Synthèse de gestionnaires mémoire pour applications Java temps-réel embarquées

Guillaume Salagnac

UNIVERSITÉ JOSEPH FOURIER – GRENOBLE I

THÈSE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : Informatique

préparée au laboratoire **Verimag**

dans le cadre de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

présentée et soutenue publiquement

par

Guillaume SALAGNAC

le 10 avril 2008

Synthèse de gestionnaires mémoire pour applications Java temps-réel embarquées

Composition du Jury

Sacha KRAKOWIAK *Université Joseph Fourier – Grenoble I*	Président
Xavier LEROY *Institut National de Recherche en Informatique et en Automatique*	Rapporteur
Gilles MULLER *École des Mines de Nantes*	Rapporteur
Gilles GRIMAUD *Université des Sciences et Technologies de Lille – Lille I*	Examinateur
Christophe RIPPERT *Institut Polytechnique de Grenoble*	Co-directeur
Sergio YOVINE *Centre National de la Recherche Scientifique*	Co-directeur

Table des matières

Chapitre 1

Introduction

1.1 Contexte

Ce travail a été réalisé au laboratoire Verimag. Les travaux menés à Verimag sont principalement axés sur le développement de logiciels et matériels pour les systèmes embarqués.

Historiquement, les systèmes embarqués ont d'abord été utilisés dans le contexte industriel, par exemple pour commander des robots dans une chaîne de production, ou dans le contexte des transports, notamment en tant que systèmes de contrôle ferroviaires. Le matériel et le logiciel étaient alors conçus ensemble, de façon à assurer au mieux la tâche précise du système. L'expertise de Verimag dans le domaine de la conception de systèmes embarqués recouvre plusieurs techniques, de la conception de langages de programmation à la modélisation, en passant par la vérification automatique (*model-checking*).

Au cours des dernières décennies, les progrès de l'électronique miniaturisée ont permis d'embarquer des systèmes informatiques dans des supports de plus en plus divers, étendant ainsi les applications de l'informatique bien au-delà des systèmes de contrôle mentionnés ci-dessus. Aujourd'hui, il est commun de trouver de l'électronique, et en particulier du logiciel, dans une grande partie des appareils que nous utilisons dans la vie quotidienne, comme les téléphones portables, l'électroménager, ou l'automobile. Cette grande diffusion des systèmes embarqués a fait apparaître de nouvelles problématiques, en particulier en ce qui concerne le développement logiciel. Ce travail se situe dans ce dernier domaine, celui du développement logiciel pour les systèmes embarqués.

1.1.1 Contraintes liées à l'embarqué

La principale caractéristique du matériel embarqué est qu'il est soumis à des limitations importantes en termes de ressources, ce qui a un grand impact sur le développement logiciel. Les raisons de ces limitations sont diverses. Il peut s'agir de contraintes de coût de production, d'un besoin de miniaturisation lié à l'application visée, ou de consommation énergétique, lorsque le système est destiné à évoluer en autonomie dans son environnement. Pour toutes ces raisons, la puissance de calcul, la mémoire, et les interfaces de communication disponibles sur un système embarqué sont souvent réduites au strict minimum.

Pourtant, les exigences qui portent sur le système en termes de performances et notamment de réactivité sont importantes. Un système embarqué est typiquement en interaction constante avec son environnement, et doit donc être capable de s'exécuter à une vitesse prévisible. On dit qu'il doit respecter un certain nombre de *contraintes temps-réel*.

Toutes ces difficultés ont amené les industriels à adopter des techniques de développement logiciel spécifiques au monde des systèmes embarqués. Par exemple, les programmes sont encore souvent écrits en assembleur ou en C pour exploiter au mieux les faibles ressources disponibles. Les systèmes d'exploitation utilisés pour servir de support d'exécution à ces programmes sont

très variés d'une plate-forme à l'autre, ce qui limite très fortement la réutilisation de code. Ces pratiques présentent un coût de développement important, et sont aujourd'hui de plus en plus remises en question au profit de technologies plus standard. En particulier, nous nous plaçons dans ce travail dans le cadre de l'utilisation de la technologie Java pour le développement de logiciels embarqués.

1.1.2 Java pour les systèmes embarqués

Le langage Java présente de nombreux atouts vis-à-vis des systèmes embarqués. Il s'agit d'un langage généraliste, très populaire car il est à la fois puissant et facile à maîtriser. Par exemple, le langage Java offre une gestion entièrement automatique de la mémoire, ce qui simplifie grandement le travail de mise au point et de maintenance des programmes. De plus, il est en général compilé non pas vers du code binaire, mais vers un *bytecode* destiné à être interprété par une *machine virtuelle*. Cette architecture apporte une certaine sécurité, car la machine virtuelle pratique des vérifications sur le programme lors de son chargement ainsi que pendant l'exécution. L'usage d'un code intermédiaire permet également une grande portabilité, puisque la machine virtuelle est indépendante du matériel sous-jacent. Un programme peut donc être réutilisé d'une architecture sur l'autre, ce qui est un atout considérable dans le monde des systèmes embarqués où les plates-formes matérielles sont très diverses. Enfin, le bytecode Java est très compact, comparé à un programme binaire équivalent. Cette caractéristique permet l'utilisation de Java sur des supports où l'espace mémoire est très restreint.

La technologie Java est donc intéressante à plus d'un titre pour le développement de programmes embarqués.

1.2 Motivation et problématique

L'utilisation de Java dans le monde de l'embarqué et du temps-réel tarde cependant à se généraliser. Initialement destiné à être universellement portable, le langage Java a en effet évolué vers une complexité peu adaptée au monde de l'embarqué et du temps-réel. Par exemple, la taille de la bibliothèque standard, de plus en plus importante, est difficilement compatible avec des plates-formes aux ressources limitées. De plus, elle ne contient pas les services typiquement utiles aux logiciels embarqués, comme l'accès au matériel.

Le langage Java, bien qu'attrayant pour toutes les raisons citées plus haut, est également considéré comme peu adapté au développement de logiciels soumis à des contraintes temps-réel, comme c'est typiquement le cas dans le contexte des systèmes embarqués.

Un tel logiciel doit présenter des durées d'exécution prévisibles, pour permettre à son concepteur de s'assurer que le système interagira correctement avec son environnement. L'implantation classique des machines virtuelles Java complique beaucoup la tâche des développeurs lorsqu'il s'agit d'écrire du code au comportement prévisible. En effet, plusieurs caractéristiques du langage rendent le temps d'exécution du programme imprévisible. Par exemple, le ramasse-miettes, c'est à dire le système de recyclage automatique de la mémoire, est typiquement amené à interrompre le programme de temps à autre pour examiner la mémoire. Ces *temps de pause* imposés à l'exécution à des moments impossibles à prédire, et pour des durées inconnues, rendent le comportement temporel du programme incompatible avec des contraintes temps-réel.

Le problème auquel on s'attaque dans ce travail est donc le suivant : comment concevoir un système de gestion mémoire qui, tout en restant automatique, présente un comportement temporel prévisible ?

1.2.1 La gestion mémoire

Pour mieux comprendre ce problème, il faut tout d'abord détailler la problématique générale de la gestion mémoire. Le programme utilise pour stocker les données nécessaires à son exécution un espace mémoire appelé le tas. Comme pour les autres ressources, il est important d'avoir une certaine discipline quant à l'utilisation de la mémoire. En particulier, le problème qui se pose est celui de tenir à jour l'inventaire des zones libres, c'est à dire inutilisées, et des zones occupées, dans lesquelles sont situées des données utiles à l'application. Tout au long de l'exécution, le programme alloue de nouvelles zones, il faut donc pouvoir réutiliser celles devenues inutiles.

La gestion mémoire manuelle demande au programmeur de déclarer explicitement toutes les allocations et les désallocations. Un composant logiciel appelé *gestionnaire mémoire* se charge de tenir le compte des zone libres ou occupées. Lorsque le programme a besoin de mémoire, il fait une requête au gestionnaire mémoire qui lui alloue une zone de la taille demandée. Lorsqu'une zone est devenue inutile, le programme la rend au gestionnaire mémoire qui la réintègre dans l'ensemble des zones libres. Cette technique est très contraignante pour le développeur, et sujette à de nombreuses erreurs de programmation.

1.2.2 Recyclage automatique de la mémoire

Le recyclage automatique de la mémoire consiste à dispenser le programmeur de déclarer les désallocations, et à repérer automatiquement les zones inutilisées grâce à un *ramasse-miettes*. Cette technique est synonyme d'un grand confort de programmation et d'une productivité accrue, car elle libère le développeur d'une préoccupation non fonctionnelle. Historiquement plutôt associé à des langages académiques, le recyclage automatique de la mémoire est aujourd'hui présent dans de très nombreux langages de programmation. En particulier, il s'agit d'une des raisons majeures du succès de Java.

Pour déterminer quelles zones sont inutilisées, le ramasse-miettes a cependant besoin d'interrompre l'exécution de temps à autre, afin de parcourir la mémoire. En effet, il lui faut savoir si chaque zone est encore référencée par les variables du programme ou si elle a été abandonnée. Ces interruptions, appelées *temps de pause*, sont déclenchées par l'environnement d'exécution à des instants arbitraires, sans rapport avec l'exécution de l'application. De plus, la tâche de parcours de la mémoire peut prendre un temps très variable, car il faut explorer l'ensemble des structures de données pour déterminer quelles zones recycler et quelles zones conserver en mémoire.

