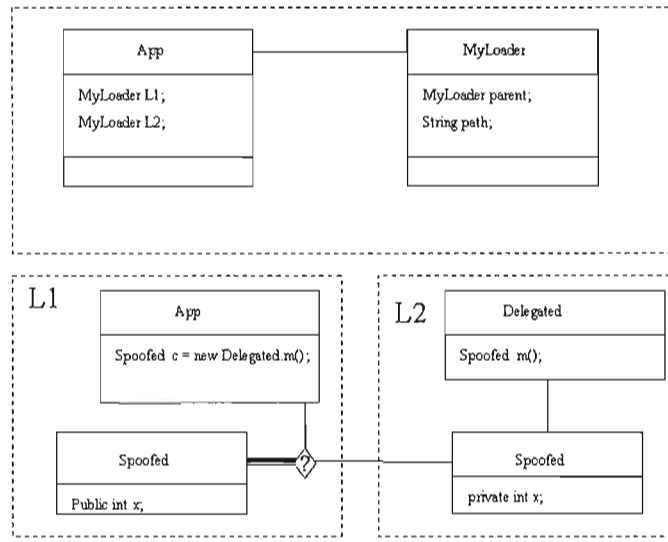


FIGURE 3.2 Schéma conceptuel d'une incohérence de types

Lorsque c invoque la méthode `m()` de la classe `Delegated` qui retourne le type `Spoofed`,

le chargeur L_1 délègue le chargement des classes `Spoofed` et `Delegeted` au `ClassLoader` L_2 qui chargera la vraie version de la classe `Spoofed` avec un attribut privé. Ce qui permet à l'objet c d'accéder à l'attribut `x` de `Spoofed` car l'attribut privé a été converti en accès public.

3.3 Solution

Nous allons représenter le type classe en utilisant la notation $< C, L_d >^{L_i}$, où C dénote le nom de la classe, L_d le `ClassLoader` qui a défini C (c'est-à-dire celui qui a chargé le code octet de la classe), et L_i le `ClassLoader` qui a initié le chargement de la classe (c'est-à-dire celui qui a provoqué le chargement du code-octet de la classe).

La solution que nous proposons repose sur l'article de Bracha [LB98]. L'idée principale est de créer pour chaque objet de type `ClassLoader` un ensemble dynamique qui mémorise les contraintes de chargement qui sont générées durant les phases de préparation et de résolution des classes.

Nous avons trois types de contraintes :

1- Lors de la préparation de chaque classe ou interface $< C, L_1 >$: Pour toute méthode g (dont la signature est de la forme $T_0\ g(T_1, ..., T_n)$) redéfinie d'une super-classe $< D, L_2 >$ on génère les contraintes suivantes :

$$\forall i \in \{0, ..., n\} \quad T_i^{L_1} = T_i^{L_2}$$

2- Lors de la résolution de chaque classe ou interface $< C, L_1 >$: Pour toute référence symbolique vers une méthode (dont la signature est de la forme $T_0\ \ g(T_1, ..., T_n)$) d'une classe $< D, L_2 >$ on génère les contraintes suivantes :

$$\forall i \in \{0, ..., n\} \quad T_i^{L_1} = T_i^{L_2}$$

3- Lors de la résolution de chaque classe ou interface $< C, L_1 >$: Pour toute référence symbolique vers un champ de type nommé T d'une classe $< D, L_2 >$ on génère les contraintes suivantes :

$$T^{L_1} = T^{L_2}$$

À chaque fois qu'on veut ajouter une contrainte à cet ensemble, on vérifie que toutes les classes qui sont déjà chargées par les `Classloaders` L_1 et L_2 vérifient bien cette nouvelle contrainte, sinon l'opération qui a lancé cette contrainte échoue. En appliquant cette solution à l'exemple de la classe `Spoofed` (Figure 3.2), lorsque on rajoute la contrainte $Spoofed^{L_1} = Spoofed^{L_2}$, on doit s'assurer que les deux classes `Spoofed` chargée par L_1 et `Spoofed` chargée par L_2 sont les mêmes.

3.4 Conclusion

Les contraintes de chargement constituent un élément important de la sécurité de bas niveau de l'architecture du langage Java. L'implémentation de cette fonctionnalité dans la machine virtuelle permet de garantir une cohérence de types lors de l'interprétation du code octet.

Nous avons présenté le concept de chargement dynamique des classes dans la MVJ, la vulnérabilité créée en présence de multiples chargeurs de classes et la manière avec laquelle la solution de Liang [LB98] a été implémentée.

Un autre intérêt de ce chapitre est la réutilisation du principe des contraintes de chargement pour effectuer une vérification de type paresseuse. En effet, lors de la vérification d'une méthode m(A a), au lieu de provoquer le chargement de la classe A pour vérifier certaine contrainte C, on pourra mémoriser cette contrainte dans le chargeur de classes et la vérifier une fois que la classe A est chargée.

Chapitre IV

VÉRIFICATION DES TYPES

Dans ce chapitre nous expliquons notre algorithme de vérification des types dans la MVJ. Afin de simplifier cet algorithme, nous l'avons décomposé en cinq étapes :

1. Associer chaque instruction `jsr` à une instruction `ret`.
2. Construire le graphe d'appel de sous-routines.
3. Calculer le point fixe de chaque instruction `ret`.
4. Calculer le point fixe de toute instruction.
5. Finalement, effectuer la vérification des types.

Ce chapitre se compose comme suit : La première section présente notre motivation, les raisons principales de nos choix et la problématique. Dans la deuxième section, nous présentons une description du vérificateur du code octet, son rôle et les cas qui compliquent sa réalisation. La troisième section, la plus importante, présente les cinq étapes de notre algorithme de vérification. Nous débutons par une vue globale de l'algorithme, suivi d'une description détaillée de chacune des étapes. Dans la quatrième section, nous présentons la preuve formelle du vérificateur, qui se base sur la preuve de monotonie et de distributivité des fonctions de transfert.

4.1 Motivation et problématique

Comme expliqué dans le deuxième chapitre, le vérificateur du code octet est une importante composante de la MVJ. Cette vérification est basée sur l'analyse du flot de données (*DataFlowAnalysis* DFA) qui propage les types dans le graphe de contrôle. Le vérificateur s'assure que les contraintes statiques et structurelles ne sont pas violées. La majorité des MVJ commerciales utilisent le vérificateur de Sun Microsystems. Par contre, plusieurs études, comme celle réalisée par Leroy [Ler03], ont démontré l'inefficacité de ce vérificateur et le rejet de certains codes Java valides principalement à cause des sous-routines, comme un des exemples cités dans l'étude réalisée par Stark [SS03] où le code octet généré par le JDK 1.3 ou 1.4 est rejeté par la majorité des vérificateurs comme JDK 1.3, JDK 1.4, Netscape 4.73-4.76, Microsoft VM pour Java 5.0 et 5.5 ainsi que le vérificateur de Kimera [EB97]. Un de ces exemples est présenté dans la figure 4.1 ; celui-ci a été refusé lors de l'exécution par Java HotSpot version 1.6.0.

```
class TestStark {
 void test(boolean b) {
   int i;
   try { i=1; }
   finally { if(b) i=2; }
   int j=i;
  }
}
java -verify TestStark
Exception in thread "main" java.lang.VerifyError :
(class : TestStark, method : test signature :
(Z)V) Register 2 contains wrong type.
```

FIGURE 4.1 Exemple de code Java refusé

En plus de ce type de bogues, il y a aussi l'inefficacité de certains vérificateurs et l'absence d'une modélisation et d'une preuve formelle qui tient compte de la présence de sous-routines. Ces problématiques nous ont motivé à présenter un environnement simple et bien fondé qui

se base sur des théories d'analyse de flot de données bien définies, lesquelles reposent sur des relations d'ensembles classiques ($\cup$ et $\subseteq$), ce qui nous permet d'obtenir des algorithmes efficaces qui restent polynomiaux dans le pire des cas.

Afin d'émuler l'effet du code octet d'une méthode, le vérificateur se base sur une analyse de flot de données et un treillis dont chaque propriété représente un environnement d'exécution formé de deux unités liées : les variables locales et la pile.

D'une part, les variables locales servent à mémoriser les types, afin qu'ils soient utilisés par les instructions du code octet. D'autre part, la pile est l'endroit où agissent ces instructions pour faire passer les paramètres ou pour récupérer le résultat. Les informations échangées entre la pile et les variables locales sont les types que nous allons définir ultérieurement.

L'une des contraintes les plus importantes lors du processus de vérification est la vérification du bon typage des instructions. D'un point de vue mathématique cela consiste à s'assurer que la fonction de transfert associée à chaque instruction reçoit les paramètres convenables. Par exemple, l'instruction *iadd* impose que la pile de l'environnement qu'elle reçoit en entrée contienne au moins deux éléments de type *int* sur la pile, comme dans l'exemple de la figure 4.2 avec un environnement composé d'une pile qui contient deux entrées s_0 et s_1 initialisées par le type u qui indique qu'elles sont inutilisées, et une variable locale v_0 initialisée par le type *int*. Lors de l'instruction *iload* 0 la pile contient le type *int* sur la pile dans les entrées s_0 et s_1. Aprés la simulation de l'instruction *iadd*, le type *int* sur la pile à l'entrée s_1 est consommé et remplacé par le type u pour indiquer que l'entrée s_1 n'est plus utilisable.

	s_0	s_1	v_0
état initial	u	u	*int*
iconst 0	*int*	u	*int*
iload 0	**int**	**int**	**int**
iadd	*int*	u	*int*
istore 0	u	u	*int*
return	u	u	*int*

FIGURE 4.2 Exemple de l'instruction iadd

Nous présentons un autre exemple dans la figure 4.3 qui contient un appel à une méthode

test(*String*) à travers l'instruction *invokestatic* qui reçoit un type bien précis sur la pile, de la classe *String*.

	s_0	s_1	v_0
état initial	u	u	int
iload 1	int	u	int
$ifeq\ 0\ L$	u	u	int
new A	A	u	int
dup	A	A	int
invokespecial < *init* >	A	u	int
goto H	A	u	int
L : *new B*	B	u	int
dup	B	B	int
invokespecial < *init* >	B	u	int
goto H	B	u	int
H : **invokestatic test(String)V**	**A ou B**	u	int
return	u	u	int

FIGURE 4.3 Exemple de l'instruction *invokestatic*

Dans ce dernier exemple, pour que ce code soit valide, nous devons vérifier que l'instruction *invokestatic* reçoive bien une valeur de la classe *String*. Dans ce cas, le vérificateur doit s'assurer que les deux classes *A* et *B* égales à la classe *String* ou héritent bien de celle-ci.

Suite à ces exemples, nous constatons que la vraie problématique du vérificateur est l'efficacité de l'algorithme de calcul des types qui atteint chaque instruction. Parmi les difficultés qui compliquent considérablement cette tâche notons la présence de sous-routines.

Une sous-routine est en réalité une séquence d'instructions qui commence à une adresse bien précise, que l'on va noter par *S*, et qui peut être appelée de n'importe quel endroit du code à travers l'instruction *jsr S*. Afin de retourner à l'instruction qui suit l'instruction *jsr*, l'adresse de retour est stockée dans une variable locale accédée par l'instruction *ret* qui est un des points de sortie de la sous-routine.

D'autres points de sortie de la sous-routine peuvent être causés par la présence d'exception

ou simplement à l'aide des instructions de branchement comme *goto* ou if qui peuvent mener vers des chemins qui ne contiennent aucune instruction *ret*, comme le cas de l'exemple ci-dessous.

```
Method void tryFinally(boolean b)

0 aload 0
1 jsr S
2 return
3 S : astore 1
4 iload 2
5 ifeq L
6 ret 1
7 L : invokevirtual 6
8 goto 2
```

FIGURE 4.4 Exemple d'une sous-routine

La technique la plus utilisée pour déterminer les types propagés est la technique de la fusion des types qui est présentée en détails par l'article de Leroy [Ler03] et qui se base sur le calcul de la classe commune la plus proche dans la hiérarchie d'héritage en considérant le fait que, toute classe hérite de la classe *Object*. Par exemple, si A et B sont deux classes qui héritent directement de C alors leur fusion est C. La perte de précision, la présence des types de base comme *int* et $float$, la possibilité qu'une classe implémente plusieurs interfaces et la présence de tableaux d'objets, tout cela complique la construction d'un treillis complet et la modélisation des fonctions de transfert monotones et distributives.

Le principe de notre approche est plus simple : nous gardons pour chaque instruction tous les types qui peuvent l'atteindre. Nous commençons par des ensembles singleton, enrichis au fur et à mesure que le DFA passe par les instructions. La figure 4.5 présente la simulation d'un code octet avec deux entrée dans la pile s_0, s_1 et une variable locale v_0. Lors de la simulation de l'instruction *invokstatic*, l'entrée de la pile consommée s_0 qui contient deux types possibles A et B.

	s_0	s_1	v_0
état initial	$\{u\}$	$\{u\}$	$\{int\}$
iload 1	$\{int\}$	$\{u\}$	$\{int\}$
ifeq 0 L	$\{u\}$	$\{u\}$	$\{int\}$
new A	$\{A\}$	$\{u\}$	$\{int\}$
dup	$\{A\}$	$\{A\}$	$\{int\}$
invokespetial <init>	$\{A\}$	$\{u\}$	$\{int\}$
goto H	$\{A\}$	$\{\}$	$\{int\}$
L : new B	$\{B\}$	$\{u\}$	$\{int\}$
dup	$\{B\}$	$\{B\}$	$\{int\}$
invokespecial <init>	$\{B\}$	$\{u\}$	$\{int\}$
goto H	$\{B\}$	$\{u\}$	$\{int\}$
H : **invokestatic test(String)V**	$\{A, B\}$	$\{u\}$	$\{int\}$
return	$\{u\}$	$\{u\}$	$\{int\}$

FIGURE 4.5 Exemple de types

Notre but est de construire pour chaque instruction le plus petit ensemble contenant tous les types qui peuvent atteindre cette instruction. Ce plus petit ensemble est le point fixe minimum de la fonction de transfert associée à cette instruction et que nous allons calculer à l'aide de l'analyse de flot de données. Dans l'exemple ci-dessus, le plus petit ensemble qui atteint l'instruction *invokestatic* à l'entrée s_0 est l'ensemble $\{A, B\}$.

4.2 Vue d'ensemble

Dans cette section nous donnons un aperçu du processus de vérification. Pour débuter, nous présentons les étapes nécessaires préalables à la vérification des contraintes statiques et structurelles. Ensuite, nous présentons les caractéristiques du graphe de contrôle utilisées pour notre analyse de flot de données. Puis, nous présentons l'analyse de flot de données, son treillis et ses propriétés. Nous terminons par un aperçu global du processus de vérification.

4.2.1 Étapes préliminaires

Avant d'entamer la vérification des types d'instructions, le vérificateur s'assure qu'aucune des contraintes statiques mentionnées au deuxième chapitre n'est violée.

Toutes ces vérifications sont effectuées lors de la lecture du fichier *.class*. À chaque fois que nous lisons un nombre d'octet attendu, nous nous assurons qu'il correspond bien au format décrit dans les spécifications de la MVJ. Par exemple, au début, nous lisons 4 octets et nous vérifions qu'ils correspondent bien au nombre magique $0xCAFEBABE$, et ainsi de suite.

Dès que le fichier *.class* est bien lu et que toutes les contraintes statiques sont vérifiées, alors, pour chaque méthode de la classe, nous procédons à la vérification des contraintes structurelles telle que cela est illustré à la figure 4.6.

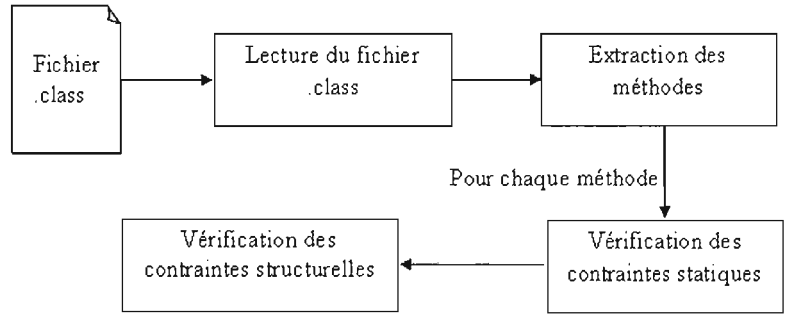


FIGURE 4.6 Étapes préliminaires de vérification des types

4.2.2 Le graphe de contrôle

Le graphe de contrôle G, sur lequel sont basées toutes les analyses présentées dans ce document, est un couple $G = (N, V)$, avec N l'ensemble de noeuds (instructions) du graphe et V l'ensemble des arêtes ($N \times N$). Les noeuds du graphe correspondent aux instructions du code octet et les arêtes aux transitions directes possibles entre deux instructions.

Le graphe est orienté et sa racine correspond à l'instruction dont l'*offset* est 0. Il existe un chemin d'exécution entre la première instruction et n'importe quelle autre instruction de la méthode. Cette condition est vérifiée par la majorité du code octet compilé. Toutefois, du

code mort peut exister sans provoquer un échec de la vérification. Les spécifications de la MVJ n'imposent pas la détection du code mort. Toutefois, sa présence peut refléter un code octet incorrect, et influencer certains algorithmes qui se basent sur un parcours séquentiel des instructions.

Afin de modéliser l'effet de chaque instruction, nous utilisons une famille de fonctions indexées par la hauteur de la pile et l'index des variables locales. Les instructions qui agissent seulement sur la pile, comme *iadd*, auront une famille de fonctions dont l'index est compris entre 0 et $h_{max} - 1$. Les instructions qui agissent seulement sur les variables locales, comme *iinc*, auront une famille de fonctions d'un seul index compris entre 0 et $e_{max} - 1$. Les autres instructions qui manipulent à la fois les variables locales et la pile auront une famille de fonctions indexées par deux paramètres qui indiquent respectivement la hauteur de la pile et l'index de la variable locale.

Nous devons signaler que le graphe de contrôle est construit durant la première étape de notre algorithme, au fur et à mesure que les instructions sont rencontrées.