C'est pour cette raison que l'utilisation d'un ramasse-miettes est en général considérée comme incompatible avec la présence de contraintes temps-réel. Si le programme est interrompu intempestivement par le ramasse-miettes pour une durée impossible à prévoir, on ne peut pas non plus prévoir son comportement temporel, et donc s'assurer qu'il respectera bien ses échéances. Les applications temps-réel sont donc en général écrites en renonçant complètement à l'utilisation de ramasse-miettes.

L'objectif de ce travail est de proposer un moyen de concilier recyclage automatique de la mémoire et comportement temporel prévisible.

1.3 Contribution

Nous proposons d'adopter un modèle mémoire au comportement temporel prévisible en organisant la mémoire sous forme de *régions* [Tof98]. Les objets sont ainsi groupés par durée de vie, et désalloués d'un seul coup au moment de la *destruction* de leur région. Une région est une zone de mémoire bien définie, dans laquelle l'allocation se fait en un temps prévisible. De même, l'opération de destruction se fait sans tenir compte des objets contenus dans la région, son comportement temporel est donc également prévisible.

Cette approche est adoptée notamment par la *Spécification Temps-Réel pour Java* (RTSJ [BBD+00]), qui est une extension du langage et de la machine virtuelle destinée aux applications temps-réel. Cependant, la RTSJ suppose de laisser le programmeur organiser lui-même les objets par régions, ce qui est contradictoire avec le modèle de programmation Java dans lequel la gestion mémoire est entièrement automatique. De plus, la RTSJ demande au développeur de fixer la taille des régions qu'il utilise, ce qui est très difficile en pratique dans le modèle de programmation Java.

Pour grouper automatiquement les objets par régions, nous proposons plutôt d'utiliser une technique de *synthèse de régions*, c'est à dire une analyse statique du programme, pour déterminer comment créer et détruire les régions, ainsi que dans quelles régions placer les objets. Des algorithmes de synthèse de régions pour Java ont déjà été proposés [CR04, CCQR04], mais ils souffrent d'un problème que nous avons nommé *syndrome d'explosion des régions*, c'est à dire qu'ils produisent des résultats trop imprécis pour certains programmes, menant à un comportement mémoire inacceptable.

Estimant que même une analyse statique très sophistiquée ne pourra pas résoudre seule le problème de la gestion mémoire, nous proposons d'utiliser un algorithme d'analyse plus simple, et donc plus rapide, afin de pouvoir l'intégrer dans le processus de compilation du programme. Nous nous basons sur une *hypothèse générationnelle* [HHDH02] selon laquelle il existe une forte corrélation entre les connexions qui relient les objets et leurs durées de vie. Des objets connectés ensemble ont en pratique beaucoup plus de chances d'avoir une durée de vie semblable que des objets non connectés. En conséquence, nous proposons de grouper les objets connectés dans une même région, et pour cela de détecter par avance les connexions par une analyse statique.

Nous proposons de détecter statiquement les risques d'explosion de région induits par cette stratégie de placement des objets, et de les faire remonter directement au programmeur. En s'insérant ainsi au plus tôt dans le cycle de développement, cette approche permet de *guider* le développement du programme de façon qu'il reste compatible avec les régions, tout en n'imposant pas au programmeur la tâche de gestion mémoire proprement dite. La simplicité de l'algorithme d'analyse est intéressante à deux niveaux, puisqu'elle permet aux résultats d'être facilement compris par le développeur, et puisqu'elle assure une exécution rapide de l'analyse. Ce dernier point est un facteur important pour l'intégration dans le cycle de développement.

Le programme est ensuite exécuté dans une machine virtuelle dotée d'un gestionnaire mémoire spécifique, dont toutes les opérations de régions s'exécutent en un temps prévisible. La taille des régions n'est pas fixée à l'avance, car elles sont implantées sous la forme de listes de pages mémoire. La taille d'une région peut donc augmenter automatiquement en rajoutant des pages lorsque le programme alloue de nouveaux objets dans la région. La mémoire est recyclée automatiquement par le gestionnaire mémoire qui détruit les régions sitôt qu'elles sont devenues inutilisées.

1.4 Plan

Le chapitre 2 présente un panorama des systèmes embarqués et temps-réel, des contraintes qui leur sont associées ainsi que des techniques utilisées pour programmer des logiciels embarqués et temps-réel. On s'attachera notamment au langage Java, aux difficultés liées à son utilisation dans l'embarqué et le temps réel, ainsi qu'à diverses variantes de Java destinées à réduire ces difficultés.

Le chapitre 3 détaille la problématique de la gestion mémoire, et explore différentes solutions apportées jusqu'ici pour la résoudre. En particulier, certaines de ces solutions sont motivées par les mêmes objectifs que les nôtres, c'est pourquoi nous les étudierons en détail.

Le chapitre 4 est une rapide présentation du domaine de l'analyse statique, et pose quelques bases théoriques de notre travail. Nous donnerons également un panorama de différentes analyses

statiques liées au domaine de la gestion mémoire, car certaines nous seront utiles par la suite.

Le chapitre 5 présente notre approche à la lumière des chapitres précédents. Nous y exposons informellement nos principales hypothèses de travail, et proposons d'utiliser deux d'analyses statiques successives pour décider du placement des objets en régions, ainsi que pour faire remonter au programmeur des informations sur le comportement mémoire probable du programme en régions.

Le chapitre 6 est consacré à la définition d'une nouvelle représentation du programme. En effet, il nous paraît plus naturel de raisonner sur les objets manipulés par un programme et leurs connexions en se basant sur une représentation du code sous forme de graphe. Nous comparons cette approche à celle de Sălcianu et Rinard [SR01] qui utilisent également une analyse de pointeurs, mais beaucoup plus complexe, comme base de plusieurs autres travaux liés à la mémoire.

Le chapitre 7 détaille de façon plus rigoureuse les analyses statiques évoquées au chapitre 5. Nous y donnons une sémantique opérationnelle du langage, ainsi qu'une preuve de correction de notre politique d'allocation en régions. Nous détaillons également l'analyse servant de base à l'interaction avec le programmeur.

Le chapitre 8 décrit l'implantation que nous avons réalisée des algorithmes d'analyse statique ainsi que de l'allocateur mémoire. Grâce à nos résultats expérimentaux, nous validons notre proposition, et montrons qu'il est possible d'exécuter en régions des programmes Java sans avoir à les transformer lourdement. Pour certains programmes, l'exécution en régions n'est pas adaptée, ainsi que l'anticipe l'analyse statique.

Le chapitre 9 conclut ce manuscrit en faisant le bilan des résultats obtenus dans ce travail. Il présente également les limites de notre approche, ainsi que certaines perspectives envisagées pour surmonter celles-ci.

Première partie

Contexte et état de l'art

Chapitre 2

Les systèmes embarqués et temps-réel

Après les ordinateurs, ainsi que les systèmes industriels automatiques, de plus en plus d'objets technologiques de notre environnement deviennent *intelligents* : téléphone portable, carte bancaire, automobile, électroménager... Grâce aux progrès de l'électronique et à la miniaturisation croissante, l'informatique ne se cantonne plus à l'ordinateur mais devient peu à peu omniprésente, utilisant des supports matériels de plus en plus variés. Les contraintes liées à ces supports (taille, performances, ou coût de fabrication), très différentes de celles rencontrées par les ordinateurs de bureau, leur valent le nom de *systèmes embarqués*.

Ces systèmes, embarqués dans leur environnement, doivent souvent interagir avec lui, pour le contrôler (automobile), le surveiller (capteur sans fil), ou pour communiquer (téléphone portable). Pour remplir leur tâche, ils doivent donc être capables de *suivre le rythme* que leur impose le monde extérieur. On parle alors de *systèmes temps-réel* pour les distinguer des systèmes informatiques classiques, dans lesquels la performance est bien sûr importante, mais ne présente pas un côté aussi essentiel. Pour un système classique, le plus important est de s'exécuter le plus vite possible, et la performance est une qualité appréciée. Pour un système temps-réel, la vitesse n'est pas nécessaire ; ce qui importe est d'offrir des durées d'exécution prévisibles, et compatibles avec le rythme de l'environnement.

Dans ce chapitre, nous faisons un panorama des problématiques liées aux systèmes embarqués et temps-réel, afin de motiver notre travail et de le situer parmi les autres solutions existantes à ces problématiques.

2.1 Les systèmes embarqués

Le terme *système embarqué* ne désigne pas une notion rigoureusement définie, mais recouvre généralement tout ou partie des caractéristiques suivantes, souvent exprimées par opposition au système *non embarqué* classique, l'ordinateur de bureau :

Un système embarqué est comme son nom l'indique *embarqué* dans un autre équipement, le plus souvent pour le *contrôler*. Contrairement à un ordinateur classique, il n'est donc pas autosuffisant, en termes de ressources, ou même de raison d'être. Le calculateur de bord embarqué dans une voiture, par exemple, est indissociable de la voiture elle-même, tant physiquement que d'un point de vue d'utilité.

Il n'a pas non plus vocation à l'universalité, contrairement à l'ordinateur, mais est plutôt *dédié* à une ou plusieurs tâches précises. Cette caractéristique est importante, car elle influe fortement sur la conception. Un système embarqué constitue un *ensemble matériel/logiciel* cohérent, souvent conçu en même temps. Par exemple, un téléphone portable sera typiquement doté d'un ensemble

de processeurs dédiés (*Domain Specific Processor*, ou DSP) au traitement du signal plutôt que d'un logiciel qui réaliserait les même fonctions.

Les contraintes de coût de fabrication, de poids, ou de consommation d'énergie limitent fortement les performances du matériel embarquable. La puissance de calcul, la quantité de mémoire, et les autres *ressources* sont donc souvent *limitées au strict minimum*. Par exemple, un industriel fabriquant un appareil destiné à être vendu en grande série comme un baladeur multimédia sera soumis à une forte concurrence. Il devra donc limiter au strict minimum les capacités du matériel embarqué pour réduire le prix de revient.

Malgré ces limitations, les *exigences de sûreté et de fiabilité* que l'on impose à un système embarqué sont souvent très fortes : il n'est pas acceptable, par exemple, de voir le correcteur de trajectoire d'une voiture se mettre à hésiter au moment critique. Ces exigences, typiquement relatives à la tâche de contrôle du système embarqué, incluent souvent des *contraintes temps-réel*, au point que les termes «temps-réel» et «embarqué» sont parfois utilisés comme synonymes.

2.2 Matériel pour l'embarqué

Les systèmes embarqués se distinguent d'emblée des ordinateurs conventionnels par leurs caractéristiques matérielles très variées. Les contraintes de coût et de spécialisation freinent considérablement l'utilisation de matériel *générique*, mais poussent plutôt à des choix spécifiques en fonction des besoins liés au produit. Si les cas les plus simples conduisent à l'adoption d'un circuit électronique classique, les systèmes plus avancés requièrent vite une programmation logicielle. Devenant alors des systèmes informatiques à part entière, ils comportent donc au minimum un processeur ainsi qu'une unité mémoire. Suivant les besoins, on rencontrera aussi des interfaces d'entrées-sorties, ou des composants spécifiques à un certain service comme un récepteur radio, ou un DSP.

Processeur Dans la conception d'un système embarqué, le choix du microprocesseur est un point sensible. En effet, il faut trouver un compromis entre la puissance de calcul nécessaire à l'application et les contraintes comme le coût de fabrication ou la consommation d'énergie. Les applications les plus légères se contentent d'un microcontrôleur, c'est à dire d'une puce qui intègre dans un même boîtier le processeur, la mémoire et les interfaces d'entrées-sorties. Cette solution permet, pour un prix modeste, l'utilisation d'informatique embarquée dans des équipements où les contraintes d'encombrement sont importantes, mais reste très limitée en puissance de calcul. A l'inverse, des systèmes plus complexes comme ceux utilisés dans les télécommunications utilisent typiquement plusieurs processeurs dont certains sont dédiés au traitement du signal. L'architecture obtenue est alors beaucoup plus complexe, mais permet de rendre des services plus avancés.

Mémoire De même que les ordinateurs, les systèmes embarqués utilisent plusieurs types de mémoire : mémoire de travail, mémoire de masse, et mémoire morte.

La mémoire vive (RWM pour *Read/Write Memory*, ou plus souvent, par abus de langage, RAM pour *Random Access Memory*) est indispensable en tant que mémoire de travail, c'est pourquoi elle est présente dans toutes les formes de systèmes embarqués. Certaines applications comme le traitement de flux multimédia (encodage ou décodage audio/vidéo) en exigent même une quantité substantielle. Cependant son coût, son encombrement important, sa volatilité, synonyme de consommation d'énergie, en limitent fortement l'usage sur les plates-formes les plus contraintes.

La mémoire morte (ROM, pour *Read-Only Memory*) au contraire, est persistante, et est économique tant en prix de revient qu'en encombrement. Elle est donc couramment employée

dans les systèmes embarqués, typiquement pour y stocker le logiciel dans les cas où celui-ci n'est pas destiné à changer pendant toute la vie du système embarqué.

Les systèmes qui nécessitent des données modifiables et persistantes doivent donc avoir recours à une forme de mémoire de masse : un disque dur, ou une mémoire non-volatile comme l'EEPROM (*Electrically-Erasable Programmable Read-Only Memory*) ou la mémoire Flash. Ces mémoires ne sont pourtant en général pas utilisées comme mémoire de travail, car leurs performances sont insuffisantes, surtout en écriture. De plus, ces mémoires ont typiquement une durée de vie assez limitée, de l'ordre de quelques milliers d'écritures.

La décision de stocker des données dans l'une ou l'autre mémoire relève donc encore une fois d'un compromis entre les performances (temps d'accès, persistance) et le coût (coût de fabrication, encombrement, consommation énergétique). Cependant, d'un genre de système embarqué à un autre, l'espace disponible peut varier considérablement : si sur un téléphone portable, on peut raisonnablement s'attendre à trouver plusieurs méga-octets, voire dizaines de méga-octets de mémoire vive, à l'inverse, une carte à puce ne disposera que de quelques kilo-octets de RAM.

Lorsque l'architecture le permet, l'utilisation de *mémoire virtuelle* [CV65] permet de simplifier ce problème. Le principe est de considérer toutes les mémoires comme un espace uniforme du point de vue du processeur, et de confier à un coprocesseur spécifique appelé *unité de gestion mémoire* (MMU, ou *Memory Management Unit*) la tâche de *traduire* les adresses virtuelles en adresses réelles. La mémoire virtuelle est en général utilisée conjointement avec un mécanisme de *pagination*, pour permettre au système de déplacer physiquement certaines pages d'une zone de mémoire à une autre. L'idée de ce *va-et-vient* (*swapping*) est de déplacer dynamiquement les pages les plus utilisées vers la mémoire la plus rapide, et de *décharger* les pages les moins utilisées vers une mémoire plus abondante mais plus lente. La présence de l'unité de gestion mémoire permet de faire cette manipulation d'une façon transparente pour le programme, puisque les adresses virtuelles des pages déplacées ne sont pas modifiées.

Dans la plupart des systèmes embarqués, cependant, l'architecture ne comporte pas d'unité de gestion mémoire, et l'utilisation des différents types de mémoire doit être programmée spécifiquement.

Entrées-sorties Un système embarqué n'est pas obligatoirement en contact direct avec son utilisateur. De nombreux systèmes embarqués travaillent en autonomie complète dans leur environnement, on parle alors parfois de systèmes *enfouis*. La présence d'une interface de communication est pourtant nécessaire, pour que le système puisse remplir sa fonction. Suivant les situations, il peut s'agir d'une liaison réseau, d'une connexion directe à l'équipement contrôlé, ou d'une interface spécifique, comme le connecteur des cartes à puce.

Par exemple, un contrôleur de portail automatique disposera comme interface d'entrée d'un récepteur radio associé avec la télécommande, et comme interface de sortie d'une connexion directe avec les pré-actionneurs du moteur.

2.3 Plates-formes d'exécution pour l'embarqué

Le caractère très varié des architectures matérielles embarquées pose des difficultés importantes dans le développement du logiciel associé. À la différence des systèmes d'exploitation classiques, les systèmes d'exploitation destinés à l'embarqué sont souvent spécifiques à une application. En effet, le système d'exploitation doit être parfaitement adapté au matériel pour exploiter au mieux les ressources disponibles.

On distingue principalement deux approches pour la conception d'environnements d'exécution. L'approche *additive* propose d'assembler des composants logiciels préexistants de façon à obtenir les fonctionnalités souhaitées tout en conservant un système minimal. Au contraire, l'approche

soustractive consiste à partir d'un système existant et à le spécialiser au maximum, c'est à dire en retirer le plus possible de fonctionnalités, de façon à réduire les ressources nécessaires.

Dans certains cas, le programme applicatif est même exécuté directement sur le matériel, sans avoir recours à un environnement d'exécution.

2.3.1 Construction de logiciel système : Approche additive

Pour obtenir un système le plus économe possible en matière de ressources, l'approche additive procède par ajouts de composants, éventuellement à partir d'une base minimaliste. Cette construction permet de n'inclure que les services nécessaires à l'application.

Construction statique Le moment le plus naturel pour décider de l'inclusion de fonctionnalités est la compilation du système. Par exemple, OSKit [FBB+97] est un canevas permettant de composer des systèmes à la carte en assemblant des composants logiciels préexistants. Les composants sont souvent issus de portions de code récupérées dans des systèmes d'exploitation classiques, comme Linux, ils n'ont donc pas toujours d'interfaces clairement définies, ce qui rend difficile le remplacement d'un certain composant par un autre. De plus, le code d'origine des composants OSKit, issu de systèmes classiques, n'est pas pas adapté au matériel embarqué.

À l'inverse, le système TinyOS [HSW+00, CHB+01] est destiné à des architectures matérielles très limitées puisqu'il s'adresse uniquement aux réseaux de capteurs. Il s'agit d'un exemple de conception conjointe du système et du langage de programmation. En effet, les ressources disponibles sur une plate-forme très contrainte comme un nœud de réseau de capteurs sont tellement limitées qu'il est impossible d'adopter une approche classique de système d'exploitation. TinyOS est écrit dans un langage *ad hoc* appelé nesC [GLvB+03] dans lequel certaines notions propres au domaine d'application sont disponibles directement au niveau langage. Par exemple, l'interaction avec le matériel ainsi que la communication entre tâches sont abstraites par une même notion d'*évènements*, afin de simplifier et d'uniformiser la programmation du système et de l'application. De même, Un programme nesC est organisé en composants logiciels, qui interagissent avec des composants spécifiques qui donnent accès au matériel. Ainsi le programmeur peut spécifier quels composants il veut inclure dans le système embarqué, ainsi que comment il veut les assembler.

Construction dynamique Dans un souci d'extensibilité, de nombreux systèmes proposent d'ajouter de nouveaux composants dynamiquement, c'est à dire pendant l'exécution du système.