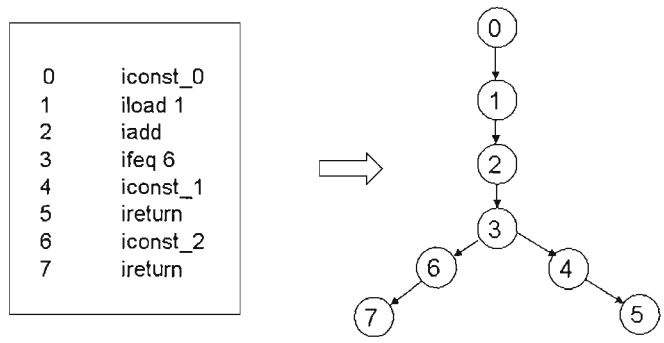


FIGURE 4.7 Exemple de graphe de contrôle

Dans l'exemple de la figure 4.7, on a trois types de branchement :

- Le successeur de l'instruction est l'instruction suivante, comme le cas des instructions 0, 1, 2, 3 et 4 ;

- Le successeur de l'instruction dépend d'une condition, comme c'est pour l'instruction 3 : si le sommet de la pile égal à zéro le successeur est l'instruction 6 sinon c'est l'instruction 4 ;

- L'instruction n'a aucun successeur puisque c'est une instruction de sortie du graphe, comme les instructions 5 et 7 (*ireturn*).

Pour plus de détails sur la construction du graphe de contrôle du code octet Java, on peut se référer au travail de Zhao citedepartment-analyzing.

4.2.3 Propriétés de l'analyse du flot de données

Dans cette section, nous présentons les propriétés de l'analyse de flot de données en nous basant sur les définitions décrites dans le livre d'Appel [App02].

Dans la suite de ce document, nous utilisons les notations suivantes :

$hmax$: la hauteur maximale de la pile,
$emax$: le nombre de variables locales,
$rmax$: le nombre maximal de sous-routines,
$m = hmax + emax$,

Définition 8 (Frame). *Un frame est une structure de données composée d'une pile de hauteur maximale hmax et de emax variables locales. Un frame contextuel est un frame qui sert principalement à garder les types qui atteignent une sous-routine et qui ne sont pas modifiés par celle-ci.*

La figure 4.8 présente un exemple d'un *frame* normal qui indique le contenu de chaque variable locale.

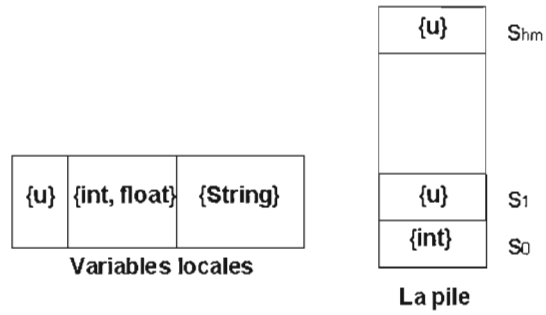


FIGURE 4.8 Exemple d'un *frame* normal

Définition 9 (in et out). *Le out d'une instruction est un frame résultat de la simulation de cette instruction sur son in. Le in est un frame résultat de l'union de tous les out des prédécesseurs de cette instruction.*

Soit n un noeud quelconque, f sa fonction de transfert et $pred(n)$ est l'ensemble des prédécesseurs de n, alors :

$$in[n] = \cup_{p \in pred(n)} out[p]$$

$$out[n] = f(in[n])$$

Pour la première instruction, son *in* prend en compte les paramètres de la méthode en cours de vérification : ainsi, si la méthode est statique et ne reçoit aucun paramètre alors son *in* est initialisé par des $\{u\}$ pour tous les emplacements de la pile et pour toutes les variable locales.

$$in(0) = (s_0, v_0)$$

Où :

$$s_0 = (\{u\}, ..., \{u\})$$
$$v_0 = (\{u\}, ..., \{u\})$$

Si la méthode reçoit p arguments $(arg_1, ..., arg_p)$, alors $in(0)$ est comme suit :

$$s_0 = (\{u\}, ..., \{u\})$$

$$v_0 = (\{arg_1\}, ..., \{arg_p\}, \{u\}, ..., \{u\})$$

Pour toute autre instruction différente de la première, chaque emplacement de son *out* est initialisé par l'ensemble vide. Ainsi, lors du calcul du *in* d'un noeud, même si on va pas encore visité tous ses prédécesseurs, nous allons garder les types de noeuds qui sont déjà visités, comme dans l'exemple de la figure 4.9 : Supposons que la hauteur de la pile est 2 et que nous avons une seule variable locale :

$$out(4) = (\{int\}, \{float\}, \{u\})$$
$$out(8) = (\{\}, \{\}, \{\})$$

Alors :

$$in(6) = out(8) \cup out(4) = (\{int\}, \{float\}, \{u\})$$

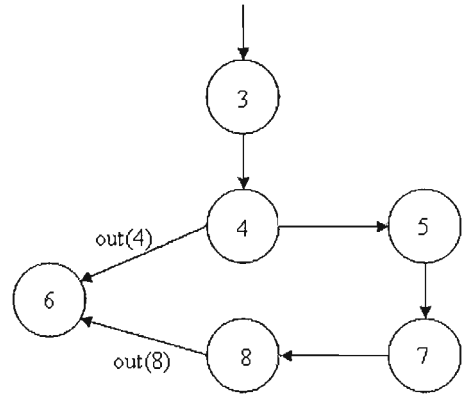


FIGURE 4.9 Exemple d'un noeud avec deux prédécesseurs

Dans la suite nous présentons tous les types générés ou manipulés par les instructions.

4.2.4 Le treillis de base

Soit T l'ensemble des types manipulés par les instructions de la méthode en cours de vérification, alors le treillis de base utilisé dans notre algorithme d'analyse du flotde données est le suivant :

$$L_0 = (E_0, \subseteq)$$

Chaque élément du treillis représente l'état de la pile et des variables locales pour une instruction donnée dans le code :

$$E_0 = \{(s, v) \mid s \in Pow(T)^{hmax} \; et \; v \in Pow(T)^{emax}\}$$

Et :

$$(s_1, v_1) \subseteq (s_2, v_2) \iff s_1 \subseteq s_2 \text{ et } v_1 \subseteq v_2$$

Où $Pow(T)$ est l'ensemble puissance de l'ensemble T.

La relation d'ordre de ce treillis est la relation d'ordre entre séquences d'ensembles. Ainsi, $(s_1, v_2) \subseteq (s_2, v_2)$ signifie que chaque composante s_{1i} de s_1 est un sous ensemble de T qui est inclus dans s_{2i}.

$$\forall i \in \{0, ..., hmax\} \text{ et } \forall j \in \{0, ..., emax\} \text{ on a } s_{1i} \subseteq s_{2i} \text{ et } v_{1j} \subseteq v_{2j}$$

Soit (s, v) un élément du treillis, alors :

$$s = (s_0, ..., s_{hmax-1})$$
$$v = (v_0, ..., v_{emax-1})$$

Chaque élément s_i représente une composante de l'élément s au i-ème rang, ce qui représente une entrée de la pile à la hauteur i ; de la même manière, v_j est une composante de l'élément v au j-ème rang qui représente dans ce cas le contenu de la variable locale dont l'indice est j.

L'élément minimum $\perp$ de ce treillis est le couple $(s_\perp, v_\perp)$ où chaque composante de $s_\perp$ et de $v_\perp$ est l'ensemble vide. D'autre part, l'élément maximum $\top$ est le couple $(s_\top, v_\top)$ où chaque composante de $s_\top$ et de $v_\top$ est l'ensemble T. De cette manière, nous pouvons constater grâce aux définitions et aux propriétés citées dans le premier chapitre que nous avons un treillis complet.

$$\forall x \in E_0, \Rightarrow (s_\perp, v_\perp) \subseteq x$$

$$\forall x \in E_0, \Rightarrow x \subseteq (s_\top, v_\top)$$

Afin qu'une analyse du flot de données basée sur ce treillis parvienne à la meilleure solution, il suffit d'avoir des fonctions de transfert monotones et distributives.

4.2.5 Les fonctions de transfert

Soient $x = (s, v) \in L_0$, l et h représentent respectivement l'index de la variable locale et la hauteur courante de la pile, $s = (s_0, ..., s_{hmax-1})$, $v = (v_0, ..., v_{emax+1})$, $hmax$ la hauteur maximale de la pile, et $emax$ le nombre de variables locales. Chaque fonction de transfert reçoit en paramètres un élément de E_o, comme indiqué dans le schéma de la figure 4.10. Nous présentons dans ce qui suit un exemple d'une fonction de transfert (Les autres fonctions sont présentées en annexe).

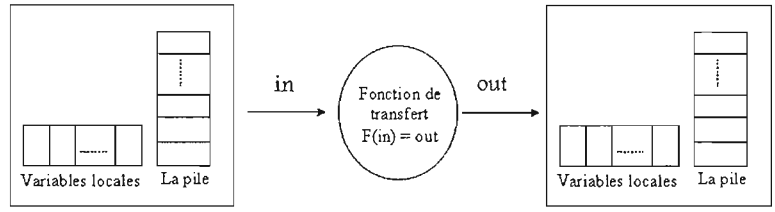


FIGURE 4.10 Fonction de transfert

4.2.5.1 Istore

Nous définissons dans cette section la fonction *istore* qui dépile le sommet de la pile et le charge dans la variable locale l :

$$istore_{h,l}(s, v) = (s', v')$$

Où :

$$s'_i = \begin{cases} s_i & \text{si } i \neq h \\ \{u\} & \text{si } i = h \end{cases}$$

$$v'_i = \begin{cases} v_i & \text{si } i \neq l \\ s_h & \text{si } i = l \end{cases}$$

Cette fonction reçoit en entrée une pile $s = (s_0, ..., s_h, ..., s_{hmax})$ et des variables locales $v = (v_0, ..., v_{emax})$, et en sortie la même pile, sauf pour l'emplacement à la hauteur h qui contient l'ensemble $\{u\}$ ce qui signifie que cet emplacement n'est plus utilisé. Les variables locales quant à elles sont les mêmes sauf pour la variable locale d'indice v_l qui contient l'ensemble des types qui était dans l'emplacement h de la pile, dans ce cas s_h. L'ensemble des fonctions de transfert est présenté à l'annexe A.

4.2.6 Processus de vérification

La plus grande difficulté rencontrée lors de cette étude fut la formalisation des fonctions de transfert, spécialement la fonction responsable de la propagation des types à l'instruction qui suit chaque `jsr` et la preuve de monotonie et de distributivité de cette fonction. Nous notons cette fonction de transfert par *jsrBis*. Cette complexité est due principalement au fait que cette fonction dépend de deux variables : le *in* du `jsr` qui représente les types qui atteignent l'instruction `jsr` et le *out* du `ret` qui représente les types propagés par le `ret` associé à cette `jsr`. Autrement dit, cela est dû à la difficulté de déterminer les types qui survivent ou qui naissent durant la l'exécution de la sous-routine.

Au début, notre approche consistait à modéliser la fonction de transfert `jsrBis` avec deux variables in_{jsr} et out_{ret}, mais la difficulté de prouver sa monotonie, sa distributivité, et la non appartenance de cette fonction à la même famille que les autres fonctions de transfert, nous a poussé à fixer la deuxième variable qui est le out_{ret} (troisième phase de l'algorithme), en effectuant une pré-analyse dont le but est d'analyser les sous-routines et savoir d'avance la forme des types qui vont atteindre l'instruction `ret` en parcourant seulement les instructions qui font partie de la sous-routine.

Pour atteindre notre objectif et pour prendre en considération le cas des sous-routines imbriquées, nous devons connaître l'ordre d'appel entre les sous-routines. Pour cette raison, nous avons consacré une étape afin de construire le graphe d'appel de sous-routines (2ème étape de l'algorithme). Ainsi, nous connaissons les types propagés d'une sous-routine appelée à la sous-

routine appelante. Avant de commencer l'analyse du graphe de contrôle pour calculer le graphe d'appel des sous-routines, nous devons déterminer toutes les instructions `ret`, leurs adresses de début et toutes les instructions `jsr` associées à chacune (première phase de l'algorithme). Une fois que nous avons le point fixe de chaque instruction `ret`, on peut déterminer les types qui seront propagés à chaque instruction qui suit une instruction `jsr` associée à une instruction `ret`. Par conséquent nous pouvons calculer le point fixe de chaque instruction. Dans la dernière étape, nous procédons à la vérification des types en nous assurant que chaque instruction a les bons types dans la pile et dans les variables locales. Toutes ces étapes sont illustrés par la figure 4.11.

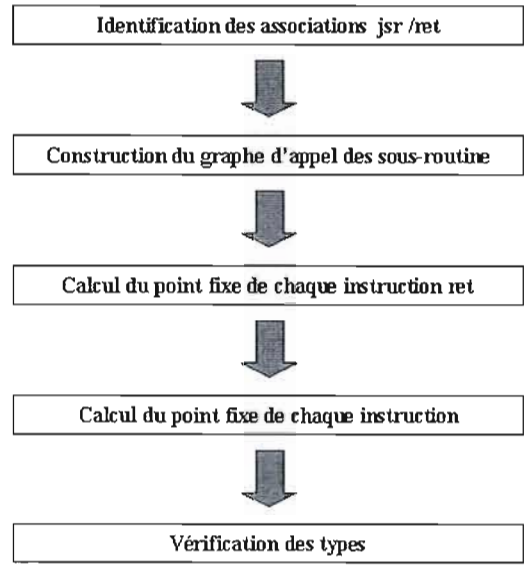


FIGURE 4.11 Les étapes de vérification des types

Toutes ces étapes se basent sur le même graphe de contrôle. Par contre, les propriétés du graphe de contrôle de la première étape sont différentes des autres puisque l'analyse du flot de données est basée sur le produit du treillis, à la différence des autres étapes qui se basent sur un seul treillis.

Dans les sections suivantes, nous expliquons les cinq étapes de notre algorithme en commençant par la première étape : l'association de chaque adresse de début d'une sous-routine à une éventuelle instruction `ret`.

4.3 Algorithme de vérification

4.3.1 Identification des associations jsr/ret

Le but de cette étape est de déterminer les sous-routines qui existent, leurs adresses de début et les instructions `ret` qui leurs sont associées.

Dans le but d'étudier l'influence de chaque sous-routine sur les types qui atteignent leurs adresses, nous associons à chaque adresse d'une sous-routine un *frame* contextuel.

public static void methodeName(int i)
0 : *jsr Y*
1 : *return*

Y : *astore* 0
iload 2
ifeq L
jsr Z
L : *ret* 0

Z : *astore* 1
ret 1

FIGURE 4.12 Exemple de code qui contient deux sous-routines

Dans l'exemple de la figure 4.12, en faisant un parcours du graphe de contrôle de ce code, nous pouvons constater qu'il y a deux adresses de sous-routines Y et Z, d'où le besoin de deux frames contextuels : le premier pour la sous-routine dont l'adresse est Y, et le second pour la sous-routine dont l'adresse est Z.

	s_0	v_0	v_1	v_2	s_0	v_0	v_1	v_2
jsr @Y	@Y	α_0	α_1	α_2	@Y	u	u	int
Y : *astore* 0	u	@Y	α_1	α_2	u	@Y	u	int
iload 2	α_2	@Y	α_1	α_2	int	@Y	u	int
ifeq L	u	@Y	α_1	α_2	u	@Y	u	int
L : ret 0	u	u	α_1	α_2	u	u	u	int
		@Y/ret 0						
return	u	u	u	int	u	u	u	int
jsr @Z	@Z	@Y	α_1	α_2	@Z	α_0	α_1	α_2
Z : *astore* 1	u	@Y	@Z	α_2	u	α_0	@Z	α_2
ret 1	u	@Y	u	α_2	u	α_0	u	α_2
							@Z/ret 1	

Lors de cette analyse, nous visitons chaque instruction une seule fois. Le fait qu'une sous-routine ne peut être associée qu'à une seule instruction `ret` et que les spécifications de la MVJ [LY99] imposent que deux sous-routines ne peuvent pas avoir la même instruction `ret` et que l'instruction `jsr` ne peut pas être associée à deux instructions `ret` assure l'unicité de l'association entre l'adresse de début de la sous-routine et l'instruction `ret`. Il se peut aussi qu'il existe des instructions `jsr` qui n'ont aucune instruction `ret` associée et qui vont être considérées comme des simples instructions de branchement.

Rappelons que l'instruction `jsr` empile l'adresse de l'instruction qu'elle la suit pour pouvoir y revenir lors de la sortie de la sous-routine. Dans notre approche, et pour avoir une bonne trace de l'adresse d'une sous-routine, on empile l'adresse de la sous-routine que nous notons S. Dans l'exemple ci-dessus, nous avons deux adresses $@Y$ et $@Z$.

Description de l'algorithme

Lors de la construction du graphe de contrôle, durant le traitement du noeud courant n_c, si l'instruction est un `jsr` alors nous remplaçons toutes les entrées du *frame* contextuel de la sous-routine associée à ce `jsr` avec des valeurs abstraites qui mémorisent leurs contenus, sauf pour l'emplacement de la hauteur courante de la pile (qui contient l'adresse de la sous-routine) et les emplacements au-dessus de la hauteur courante.

Si le noeud courant est une instruction `ret`, nous devons trouver dans la variable locale

de ce `ret` un seul élément qui est l'adresse de la sous-routine associée à ce `ret`. Dans le cas contraire le code sera considéré comme invalide.

Afin d'initialiser le *in* de la racine, nous prenons en considération les types reçus par la méthode en cours de vérification. En effet, si $in = (s, v)$ et les paramètres de la méthodes sont $t_1, ..., t_p$ où t_i un élément de T pour tout i dans $\{1, ..., p\}$, alors :

$$\forall i \in \{0, ..., hmax - 1\}\ s_i = \{u\}$$

Si la méthode n'est pas statique, alors :

$$\forall i \in \{1, ..., p\}\ v_{i-1} = \{t_i\}$$
$$\forall i \in \{p, ..., emax - 1\}\ v_i = \{u\}$$

Sinon :

$$v_0 = \{this\}$$

$$\forall i \in \{1, ..., p\}\ v_i = \{t_i\}$$

$$\forall i \in \{p + 1, ..., emax\}\ v_i = \{u\}$$

Où *this* est la classe qui contient cette méthode en cours de vérification.