Cette idée découle du principe des micro-noyaux [Tan87, RBF+89], qui supposent de ne conserver dans le noyau que les services indispensables (gestion des processus, allocation des ressources), et de reléguer tous les autres (pilotes de périphériques, pile réseau) dans des modules exécutés en mode non privilégié.

Par exemple, l'architecture Think [Fas01, FSLM02] propose de construire *à la carte* un noyau de système d'exploitation grâce à un mécanisme d'association très flexible [BCS02]. En effet, les composants peuvent non seulement être assemblés lors de la compilation, mais également être modifiés pendant l'exécution par le programmeur. Un système Think est construit autour d'un exo-noyau [EKO95], c'est à dire un noyau qui ne définit absolument aucune abstraction système, mais ne fait que fournir des interfaces logicielles pour accéder au matériel sous-jacent.

Les différents services de base du système sont implantés sous forme de composants à ajouter au-dessus du noyau. Par exemple, la bibliothèque de composants de Think inclut plusieurs composants d'abstraction du matériel (*Hardware Abstraction Layer*), des composants implantant des pilotes, ou des services de plus haut niveau comme des ordonnanceurs de processus ou des gestionnaires de sécurité.

La caractéristique essentielle de l'architecture Think est la flexibilité des systèmes obtenus. Elle permet de substituer à la volée certains composants, par exemple pour remplacer un pilote

matériel générique par une version différente. Néanmoins cette souplesse entraîne des surcoûts à l'exécution puisque les communications entre composants sont réalisés par des interfaces complexes. Les systèmes basés sur Think sont donc plutôt destinés à des systèmes embarqués disposant de ressources assez importantes.

La structure d'exo-noyau, par ses prérequis très réduits, est pourtant particulièrement adaptée aux systèmes embarqués très contraints, tels que la carte à puce. Le mode de développement traditionnel de logiciel pour carte à puce est très coûteux. Les applications sont souvent conçues de façon complètement spécifique, c'est à dire écrites directement dans le langage assembleur propre à la carte, sans perspective d'évolution ou d'extensibilité. Dans l'optique d'une utilisation plus ouverte des cartes à puce, le système Camille [Gri00], basé sur un exo-noyau, est conçu pour permettre le chargement de nouvelles applications *après* la fabrication de la carte. Les nouveaux programmes sont transférés sur le système en utilisant un format de pseudo-code binaire (*bytecode*) appelé Façade. De cette façon, il devient possible de pratiquer sur ces programmes certaines vérifications afin d'assurer l'intégrité du système. Le code Façade ensuite est traduit vers le code binaire de la carte par un compilateur à la volée (*just-in-time compiler*, ou JIT).

Discussion L'approche additive permet d'obtenir un système minimal, qui ne contient que les services nécessaires à l'application. Certains environnements permettent même de remplacer dynamiquement des composants, ce qui permet de faire évoluer l'application après la mise en route du système embarqué. Cependant, si une fonctionnalité n'est pas offerte par la bibliothèque de composants, le programmeur doit l'implanter lui-même, ce qui implique un effort de développement important. De plus, les systèmes obtenus avec cette approche sont très spécifiques. La portabilité du code applicatif est donc grandement compromise.

2.3.2 Spécialisation de logiciel système : Approche soustractive

À l'inverse de la précédente, l'approche soustractive propose de partir d'une base générique, la plus complète possible, et de procéder à une *spécialisation* du système afin de ne conserver que les services utiles.

Adaptation de systèmes d'exploitation classiques Pour doter un système embarqué d'un environnement de développement et d'exécution, une solution classique est de *spécialiser* un système d'exploitation généraliste. Les systèmes *Embedded Linux*, *Windows CE*, ou *Windows XP Embedded* sont des exemples de telles adaptations. Les avantages de ce genre d'approche sont multiples : avec une simple chaîne de compilation croisée, il est facile de développer de nouvelles applications pour le système. De plus, l'environnement d'exécution est proche de la version *classique* du système en question, il est donc possible de porter des programmes existants sans avoir à les réécrire entièrement. Cependant, les ressources matérielles requises pour exécuter ce genre de systèmes sont considérables, par exemple, un système basé sur *Embedded Linux* a typiquement besoin de plusieurs méga-octets de mémoire vive. Ces approches sont donc essentiellement utilisées sur des architectures assez puissantes, comme les téléphones portables, ou les assistants personnels.

Systèmes d'exploitation spécifiques Pour des plates-formes plus contraintes, les systèmes d'exploitation classiques, même une fois adaptés, sont souvent trop exigeants. On utilise alors des systèmes d'exploitation plus spécifiques, conçus directement à destination de matériels aux capacités limitées.

Par exemple, le système eCos [Mas03], tout en s'adressant à des plates-formes aux capacités plus modestes qu'*Embedded Linux*, vise lui aussi une compatibilité matérielle la plus large possible. À la construction du système, un mécanisme de personnalisation permet de désactiver certaines fonctionnalités (pile de protocoles réseau, gestion de la concurrence). Il s'agit cependant d'un

paramétrage manuel, ce qui suppose un faible degré de spécialisation. En effet, il serait très difficile de décider finement quoi désactiver dans le système sans l'aide d'un outil automatique.

A l'inverse, des outils comme Tempo [CHM+98, CLM04] analysent automatiquement le programme, de façon à le spécialiser pour son contexte d'exécution.

Cette démarche de spécialisation automatique est une des caractéristiques essentielles du projet JITS [MCG05, CGVSR05], qui vise à offrir un environnement d'exécution Java Standard pour des systèmes embarqués aux ressources très limitées, comme les cartes à puce ou les nœuds de réseaux de capteurs. Plutôt que de partir d'une spécification restreinte de Java, comme c'est le cas dans les technologies Java Micro Edition et JavaCard, JITS part de la spécification Java d'origine, et compte sur une analyse poussée du système pour réduire par spécialisation les ressources nécessaires à son exécution embarquée. Les différentes variantes embarquées de Java, y compris JITS, sont présentées plus en détail à la section 2.4.

Discussion L'approche soustractive est plus facile à mettre en œuvre que l'approche additive, puisque le système générique qui sert de base est par définition capable de rendre tous les services nécessaires à l'application. Cette caractéristique est importante car elle permet la portabilité d'une application d'un système à l'autre. La difficulté se situe cependant à un niveau non fonctionnel, celui de la taille du système obtenu. En effet, il peut s'avérer difficile de déterminer quelles parties du système peuvent être retirées sans compromettre son exécution future. Le problème peut cependant être attaqué par une analyse automatique du système, ce qui reporte la difficulté vers la conception de cette analyse. L'élaboration du système proprement dit est alors facilitée.

2.3.3 Exécution sans système d'exploitation

Dans un système embarqué, la présence d'un système d'exploitation ne va pas de soi, car il est préférable dans certaines situations d'exécuter le code tel quel, sur machine nue. En effet, les services rendus par le système (gestion des ressources, synchronisation entre tâches concurrentes) ne sont pas toujours nécessaires. L'inconvénient de ce genre d'approche est cependant une difficulté accrue dans le développement et la maintenance du code, puisqu'il devient de plus en plus étroitement lié au matériel.

Programmation en C Plutôt que de programmer entièrement en assembleur, de nombreux projets adoptent le langage C qui permet de raisonner à un niveau d'abstraction plus élevé. Le développement s'en trouve facilité, et de plus il devient possible dans une certaine mesure de réutiliser le code source lors du portage de l'application d'une architecture matérielle sur l'autre.

Langages spécifiques Le langage C n'est cependant pas adapté à tous les types de programmation. Certains domaines d'application se sont dotés de langages *ad hoc* (DSL, ou *Domain-Specific Languages*) pour faciliter le développement. Par exemple, le langage Devil [MM01] permet de générer automatiquement le code de la couche de communication entre le programme et le matériel, de façon à simplifier l'écriture de pilotes matériels. On peut également citer nesC [GLvB+03], le langage d'implantation de TinyOS, destiné aux réseaux de capteurs, ou des langages fonctionnels comme Timber [BCJ+02] ou Hume [Ham00], conçus pour la programmation de systèmes de contrôle temps-réel.

Ingénierie basée sur les modèles Dans le domaine des systèmes critiques, un effort particulier est consacré à la production de logiciels *vérifiés*, c'est à dire pour lesquels on peut garantir certaines propriétés lors de l'exécution. La plupart des langages de programmation conventionnels sont trop expressifs pour permettre les vérifications souhaitées. Pour cette raison, une autre approche du problème propose de travailler à tous les stades du développement en utilisant des

modèles, peut-être moins expressifs, mais plus faciles à vérifier automatiquement. Le modèle n'est traduit en code exécutable que le plus tard possible.

Par exemple, le langage Stateflow [HR04] permet de décrire le comportement du système sous forme d'automates. Le langage Lustre [CPHP87] est un langage dit *flot-de-données*, c'est à dire dans lequel on décrit le système sous forme d'un ensemble de composants qui travaillent chacun sur une séquence (infinie) de valeurs d'entrées et produisent une séquence infinie de valeurs de sortie. Ce modèle est inspiré des circuits électroniques, dans lesquels les composants physiques manipulent également des flots infinis de valeurs. Le projet Ptolemy [EJL+03] vise même à permettre la description du système sous forme de composants hétérogènes, c'est à dire ne partageant pas nécessairement le même modèle d'exécution.

Dans tous ces exemples, une chaîne de compilation est disponible qui permet de générer automatiquement à partir des modèles de haut niveau du code exécutable par le système embarqué. Ce n'est cependant pas le cas dans tous les contextes, le passage d'un niveau à l'autre doit alors être entièrement fait par le développeur.