4.3.1.1 Environnement

L'environnement de cet algorithme contient un *frame* contextuel pour chaque sous-routine. Pour connaître le nombre de frames secondaires, nous comptons le nombre d'adresses de branchement des instructions `jsr` durant la construction du graphe de contrôle.

Afin d'associer chaque adresse à son `ret`, nous utilisons un *frame* contextuel pour chaque sous-routine, et ce dans le but de ne pas perdre les types non modifiés par la sous-routine. Nous initialisons tous les composants du *frame* de celle-ci par des valeurs abstraites sauf pour l'emplacement de l'adresse de cette sous-routine. Ainsi, nous détectons tout nouveau type généré par la sous-routine.

Soient @ l'adresse de la sous-routine et h la hauteur courante de la pile. Alors, le *in* de la sous-routine est (s, v), où :

$$s = (\beta_0, ..., \beta_{h-1}, @, \beta_{h+1}, ..., \beta_{hmax-1})$$
$$v = (\alpha_0, ..., \alpha_{emax-1})$$

Pour connaître l'instruction `ret` de cette sous-routine, lors du `ret`, nous devons trouver un ensemble singleton qui contient l'adresse d'une sous-routine. Dans le cas contraire le code sera rejeté.

Le fait d'utiliser plusieurs *frames* engendre un treillis basé sur le produit de treillis L_0 :

$$L = (E, \cup, \cap)$$

Où :

$$L = L_0 \times ... \times L_0, \; rmax \text{ fois}$$
$$E = E_0 \times ... \times E_0, \; rmax \text{ fois}$$

$rmax$ est le nombre de sous-routines de la méthode en cours de vérification et E_0 le treillis de base. Chaque instruction est simulée sur tous les frames et chaque *frame* correspond à une sous-routine détectée dans le code.

Soit x un élément de E, alors la fonction de transfert $ASTORE$ de E dans E est définie comme suit :

$$ASTORE_{h,l}(x_1, ..., x_{rmax}) = (astore_{hl}(x_1), ..., astore_{hl}(x_{rmax}))$$

Nous décrivons dans ce qui suit l'algorithme qui associe chaque instruction `jsr` à une instruction `ret`.

```
Algorithme :

L : liste de travail
M : ensemble pour marquer les noeuds visités
n_c : noeud courant
f_{n_c} : fonction de transfert associée à l'instruction du noeud n_c
in(n_c), out(n_c) : deux éléments de E_0 × ... × E_0, rmax fois

Initialiser le in de la racine en fonction des paramètres de la méthode
Ajouter la première instruction dans L

Tant que L ≠ ∅
  n_c = défiler L
  in(n_c) = ∪_{p∈pred(n_c)} out(p)
  out(n_c) = f_{n_c}(in(n_c))
  traitement de n_c
  ajouter n_c dans M
  Pour tout s successeur de n_c :
    si s ∉ M alors
      ajouter s dans L
      ajouter l'arc n_c → s dans le graphe de contrôle
Fin Tant que.
```

Pour plus de détails sur la construction du graphe de contrôle et des successeurs veuillez vous référer à [Zha99].

Lors de cette étape, nous ne pouvons pas connaître l'ordre des appels entre les sous-routines. En effet, nous ne pouvons pas savoir si une instruction `jsr` fait partie ou non d'une sous-routine tant que nous n'avons pas encore visité l'instruction `ret` de cette sous-routine. Pour cette raison, nous consacrons l'étape suivante à la construction du graphe d'appel des sous-routines.

4.3.2 Construction du graphe d'appel des sous-routines

L'objectif de cette phase est la construction du graphe d'appel des sous-routines en faisant un parcours par arrière du graphe de contrôle. Durant cet algorithme nous utilisons une liste de travail initialisée en enfilant toute les instructions `ret` rencontrées lors de la première phase.

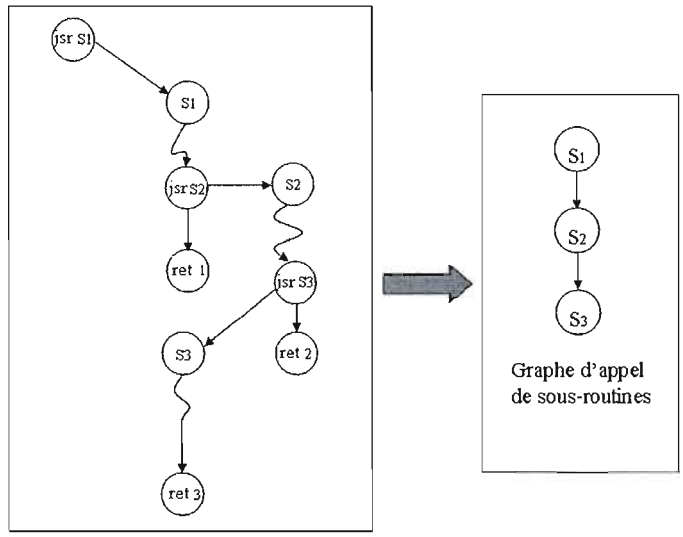


FIGURE 4.13 Exemple d'un graphe d'appel de sous-routines

La possibilité de se brancher sur une sous-routine et de sortir par une instruction de branchement comme *goto* ou if ou par une exception complique considérablement la construction du graphe d'appel des sous-routines. En utilisant un parcours normal, sur un chemin quelconque qui passe par le début d'une sous-routine, nous ne pouvons pas déterminer si une instruction fait partie ou non de cette sous-routine tant que nous n'avons pas visité le `ret` de cette sous-routine. C'est pour cette raison que nous avons décidé de parcourir le graphe de flot de données à l'envers, c'est-à-dire, au lieu d'insérer le successeur comme dans le cas de la première analyse, nous insérons le prédécesseur, et pour un code valide nous sommes sûrs d'atteindre l'adresse de la sous-routine lors de ce parcours. Dans la figure 4.13, en remontant depuis le noeud correspondant à l'instruction *ret* 1, nous rencontrons le noeud $jsr\ S_2$, ce qui indique qu'il y a un appel de la sous-routine d'adresse S_1 à la sous-routine d'adresse S_2, par conséquent nous ajoutons l'arc $S_1 \rightarrow S_2$ au graphe d'appel des sous-routines.

Dans l'exemple de la figure 4.14, en remontant de l'instruction ret_1, nous trouvons une instruction `jsr` qui dans ce cas représente un branchement normal puisqu'il n'y a pas d'instruction `ret` reliée. Par conséquent, nous ne rajoutons pas un arc dans le graphe des sous-routines.

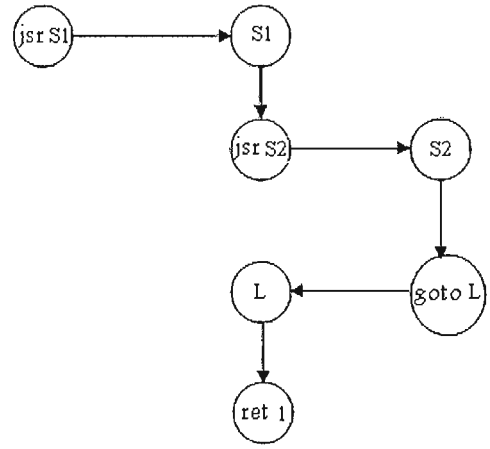


FIGURE 4.14 Branchement invalide à une sous-routine

Pour éviter ce cas, un arc d'appel entre deux sous-routines est ajouté si et seulement si on visite une instruction qui suit une instruction *jsr* et que l'adresse de cette sous-routine est reliée à une instruction `ret`.

Lors du calcul des appels d'une sous-routine vers d'autres sous-routines, nous marquons toutes les instructions visitées ce qui nous donnera l'ensemble des instructions qui font partie de chaque sous-routine.

Pour que l'algorithme visite une instruction qui ne fait pas partie de la sous-routine, un chemin doit être crée qui mènera à l'instruction `ret` sans empiler l'adresse de cette sous-routine dans la variable locale de cette instruction `ret`. Dans un tel cas, nous aurons une erreur de vérification. Comme le montre l'exemple de la figure 4.15, nous avons un branchement (*Goto L*) à l'intérieur de la sous-routine qui atteint le noeud *ret* 1 sans empiler l'adresse de la sous-routine dans la variable locale 1.

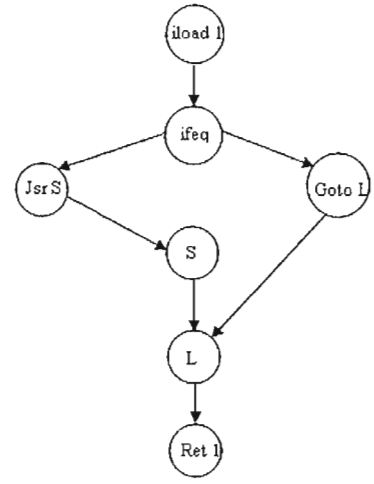


FIGURE 4.15 Exemple d'un branchement invalide à une sous-routine

4.3.2.1 Algorithme

Nous avons un seul *frame* à utiliser pour chaque *Ret*, seulement nous devons l'initialiser à chaque fois que l'on commence l'analyse. Soient :

R : Ensemble des instructions *ret* rencontrées durant la première phase.
L : Liste de travail
$ret_{c,@_c}$: Instruction ret courante
$@_c$: Adresse de la sous-routine associée à $ret_{c,@_c}$
n_c : Noeud courant
$n_@$: Noeud début de la sous-routine courante
$M(ret_c)$: Liste d'instructions associée au ret_c
G_s : Graphe des sous-routines
SR : Ensemble de toutes les adresses des sous-routines
$next_{jsr,@}$: Instruction qui suit l'instruction jsr @
$ret_@$: Instruction ret associée à la sous-routine dont l'adresse est @
n_0 : Le noeud qui correspond à la première instruction.

```
Tant que R ≠ ∅ alors
    enlever ret_{c,@_c} de R
    ajouter ret_{c,@_c} dans la liste L

    Tant que L ≠ ∅ alors
       n_c : enlever L
       si n_c = next_{jsr,@} alors ajouter jsr dans L si jsr ∉ M(ret_{c,@_c})
       sinon si n_c = jsr@ alors ajouter l'arc (@, @_c) dans G_s
       sinon si n_c = n_0 alors erreur
       si n_c ≠ @_c alors pour tout p prédécessur de n_c :
          si p ∉ M(ret_{c,@_c}) alors ajouter p dans L
          ajouter p dans M(ret_{c,@_c})
     Fin Tant que
Fin Tant que
```

Vérification de la récursivité des sous-routines

À la fin de cette étape, nous vérifions qu'il n'y a pas de cycles dans le graphe de sous-routines ce qui nous assure qu'il n'y a pas d'appels récursifs entre les sous-routines. Dans la figure 4.16, nous exposons le cas de deux sous-routines qui s'appellent mutuellement.

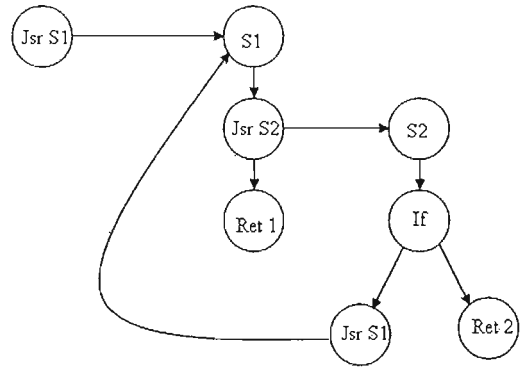


FIGURE 4.16 Exemple de deux sous-routines récursives

4.3.3 Calcul du point fixe des sous-routines

Cette phase a comme objectif le calcul du point fixe de chaque instruction `ret` en utilisant un seul *frame* et en commençant par les sous-routines les plus imbriquées, puisque l'ordre est important pour propager les bons types générés par les sous-routines.

L'initialisation du *in* de l'adresse de chaque sous-routine se fait de la façon suivante :

$$in(S) = (\alpha, \beta)$$
$$\alpha = (\alpha_0, ..., \alpha_{emax}) \text{ et } \beta = (\beta_0, ..., \beta_{h-1}, @, \beta_{h+1}, ..., \beta_{hmax})$$

Où : $\{\alpha_i\}_{1 \leq i \leq emax}$ désigne les variables locales, et $\{\beta_j\}_{1 \leq j \leq hmax}$ les emplacements de la pile, et h la hauteur de la pile du *in* de l'adresse S.

Nous sommes certains que le *in* du début de la sous-routine est le même dans tous les cas. Pour cette raison, nous avons restreint notre analyse seulement aux instructions qui font partie de la sous-routine et qui sont mémorisées lors de la deuxième étape. Cette analyse est initialisée par $in(S)$ en enfilant dans la liste de travail uniquement les instructions qui font partie de la sous-routine jusqu'à ce que soit atteint le point fixe de l'instruction *ret* associée à S.

Lors de la visite d'une instruction *jsr* associée à une instruction *ret* (figure 4.17), nous sommes certains que nous avons déjà calculé le *out* de ce *ret*, puisque nous ne pouvons pas

traiter une sous-routine avant de traiter toutes les sous-routines qu'elle appelle. Ce qui ne nous oblige pas à parcourir cette sous-routine appelée.

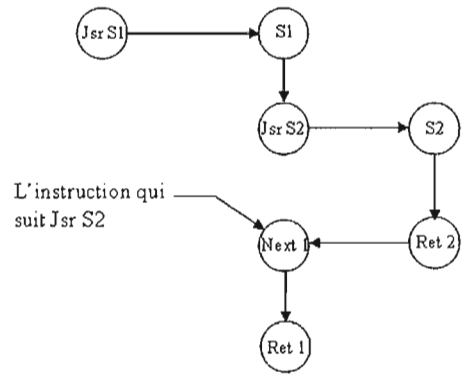


FIGURE 4.17 Appel entre deux sous-routines

Dans le cas de la visite d'une instruction *jsr* qui n'est associée à aucun des *ret* détectés durant la première étape, elle sera traitée comme une simple instruction de branchement qui met une adresse sur la pile.

Un des cas à traiter avec prudence est le cas où il y a un branchement à l'intérieur de la sous-routine vers un autre chemin qui ne mène pas au *ret* de cette sous-routine. Grâce à l'ensemble $M(ret)$ calculé durant la deuxième étape et qui mémorise toutes les instructions qui font partie d'une sous-routine, nous pouvons éviter que notre analyse du flot soit propagée aux autres instructions qui ne font pas partie de cette sous-routine. Pour cela, lors de l'analyse du flot, il faut insérer dans la liste de travail seulement les instructions qui font partie du $M(ret)$.

4.3.3.1 L'instruction qui suit le jsr : JsrBis

Lors de l'analyse du flot de cette troisième étape, nous allons utiliser les mêmes fonctions de transfert dont le domaine de départ est E_0, et qui sont définies dans la section 4.2.5.

Définissons dans ce qui suit la fonction de transfert qui propage les types générés par une sous-routine donnée à chaque instruction *next* qui suit chaque *jsr* qui se branche à cette sous-routine.

Remarque 10. *Soit e un élément de E_0, alors tout composant e_i de e peut être écrit sous la forme suivante :*

$$e_i = t_i \cup (\bigcup_{k \in I_i} \{\beta_k\}) \cup (\bigcup_{k \in J_i} \{\alpha_k\})$$

Où :

$$I_i, J_i \subseteq \{1, ..., m\}, \text{ et } t_i \in Pow(T) - (\alpha \cup \beta)$$

Définition 11 (Fonction $jsrBis$). *Soient $x = (s, v)$ et $y = (s', v') \in L_0$, @ l'adresse associée à cette instruction et r l'*offset *de l'instruction* `ret` *de cette sous-routine, tel que :*

$$r_i = t_i \cup (\bigcup_{k \in I_i} \{\beta_k\}) \cup (\bigcup_{k \in J_i} \{\alpha_k\})$$

Alors :

$$jsrBis_{@,r}(x) = (s', v')$$

Où :

$$\forall i \in \{0, ..., hmax - 1\} \;\; s'_i = t_i \cup (\bigcup_{k \in I_i} s_k) \cup (\bigcup_{k \in J_i} v_k)$$
$$\forall i \in \{0, ..., emax - 1\} \;\; v'_i = t_i \cup (\bigcup_{k \in I_i} s_k) \cup (\bigcup_{k \in J_i} v_k)$$

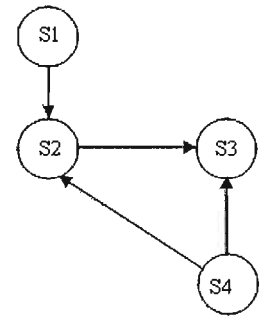
Nous présentons dans ce qui suit l'algorithme du calcul du point fixe de chaque instruction `ret` détectée durant la première étape.

4.3.3.2 Algorithme

En utilisant un algorithme de traitement de graphes, nous construisons une liste de travail ordonnée qui contient tous les noeuds du graphe des sous-routines de façon à ce que les noeuds les plus imbriqués soient traités en premier.

Exemple :

Si le graphe des sous-routines est comme ci-dessus, alors : $L = [S_3, S_2, S_4, S_1]$



Soient :

R : l'ensemble des instructions ret rencontrées lors
de la première phase.
L : liste de travail
$@_c$: l'adresse de la sous-routine courante
G_s : le graphe des sous-routines
ret_c : le ret de la sous-routine courante

Enfiler dans L les noeuds du graphe G_s dans l'ordre des plus imbriqués
au moins imbriqués
Tant que $L \neq \emptyset$ alors
    défiler $@_c$ de L
    calculer_point_fixe($@_c$)
Fin Tant que

PROCÉDURE *calculer_point_fixe*($@_c$) :

n_c : noeud courant
$pred(n_c)$: ensemble des prédécesseurs du noeud n_c
@ : adresse d'une sous-routine
f_{n_c} : fonction de transfert associée au noeud n_c
$jsr_{@,ret}$: instruction jsr d'adresse @ et reliée à une instruction ret
$W(ret_c)$: liste de travail
Insérer le noeud de l'adresse de la sous-routine $@_c$ dans la liste $W(ret_c)$.
Initialiser le *frame* de $@_c$

Tant que $W(ret_c) \neq \emptyset$ faire
    n_c : défiler $W(ret_c)$
    $old = out[n_c]$

    $in[n_c] = \bigcup_{p \in pred(n_c)} out[p]$

    Si n_c est une instruction qui suit un $jsr_{@,ret}$ alors
        $in[n_c] = in[n_c] \cup jsrBis(ret_@)$
    $out[n_c] = f_{n_c}(in[n_c])$
    Si $old \neq out[n_c]$ alors
      Pour tout successeur s de n_c
          si s $\in M(ret_c)$ alors ajouter s dans $W(ret_c)$
      Fin Pour
      si n_c est une instruction offset $jsr_{@,ret}$ alors
          ajouter l'instruction d'adresse offset+1 dans $W(ret_c)$
Fin Tant que

4.3.4 Calcul du point fixe de toutes les instructions

Cette phase consiste à calculer le point fixe de chaque instruction. Elle est basée sur une analyse normale du flot de données, qui se sert d'un seul *frame* et des mêmes fonctions de transfert que celles utilisées à l'étape du calcul du point fixe des instructions ret. Puisque le *out* de l'instruction ret est déjà calculé dans l'étape précédente, l'instruction *offset jsr S* a deux successeurs : l'adresse S et l'instruction qui suit dont l'*offset* est *offset+1*. D'autre part, l'instruction ret n'a aucun successeur dans cette analyse du flot.

4.3.4.1 Algorithme

n_c : noeud courant
$pred(n_c)$: ensemble des prédécesseurs du noeud n_c
@ : adresse d'une sous-routine
f_{n_c} : fonction de transfert associée au noeud n_c
$jsr_{@,ret}$: instruction jsr d'adresse @ avec une instruction ret reliée
L : liste de travail

```
Ajouter la première instruction dans L
Tant que L ≠ ∅
     enlever n_c de L
     old = out[n_c]
     in = ∪_{p∈pred(n_c)} out[p]
     si n_c est une instruction qui suit un jsr_{@,ret} alors
        in = in ∪ jsrBis(ret_@)
     out[n_c] = f_{n_c}(in[n_c])
     si old ≠ out[n_c] alors
         Pour tout successeur s de n_c faire
           si s ∉ L alors ajouter s dans L
           si n_c est une instruction offset jsr_{@,ret} alors
              si l'instruction offset+1 n'existe pas dans la liste L alors
                 ajouter l'instruction d'adresse offset+1 dans L
         Fin Pour
Fin Tant que
```

4.3.5 Vérification

Une fois que nous avons calculé le point fixe de chaque instruction, nous pouvons déterminer tous les types qui peuvent les atteindre. Ainsi, nous pouvons procéder à la vérification des contraintes structurelles définies dans le troisième chapitre.

Pour atteindre notre but, nous parcourons séquentiellement les instructions, en ne visitant que les instructions vivantes et en vérifiant les points suivants :
- Si l'instruction accède à la pile, il faut s'assurer de n'avoir que les bons types sur la pile ;
- Si on accède à une valeur dans une variable locale, il faut s'assurer que nous avons le bon type dans cette variable locale ;
- Nous vérifions aussi que les objets sont bien initialisés en les identifiant avant leur initialisation

par l'*offset* de l'instruction *new*. De cette façon on peut initilialser un objet ainsi que tous ses alias [Ler01, Section 4].

Il est à noter que certaines vérifications sont effectuées lors des étapes précédentes comme la récursivité des sous-routines qui est vérifiée durant la deuxième étape (construction du graphe des sous-routines), ou le débordement de la pile ou le dépilement d'une pile vide. Ces violations peuvent être détectées au niveau de l'étape du calcul du point fixe du `ret` (troisième étape), ou durant le calcul du point fixe de chaque instruction (quatrième étape).

4.4 Validité du DFA

Aucun des travaux d'analyse du flot de données utilisés dans la réalisation du vérificateur Java ne propose de preuve de validité ou de preuve formelle de la monotonie ou de la distributivité des fonctions de transfert. Notre étude se base simplement sur des propriétés standards de la théorie des ensembles pour démontrer que toutes les fonctions de transfert sont monotones et distributives. Ces deux dernières conditions sont suffisantes pour atteindre un point fixe (monotonie) et pour obtenir le meilleur point fixe (distributivité).

Les troisième et quatrième phases, qui permettent respectivement le calcul du point fixe de chaque instruction `ret` et le calcul du point fixe de toute instruction se basent sur une analyse du flot de données. Pour que ces deux analyses atteignent une meilleure solution, il suffit de montrer que les fonctions de transfert sont monotones et distributives.

Notation :
Dans cette section, on note par $\bigcup$ la relation d'union entre deux ensembles d'éléments quelconques.

Soient x et x' deux éléments de L_0, l et h représentant respectivement l'indice de la variable locale et la hauteur courante de la pile, tel que :

$$x = ((s_0, ..., s_{hmax-1}), (v_0, ..., v_{emax-1}))$$
$$x' = ((s'_0, ..., s'_{hmax-1}), (v'_0, ..., v'_{emax-1}))$$

4.4.1 istore

Monotonie :

Soit :

$$x \subseteq x'$$
$$istore(x) = (y, z)$$
$$istore(x') = (y', z')$$

Montrons que :

$$istore(x) \subseteq istore(x')$$

Par définition de la fonction $istore$, on a :

$$y_i = \begin{cases} s_i & \text{si } i \neq h \\ \{u\} & \text{sinon} \end{cases}$$

$$z_j = \begin{cases} v_j & \text{si } j \neq l \\ s_h & \text{sinon} \end{cases}$$

Et :

$$y'_i = \begin{cases} s'_i & \text{si } i \neq h \\ \{u\} & \text{sinon} \end{cases}$$

$$z'_j = \begin{cases} v'_j & \text{si } j \neq l \\ s'_h & \text{sinon} \end{cases}$$

Or, pour tout $(i, j) \in \{0, ..., hmax - 1\}$ et $\{0, ..., emax - 1\}$:

$$s_i \subseteq s'_i \text{ et } v_j \subseteq v'_j$$

D'où :

$$y_i \subseteq y'_i \text{ et } z_i \subseteq z'_i$$

Cette fonction est monotone, puisque chaque composante de $istore(x)$ est incluse dans $istore(x')$.

Distributivité :

Montrons que :

$$istore(x \cup x') = istore(x) \cup istore(x')$$

En effet, si $i \in \{0, ..., hmax - 1\}$ et $i \neq h$, autrement dit, si un emplacement de la pile est différent de la hauteur courante, alors :

$$istore(x)_i \cup istore(x')_i = s_i \cup s'_i$$

D'autre part :

$$istore(x \cup x')_i = s_i \cup s'_i$$

D'où :

$$istore(x \cup x')_i = istore(x)_i \cup istore(x')_i$$

sinon si $j \in \{0, ..., emax - 1\}$ et $j \neq l$, autrement dit, un emplacement d'une variable locale différent de la variable locale d'indice l, alors :

$$istore(x)_j \cup istore(x')_j = v_i \cup v'_i$$

D'autre part :

$$istore(x \cup x')_i = v_i \cup v'_i$$

Sinon si $i = h$, alors :

$$istore(x)_i \cup istore(x')_h = \{u\} \bigcup \{u\} = \{u\}$$

D'autre part :

$$istore(x \cup x')_h = \{u\}$$

Sinon $(j = l)$, alors :

$$istore(x)_l \cup istore(x')_l = s_h \cup s'_h$$

D'autre part :

$$istore(x \cup x')_l = s_h \cup s'_h$$

Dans tous les cas $istore(x)_i \cup istore(x')_i = istore(x \cup x')_i$, d'où la fonction $istore_{hl}$ est bien distributive.

4.4.2 jsrBis

Dans cette section nous présentons la preuve de la monotonie et la distributivité de la fonction de transfert $jsrBis_{ret}$ responsable du calcul du *in* de l'instruction $next_{jsr,@}$, l'instruc-

tion qui suit chaque `jsr` associée au `ret`. La figure 4.18 présente une abstraction du graphe de flux et l'arc de l'instruction *jsrBis* qui constitue une sortie de la sous-routine à travers l'instruction *ret*.

Dans le but de différencier les types qui sont générés durant la sous-routine et les types qui survivent durant celle-ci, nous notons le *out* du `ret` comme suit :

$$ret = (ret_1, ..., ret_m)$$

Et :

$$ret_i = T_i \cup \{\alpha_k\}_{k \in I_i} \cup \{\beta_k\}_{k \in J_i}$$

Où :

$\{\alpha_k\}_{k \in I_i}$: L'ensemble des types propagés par les variables locales du *out* du `jsr` et qui sont non modifiables durant la sous-routine,

I_i : L'ensemble des indices des α_i qui appartiennent à ret_i,

$\{\beta_k\}_{k \in J_i}$: L'ensemble des types propagés par les emplacements de la pile du *out* du `jsr` et qui sont non modifiables durant la sous-routine,

J_i : L'ensemble des indices des β_j qui appartiennent à ret_i,

T_i : Un sous-ensemble de T, qui contient les types générés lors de la sous-routine.

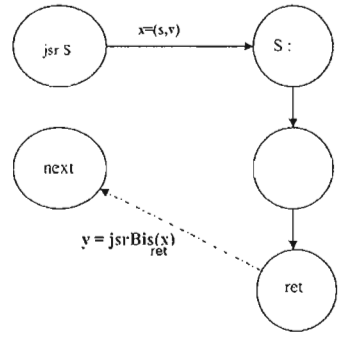


FIGURE 4.18 La fonction jsrBis

Soient x et y deux éléments du treillis L_0, qui représentent respectivement le *out* du `jsr` et le *in* de l'instruction `ret`, tel que :

$$x = (s, v)$$
$$y = (s', v') = (y_1, ..., y_m)$$
$$m = hmax + emax$$

Où :

$$s = (s_0, ..., s_{hmax-1})$$
$$v = (v_0, ..., v_{emax-1})$$

Alors, le *in* de l'instruction $next_{jsr,@}$ est calculé comme suivant :

$$jsrBis_{ret}(x) = y$$

Où :

$$y_i = T_i \cup (\cup_{k \in I_i} v_k) \cup (\cup_{k \in J_i} s_k)$$

Monotonie :

Soient x et x' deux éléments de L_0 tel que :

$$x \subseteq x'$$
$$x = (s, v),\ x' = (s', v')$$

Où :

$$s = (s_1, ..., s_{hmax})$$
$$v = (v_1, ..., v_{emax})$$
$$s' = (s'_1, ..., s'_{hmax})$$
$$v' = (v'_1, ..., v'_{emax})$$

Montrons que :

$$y = jsrBis_{ret}(x) \subseteq y' = jsrBis_{ret}(x')$$

Démonstration. Par définition de la relation d'inclusion, on a :

$$(s, v) \subseteq (s', v')$$

$$s \subseteq s'$$
$$v \subseteq v'$$

D'où : $\forall k \in \{1, ..., hmax\}$, et $\forall j \in \{1, ..., emax\}$, on a :

$$s_k \subseteq s'_k \text{ et } v_j \subseteq v'_j$$

Par conséquent, pour tout $i \in \{1, ..., m\}$, $\forall k \in \{1, ..., hmax\}$, et $\forall j \in \{1, ..., emax\}$, on a :

$$T_i \cup s_k \subseteq T_i \cup s'_k$$
$$T_i \cup v_j \subseteq T_i \cup v'_j$$

D'où, nous pouvons conclure que :

$$T_i \cup (\cup_{j \in I_i} v_j) \subseteq T_i \cup (\cup_{j \in I_i} v'_j) \quad (1)$$
$$T_i \cup (\cup_{k \in J_i} s_k) \subseteq T_i \cup (\cup_{k \in J_i} s'_k) \quad (2)$$

En faisant l'union des deux cotés des relations (1) et (2), on a :

$$T_i \cup (\cup_{j \in I_i} v_j) \cup (\cup_{k \in J_i} s_k) \subseteq T_i \cup (\cup_{j \in I_i} v'_j) \cup (\cup_{k \in J_i} s'_k)$$

D'où :

$$y_i \subseteq y'_i$$

□

Distributivité :

Soient x et x' deux éléments de L_0 tel que :

$$x = (s, v),\ x' = (s', v')$$

Où :

$$s = (s_1, ..., s_{hmax})$$
$$v = (v_1, ..., v_{emax})$$
$$s' = (s'_1, ..., s'_{hmax})$$
$$v' = (v'_1, ..., v'_{emax})$$

Montrons que :

$$jsrBis_{ret}(x \cup x') = jsrBis_{ret}(x) \cup jsrBis_{ret}(x')$$

Démonstration. En effet, soit :

$$x \cup x' = (s'', v'')$$

Avec :

$$s'' = (s_1 \cup s'_1, ..., s_{hmax} \cup s'_{hmax})$$
$$v'' = (v_1 \cup v'_1, ..., v_{emax} \cup v'_{emax})$$

Notons :

$$jsrBis_{ret}(x) = y$$
$$jsrBis_{ret}(x') = y'$$
$$jsrBis_{ret}(x \cup x') = y''$$

Alors :

$y''_i = T_i \cup [\cup_{k \in I_i} (v_k \cup v'_k)] \cup [\cup_{k \in J_i} (s_k \cup s'_k)]$

$= T_i \cup [(\cup_{k \in I_i} v_k) \cup (\cup_{k \in I_i} v'_k)] \cup [(\cup_{k \in J_i} s_k) \cup (\cup_{k \in J_i} s'_k)]$

$= T_i \cup [(\cup_{k \in I_i} v_k) \cup (\cup_{k \in J_i} s_k)] \cup [(\cup_{k \in I_i} v'_k) \cup (\cup_{k \in J_i} s'_k)]$

$= [T_i \cup (\cup_{k \in I_i} v_k) \cup (\cup_{k \in J_i} s_k)] \cup [T_i \cup (\cup_{k \in I_i} v'_k) \cup (\cup_{k \in J_i} s'_k)]$

$= y_i \cup y'_i$

□

4.5 Conclusion

Dans ce chapitre, nous avons élaboré un vérificateur flexible capable de traiter les sous-routines et de déterminer les types qui peuvent être propagés à l'instruction qui suit chaque instruction `jsr` qui appelle une sous-routine.

En décomposant le problème en plusieurs étapes, nous avons élaboré une preuve formelle de validité de cet algorithme de vérification des types dans la MVJ. Cette preuve est basée sur une analyse de flot de données dont chaque fonction de transfert est monotone et distributive, ce qui nous assure d'atteindre un point fixe minimum unique pour chaque fonction de transfert et par conséquent, pour chaque instruction, le plus petit ensemble de types qui l'atteint.

Chapitre V

VÉRIFICATION DE LA SYNCHRONISATION

Dans ce chapitre nous présentons l'algorithme qui a pour but d'assurer qu'aucune règle de synchronisation n'est violée dans une méthode. Dans la première section, nous présentons notre motivation et la problématique de la vérification de la synchronisation. Dans la deuxième section, nous donnons une vue globale de l'algorithme de vérification. Ensuite, dans la troisième section, nous présentons l'algorithme de vérification en détails. Au début de cette section, nous présentons une vue globale de l'algorithme, suivie d'une description du traitement des sous-routines. Dans la quatrième section, nous présentons les fonctions de transfert. Dans la cinquième section, nous présentons l'étape de vérification.

5.1 Motivation et problématique

Le langage Java est un langage qui supporte la programmation concurrente à travers l'utilisation de *threads* et des verrous. Le modificateur *synchronized* permet de synchroniser des méthodes et des blocs de codes. Lorsqu'une classe déclare une ou plusieurs zones synchronisées, un verrou mutex est créé par la machine virtuelle Java pour la classe ou pour son instance. Ce verrou doit être acquis par le *thread* pour accéder au code synchronisé.

Dans la machine virtuelle Java, la synchronisation des blocs de code est gérée par les instructions *monitorenter* et *monitorexit*. Pour mieux protéger l'intégrité de la MVJ, le vérificateur doit s'assurer que les opérations de verrouillage ne mettent pas en danger la MVJ.
Comme illustré dans l'exemple de la figure 5.1, la synchronisation d'une variable se produit en chargeant celle-ci sur la pile et en la verrouillant avec l'instruction *monitorenter*. Avant de sortir du bloc synchronisé, la même variable stockée quelque part dans une variable locale est chargée sur la pile et un *monitorexit* est exécuté pour relâcher le verrou. Nous constatons aussi que

pour un code Java généré par un compilateur fiable, il y a toujours un bloc de code octet pour traiter les exceptions soulevées dans le bloc de code octet synchronisé pour relâcher le verrou avant de sortir de la méthode à travers l'instruction *monitorexit*.

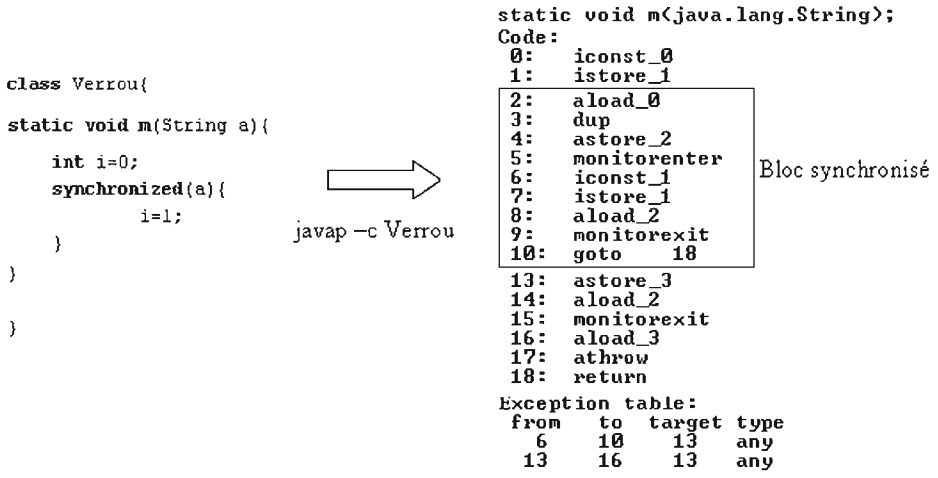


FIGURE 5.1 Exemple d'un code synchronisé

Dans l'exemple de la figure 5.2, nous avons remplacé l'instruction *monitorexit* par une simple instruction *pop* qui fait dépiler le sommet de la pile. Nous avons vérifié ce code octet avec le vérificateur du code octet Bcel du projet Apache sans que celui-ci soit capable de détecter cette erreur. Par contre, lors de l'exécution du code octet, nous avons eu une exception de type `IllegalMonitorStateException`.

```
void m(java.lang.String);
  Code:
   0:   iconst_0
   1:   istore_2
   2:   aload_1
   3:   dup
   4:   astore_3
   5:   monitorenter
   6:   iconst_1
   7:   istore_2
   8:   aload_3
   9:   pop
  10:   goto    20
  13:   astore  4
  15:   aload_3
  16:   pop
  17:   aload   4
  19:   athrow
  20:   return
 Exception table:
  from   to  target type
    6    10    13   any
   13    17    13   any
```

FIGURE 5.2 Exemple d'un code mal synchronisé

Notre but est de mettre en place un mécanisme intégré dans le vérificateur du code octet de la MVJ pour s'assurer qu'aucun code mal synchronisé ne puisse être exécuté par la MVJ.