2.3.4 Discussion

La grande hétérogénéité du matériel utilisé dans le domaine des systèmes embarqués se traduit par une grande hétérogénéité dans le logiciel correspondant. Les technologies disponibles sont différentes sur chaque type de plates-formes, et les programmes sont souvent réécrits spécifiquement pour chaque architecture. Dans une optique industrielle, cette diversité engendre des coûts de conception assez élevés, tant en ce qui concerne les choix de matériel que les développements logiciels associés.

Les systèmes embarqués sont de plus en plus répandus, et leur mise au point subit des contraintes croissantes en matière de coûts de production et de délais de mise sur le marché. Cette évolution conduit les industriels à se pencher de plus en plus vers des technologies standard, au détriment des technologies très spécifiques utilisées traditionnellement. En particulier, on se place dans ce travail dans le contexte de la technologie Java.

2.4 Java pour l'embarqué

Le langage Java, initialement développé par *Sun Microsystems* dans les années 90, est un langage de programmation généraliste qui a rencontré un immense succès. Sa syntaxe inspirée de celle de C et C++, le rend en effet facile d'accès pour des programmeurs habitués à ces langages eux-mêmes très populaires. Le modèle de programmation, en particulier en ce qui concerne la gestion mémoire, est plus simple que celui de C++, ce qui rend le développement bien plus rapide. En particulier, le programmeur n'a pas à se soucier des désallocations mémoire, qui sont gérées de façon transparente par un *ramasse-miettes* (cf section 3.4).

De plus, ce succès est dû notamment à la réputation de grande portabilité du code binaire Java (le *bytecode*), qui n'est pas exécuté par du matériel, mais est interprété par une *machine virtuelle*. L'auteur d'un programme n'a donc en principe pas à se soucier des plates-formes d'exécution sur lesquelles son programme va être utilisé, du moment qu'elles disposent d'une machine virtuelle Java. L'un des slogans commerciaux de Java est d'ailleurs « *Write Once, Run Anywhere* ».

Ce dernier point est aujourd'hui considéré comme allant de soi pour les machines de bureau. Plusieurs systèmes d'exploitation grand public sont même livrés avec une machine virtuelle Java préinstallée, de façon à pouvoir exécuter les applications Java ainsi que *applettes* rencontrées sur le web sans requérir l'installation de nouveau logiciel.

Le langage Java intéresse les concepteurs de systèmes embarqués à plusieurs titres. La grande popularité du langage implique des coûts de développement réduits, puisque des développeurs Java sont très répandus. De plus, la portabilité des programmes permet d'économiser l'effort

de portage pour réutiliser une application sur un nouveau support matériel, ce qui est très recherché dans ce secteur très concurrentiel. Un dernier facteur essentiel est la grande compacité du *bytecode*, en comparaison avec le code binaire natif. Cette caractéristiques est très importante dans un domaine où l'espace mémoire est une ressource précieuse.

Pourtant, l'universalité promise par Java, si elle est bien réelle dans le mode des ordinateurs classiques, est loin d'être acquise lorsqu'on s'intéresse à des plates-formes embarquées. En effet, les différents éléments qui composent un environnement d'exécution Java ne sont pas bien appropriés à un contexte où les ressources sont limitées.

Machine virtuelle Bien que la spécification du langage n'impose pas de contrainte sur l'implantation de la plate-forme d'exécution, de nombreuses machines virtuelles java (JVM, *Java Virtual Machine*) modernes sont conçues pour des raisons de performances autour d'un compilateur à la volée (*just-in-time*, ou JIT) [DS84, ATCL$^+$98] qui traduit dynamiquement le bytecode Java vers du code binaire natif de l'architecture hôte. Si cette technique est bien adaptée à des machines classiques, le volume du code natif obtenu peut se révéler problématique si l'espace disponible sur le système embarqué est limité.

De même, le ramasse-miettes utilisé est souvent du type *générationnel* (cf section 3.4.2.2 page 44), ce qui favorise les performances globales mais présente un coût élevé en espace mémoire. Suivant la puissance de calcul et l'espace mémoire dont dispose le système embarqué, ces techniques peuvent donc s'avérer impraticables.

Bibliothèque standard La bibliothèque standard de Java (appelée parfois «Java Class Library», ou «Classpath») est un de ses atouts majeurs en tant que langage de programmation généraliste. Pourtant, elle constitue plutôt un obstacle à son utilisation dans le monde de l'embarqué. La très grande diversité des services proposés au programmeur se traduit par un volume de code (plusieurs dizaines de méga-octets pour l'édition standard) bien trop important pour un système embarqué. De plus, de nombreuses fonctionnalités d'un ordinateur classique (interface graphique...) n'ont pas de sens dans un système embarqué, ce qui rend inutile une bonne partie de l'API. Au contraire, certains besoins typiques des systèmes embarqués (accès direct au matériel...) ne sont pas couverts par la bibliothèque standard, car ils sont très dépendants de la plate-forme physique d'exécution.

Outils de développement L'environnement d'exécution Java est indissociable des outils de développement et de déploiement correspondants. Cependant, le compilateur, ou le chargeur, ne sont pas utilisables tels quels pour un système embarqué. Par exemple, le format de distribution traditionnel des programmes Java est le fichier `.class`, parfois réuni par lot en une archive `.jar`. Ce format, conçu dans l'optique des *applettes* web, n'est pas adapté au chargement de classes sur une plate-forme aux ressources restreintes, car il requiert de nombreux traitements complexes (vérifications de type, édition de liens...) de la part de la machine virtuelle, et repose sur un accès permanent au réseau.

2.4.1 Compilation statique

Pour réduire la charge de travail de la machine virtuelle, notamment en matière d'efficacité à l'exécution, une approche assez répandue consiste à *compiler* le programme pour l'architecture visée, comme on le ferait pour un langage de programmation classique. On parle alors de compilation *ahead of time* (AOT), par opposition à la compilation *just-in-time*. Le code obtenu est considéré comme étant plus efficace, mais beaucoup plus volumineux. Pour trouver un compromis entre les deux, des outils comme TurboJ [WDDF98, CCF$^+$02] ou Harissa [MMBC97] permettent de ne compiler que certaines parties du programme, et de conserver la majorité du code sous forme interprétée.

Spécifications Java dédiées à l'embarqué

L'ensemble de ces difficultés a conduit la communauté à délimiter plusieurs versions distinctes de la spécification : Java SE (Standard Edition) est la plate-forme destinée aux ordinateurs de bureau classiques, et Java EE (Enterprise Edition) aux serveurs d'entreprise. Pour le monde de l'embarqué, Java ME (Micro Edition) s'adresse aux applications mobiles et cible des architectures comme les téléphones portables ou les assistants personnels (PDA), tandis que JavaCard s'adresse aux matériels très contraints comme les cartes à puce. Ces deux dernières versions de la plate-forme Java n'offrent qu'une version restreinte des possibilités du langage, avec pour objectif de rendre possible son utilisation dans des contextes les plus variés possible.

2.4.2 Java Micro Edition

La spécification Java Micro Edition[1] est née de la volonté de Sun d'adapter la technologie Java à des équipements embarqués qui ne disposent pas de suffisamment de ressources pour proposer la compatibilité Java Standard.

Pour s'accorder le mieux possible aux différentes familles d'équipements, la spécification distingue différentes *configurations*, elles-mêmes découpées en *profils*. Les deux configurations utilisées en pratique sont CDC (*Connected Device Configuration*), qui s'adresse à des équipements fixes, comme les décodeurs de télévision numérique, et CLDC (pour *Connected Limited Device Configuration*), qui vise les équipements mobiles comme les téléphones portables.

Chacune de ces configurations correspond à une certaine machine virtuelle ainsi qu'à une certaine bibliothèque standard. CDC requiert une machine virtuelle complète, et offre une API assez riche, dans laquelle les différents profils correspondent essentiellement à la présence ou non d'une interface graphique. CLDC, par contre, est destinée à des équipement plus contraints, et n'offre plus qu'une version restreinte du langage. Pour simplifier l'implantation de la machine virtuelle, la réflexivité, la finalisation, les nombres à virgule flottante ainsi que les chargeurs de classes personnalisés sont éliminés. Une grande partie des APIs standard sont aussi supprimées, au profit des *profils* spécifiques à chaque type d'application. Par exemple, le profil le plus utilisé est le MIDP (*Mobile Information Device Profile*) qui donne accès à l'écran du téléphone portable, ainsi qu'à une certaine connectivité réseau (requêtes HTTP).

2.4.3 JavaCard

La spécification JavaCard[2] est destinée à des architectures encore plus contraintes, comme par exemple les cartes à puce. C'est la plus limitée des quatre versions officielles de Java, mais elle reste très utilisée, car elle permet de bénéficier de certains avantages de Java, comme la réutilisabilité du code, ou la sécurité à l'exécution, sur des plates-formes où ce genre de propriétés est très recherché. En effet, les cartes à puce sont souvent utilisées dans des contextes où la sécurité est une préoccupation importante, tels que les cartes bancaires, ou la téléphonie.

Cette technologie, même si elle bénéficie d'un soutien très important de la part de Sun, offre seulement des possibilités beaucoup plus limitées que l'édition standard. Par exemple, elle ne supporte pas la programmation concurrente (pas de *threads*), ni l'utilisation de mémoire dynamique (pas de ramasse-miettes, cf chapitre 3), ni une grande partie des types de données du langage (nombres à virgule flottante, chaînes de caractères, etc.). Le chargement dynamique de classes, une opération assez lourde en termes de ressources, est également écarté, abandonnant ainsi l'extensibilité des applications.