Dans le but de vérifier la bonne structure de la synchronisation dans un code octet Java, nous devons identifier toutes les instances et leur alias au niveau de chaque noeud du graphe de flot de données. Ainsi, lors du verrouillage d'une instance quelconque, nous pouvons localiser tous ses alias et, par conséquent, lors du déverrouillage d'une instance déjà verrouillée, nous pouvons modifier tous ses alias pour qu'ils deviennent déverrouillés. L'idée principale est donc de rassembler les indices de la pile et des variables locales qui contiennent la même instance.

```
public  static void  method(A a)
    Hauteur maximale de la pile : 3
    Nombre de variables locales : 2

 0 aload_0          // Empiler l'instance a0
 1 astore_1         // Stocker le sommet de la pile dans
                    // la variable locale 1
 2 aload_1          // Empiler le contenu de variable locale 1
 3 dup              // Dupliquer l'opérande sur le haut de la pile
 4 monitorenter     // Verrouiller l'instance a0
 5 new A            // Créer un objet de A: a4
 6 dup              // Dupliquer l'opérande sur le haut de la pile
 7 invokespecial    // Appeler son constructeur A.<init>
 8 aload_1          // Empiler le contenu de variable locale 1
 9 monitorexit      // Déverrouiller l'instance a0
10 return           // sortir de la méthode
```

FIGURE 5.3 Exemple de *monitorenter* et *monitorexit*

Dans l'exemple de la figure 5.3, lors de l'instruction 8 *aload_1* nous pouvons distinguer les instances verrouillées $A(4)$ de celles qui ne le sont pas, même si elles représentent la même classe. les figures 5.4 et 5.5 expliquent le processus de verrouillage et déverrouillage d'une référence.

	S_0	S_1	S_2	V_0	V_1
0 aload_0	A	u	u	A	u
1 astore_1	u	u	u	A	A
2 aload_1	A	u	u	A	A
3 dup	A	A	u	A	A
4 monitorenter	A(4)	u	u	A(4)	A(4)
5 new A	A(4)	A	u	A(4)	A(4)
6 dup	A(4)	A	A	A(4)	A(4)
7 invokespecial	A(4)	A	u	A(4)	A(4)
8 alaod_1	A(4)	A	A(4)	A(4)	A(4)
9 monitoexit	A	A	u	A	A
10 return	A	A	u	A	A

S_0, S_1, S_2 : les emplacements de la pile
V_0, V_1 : les variables locales

FIGURE 5.4 Tableau d'alias

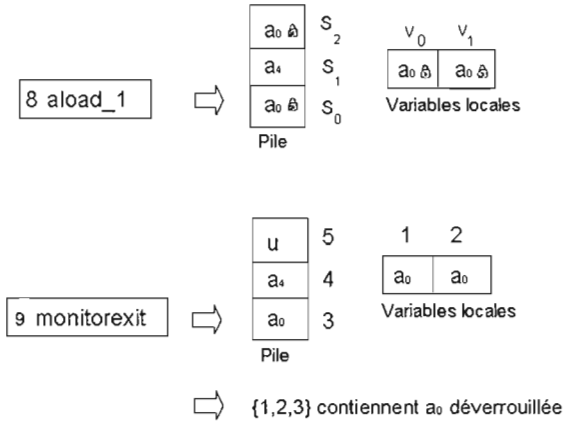


FIGURE 5.5 Exemple d'alias

Lors de cette vérification, nous supposons que la vérification de types a préalablement été effectuée. Nous utilisons aussi le résultat des algorithmes du chapitre de vérification des types, comme les relations entre *jsr* et *ret* et le graphe de sous-routines.
En suivant la même logique que celle de la vérification des types, notre environnement se compose d'un *frame* de base qui représente les variables locales et une pile d'opérandes.

5.2 Vue d'ensemble

5.2.1 Étapes préliminaire

En suivant la même logique que celle de la vérification des types, l'algorithme de vérification de la synchronisation se déroule en trois étapes :

1. Calcul du point fixe de chaque instruction *ret* ;
2. Calcul du point fixe de chaque instruction ;
3. Vérification de la synchronisation.

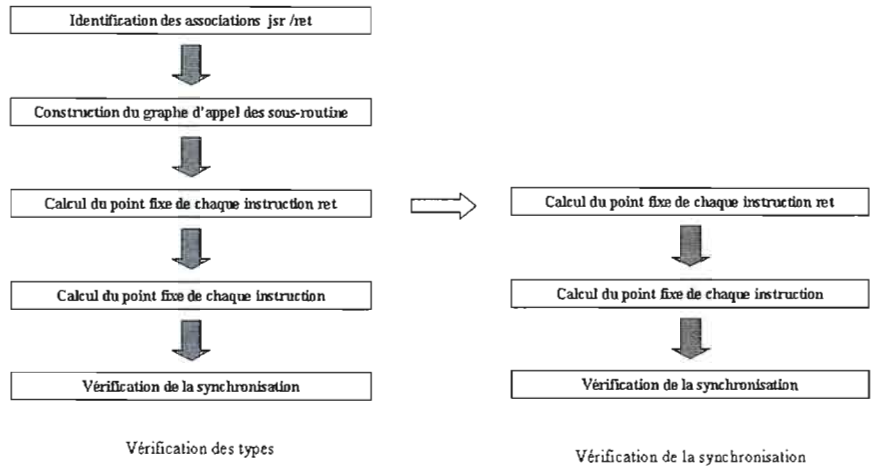


FIGURE 5.6 Les phases de la vérification de la synchronisation

Les trois étapes présentées dans ce chapitre illustrées par la figure 5.6 s'exécutent après celles de la vérification des types. Le treillis et les fonctions de transfert sont modifiés pour modéliser l'effet de la synchronisation.

Dans le but de définir les fonctions qui prennent en compte l'$offset$ des instructions *monitorenter* et l'ordre dans lequel ces instructions sont effectuées, nous devons définir un ensemble de permutations de l'ensemble de ces $offset$.

Définition 12 (Pile des instructions *monitorenter*). *Soit O l'ensemble d'$offset$ des instructions monitorenter, dont le cardinal est égal à d, et k un entier strictement positif inférieur ou égal à d. Nous notons par $P(O,k)$ l'ensemble de k-tuples de toutes les combinaisons possibles d'éléments de O tel qu'il n'y aura pas de répétition dans le tuple.*

Exemple :

Si l'ensemble des $offset$ est $O = \{1,2,3\}$ alors :

$P(O,2) = \{(1,2),(2,1),(2,3),(3,2),(1,3),(3,1)\}$

$P(O,1) = \{(1),(2),(3)\}$

$P(O,0) = \emptyset$ (ce qui représente le cas normal où il n'y a pas de verrous)

Dans la suite, l'ensemble P défini ci-dessous, représente tous les tuples possibles des $offsets$ des instructions *monitorenter* :

$$P = \bigcup_{k=0}^{card(O)} P(O,k)$$

Où $card(O)$ est la cardinalité de l'ensemble O (le nombre d'éléments de O).

5.2.2 Construction du treillis L_0

Définition 13 (Instances abstraites et treillis). *Les instances contenues dans la pile ou dans les variables locales sont réduites aux éléments suivants que nous appelons les instances abstraites :*

$\emptyset$ *: instance quelconque*
$\emptyset_{[p]}$ *: instance verrouillée par une ou plusieurs instructions monitorenter dont la pile d'offset est* $p \in P$
s_i *: valeur abstraite qui garde le contenu d'une entrée de la pile ou d'une variable locale du in de la sous-routine* $(1 \leq i \leq m,\ m = emax + hmax)$.
$s_{i[p]}$ *: valeur abstraite verrouillée par une ou plusieurs instructions monitorenter dont la pile d'offset est* $p \in P$.

Rappelons que le but de ce treillis est de déterminer pour chaque instruction les alias d'instances qui l'atteignent. Pour cette raison, l'ensemble E_0 du treillis de base L_0 est composé des éléments e de la forme :

$$e = \{e_1, ..., e_k\},\ 1 \leq\ k \leq m$$
$$e_j = (N_j : t_j),\quad 1 \leq\ j \leq k$$

Où :

- $\{N_j\}_{1 \leq j \leq k}$ est une partition de l'ensemble $\{1, ..., m\}$;

- t_i est un ensemble singleton qui contient une instance abstraite ;

Alors $L_0 = (E_0, \cup, \cap, \subseteq)$ est le treillis défini par l'ensemble E_0 et les relations $\cup$ (union), $\cap$ (intersection) et $\subseteq$ (inclusion) définies dans les sections suivantes.

Exemple d'un élément du treillis :

$$x_1 = \{(\{1,5\} : \emptyset), (\{3\} : \emptyset[5,17]), (\{4,2\} : \emptyset[15])\}$$

Cet élément représente l'état d'un *frame* qui nous indique que les emplacements 1 et 5 réfèrent à la même instance, et que les emplacements 4, 2 réfèrent à une instance verrouillée par une instruction *monitorenter* dont l'$offset$ est 15 et que l'emplacement 3 réfère à une instance verrouillée par une deux instructions *monitorenter* dont leur $offset$ est respectivement 5 et 17. Dans le but d'alléger la notation nous omettrons d'écrire les accolades des partitions :

$$x_1 = \{(1,5 : \emptyset), (3 : \emptyset[5,17]), (4,2 : \emptyset[15])\}$$

Il est à noter que l'ordre dans lequel est effectué le verrouillage est important, ainsi $(3 : \emptyset[15,17])$ est différent de $(3 : \emptyset[17,15])$.

Avant de présenter les définitions des relations du treillis, nous présentons une définition formelle d'un alias et autres définitions qui nous sont utiles.

Définition 14 (Alias). *Un alias d'instance t est un élément de la forme $(e : t)$ où e est un ensemble de la forme $\{i, j\}$ avec i et j désignant une entrée dans la pile ou l'emplacement d'une variable locale et qui réfèrent à la même instance t.*

Définition 15 (Regroupement d'alias). *Un regroupement d'alias est un élément de la forme $(N : t)$ où N est un sous-ensemble de $\{1, ..., m\}$, tel que :*

$$\forall i, j \in N \; i \neq j \Rightarrow (i, j : t) \text{ est un alias}$$

Il faut bien noter que, par définition, nous avons l'égalité suivante :

$$\{(3,4,5:\emptyset)\} = \{(3,4:\emptyset),(3,5:\emptyset),(4,5:\emptyset)\}$$

Définition 16 (Regroupement d'alias d'un emplacement). *Soient x un élément du treillis L_0 et i l'indice d'une variable locale ou d'un emplacement dans la pile, alors $N(x,i)$ est l'ensemble du regroupement d'alias auquel appartient i, et $t(x,i)$ l'instance du type de l'emplacement i.*

Exemple :

Soit x un élément du treillis tel que :

$$x = \{(1:\emptyset),(2,3:\emptyset[16])\}$$

Alors :

$$N(x,2) = N(x,3) = \{2,3\} \text{ et } t(x,2) = \emptyset[16]$$

Afin de mieux gérer la relation d'union et éviter des cas dans lesquels deux instances abstraites sont dans le même type d'un élément du treillis, nous introduisons l'élément $\top$ qui désigne le plus grand élément de l'ensemble $\{\emptyset, \emptyset_p, s_i, s_{ip} : p \in P \text{ et } i \in O\}$ et qui sera utile lors du calcul de l'union de deux emplacements qui proviennent de deux chemins différents et qui ont deux types différents, comme dans l'exemple suivant :

$$(1:s_1) \cup (1:s_2) = (1:\top)$$

$$(1:s_1[17]) \cup (1:s_1[18]) = (1:\top)$$

Dans le but de garder la même logique que nous avons suivi durant la définition de la relation d'intersection et puisque les deux types $\emptyset[5]$ et s_1 sont différents leur union est l'élément $\top$.

$$(1:\emptyset[5]) \cup (1:s_1) = (1:\top)$$

Définition 17 (Relation entre les instances abstraites). *L'intersection entre deux instances abstraites est notée par* $\wedge$, *et leur union par* $\vee$. *Soient* a *et* b *deux instances abstraites, alors :*

$$a \neq b \Rightarrow a \wedge b = \emptyset \text{ et } a \vee b = \top$$
$$a \preceq b \Leftrightarrow a \wedge b = a$$

Définissons maintenant les relations du treillis de base L_o.

5.2.3 Les relations du treillis

Le but de la relation d'intersection est de calculer tous les alias qui peuvent atteindre un point donné dans le graphe de contrôle.

Définition 18 (Intersection). *L'intersection de deux éléments* x_1 *et* x_2 *de* L_0 *est l'élément généré à partir de l'intersection de tous les éléments de* x_1 *et* x_2 *en respectant les conditions suivantes :*

Soit :

$$x_1 = \{(N_1 : t_1), ..., (N_p : t_p)\}, \text{ où } 1 \leq p \leq m$$

$$x_2 = \{(M_1 : f_1), ..., (M_q : f_q)\}, \text{ où } 1 \leq q \leq m$$

Alors :

$$x_1 \cap x_2 = \{(N_i \sqcap M_j : t_i \wedge f_j) \ / \ N_i \sqcap M_j \neq \emptyset\}_{i,j}$$

Où $\sqcap$ *est la relation d'intersection normale entre deux ensembles d'éléments.*

Exemples :

$\{(1, 2 : \emptyset), (3, 4 : \emptyset[15])\} \cap \{(1 : \emptyset), (2 : \emptyset), (3, 4 : \emptyset[17])\} =$

$\{(1 : \emptyset), (2 : \emptyset), (3, 4 : \emptyset)\}$

$\{(1 : \emptyset), (2 : \emptyset), (3, 4 : \{s_1\}[15])\} \cap \{(1 : \emptyset), (2 : \emptyset), (3, 4 : \{s_1\}[17])\} =$

$\{(1:\emptyset),(2:\emptyset),(3,4:\emptyset)\}$

Lors de l'intersection de ces deux éléments, nous perdons de l'information sur le *monitorenter* qui a verrouillé cette référence. Donc, nous gardons seulement l'information que c'est un *alias*.

Définition 19 (Union). *L'union entre deux éléments x_1 et x_2 est l'élément qui contient tous les alias de x_1 et de x_2. Ce résultat est atteint en faisant l'union entre les éléments qui ont un élément en commun entre leurs regroupements d'alias .*

Soient :

$$x_1 = \{(N_1 : t_1), ..., (N_p : t_p)\} \text{ , } où\ 1 \leq p \leq m$$

$$x_2 = \{(M_1 : f_1), ..., (M_q : f_q)\}, \ où\ 1 \leq q \leq m$$

Alors :

$$x_1 \cup x_2 = \{(N_i \sqcup M_j : t_i \vee f_j) \ / \ N_i \sqcap M_j \neq \emptyset\}_{i,j}$$

Où $\sqcap$ est la relation d'intersection normale entre deux ensembles d'éléments.

Exemple :

Soient x et y deux éléments du treillis tels que :

$$x = \{(1:\emptyset),(2:\emptyset),(3,4:\emptyset)\},\ y = \{(1:\emptyset),(2,3:\emptyset),(4:\emptyset)\}$$

Alors :

$$x \cup y = (1:\emptyset),(2,3,4:\emptyset),(2,3:\emptyset),(3,4:\emptyset)$$

Afin de garder la bonne partition de l'ensemble $\{1, ..., m\}$, nous faisons l'union des ensembles qui contiennent au moins un élément en commun :

Soient $W_{i,j}$ et $q_{i,j}$, tel que :

$$W_{ij} = N_i \sqcup M_j \text{ et } q_{ij} = t_i \vee f_j$$

Alors :

$$x_1 \cup x_2 = \{(N_i \sqcup M_j : t_i \vee f_j) : N_i \sqcap M_j \neq \emptyset\}$$

$$x_1 \cup x_2 = \{(W_{ij} : q_{ij})\}_{i,j}$$

Alors :

$$x_1 \cup x_2 = \{(\cup_{i,j} W_{ij} : q_{ij})\}, \cap_{i,j} W_{i,j} \neq \emptyset$$

En continuant avec le même exemple :

$$x \cup y = \{(1 : \emptyset), (2,3,4 : \emptyset), (2,3 : \emptyset), (3,4 : \emptyset)\}$$

$$x \cup y = \{(1 : \emptyset), (2,3,4 : \emptyset)\}$$

Une fois que l'on a bien défini les relations du treillis de base, l'unicité de la borne inférieure et de la borne supérieure de deux éléments x_1 et x_2, découle respectivement de l'union et de l'intersection des relations d'ensembles déjà bien définies.

Définition 20 (Inclusion). *Soient x_1 et x_2 deux éléments de L_0, alors :*

$$x_2 \subseteq x_1 \Leftrightarrow x_1 \cap x_2 = x_2$$

Autrement dit : chaque alias dans x_2 existe nécessairement dans x_1.