Pré-chargement En effet, le chargement de classes par la machine virtuelle Java est un processus assez lourd, comportant la vérification d'intégrité du bytecode, la résolution de références

[1] `http://java.sun.com/javame/`
[2] `http://java.sun.com/products/javacard/`

symboliques (édition de liens) vers les autres classes, ainsi que bien sûr la création des structures de données internes de la machine virtuelle destinées à supporter l'exécution des méthodes. Ces opérations étant réputées trop complexes pour la plate-forme cible, la spécification JavaCard décompose le déploiement d'application en deux phases : pour simplifier au maximum la tâche de chargement exécutée par la carte, une phase de *pré-chargement* est introduite, exécutée sur une station de travail. La vérification du bytecode, l'édition de liens, ainsi qu'une partie de l'initialisation de l'application sont alors exécutées par un outil appelé *convertisseur*. Celui-ci enregistre alors le système obtenu dans un fichier, dans un format dit *pré-chargé* (fichiers `.cap` ou *Converted Applet*). Ce format est très proche de la représentation interne de la machine virtuelle, celle-ci n'a plus au moment du chargement que quelques opérations très simples à réaliser avant de pouvoir exécuter le programme.

Cette architecture de chargement *distribué* permet de déporter la majeure partie du travail de chargement de classes hors de la plate-forme cible, et donc de rendre possible l'utilisation de Java sur des architecture très contraintes. Cependant, elle prive les applications JavaCard de la possibilité de charger de nouvelles classes par la suite, ce qui est pourtant un des atouts importants de la technologie Java. De plus, les opérations effectuées par le convertisseur sont limitées à des traitements très simples (affectations de valeurs constantes), car il n'est pas autorisé à exécuter de bytecode. Si des opérations d'initialisation plus complexes sont nécessaires, elles doivent être programmées explicitement et exécutées sur le système embarqué, ce qui peut à nouveau poser le problème des ressources.

2.4.4 Java In The Small – JITS

Le projet Jits [MCG05, CGVSR05] vise à éliminer le plus possible les restrictions imposées *a priori* à Java sur les plates-formes embarquées. Contrairement aux spécifications pour Java embarqué issues de Sun, l'approche retenue par Jits est de proposer un environnement d'exécution Java Standard, pour faciliter au maximum le développement, tout en visant des plates-formes très contraintes, comme les cartes à puce ou les nœuds de réseaux de capteurs.

Déploiement *in vitro* Dans un esprit analogue au pré-chargement utilisé par JavaCard, le déploiement d'une application utilisant Jits est séparé en plusieurs phases, afin de limiter au maximum les besoins en ressources (mémoire, puissance de calcul) sur le système embarqué. L'objectif est d'exécuter la plus grande partie possible du déploiement *hors* du système embarqué, et de transférer sur celui-ci le logiciel dans un état le plus proche possible de son état dit *utile*, c'est à dire l'état dans lequel le système embarqué est en mesure de rendre le service pour lequel il a été conçu. Il s'agit donc d'une *migration forte*, qui inclut non seulement le code du programme et les données, mais aussi son état d'exécution complet.

L'environnement Jits propose à cet effet un ensemble d'outils, destinés à pré-charger et à démarrer l'application *in vitro*, c'est à dire sur une station de travail classique, là où les ressources sont disponibles en abondance. Cette exécution permet d'effectuer les tâches d'initialisation de la machine virtuelle et de l'application ; elle n'est interrompue qu'au dernier moment. Le système est ensuite *capturé*, c'est à dire qu'une représentation exhaustive de son état interne est établie, puis *transféré* sur la plate-forme embarquée.

Romization Cette dernière opération est appelée *romization*, car l'objectif de Jits est de placer au maximum les données et le code dans la ROM du système embarqué, bien plus vaste que la mémoire vive. A cet effet, un grand nombre d'analyses et d'optimisations sont effectuées sur la représentation du système avant son transfert. Par exemple, le code des initialisateurs de classes, vu qu'il a déjà été exécuté, ne sera plus jamais d'aucun intérêt et peut être éliminé. Une autre optimisation essentielle est la factorisation des données constantes [RD04]. En Java, chaque fichier `.class` définit un certain ensemble de valeurs constantes utilisées par les méthodes de la classe. Après le chargement des classes et le déploiement de la machine virtuelle, une analyse du

système permet d'éliminer les données constantes redondantes, de façon à réduire l'empreinte mémoire du système.

Pour aller plus loin, les outils qui composent JITS permettent de retirer manuellement du système certaines fonctionnalités : par exemple, si l'application ne fait aucun usage des nombres à virgule flottante, leur support peut être désactivé, allégeant ainsi la machine virtuelle embarquée.

Cette démarche, appelée *spécialisation tardive* [Cou06], permet de réduire considérablement les impératifs de ressources de la plate-forme cible, tout en ne renonçant pas *a priori* à telle ou telle fonctionnalité de Java.

2.4.5 Discussion

Comme on vient de le voir, l'intérêt de la communauté des systèmes embarqués pour le langage Java est très important. La facilité de développement, la grande popularité du langage, la portabilité et la compacité du *bytecode* sont des atouts majeurs du langage Java.

Cependant, les obstacles à son adoption sont nombreux, à commencer par la difficulté d'offrir un environnement d'exécution uniforme sur les différentes architectures. La technologie Java, tant du point de vue langage que du point de vue bibliothèque standard, se voit alors déclinée en un ensemble de variantes plus ou moins dégradées de la spécification d'origine. Cette diversité des environnements dits «Java», causée par la diversité encore plus grande du matériel, pèse considérablement sur le succès de cette technologie. En effet, l'incompatibilité qui en découle empêche la réutilisation de code, et nuit à l'efficacité du développement.

Au contraire, nous cherchons dans ce travail à rendre possible la programmation Java Standard dans un contexte embarqué. C'est pour cette raison que nous nous sommes tournés vers l'approche proposée par le projet JITS, qui vise à concilier au cas par cas les contraintes de ressources inhérentes au matériel visé et les besoins en fonctionnalités logicielles liés à l'application.

2.5 Les systèmes temps-réel

Un système *temps-réel* est un système qui est soumis à des contraintes temporelles venues du monde réel. Cette formulation est assez vague, et souvent les notions d'*embarqué* et de *temps-réel* se recouvrent, car elles se rapportent toutes les deux à la relation entre le système et son environnement extérieur.

Ces contraintes temporelles font partie des spécifications non fonctionnelles du système, qui doit donc non seulement produire certains résultats, mais en plus les fournir *au bon moment*. Elles sont généralement exprimées en termes de débit, ou de temps de réponse. Par exemple, un système de vidéo-conférence devra transmettre 30 images par seconde ou un système d'alarme incendie devra alerter les secours en moins d'une minute. Dans le premier cas, une échéance ratée provoquerait une dégradation dans la qualité du service, même si l'application resterait utilisable. On parle alors de temps-réel *souple*, ou *lâche* (*soft real-time*). La plupart des applications multimédia, ainsi que les logiciels interactifs appartiennent à cette catégorie. Dans le second cas au contraire, un retard serait inacceptable car il pourrait causer des dégâts considérables. Les systèmes qui sont soumis à de telles contraintes strictes sont appelés systèmes temps-réel *dur* (*hard real-time*), et nécessitent typiquement l'usage de technologies spécifiques dès la phase de conception. En général, on distingue dans l'exécution d'un système temps-réel au moins deux phases distinctes dans l'exécution. Dans un premier temps, le système *démarre*, se configure, et s'apprête à rendre le service pour lequel il est conçu. Pendant cette phase d'*initialisation*, le temps d'exécution n'est pas un paramètre essentiel. Dans un second temps, le système entre dans une phase *mission* pendant laquelle il va remplir sa tâche proprement dite.

Pour s'assurer du respect des échéances pendant cette phase, il est naturellement nécessaire de disposer d'une connaissance précise, et quantitative, des caractéristiques temporelles des différentes opérations concernées. En effet des performances imprévisibles, tant au niveau matériel que

logiciel, sont un obstacle considérable lorsque l'on cherche à garantir un certain comportement temporel. Contrairement à une idée reçue assez répandue, la vitesse de traitement proprement dite n'est pas un critère essentiel des systèmes temps-réel. La *prévisibilité*, par contre, est indispensable. C'est pourquoi les concepteurs de systèmes temps-réel ou d'environnements d'exécution dédiés au temps-réel, fournissent des efforts considérables pour éliminer autant que possible toutes les sources de non-déterminisme.

2.5.1 Calcul de temps d'exécution

Pouvoir garantir le respect des temps de réponse, les temps d'exécution correspondants doivent être connus à l'avance. Différentes techniques sont utilisées à cet effet, qui cherchent en général seulement à déterminer une approximation supérieure de ces temps d'exécution, appelée *temps d'exécution au pire cas* (*Worst Case Execution Time*, ou WCET). En effet, le calcul exact du temps d'exécution d'un algorithme constitue un problème indécidable, car dans le cas général, il est équivalent au problème de l'arrêt. Les approches qui abordent ce sujet utilisent deux sortes de techniques : d'une part, les méthodes expérimentales, basées sur l'observation du programme à l'exécution et sur des mesures de temps empiriques, et d'autre part, les approches basées sur l'analyse statique du code source du programme.

Méthodes dynamiques La limitation principale des approches *empiriques* est la difficulté de garantir qu'un certain jeu de données d'entrées occasionne bien les comportements au pire cas du système. Puisque le temps mesuré correspond à un comportement réel, les résultats sont fidèles à la réalité, mais constituent seulement un minorant du temps d'exécution au pire cas. Ce genre de techniques est néanmoins beaucoup utilisé, par exemple lorsque le programme est assez simple, et donc que son comportement temporel varie peu en fonction des entrées, ou lorsque le domaine d'entrée est assez réduit [UML06]. Les résultats obtenus correspondent alors au *temps d'exécution dans le cas presque le pire*, ce qui peut être suffisant dans un contexte de temps-réel souple, où un dépassement sporadique des échéances n'est pas catastrophique.

De nombreux appareils multimédia sont conçus en suivant cette approche. Par exemple, lorsqu'on cherche à décider à quelle fréquence doit fonctionner le processeur d'un baladeur numérique, on doit faire face à deux contraintes contradictoires : il faut la réduire le plus possible, de façon à augmenter l'autonomie de la batterie, tout en la maintenant assez importante pour que chaque échantillon puisse être décodé à temps. En d'autres termes, il s'agit de choisir la fréquence la plus basse pour laquelle la durée au pire cas de décodage d'un échantillon reste inférieure à la durée réelle de l'échantillon. Dans une optique pragmatique, et en supposant que le logiciel se comporte de façon suffisamment régulière, le tâtonnement est alors une solution tout à fait valable.

Méthodes statiques A l'inverse, les approches basées sur l'analyse statique ne cherchent pas à *mesurer* les temps d'exécution, mais à les *calculer* par des méthodes formelles. Elles nécessitent donc de disposer d'un *modèle* du programme, obtenu par exemple à partir de son graphe de flot de contrôle. Essentiellement, il s'agit alors de déterminer l'ensemble des différents chemins d'exécution possibles, puis de calculer le temps d'exécution impliqué par chacun d'entre eux. On fait alors face à des difficultés considérables, provenant tant de l'une ou l'autre de ces deux étapes. Évaluer l'ensemble de ces chemins d'exécution est déjà complexe. En effet, dans les langages de programmation classiques, la présence de boucles de contrôle rend souvent l'énumération des chemins impossible. On recourt alors à des techniques symboliques, qui énumèrent seulement *implicitement* ces chemins. Par ailleurs, on demande souvent au programmeur de fournir par des annotations des informations supplémentaires sur le comportement du programme, par exemple pour décrire le nombre maximal d'itérations dans chaque boucle.

Mais d'autre part, un effort substantiel est également nécessaire pour évaluer le temps d'exécution d'un certain chemin. En effet, certaines opérations ont intrinsèquement une durée

d'exécution imprévisible. Par exemple, en Java, chaque instruction `new` risque de provoquer des opérations complexes dans le gestionnaire mémoire, son temps d'exécution au pire cas est donc déraisonnablement élevé. De plus, la sophistication croissante du matériel (mémoire cache, prédicteur de branchement...) rend les performances très variables, et sensibles à des facteurs difficiles à maîtriser. Les temps d'exécution pire cas obtenus par ces approches sont donc par construction *sûrs*, mais sont parfois des sur-approximations considérables par rapport à la réalité.

La recherche autour du calcul des temps d'exécution au pire cas est très active, car les applications de ces résultats sont importants dans de nombreux domaines. Un tour d'horizon plus détaillé de ces travaux est présenté par Colin *et al.* [CPRS03].

2.5.2 Ordonnancement

Suivant la nature de l'application, un système embarqué peut être amené à exécuter plusieurs tâches de façon concurrente. Ces tâches ont en général chacune des caractéristiques différentes en termes de durée, de périodicité, et de contraintes temporelles. La connaissance des temps d'exécution de ces opérations n'est pas en elle-même suffisante pour assurer le respect des échéances : encore faut-il que le support d'exécution sache quelle tâche exécuter à quel moment, c'est à dire sache comment les *ordonnancer*.

On distingue classiquement, en plus des programmes qui tournent en «tâche de fond», sans contraintes temps-réel, deux types de tâches temps-réel, en fonction de leur *loi d'arrivée*. Les tâches *périodiques* doivent être exécutées de façon répétitive, avec un intervalle régulier et connu à l'avance, par exemple «toutes les 20ms». Il s'agit typiquement de traitements en rapport avec le contrôle d'un environnement physique, qui évolue continûment. À l'inverse, les tâches *apériodiques* doivent être exécutées en réponse à un certain évènement dont l'occurrence n'est pas régulière, par exemple «chaque fois que la température dépasse 30°C». Il existe bien entendu dans ces deux catégories de nombreux types de tâches, définis par des lois d'arrivée plus ou moins complexes.

Les caractéristiques d'une tâche temps-réel sont donc définies, d'une part par son temps d'exécution au pire cas, et d'autre part par sa loi d'arrivée. Le problème de l'ordonnancement consiste à planifier l'exécution des différentes tâches du système, dans les limites des ressources disponibles, et de façon à respecter les contraintes temporelles. Ces contraintes peuvent être des échéances, des temps de réponse, et peuvent aussi faire intervenir des dépendances entre les tâches. Selon qu'on s'autorise à interrompre une tâche en cours d'exécution pour la remettre à plus tard ou non, on parlera d'ordonnancement *préemptif* ou *non-préemptif*.

Cette grande variété des hypothèses et la difficulté considérable du problème font de l'ordonnancement un domaine de recherche inépuisable. Historiquement, il s'agit d'une discipline plus ancienne que l'informatique. En effet, la problématique de devoir organiser dans le temps plusieurs tâches aux caractéristiques complexes, par exemple incompatibles entre elles, ou interdépendantes, se pose également dans le contexte de la production industrielle, ou celui des transports. De nombreuses heuristiques ont été mises au point pour résoudre ces problèmes.

Stratégies d'ordonnancement L'algorithme d'ordonnancement le plus simple qui soit est l'*ordonnancement en file* (FIFO, ou *First-In First-Out*), qui consiste à exécuter chaque tâche dans l'ordre d'arrivée. Cette approche est cependant assez rigide, et on utilise en général plutôt un système de *priorités* pour permettre à des tâches plus urgentes, même arrivant plus tard, de prendre la main sur l'exécution. On parle alors *d'ordonnancement à priorité*. Par exemple, le *Rate Monotonic Scheduling* [LL73] consiste à ordonnancer un ensemble de tâches périodiques selon leur fréquence : la tâche la plus fréquente a la priorité la plus élevée, etc. Lorsque les tâches sont plus irrégulières, on utilise par exemple des *priorités dynamiques*. La stratégie *Earliest Deadline First* [LL73] propose par exemple d'exécuter systématiquement la tâche dont l'échéance est la plus proche. Il faut ici donc disposer d'un ordonnanceur préemptif, puisque l'arrivée d'une nouvelle tâche peut en interrompre une autre.

Inversion de priorité Lorsque les différentes tâches du système ne sont pas indépendantes mais partagent un ensemble de ressources, un phénomène appelé *inversion de priorité* peut venir perturber considérablement l'ordonnancement. Par exemple, si une certaine ressource est allouée à une tâche de basse priorité T_1, une tâche de haute priorité T_2 peut se retrouver bloquée à attendre la libération de la ressource, ce qui revient à inverser les priorités des deux tâches. En outre, si une tâche de priorité moyenne T_3 arrive à ce moment-là, et qu'elle ne dépend pas de la ressource en question, alors l'ordonnanceur va interrompre T_1 pour exécuter T_3. On a donc une situation où une tâche de haute priorité T_2 est obligée d'attendre la fin d'une tâche moins prioritaire T_3, alors qu'elles ne dépendent pas des mêmes ressources.

Ce problème est connu notamment pour avoir détérioré considérablement le comportement du robot PathFinder de la NASA [Jon97].

Un certain nombre de solutions ont été proposées pour y remédier, dont notamment le protocole dit d'*héritage de priorité* (*priority inheritance* [SRL90]). Le principe est de modifier la priorité de T_1 au moment où T_2 demande la ressource, en donnant temporairement à T_1 la même priorité que celle de T_2. Cette technique empêche la préemption par T_3, et permet de libérer «le plus vite possible» la ressource pour T_2.

2.5.3 Discussion

Dans le cadre de ce manuscrit, nous ne rentrerons pas plus avant dans les détails de ces problématiques. En effet, ce travail n'apporte pas de nouveau développement en matière d'ordonnancement, ni de techniques de calcul de temps d'exécution au pire cas.

Cette thèse porte sur un autre problème, celui de la prévisibilité des opérations mémoire dans le contexte du langage Java. En effet, comme on le verra par la suite, les techniques actuelles de gestion de la mémoire dynamique ont un comportement temporel imprévisible. En particulier, l'instruction `new` a en général un temps d'exécution pire cas démesuré, très éloigné de son temps d'exécution moyen. Pour cette raison, son utilisation est souvent proscrite dans un contexte temps-réel, car elle rend l'estimation des temps d'exécution, et donc le problème de l'ordonnancement très difficile.

2.6 Java pour le temps-réel

Le langage Java est très populaire, ce qui pousse les industriels à l'utiliser dans toutes sortes d'applications. En effet, le grand nombre de programmeurs Java disponibles sur le marché ainsi que la rapidité et la facilité de programmation réduisent fortement les coûts de développement. Naturellement, un besoin se fait sentir d'utiliser la technologie Java dans des contextes où les applications sont soumises à des contraintes temps-réel.

Cependant, une application Java Standard, exécutée sur une machine virtuelle Java classique au-dessus d'un système d'exploitation généraliste, ne peut espérer garantir que des performances temps-réel souples, avec des temps de réponse de l'ordre de plusieurs centaines de millisecondes. En effet, de nombreux aspects du langage Java ainsi que de ses implantations courantes sont intrinsèquement chargés de non-déterminisme temporel : l'ordonnancement, le chargement de classes, la gestion mémoire, la compilation tardive...