Exemple :

$$\{(1,2 : \emptyset), (3 : \emptyset), (4 : \emptyset)\} \subseteq \{(1,2 : \emptyset), (3,4 : \emptyset)\}$$

$$\{(1 : \emptyset), (2 : \emptyset), (3 : \emptyset)\} \subseteq \{(1,2 : \emptyset), (3 : \emptyset)\}$$

$$\{(1,2 : \emptyset), (3 : \emptyset)\} \subseteq \{(1,2; \emptyset[15]), (3 : \emptyset)\}$$

Remarque :

Soient x_1 et x_2 deux éléments de L_0, tels que :

$$x_1 \subseteq x_2 \text{ et } x_2 \subseteq x_1$$

Alors :

$$x_1 = x_2$$

Donc, si deux éléments de L_0 ont les mêmes regroupements d'alias, alors ils sont égaux. Par conséquent, tout élément du treillis est identifié d'une manière unique par ses regroupements d'alias et ses instances.

$$x_1 = x_2 \quad \Leftrightarrow \quad \forall i \in \{1, ..., m\} \quad N(x_1, i) = N(x_2, i) \text{ et } t(x_1, i) = t(x_2, i)$$

Proposition 21. *Soient x_1 et x_2 deux éléments du treillis tels que :*

$$x_1 \subseteq x_2$$

Alors, pour tout emplacement i dans la pile ou toute variable locale, nous avons :

$$N(x_1, i) \sqsubseteq N(x_2, i)$$
$$t(x_1, i) \preceq t(x_2, i)$$

Où $\sqsubseteq$ est la relation d'inclusion normale entre deux ensembles d'éléments.

Démonstration. Soit k un élément de $N(x_1, i)$, alors l'ensemble $\{k, i\}$ forme un alias dans x_1. Par définition de la relation d'inclusion, cet ensemble doit être aussi un alias dans x_2, et $t(x_1, i) \wedge t(x_2, i) = t(x_1, i)$. Par conséquent k et i appartiennent au même regroupement d'alias . D'où nous pouvons conclure que : $k \in N(x_2, i)$ et $t(x_1, i) \preceq t(x_2, i)$.

□

Proposition 22. *Soient x_1 et x_2 deux éléments du treillis, alors :*

$$N(x_1 \cap x_2, i) = N(x_1, i) \sqcap N(x_2, i)$$
$$t(x_1 \cap x_2, i) = t(x_1, i) \wedge t(x_2, i)$$

Démonstration. Soient x_1 et x_2 deux éléments du treillis, alors :

$$x_1 \cap x_2 \subseteq x_1$$
$$x_1 \cap x_2 \subseteq x_2$$

Grâce à la proposition 21, on trouve que :

$$N(x_1 \cap x_2, i) \subseteq N(x_1, i)$$
$$N(x_1 \cap x_2, i) \subseteq N(x_2, i)$$

D'où :

$$N(x_1 \cap x_2, i) \sqsubseteq N(x_1, i) \sqcap N(x_2, i)$$

Afin de montrer que :

$$N(x_1 \cap x_2, i) = N(x_1, i) \sqcap N(x_2, i)$$

il suffit de montrer que :

$$N(x_1 \cap x_2, i) \sqsupseteq N(x_1, i) \sqcap N(x_2, i)$$

En effet, $k \in N(x_1, i) \sqcap N(x_2, i)$, alors :

$$k \in N(x_1, i)$$
$$k \in N(x_2, i)$$

D'où :

(k, i) est un alias dans x_1 et x_2, et donc, par définition de la relation d'intersection, (k, i) est un alias de $x_1 \cap x_2$.

D'où :

$$k \in N(x_1 \cap x_2, i)$$

En conclusion :

$$N(x_1 \cap x_2, i) = N(x_1, i) \sqcap N(x_2, i)$$

D'autre part, par définition de la relation d'intersection du treillis $\cap$, nous avons :

$$t(x_1 \cap x_2, i) = t(x_1, i) \wedge t(x_2, i)$$

□

5.3 Algorithme de vérification

5.3.1 Contraintes de vérification

La syntaxe du langage Java impose que la synchronisation et le *finally* soient par bloc, et qu'on ne peut pas croiser le $try - finally$ et le bloc *synchronized* comme le cas de l'exemple de la figure 5.7

```
public void m(String a){

    synchronized(a){
        try{int j=2;}
    }

    finally{
        int j=2;
    }
}
```

FIGURE 5.7 Exemple d'un try-finally invalide

D'autre part, pour un code octet généré par un compilateur fiable, chaque instance verrouillée dans un bloc synchronisé doit être déverrouillée dans celui-ci avant de se brancher dans le bloc *finally*. Pour cette raison durant la vérification de la synchronisation nous imposons des contraintes pour simplifier notre analyse et le calcul du point fixe :

- Chaque référence verrouillée dans une sous-routine doit être déverrouillée dans celle-ci ;

- Chaque référence verrouillée en dehors d'une sous-routine doit être déverrouillée en dehors de celle-ci, comme dans l'exemple de la figure 5.8 ;

- Lors du calcul du *in* de l'instruction *next* du *jsr*, il ne faut pas perdre un alias qui apparaît dans le *in* de l'instruction *jsr* et qui n'a pas été modifié par la sous-routine qui est associée à ce *jsr* ;

- Il ne faut pas perdre un verrou lors du calcul du *in* d'une instruction quelconque.

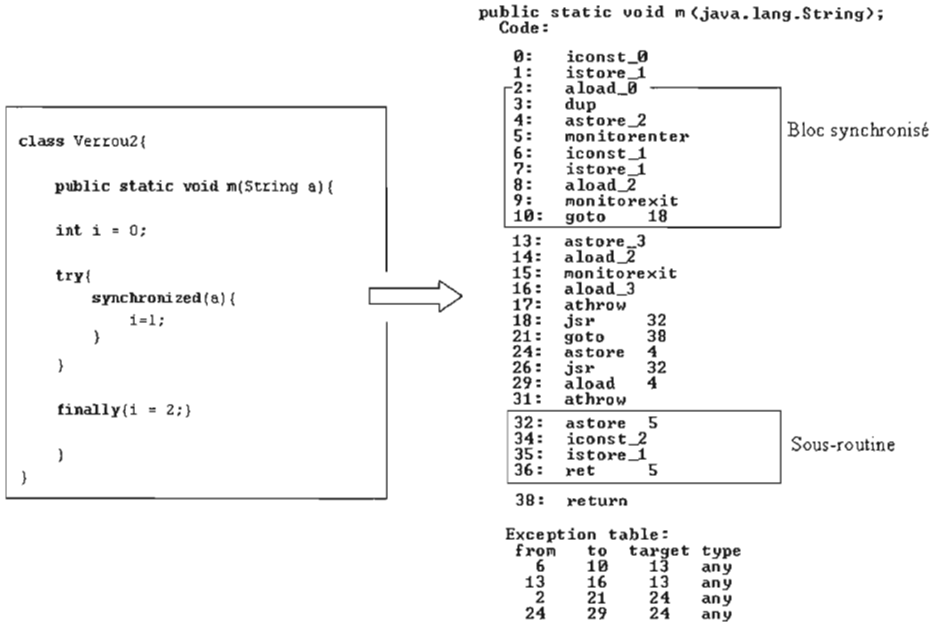


FIGURE 5.8 Exemple d'une sous-routine synchronisée

Pour réaliser ce genre d'analyse et de vérification, nous suivons la même approche que celle de la vérification des types ; il suffit de modifier le treillis et les fonctions de transfert pour modéliser l'effet de la synchronisation sur l'analyse de flot de données.

5.3.2 Traitement des sous-routines

Lors de l'initialisation du $frame$ associé à la sous-routine, nous initialisons chaque emplacement i par s_i. L'emplacement à la hauteur courante contient l'adresse de retour de la sous-routine.

Durant l'exécution de la sous-routine, si un verrou est relâché, son emplacement contient l'élément $\emptyset$ au lieu d'un élément de $S = \{s_1, ..., s_{hmax+emax}\}$ d'où, lors du calcul du in de l'instruction $jsrBis$, nous nous apercevons qu'il y a une différence au niveau des verrous entre le in du jsr et le in du $jsrBis$.

En suivant la même logique, nous pouvons détecter si un verrou est acquis lors d'une sous-routine sans qu'il soit relâché, puisqu'il apparaît dans le out du ret sous forme d'un $s_{i[p]}$, ce qui va engendrera l'échec de la vérification.

Dans ce qui suit, nous présentons quelques exemples pour illustrer l'initialisation du $frame$ de la sous-routine et le calcul du in de l'instruction qui suit l'instruction $(jsr\ @)$, que nous notons dans le reste du chapitre par $next_{jsr,@}$.

Exemple :

$jsr\ @ : \{(1, 2 : \emptyset), (3 : \emptyset), (4 : \emptyset)\}$

$@ : \{(1 : s_1), (2 : \emptyset)(3 : s_3), (4 : s_4)\}$, l'emplacement 2 contient l'adresse de la sous-routine

$astore\ 0 : \{(1 : s_1), (2 : \emptyset), l_0 : (3 : \emptyset), (4 : s_4)\}$

Afin de propager la bonne information à l'instruction $jsrBis$ nous ne devons perdre aucun alias apparaissant dans le in de l'instruction jsr et qui n'est pas modifié par la sous-routine.

Exemple :

$jsr@ : \{(1, 2 : \emptyset), (3 : \emptyset), (4 : \emptyset)\}$

$@ :$

...

$ret : \{(1 : \emptyset), (2 : \emptyset), (3; s_1), (4; s_2)\}$

Alors :

$next_{jsr,@}\ \{(1 : \emptyset), (2 : \emptyset), (3, 4 : \emptyset)\}$

L'autre aspect aussi important que le traitement des sous-routines est que, lors du calcul du *in* d'une instruction, il faut s'assurer de ne pas perdre un verrou, autrement dit tous les prédécesseurs doivent avoir les mêmes piles de verrous. Notre façon de détecter ce genre d'infraction est d'imposer que les seules instructions dont les verrous de leur *in* est différent de celui du *out* sont les instructions *monitorenter* et *monitorexit*.

La différence avec l'algorithme de vérification de types du chapitre précédent, est que nous manipulons les regroupements d'alias au lieu des types, ce qui nous aidera à différencier les instances qui sont verrouillées de celles qui ne le sont pas.

5.4 Les fonctions de transfert

Dans cette section nous présentons quelques fonctions de transfert comme *astore*, *jsr* et *aload*. Toute instruction qui ne gère pas les références comme *iadd*, *ret*, *iload*, ... a une fonction de transfert constante puisqu'elle ne peut pas modifier les relations d'alias qui existent dans l'environnement.

5.4.1 Initialisation de l'environnement

Soit M la méthode à vérifier et $m = emax + hmax$, alors le premier état de l'environnement est initialisé comme suit :

$$In(0) = \{(1 : \emptyset), (2 : \emptyset), ..., (m, \emptyset)\}$$

Définissons par la suite quelques fonctions de transfert qui sont les plus importantes pour notre étude avec la preuve de leur monotonie et distributivité.

5.4.2 Exemples de fonctions de transfert

Soit x et y deux éléments du treillis L_0, tel que :

$$x = \{(N_1 : t_1), ..., (N_q : t_q)\}$$

$$y = \{(M_1 : f_1), ..., (N_p : t_p)\}$$

5.4.2.1 Aload

Rappelons que l'instruction *aload* charge le contenu de la variable locale sur la pile. C'est pour cette raison que le sommet de la pile à l'emplacement h et la variable locale l sont des alias.

Alors :

$$aload_{h,l}(x) = y$$

Où :

$$N(y,i) = \begin{cases} N(x,l) \sqcup \{h\} & \text{si } i \in N(x,l) \sqcup \{h\} \\ N(x,i) & \text{sinon} \end{cases}$$

Et :

$$t(y,i) = \begin{cases} t(x,l) & \text{si } i \in N(x,l) \sqcup \{h\} \\ t(x,i) & \text{sinon} \end{cases}$$

Monotonie :

Soient $x_1 \subseteq x_2$, et $i \in N(x_1,l) \sqcup \{h\}$, alors grâce à la proposition 21, nous avons :

$$N(x_1,l) \sqsubseteq N(x_2,l)$$

$$t(x_1, l) \preceq t(x_2, l)$$

Et :

$$N(x_1, i) \sqsubseteq N(x_2, i)$$
$$t(x_1, i) \preceq t(x_2, i)$$

D'où :

$$N(x_1, l) \sqcup \{h\} \sqsubseteq N(x_2, l) \sqcup \{h\}$$

Ainsi, nous pouvons conclure que :

$$aload_{h,l}(x_1) \subseteq aload_{h,l}(x_2)$$

Distributivité : Soient x_1 et x_2 deux éléments du treillis, tel que :

$$aload_{h,l}(x_1) = y_1$$
$$aload_{h,l}(x_2) = y_2$$
$$aload_{h,l}(x_1 \cap x_2) = y$$

Montrons dans ce qui suit que :

$$y = y_1 \cap y_2$$

D'après la définition de la fonction *aload* nous avons :

$$N(y_1, i) = \begin{cases} N(x_1, l) \sqcup \{h\} & \text{si } i \in N(x_1, l) \sqcup \{h\} \\ N(x_1, i) & \text{sinon} \end{cases}$$

$$N(y_2, i) = \begin{cases} N(x_2, l) \sqcup \{h\} & \text{si } i \in N(x_2, l) \sqcup \{h\} \\ N(x_2, i) & \text{sinon} \end{cases}$$

Alors :

$$N(y_1,i) \sqcap N(y_2,i) = \begin{cases} [N(x_1,l) \sqcup \{h\}] \sqcap [N(x_2,l) \sqcup \{h\}] \\ \qquad \text{si } i \in [N(x_1,l) \sqcup \{h\}] \sqcap [N(x_2,l) \sqcup \{h\}] \\ [N(x_1,l) \sqcup \{h\}] \sqcap N(x_2,l) \\ \qquad \text{si } i \in [N(x_1,l) \sqcup \{h\}]\ i \notin [N(x_2,l) \sqcup \{h\}] \\ N(x_1,l) \sqcap [N(x_2,l) \sqcup \{h\}] \\ \qquad \text{si } i \notin [N(x_1,l) \sqcup \{h\}],\ i \in [N(x_2,l) \sqcup \{h\}] \\ N(x_1,i) \sqcap N(x_2,i) \\ \qquad \text{sinon} \end{cases}$$

Or, du fait que l'emplacement h est empilé toujours au-dessus des valeurs qui existent déjà dans la pile, il n'y a aucune relation d'alias qui dépend de h dans x_1 et x_2. Pour cette raison nous avons :

$$N(x_1,l) \sqcap \{h\} = \emptyset \text{ et } N(x_2,l) \sqcap \{h\} = \emptyset$$

D'où :

$$[N(x_1,l) \sqcup \{h\}] \sqcap N(x_2,l) = N(x_1,l) \sqcap N(x_2,l)$$
$$N(x_1,l) \sqcap [N(x_2,l) \sqcup \{h\}] = N(x_1,l) \sqcap N(x_2,l)$$

Par conséquent :

$$N(y_1,i) \sqcap N(y_2,i) = \begin{cases} [N(x_1,l) \sqcup \{h\}] \sqcap [N(x_2,l) \sqcup \{h\}] \\ \qquad \text{si } i \in [N(x_1,l) \sqcup \{h\}] \sqcap [N(x_2,l) \sqcup \{h\}] \\ N(x_1,i) \sqcap N(x_2,i) \\ \qquad \text{sinon} \end{cases}$$

D'autre part, grâce à la distributivité de la relation d'intersection ordinaire, pour tout ensemble A, B et H tel que :

$$A \sqcap H = \emptyset \text{ et } B \sqcap H = \emptyset$$

Nous avons :

$$(A \sqcup H) \sqcap (B \sqcup H) = (A \sqcap B) \sqcup H$$

Or, puisque :

$$N(x_1, l) \sqcap \{h\} = \emptyset \text{ et } N(x_2, l) \sqcap \{h\} = \emptyset$$

Alors :

$$N(y_1, i) \sqcap N(y_2, i) = \begin{cases} [N(x_1, l) \sqcap N(x_2, l)] \sqcup \{h\} & \text{si } i \in [N(x_1, l) \sqcap N(x_2, l)] \sqcup \{h\} \\ N(x_1, i) \sqcap N(x_2, i) & \text{sinon} \end{cases}$$

Or, d'après la proposition 22, nous avons :

$$N(x_1, l) \sqcap N(x_2, l) = N(x_1 \sqcap x_2, i)$$

D'où :

$$N(y_1, i) \sqcap N(y_2, i) = \begin{cases} N(x_1 \sqcap x_2, l) \sqcup \{h\} & \text{si } i \in N(x_1 \sqcap x_2, l) \sqcup \{h\} \\ N(x_1 \sqcap x_2, i) & \text{sinon} \end{cases}$$

À partir de la définition de la fonction $aload(x_1 \sqcap x_2)$ nous déduisons que :

$$N(y, i) = N(y_1, i) \sqcap N(y_2, i), \forall i$$

Reste à prouver que :

$$t(y, i) = t(y_1 \sqcap y_2, i), \forall i$$

Or :

$$t(y_1 \sqcap y_2, i) = t(y_1, i) \wedge t(y_2, i), \forall i$$

D'autre part, par définition de la fonction *aload*, on a :

$$t(y_1, i) = \begin{cases} t(x_1, l) & \text{si } i \in N(x_1, l) \sqcup \{h\} \\ t(x_1, i) & \text{sinon} \end{cases}$$

$$t(y_2,i) = \begin{cases} t(x_2,l) & \text{si } i \in N(x_2,l) \sqcup \{h\} \\ t(x_2,i) & \text{sinon} \end{cases}$$

Or, si $i \in N(x_1,l) \sqcup \{h\}$ et $i \notin N(x_2,l) \sqcup \{h\}$, alors :

$$i \in N(x_1,l),\ i \notin N(x_2,l)$$

Et, par définition :

$$t(y_1,i) \wedge t(y_2,i) = t(x_1,l) \wedge t(x_2,i)$$

Puisque i et l appartiennent au même ensemble de regroupement d'alias dans x_1, alors :

$$t(x_1,i) = t(x_1,l)$$

Et par conséquent :

$$t(y_1,i) \wedge t(y_2,i) = t(x_1,i) \wedge t(x_2,i)$$

En suivant le même raisonnement, nous pouvons prouver que :

Si $i \notin N(x_1,l) \sqcup \{h\}$ et $i \in N(x_2,l) \sqcup \{h\}$ alors $t(y_1,i) \wedge t(y_2,i) = t(x_1,i) \wedge t(x_2,i)$

Par conséquent :

$$t(y_1,i) \wedge t(y_2,i) = \begin{cases} t(x_1,l) \wedge t(x_2,l) & \text{si } i \in [N(x_1,l) \sqcup \{h\}] \sqcap [N(x_2,l) \sqcup \{h\}] \\ t(x_1,i) \wedge t(x_2,i) & \text{sinon} \end{cases}$$

$$t(y_1,i) \wedge t(y_2,i) = \begin{cases} t(x_1,l) \wedge t(x_2,l) & \text{si } i \in [N(x_1,l) \sqcap N(x_2,l)] \sqcup \{h\} \\ t(x_1,i) \wedge t(x_2,i) & \text{sinon} \end{cases}$$

$$t(y_1, i) \wedge t(y_2, i) = \begin{cases} t(x_1 \cap x_2, l) & \text{si } i \in N(x_1 \cap x_2, l) \sqcup \{h\} \\ t(x_1 \cap x_2, i) & \text{sinon} \end{cases}$$

D'où :

$$t(y_1, i) \wedge t(y_2, i) = t(y, i)$$

Conclusion :

$$aload(x_1 \cap x_2) = y_1 \cap y_2$$

5.4.2.2 Jsr

Rappelons que pour le frame relié à l'instruction jsr, nous initialisons tous les emplacements de la pile et des variables locales par les valeurs abstraites (s_i) sauf l'emplacement qui représente celui de l'adresse de retour, en tenant compte des références verrouillées qui atteignent cette sous-routine, puisque nous devons porter attention au cas où elle sont déverrouillées durant l'exécution de la sous-routine.

Alors :

$$jsr_h(x) = y$$

Où :

$$N(y, i) = \{i\}$$

Et :

$$t(y, i) = \begin{cases} s_i & \text{si } \quad i \neq h \\ \emptyset & \text{sinon} \end{cases}$$

5.4.3 jsrBis

Tout élément x de L_0 est une union de deux ensembles : un ensemble concret que nous notons $C(x)$ et dont les valeurs des emplacements sont $\emptyset$, $\emptyset[p]$, et un ensemble complémentaire que nous appelons la partie abstraite et que nous notons par $A(x)$:

$$x = C(x) \cup A(x)$$

Exemple :

Soit :

$$x = \{(1 : \emptyset), (2 : \emptyset), (3, 5 : s_1), (4 : s_2)\}$$

Alors :

$$C(x) = \{(1 : \emptyset), (2 : \emptyset)\} \text{ et } A(x) = \{(3, 5 : s_1), (4 : s_2)\}$$

Pour calculer la fonction de transfert de $jsrBis$, il faut tenir compte des aspects suivants :

1. Elle doit maintenir la propriété d'alias de la sous-routine, autrement dit, toute nouvelle relation d'alias qui apparaît dans le *out* du *ret* doit être propagée à l'instruction qui suit le jsr.

2. Maintenir les références verrouillées qui atteignent la sous-routine et qui ne sont pas modifiées par celle-ci.

Exemple :

$$x = \{(1, 2; \emptyset_{[15]}), (3 : \emptyset), (4 : \emptyset), (5 : \emptyset)\}$$

$$ret = \{(1 : \emptyset), (2 : \emptyset), (3, 5 : s_1), (4 : s_2)\}$$

Alors :

$$jsrBis_{ret}(x) = \{(1 : \emptyset), (2 : \emptyset), (3, 4, 5 : \emptyset_{[15]})\}$$

Le calcul de la fonction $jsrBis$ se base sur les actions suivantes :

1. Déterminer J l'ensemble des indices des s_j (déterminés lors du calcul du point fixe de chaque instruction *ret*).

2. Calculer les alias de J dans x : $(x \cap \{(J : \top)\})$.

3. Remplacer chaque j de J dans $(x \cap \{(J : \top)\})$ par le regroupement d'alias de s_j en uti-

lisant la fonction F_r.

4. Calculer l'union de ce dernier résultat avec $C(r)$.

Dans l'exemple suivant :

$$x = \{(1, 2; \emptyset_{[15]}), (3 : \emptyset), (4 : \emptyset), (5 : \emptyset)\}$$

$$ret = \{(1 : \emptyset), (2 : \emptyset), (3, 5 : s_1), (4 : s_2)\}$$

Nous avons :

$$J = \{1, 2\}$$

$$F_r(1, 2 : \emptyset_{[15]}) = \{3, 5, 4 : \emptyset_{[15]}\}$$

Définissons maintenant la fonction F_r responsable de transformer les instances abstraites s_i de r en fonction de x.

Définition 23 (Fonction F_r). *Soient r une instruction ret et $x = \{(N_i : t_i)\}_i$ un élément de L_0 tel que :*

$$N_i = \{n_1, ..., n_p\}$$

$$A(r) = \{(N_j : s_j)\}_{j \in J}$$

Et s_i et t_i des instances abstraites. Alors $F_r : L_0 \rightarrow L_o$ est la fonction définie comme suit :

$$F_r(x) = \{F_r(N_i : t_i)\}_i$$

Et :

$$F_r(N_i : t_i) = (N(r, s_{n_1}) \sqcup ... \sqcup N(r, s_{n_p}) : t_i)$$

Définition 24 (Fonction $jsrBis$). *Soient $x \in L_0$, @ l'adresse de la sous-routine du jsr et r l'instruction ret associée, tels que :*

$$A(r) = \{(N_j : s_j)\}_{j \in J}$$

Alors :

$$jsrBis_{@,r}(x) = y$$

Où :

$$y = C(r) \sqcup F_r(x \cap \{(J : \top)\})$$

Exemple :

$$x = \{(1, 2 : t), (3 : \emptyset), (4 : \emptyset), (5 : \emptyset)\}$$

$$r = \{(1 : \emptyset), (2 : \emptyset), (3, 5 : s_1), (4 : s_2)\}$$

Alors :

$$A(r) = \{(3, 5 : s_1), (4 : s_2)\}$$

$$x \cap (J : \top) = \{(1, 2 : t), (3 : \emptyset), (4 : \emptyset), (5 : \emptyset)\} \cap \{(\{1, 2\} : \top)\}$$

$$x \cap (J : \top) = \{(\{1, 2\} : t)\}$$

D'où :

$$F_r(1, 2 : t) = (3, 5, 4 : t) \text{ (nous remplaçons 1 par 3 et 5, et 2 par 4)}$$

Et :

$$jsrBis_r(x) = \{(1 : \emptyset), (2 : \emptyset), (3, 4, 5 : t)\}$$

Afin de prouver la monotonie et la distributivité de la fonction $jsrBis$, il suffit de le prouver pour la fonction F_r. Lors de cette preuve nous utilisons quelques propriétés que nous présentons, ainsi que leurs preuves à la fin de cette section.

Monotonie

Soient x_1 et x_2 deux éléments de L_0, tel que $x_1 \subseteq x_2$ et :

$$x_1 = \{(N_i : t_i)\}_{i \in I}$$

$$x_2 = \{(M_j : f_j)\}_{j \in J}$$

Montrons que :

$$F_r(x_1) \subseteq F_r(x_2)$$

Or, par définition de la fonction F_r, nous avons :

$$F_r(x_1) = \{F_r(N_i : t_i)\}_{i \in I}$$

Soit $i \in I$, alors, par définition de la relation d'inclusion :

$$\exists j \in J \ \ tel\ que \ \ (N_i : t_i) \subseteq (M_j : f_j)$$

Afin de simplifier la preuve, nous supposons que :

$$N_i = (n_1, n_2 : t_i)$$
$$M_j = (n_1, n_2, m_3 : f_j)$$

$\Rightarrow (n_1, n_2 : t_i) \sqsubseteq (n_1, n_2, m_3 : f_j)$

$\Rightarrow (N(r, s_{n_1}) \sqcup N(r, s_{n_2}) : t_i) \sqsubseteq (N(r, s_{n_1}) \sqcup N(r, s_{n_2}) \sqcup N(r, s_{m_3}) : f_j)$

$\Rightarrow F_r(N_i : t_i) \sqsubseteq F_r(M_j : f_j)$

Conclusion :
Pour tout $i \in I$, $\exists j \in J$, tel que :

$$F_r(N_i : t_i) \sqsubseteq F_r(M_j : f_j)$$

Par conséquent, la fonction F_r est bien monotone.

Distributivité :

Démonstration. Montrons dans ce qui suit que :

$$F_r(x_1) \cap F_r(x_2) = F_r(x_1 \cap x_2)$$

Nous allons développer les deux côtés de cette équation pour prouver qu'on atteint le même résultat.

Par définition de la relation d'intersection $\cap$, on a :

$F_r(x_1 \cap x_2) = F_r(\{N_i \sqcap M_j : t_i \wedge f_j\}_{i,j})$, où $N_i \sqcap M_j \neq \emptyset$

$= \{F_r(N_i \sqcap M_j : t_i \wedge f_j)\}_{i,j}$

Soient :

$$N_i = \{n_1, ..., n_p\}$$

$$M_j = \{m_1, ..., m_q\}$$

$$N_i \sqcap M_j = \{z_1, ..., z_h\}$$

Alors :

$$F_r(x_1 \cap x_2) = \{(N(r, s_{z_1}) \sqcup ... \sqcup N(r, s_{z_h} : t_i \wedge f_j)\}_{i,j}$$

D'autre part :

$F_r(x_1) \cap F_r(x_2) = \{F_r(N_i : t_i)\}_i \ \cap \ \{F_r(M_j : f_j)\}_j$

$= \{ \ (\bigcup_\alpha N(r, s_{n_\alpha}) : t_i) \ \}_i \ \cap \ \{ \ (\bigcup_\beta N(r, s_{m_\beta}) : f_j) \ \}_j$

$= \{ \ (\bigcup_\alpha N(r, s_{n_\alpha}) \ \sqcap \ \bigcup_\beta N(r, s_{m_\beta}) : t_i \wedge f_j) \ \}_{i,j}$

$= \{ \ (\bigcup_{\alpha,\beta} N(r, s_{n_\alpha}) \sqcap N(r, s_{m_\beta}) : t_i \wedge f_j) \ \}_{i,j}$

Or, grâce à la définition 16, nous avons :

$$\forall n_\alpha, m_\beta \in N_i \times M_j \ : \ n_\alpha \neq m_\beta \implies N(r, n_\alpha) \sqcap N(r, m_\beta) = \emptyset$$

D'où :

$$F_r(x_1) \cap F_r(x_2) = \{(\bigcup_z N(r, s_z) : t_i \wedge f_j)\}_{i,j} \ ; \ z \in N_i \cap M_j$$

Grâce à la définition 15, nous pouvons déduire que les deux côtés de l'équation de distributivité de la fonction F_r sont bien égaux.

□

Propriété 25. *Pour toutes suites d'ensembles $(N_i)_{i\in I}$ et $(M_j)_{j\in J}$, nous avons :*

$$\cup_{i\in I} N_i \bigcap \cup_{j\in J} M_j = \bigcup_{(i,j)\in I\times J}(N_i \cap M_j)$$

Proposition 26. *Soient $r \in L_o$, N_i et M_j deux regroupements d'alias quelconques, tel que :*

$$N_i = \{n_1, ..., n_p\}$$
$$M_i = \{m_1, ..., m_q\}$$

Alors :

$$N_i \sqcap M_j \neq \emptyset \Leftrightarrow \cup_{1\leq\alpha\leq p} N(r, s_{n_\alpha}) \ \sqcap \ \cup_{1\leq\beta\leq q} N(r, s_{m_\beta}) \neq \emptyset$$

Démonstration. Soit $k \in N_i \cap M_j$, alors :

$$N(r, s_k) \in \cup_{1\leq\alpha\leq p} N(r, s_{n_\alpha})$$
$$N(r, s_k) \in \cup_{1\leq\beta\leq q} N(r, s_{m_\beta})$$

Par conséquent :

$$\cup_{1\leq\alpha\leq p} N(r, s_{n_\alpha}) \sqcap \cup_{1\leq\beta\leq q} N(r, s_{m_\beta}) \neq \emptyset$$

Prouvons maintenant l'autre sens de la relation d'équivalence.
Soit :

$$w \in \cup_{1\leq\alpha\leq p} N(r, s_{n_\alpha}) \ \sqcap \ \cup_{1\leq\beta\leq q} N(r, s_{m_\beta})$$

Alors, il existe i et j, tel que $N(r, s_{n_i}) = N(r, s_{m_i})$, ce qui entraine que $s_{n_i} = s_{m_j}$ et par

conséquent $n_i = m_j$. D'où :

$$N_i \sqcap M_j \neq \emptyset$$

□

Proposition 27. *$\forall i, j \in 1, ..., m$, tel que $i \neq j$ et $r \in L_0$, alors :*

$$N(r, s_i) \sqcap N(r, s_j) = \emptyset$$

Démonstration. Soient i et j, tel que $i \neq j$.

Supposons que :

$$N(r, s_i) \sqcap N(r, s_j) \neq \emptyset$$

Alors il existe $k \in N(r, s_i) \sqcap N(r, s_j)$, tel que : $i \neq k$

D'où : $k \in N(r, s_i)$ et $k \in N(r, s_j)$

Ce qui entraîne que $s_i = s_j$ et par conséquent $i = j$, ce qui est contradictoire.

Donc : $N(r, s_i) \sqcap N(r, s_j) = \emptyset$ □

5.5 Vérification

Une fois le calcul du point fixe atteint, nous devons effectuer certaines vérifications, pour nous assurer du bon fonctionnement de la synchronisation. Pour cela, nous devons vérifier les conditions suivantes :

- Le *out* de chaque point de sortie du graphe (*return* ou *athrow*) ne contient aucun élément verrouillé ;
- L'ensemble des références verrouillées du *in* de toute instruction différente de *monitorenter* et *monitorexit* est égal à l'ensemble des références verrouillées de son *out* ;
- Chaque *monitorexit* est fait sur une référence verrouillée.

5.6 Conclusion

Dans ce chapitre nous avons présenté un algorithme de vérification du mécanisme de la synchronisation dans la MVJ.

Nous avons suivi la même approche que la vérification des types présentée dans le chapitre précédent : en décomposant le problème en plusieurs étapes nous avons déterminé d'abord le point fixe de toute instruction *ret*, ensuite, le point fixe de chaque instruction. Ce qui nous a permis de déterminer pour chaque instruction l'ensemble des instances qu'il atteint, ainsi que leur statut (verrouillées ou non) et les instructions *monitorenter* qu'elles ont verrouillées. Ce genre d'information nous a aidé à rendre le mécanisme de synchronisation vérifiable et le code octet plus sûr et plus fiable avant son exécution par la MVJ.

Chapitre VI

TRAVAUX RELIÉS

Ce chapitre présente les travaux réalisés en relation avec le travail présenté dans ce mémoire. Dans la première section nous présentons les articles et travaux les plus importants dans le domaine de la vérification du code octet Java. Dans la deuxième section nous présentons les travaux réalisés en relation avec la vérification de la synchronisation au niveau du code octet Java.

6.1 Vérificateur du code octet

Comme décrit dans les spécifications de la MVJ [LY99, Section 4.9.6], le principe du vérificateur du code octet Java consiste à calculer l'ensemble des variables locales non-utilisées par la sous-routine. Cette approche oblige à délimiter la structure des sous-routines et impose des contraintes supplémentaires comme le fait que toute instruction *ret* peut être associée qu'à une seule sous-routine.

Dans le but de présenter une preuve formelle de validation du vérificateur, plusieurs chercheurs ont essayé de formaliser le vérificateur Java, comme Stata [SA99], Qian [Qia99], Starck [RSB01]. Nous citons dans ce chapitre les travaux des approches les plus connues, dont la vérification de modèles et l'approche que nous avons adopté dans ce mémoire, l'analyse de flot de données.

6.1.1 La vérification de modèles (Model checking)

D'abord, donnons un aperçu de cette approche. D'après Clarke [EC01], la vérification de model est une technique automatique pour vérifier des systèmes à état fini. Ces principaux

avantages sont la simulation et le test. Son but est de vérifier algorithmiquement si un modèle donné satisfait une spécification.

Dans un travail pour présenter un vérificateur pour les cartes Java Smart, Posegga et Vogt [PV98] présentent une approche basée sur la vérification de modèles. Ils décrivent ensuite un algorithme qui prend le code octet pour une méthode et génère une formule de logique temporelle qui est valide si et seulement si le code octet est valide. Finalement, ils utilisent un outil existant de vérification de modéles pour déterminer la validité de la formule.

6.1.2 Analyse de flot de données

Qian [Qia00] [Qia99] présente une spécification formelle d'un sous-ensemble des instructions de la MVJ. Il présente des règles de types pour la majorité des instructions de la MVJ. En adoptant la même approche que celle de vérificateur de Sun, il calcule les variables locales modifiées par la sous-routine, et la fusion des types pour déterminer les types qui atteignent chaque instructions, ce qui rend la fonction de transfert de l'instruction *ret* non monotone. Son algorithme se base sur une analyse de flot de données et le théorème du calcul du point fixe pour des fonctions non monotones. Par contre, son algorithme rejette quelques types de code Java valides comme démontré par Coglio [Cog04].

Stata et Abadi [SA99] traitent les cas simples de sous-routines, en ignorant les cas où on peut sortir de la sous-routine par des instructions comme *athrow* ou *goto*. Leur approche est en continuité avec celle de Sun ; ils se basent sur une analyse par étapes. Ils font un pré-calcul de l'ensemble des variables utilisées par chaque sous-routine et ensuite, ils utilisent ces informations dans les équations d'analyse de flot de données.

Coglio [Cog04] propose une implémentation correcte de presque tous les aspects d'un vérificateur du code octet Java. Il considère le problème de la vérification comme un problème d'analyse de flots de données. Il en fournit une spécification et extrait le code correspondant grâce à l'outil SpecWare [JA01] ce qui lui a permet de dériver les spécifications en langage Lisp ou C++.

En utilisant Specware, Coglio, Goldberg et Qian ont analysé une grande partie du code octet Java. Qian [Qia00] a prouvé la validité de son algorithme en utilisant une analyse de flot de

données pour valider des règles de vérification de types décrites dans un autre article [Qia99]. Ensuite, Coglio [Cog01] [Cog04] a fourni une solution au problème des sous-routines. En suivant les travaux de Posegga et Vogt [PV98], Coglio se base sur une analyse de flot de données. Son algorithme n'accepte pas la fusion de types à l'entrée des sous-routines. Par contre, il garde tous les types de chaque variable locale durant l'analyse des sous-routines, ce qui rend son algorithme plus flexible et admet tous les types de classes qui sont rejetées par les autres vérificateur [FM99]. Stark [SS03] [RSB01] se base sur l'idée du calcul de l'ensemble des variables locales qui ne sont pas utilisées par une sous-routine. La difficulté à déterminer la structure et les limites d'une sous-routine ainsi que les instructions qui en font partie ont compliqué considérablement la présentation d'une formalisation complète et la preuve de la monotonie des fonctions de transfert.

Freund et Mitchell [FM03] [FM99] développent les règles de vérification de types pour la majorité des instructions du code octet Java. Leur approche est moins restrictive. Ils ont aussi traité l'impact des exceptions sur les sous-routines. Par contre, leur approche rejette certains types de code [Cog04].

Hagiya et Tozawa [HT98] utilisent une approche similaire à celle de Stata et Abadi [SA99], par contre ils utilisent un nouveau type, qu'ils ont noté par $last(x)$, pour modéliser la variable locale n de l'appelant de la sous-routine.

Leroy [Ler01][Ler03] présente une bonne explication du problème des sous-routines et de leur impact sur l'implémentation du vérificateur de la MVJ. Il propose une analyse de flot de données se basant sur le principe de la fusion des types.

O'Calahan [O'C99] propose une approche différente basée sur la continuation des types. Il assigne à chaque adresse de retour la pile et les variables locales attendues par l'instruction qui suit l'instruction jsr. Il propose des règles de vérification des types, mais son algorithme n'est pas efficace.

Le projet Bali se base sur le théorème Isabelle/HOL. En suivant les travaux de Qian [Qia99], les chercheurs du projet Bali fournissent la spécification du vérificateur et la preuve de sa validité. Plus tard, ils définissent [NOP00] un sous-ensemble de Java appelé μJava sur lequel ils formalisent le système de typage et la sémantique de ce langage.

6.2 Vérification de la synchronisation

Concernant la vérification de la synchronisation, nous présentons les deux travaux les plus importants dans ce domaine, ceux de Bilgiardi-Laneve [BL00] et Futoshi-Naoki [IK02]. Les deux travaux étendent le système de types de Stata-Abadi [SA99] pour modéliser l'effet de la synchronisation sur les objets. Le premier travail, celui de Bigliardi-Laneve [BL00], propose un système restrictif. Ainsi, il peut rejeter un code valide comme dans les deux cas pésentés à la figure 6.1 et la figure 6.2 [IK02].

```
0 aload 1
1 dup
2 monitorenter
3 iload 2
4 ifeq 7
5 monitorexit
6 return
7 monitorexit
8 return
```

FIGURE 6.1 Exemple de code valide rejeté par le système de Bilgiardi-Laneve

Dans l'exemple 6.1, leur système considère que l'instruction *monitorenter* a deux instructions *monitorexit* associées en tout temps, ce qui entraîne le rejet du code.

```
0 aload 1
1 dup
2 monitorenter
3 alaod 2
4 dup
5 monitorenter
6 swap
7 monitorexit
8 monitorexit
9 return
```

FIGURE 6.2 Exemple de sections synchronisées croisées

L'exemple du code octet de la figure 6.2 est aussi rejeté puisque leur système n'admet pas que les sections synchronisées soient croisées. Il est difficile de trouver du code Java produit par

un compilateur fiable et produisont ce genre de structure. Par contre, certaines optimisations du code peuvent produire ce type de code. En appliquant ces deux exemples sur notre système de vérifications, nous constatons que notre algorithme de vérification de la synchronisation est assez flexible pour les vérifier sans les rejeter.

Futoshi et Naoki [IK02] poursuivent le travail de Bigliardi et Laneve, en couvrant la majorité des problèmes. Par contre, leur travail ne traite pas les sous-routines, ils présentent une formalisation complexe et, comme expliqué par les auteurs eux-mêmes, leur système peut rejeter certains types de code valide.

CONCLUSION

Les apports de ce mémoire tiennent en deux points. D'une part, ce mémoire apporte une preuve formelle et complète d'un processus de vérification du code octet au sein de la machine virtuelle Java (MVJ). D'autre part, nous fournissons une nouvelle approche de la vérification de la synchronisation dans la MVJ.

6.3 Vérificateur de types du code octet Java

Nous avons illustré l'importance du vérificateur du code octet au sein de l'architecture de la MVJ. Aussi, nous avons expliqué les principales contraintes que le vérificateur doit satisfaire pour respecter les spécifications de la MVJ, et la complexité engendrée par les sous-routines pour la vérification des types.

Plusieurs approches ont été proposées pour mettre en place une preuve formelle du fonctionnement du vérificateur des types. Notre choix de procéder par analyse de flot de données nous a aidé à fournir une preuve mathématique simple de la monotonie et de la distributivité des fonctions de transfert.

Afin de construire un treillis simple et bien fondé, au lieu de la fusion de type comme proposé dans plusieurs études [Ler01], la relation d'ordre de notre treillis est une simple union d'ensemble d'éléments de types :

$$\{type1\} \cup \{type2\} = \{type1, type2\}$$

Grâce à la définition de cette relation nous avons évité le problème de la fusion des tableaux et des interfaces. De plus, nous avons construit un treillis bien défini en bénéficiant de tous les théorèmes associés.

Dans le but de contourner le problème de la non monotonie de la fonction de transfert associée à l'instruction `ret`, nous avons décomposé notre algorithme en plusieurs étapes, ce qui

nous a aidé à recueillir diverses informations que nous avons utilisées pour optimiser l'algorithme de la vérification de la synchronisation, comme le graphe d'appel de sous-routines et l'ensemble d'instructions qui font partie de chaque sous-routine.

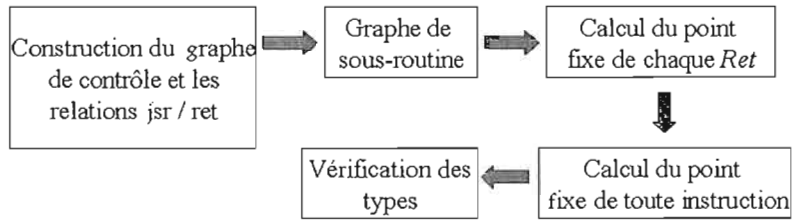


FIGURE 6.3 Les étapes de vérification des types

Nous avons fourni une preuve formelle de la vérification des types en nous basant sur l'analyse de flot de données où chaque fonction de transfert est monotone et distributive, ce qui nous a assuré la construction du plus petit ensemble de types qui atteint chaque instruction, et ainsi d'obtenir une vérification précise et fiable.

6.4 Vérificateur de la synchronisation

La présente spécification de Sun de la MVJ [LY99] ne contient aucune description de la vérification des verrous ni de contraintes sur les instructions `monitorenter` et `monitorexit` lors du processus de vérification du code octet Java. De plus, si il y a un code corrompu au niveau de la gestion des verrous comme dans l'exemple de la figure 6.4, le vérificateur ne le rejettera pas et une erreur sera produite lors de l'exécution de ce code en lançant un `IllegalMonitorStateException` ce qui peut causer l'instabilité la MVJ.

```
0 aload 0
1 monitorexit
2 return
```

FIGURE 6.4 Exemple de code mal synchronisé

Notre apport dans ce mémoire a consisté à améliorer le processus de vérification du code octet Java en intégrant aussi la vérification de la synchronisation.

Notre algorithme utilise le résultat des deux premières étapes réalisées lors de la vérification des types : l'association `jsr/ret` et le graphe d'appel de sous-routines. Ensuite, pour garder la trace des instances de types qui sont verrouillées, nous avons utilisé le même algorithme que celui de la vérification des types mais en adaptant le treillis pour répondre aux nouvelles exigences. Notre abstraction fait ressortir les instances de types d'une manière simple, ce qui rend l'algorithme plus efficace que la vérification de types puisque le point fixe est atteint plus rapidement grâce à la manipulation d'un treillis plus petit que celui utilisé durant la vérification des types.

Finalement, nous espérons que notre travail inspire d'autres projets pour améliorer la vérification du code octet Java et, par conséquent, rendre plus robuste cette riche et intéressante plateforme.

Annexe A

FONCTIONS DE TRANSFERT

Soit C une classe et M une de ses méthodes à vérifier, alors les types manipulés par le vérificateur sont les suivants :

$Primitif = \{int, float, ldouble, hdouble, llong, hlong\}$;

$ref(s)$: référence de signature s ;

Ref : l'ensemble de toutes les références ;

D = l'ensemble des adresses de retour des sous-routines ;

u = le type inutilisé ;

$New = \{(ref(s), o) | ref(s)$ est une référence engendrée par l'instruction
new et o son $offset\}$;

$arrayRef(ref)$ = tableau de références de type ref ;

$arrayOf(t)$ = tableau de références de type t, où t est un type différent du type référence ;

$Array$: l'ensemble des éléments de type $arrayRef(ref)$ et ou de type $arrayOf(t)$;

$amultiNewArray(d, C)$ = l'ensemble de types, composé du tableau de dimension d de références

de type C, de tous les tableaux de dimension inférieure à d et de type C ;

$AmultiNewArray$: l'ensemble de tous les types générés par l'instruction $amultiNewArray$;

$\beta = \{\beta_0, ..., \beta_{hamx-1}\}$: l'ensemble des variables qui représentent les types de la pile ;

$\alpha = \{\alpha_0, ..., \alpha_{emax-1}\}$: l'ensemble des variables qui représentent des variables locales ;

La signature de la méthode à vérifier :

$$Signature = (MethodName, returnType, arg_1, ..., arg_p)$$

Le corps de la méthode est représenté par une liste d'instructions de longueur $codeLength$, et l'instruction à l'index $offset$ est notée par $inst_{offset}$.

D'où l'ensemble des types manipulés par le vérificateur pour une méthode M d'une classe C est le suivant :

$T = \{returnType, arg_1, ..., arg_n\} \cup$
{Tous les types qui sont référés dans le `constant pool` du fichier *.class*}
$\cup\ New \cup Array \cup AmultiNewArray \cup Signature \cup \beta \cup \alpha$

Dans cette annexe, nous utilisons les notations suivantes :

h : la hauteur courante de la pile
l : l'index de la variable locale
$s = (s_0, ..., s_{hmax})$: la pile
$v = (v_0, ..., v_{max})$: les variables locales
ref : un élément de l'ensemble de références R.

aaload

$$aaload_h(s, v) = (s', v)$$

Où :

$$s_h = \{int\}$$
$$s_{h-1} = \{arrayref(ref)\}$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ \{ref\} & \text{si } i = h-1 \\ s_i & \text{sinon} \end{cases}$$

aastore

$$aastore_h(s, v) = (s', v)$$

Où :

$$s_h = \{arrayref(ref)\}$$
$$s_{h-1} = \{int\}$$
$$s_{h-2} = \{ref\}$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i \in \{h, h-1, h-2\} \\ s_i & \text{sinon} \end{cases}$$

aload

$$aload_{h,l}(s, v) = (s', v)$$

Alors :

$$s'_i = \begin{cases} \{v_l\} & \text{si } i = h+1 \\ s_i & \text{sinon} \end{cases}$$

anewarray

$$anewarray_{h,ref}(s, v) = (s', v)$$

Alors :

$$s'_i = \begin{cases} \{ref\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

astore

$$astore_{h,l}(s, v) = (s', v')$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

Et :

$$v'_i = \begin{cases} s_h & \text{si } i = l \\ v_i & \text{sinon} \end{cases}$$

athrow

$$athrow_h(s, v) = (s', v)$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i = 0 \\ s_h & \text{sinon} \end{cases}$$

baload

$$baload_h(s, v) = (s', v)$$

Où :

$$s_h = \{int\}$$
$$s_{h-1} = \{arrayref(ref)\}$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ \{int\} & \text{si } i = h - 1 \\ s_i & \text{sinon} \end{cases}$$

bipush

$$bipush_h(s, v) = (s', v)$$

Où :

$$s'_i = \begin{cases} \{int\} & \text{si } i = h + 1 \\ s_i & \text{sinon} \end{cases}$$

checkcast

$$checkcast(s, v) = (s, v)$$

getfield

$$getfield_{h,t}(s, v) = (s', v)$$

Où :

$$t \in T$$

$$s'_i = \begin{cases} \{t\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

getstatic

$$getstatic_{h,t}(s, v) = (s', v)$$

Où :

$$t \in T$$

$$s'_i = \begin{cases} \{t\} & \text{si } i = h + 1 \\ s_i & \text{sinon} \end{cases}$$

goto

$$goto(s, v) = (s, v)$$

iadd

$$iadd_h(s, v) = (s', v)$$

Où :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ \{int\} & \text{si } i = h - 1 \\ s_i & \text{sinon} \end{cases}$$

if_acmpeq

$$if\_acmpeq_h(s, v) = (s', v)$$

Où :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ \{u\} & \text{si } i = h - 1 \\ s_i & \text{sinon} \end{cases}$$

iinc

$$iinc(s, v) = (s, v)$$

instanceof

$$instanceof_h(s, v) = (s', v)$$

Où :

$$s'_i = \begin{cases} \{int\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

invokeinterface

$$invokeinterface_{h,count}(s,v) = (s',v)$$

Où :

count : est le nombre d'arguments de la méthode invoquée par cette instruction.

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } h - count \leq i \leq h \\ s_i & \text{sinon} \end{cases}$$

invokestatic

$$invokestatic_{h,count}(s,v) = (s',v)$$

Où :

count est le nombre d'arguments de la méthode invoquée par cette instruction.

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } h - count + 1 \leq i \leq h \\ s_i & \text{sinon} \end{cases}$$

lookupswitch

$$lookupswitch_{h,count}(s,v) = (s',v)$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

monitorenter

$$monitorenter_h(s,v) = (s',v)$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

monitorexit

$$monitorexit_h(s, v) = (s', v)$$

Alors :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

new

$$new_{h,ref}(s, v) = (s', v)$$

Où :

$$s'_i = \begin{cases} \{ref\} & \text{si } i = h \\ s_i & \text{sinon} \end{cases}$$

putfield

$$putfield_h(s, v) = (s', v)$$

Où :

$$s'_i = \begin{cases} \{u\} & \text{si } i = h \\ \{u\} & \text{si } i = h - 1 \\ s_i & \text{sinon} \end{cases}$$

swap

$$swap_h(s, v) = (s', v)$$

Où :

$$s'_i = \begin{cases} s_{h-1} & \text{si } i = h \\ s_h & \text{si } i = h-1 \\ s_i & \text{sinon} \end{cases}$$

Bibliographie

[App02] A. W. Appel. *Modern compiler implementation in Java.* Cambridge university press, Second edition, 2002.

[BL00] G. Bigliardi and C. Laneve. *A type system for JVM threads.* July 2000. G. Bigliardi and C. Laneve. A type system for JVM threads. Dept. of Computer Science, University of Bologna.

[Cog01] A. Coglio. *Improving the official specification of Java bytecode verification.* June 2001.

[Cog04] A. Coglio. Simple verification technique for complex java bytecode subroutines. *Concurr. Comput. : Pract. Exper.*, 16(7) :647–670, 2004.

[EB97] S. McDirmid E. and B. Bershard. *Kimera : A Java System security architecture.* 1997.

[EC01] D. Peled E.M. Clarke, O. Grumberg. *Model Checking.* Addison-Wesley, 2001.

[FM99] S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6) :1196–1250, 1999.

[FM03] S. N. Freund and J. C. Mitchell. A type system for the java bytecode language and verifier. *J. Autom. Reason.*, 30(3-4) :147–166, 2003.

[HT98] M. Hagiya and A. Tozawa. *On a new method for dataflow analysis of Java Virtual Machine subroutines.* Information Processing Society of Japan, 1998.

[IK02] F. Iwama and N. Kobayashi. *A new type system for lock primitives.* Tokyo, Japan, 2002.

[JA01] J. McDonald J. Anton. *SPECWARE, Producing Software Correct by Construction.* Kestrel Institute, Palo Alto, California, USA, 2001.

[Jen06] T. Jensen. *Analyse statique de programmes. Présentation du Projet Land au CNRS.* 2006.

[LB98] S. Liang and G. Bracha. *Dynamic class loading in the Java virtual machine.* 1998.

[Ler01] X. Leroy. Java bytecode verification : An overview. *Lecture Notes in Computer Science*, 2102 :265+, 2001.

[Ler03] X. Leroy. Java bytecode verification : Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4) :235–269, 2003.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Second edition, 1999.

[MB97] J. Meyer and T. Bowning. *Java virtual machine.* 1997.

[NOP00] T. Nipkow, D. Von Oheimb, and C. Pusch. *μJava : Embedding a Programming Language in a Theorem Prover.* Foundations of Secure Computation,IOS Press, 2000.

[O'C99] R. O'Callahan. A simple, comprehensive type system for java bytecode subroutines. *ACM*, 1999.

[Pot00] F. Pottier. *Compilation.* 2000. Présentation du cours inf553, compilation, à l'INRIA (Paris).

[PV98] J. Posegga and H. Vogt. *Java bytecode verification using model checking.* 1998.

[Qia99] Z. Qian. *A formal specification of Java virtual machine instructions for Objects, Methods and Subroutines.* Springer-Verlag, London, UK, 1999.

[Qia00] Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4) :638–672, 2000.

[RSB01] J. Schmid R. Stark and E. Borgen. Java and the java virual machine. *Springer-Verlag*, 2001.

[SA99] R. Stata and M. Abadi. A type system for java bytecode subroutines. *ACM Trans. Program. Lang. Syst.*, 21(1) :90–137, 1999.

[SS03] R. F. Stark and J. Schmid. Completeness of a bytecode verifier and a certifying java-to-jvm compiler. *J. Autom. Reason.*, 30(3-4) :323–361, 2003.

[Zha99] J. Zhao. *Analyzing Control Flow in Java Bytecode.* 1999.