Ordonnancement On a vu que l'ordonnancement est un facteur essentiel d'un système temps-réel. Or, dans un système d'exploitation classique, les différents processus sont ordonnancés selon une politique de *temps partagé*, incompatible avec des contraintes de temps de réponse. Même si on se place dans le contexte d'un système embarqué, dans lequel la machine virtuelle s'exécute en général directement sur le matériel, la spécification Java ne donne aucune garantie sur la qualité de l'ordonnanceur. Le programmeur peut affecter certaines priorités à chacune de ses

tâches, mais ne dispose d'aucun moyen pour s'assurer qu'une tâche de basse priorité ne viendra pas interrompre une tâche de haute priorité.

Chargement de classes Avant de charger une certaine classe, une machine virtuelle respectant la spécification Java doit attendre la première utilisation active de celle-ci par le programme utilisateur. Ce chargement peut prendre un temps très variable, en fonction de la vitesse du support (disque, réseau...) depuis lequel la classe est chargée, de la complexité de la classe, et surtout du nombre d'autres chargements déclenchés transitivement par celui-ci. Comme tous ces chargements sont implicites, leur activité est difficile à prévoir au moment du développement, et peut introduire des délais considérables dans l'exécution du programme à des moments pourtant critiques.

Compilation «juste à temps» Les programmes Java sont distribués compilés sous forme de *bytecode*, un langage intermédiaire indépendant des plates-formes matérielles cibles. Pour les exécuter, une machine virtuelle doit donc *interpréter* chacune de ces instructions bytecodes l'une après l'autre. Pour des raisons de performances, les machines virtuelles Java classiques ont recours une technique de compilation à la volée appelée *compilation juste à temps* (*Just-in-time compilation*, ou JIT) dont l'objectif est de traduire dynamiquement certaines méthodes en code natif de la plate-forme d'exécution. Une fois ainsi compilée, une méthode peut être exécutée directement par le processeur, ce qui est bien plus efficace que de l'interpréter.

S'il est globalement plus rapide, le comportement temporel d'une application est beaucoup plus difficile à prédire en présence de compilation à la volée, car les versions interprétées et compilées de chaque fragment de code présentent des performances très différentes, et il est impossible en pratique de savoir à l'avance quand et où vont intervenir ces compilations. De plus, la tâche de compilation proprement dite peut elle aussi provoquer une interruption à un moment défavorable dans l'exécution du programme.

Gestion mémoire La gestion mémoire automatique est un aspect essentiel du langage Java, et est souvent considérée comme l'un de ses atouts majeurs, notamment vis à vis du langage C++. La programmation Java épargne au programmeur la tâche complexe de décider où et quand désallouer chaque bloc de mémoire, ce qui élimine de nombreuses erreurs de programmation.

Pourtant, utiliser ainsi un *ramasse-miettes* pour récupérer automatiquement la mémoire inutilisée est impopulaire dans un contexte temps-réel. En effet, les ramasse-miettes traditionnels provoquent des interruptions dans l'exécution du programme, à des instants et pour des durées très difficiles à prédire. Pour éviter ce problème, les programmeurs utilisent en général des techniques de gestion mémoire *ad hoc*, comme par exemple des caches d'objets préalloués (*object pooling*).

Les techniques de ce genre sont assez contraignantes, non seulement car elles rompent avec les importants bénéfices associés au ramasse-miettes en termes de génie logiciel, mais également car elles sont souvent incompatibles avec la bibliothèque standard qui crée beaucoup d'objets temporaires. En effet, le programmeur Java est encouragé par le langage à utiliser de nombreux objets auxiliaires. Par exemple, l'opération de concaténation de chaînes de caractères, offerte par un simple opérateur syntaxique, est traduite par le compilateur en une séquence d'opérations qui provoque l'allocation de plusieurs objets éphémères ainsi que des appels de méthodes..

2.6.1 Spécification Temps-Réel pour Java – RTSJ

Toutes ces limitations de la technologie Java ont conduit très vite, dans le cadre du *Java Community Process*, à l'émergence d'un groupe de travail consacré à son adaptation au contexte temps-réel. La *Spécification Temps-Réel pour Java* [BBD+00] (*Real-Time Specification for Java*, ou RTSJ), issue des travaux de ce groupe d'experts, propose d'apporter des modifications

importantes au langage et à la machine virtuelle Java, dans plusieurs domaines : ordonnancement, gestion de la mémoire, synchronisation, gestion des ressources, etc.

Ordonnancement Un système temps-réel requiert un contrôle précis sur la manière dont les différentes tâches sont ordonnancées. Pour pallier la trop grande imprécision de la spécification Java à ce sujet, la RTSJ introduit un mécanisme de priorités strict, et impose la présence d'un ordonnanceur préemptif. Chaque tâche est caractérisée par une priorité fixe, déterminée par le programmeur, et l'ordonnanceur garantit qu'à chaque instant, c'est la tâche active de plus haute priorité qui s'exécute. Elle ne sera interrompue que par l'activation d'une autre tâche plus prioritaire, ou si elle-même décide de rendre la main. La RTSJ utilise également un protocole d'*héritage de priorité* pour éviter les problèmes d'*inversion de priorité*.

Gestion mémoire Les interruptions intempestives causées par le ramasse-miettes sont inacceptables dans un contexte temps-réel. Pour éviter ce problème, les niveaux de priorité associés aux tâches temps-réel sont tous *supérieurs* à celui du ramasse-miettes. Celui-ci ne sera donc jamais susceptible de perturber une tâche critique. Pour être complètement indépendantes du ramasse-miettes, les tâches temps-réel de la RTSJ ne s'exécutent pas en utilisant la mémoire classique de la machine virtuelle, mais dans des *régions* préallouées, de taille fixe, appelées *ScopedMemory areas*. Ces régions sont créées et détruites par le programmeur, qui doit suivre des règles assez complexes pour assurer l'intégrité des données : pour simplifier, ces régions doivent être hiérarchisées entre elles pour former un arbre, et les objets de ces régions ne peuvent pointer que vers des régions parentes. Le modèle mémoire de la RTSJ est étudié avec plus de détails dans la section 3.5.4.

Discussion Toutes ces restrictions rendent en tous cas la programmation RTSJ très complexe, et en particulier fort éloignée de la programmation Java habituelle. En effet, la plupart des motifs de programmation classiques de Java sont incompatibles avec le modèle mémoire de la RTSJ. En particulier, la bibliothèque standard doit être réécrite entièrement[Dau05] pour pouvoir être utilisée dans une machine virtuelle RTSJ. Les patrons de conception traditionnels ne sont également plus valables. Pour combler cette lacune, Pizlo *et al.* [PFHV04] proposent de nouvelles manières de programmer certaines tâches simples, comme par exemple la boucle infinie, ou le partage de données entre deux processus. Nous reviendrons plus précisément sur ce sujet dans la section 3.5.4 consacrée à la gestion mémoire dans la RTSJ.

La spécification RTSJ est tellement complexe que les premières implantations y ont révélé quantité d'erreurs et d'incohérences [Wel04]. Une version révisée a dû être publiée en 2004 [BBD+04], suivie d'une seconde en 2006 [BBD+06]. Dans un esprit de modération, le profil Ravenscar-java [KWK02] définit un sous-ensemble restreint des possibilités de la RTSJ de façon à rendre plus raisonnables les possibilités de programmation.

2.7 Conclusion

On a vu dans ce chapitre un aperçu du monde des systèmes embarqués et temps-réel. Ces systèmes, à cause de leur intégration dans leur environnement, font face à des contraintes en termes de ressources et de performances qui sont inconnues dans le monde des systèmes d'information et de gestion classiques, notamment pour les ordinateurs de bureau.

Longtemps cantonnés à des domaines confidentiels, les systèmes embarqués sont de plus en plus répandus dans la vie quotidienne. Leur coût de développement et de fabrication sont donc des problématiques de plus en plus importantes pour les industriels.

La très grande diversité du matériel complique considérablement le développement, et a amené au cours du temps à l'utilisation d'un grand nombre de technologies logicielles très

spécifiques. Cette hétérogénéité est synonyme de coût, c'est pourquoi des approches basées sur des technologies plus standard telles que Java suscitent de plus en plus d'intérêt.

De nombreux obstacles demeurent cependant à l'adoption de Java dans le monde de l'embarqué temps-réel. Parmi ceux-ci, nous nous intéressons ici plus spécifiquement au problème de la gestion mémoire, et spécifiquement à son comportement temporel, réputé trop imprévisible pour le domaine du temps-réel.

Chapitre 3

La gestion mémoire

Pour stocker les données nécessaires à son exécution, un programme informatique utilise des *variables*, qui correspondent à autant d'emplacements dans la mémoire physique du système.

Les différents langages de programmation offrent au programmeur diverses abstractions permettant d'accéder à cette mémoire, ainsi que des opérations primitives permettant d'y réserver de l'espace ou d'en libérer. Un composant logiciel appelé *gestionnaire mémoire* se charge alors des tâches de comptabilité associées, comme l'inventaire des zones libres et occupées.

L'utilisation de mémoire dynamique offre un confort de programmation considérable au développeur, mais soulève des questions d'implantation parfois difficiles, en particulier dans un contexte embarqué et temps-réel.

Dans ce chapitre, on explore les différentes approches du problème de la gestion mémoire, et leurs avantages et inconvénients respectifs, en particulier vis-à-vis de la prévisibilité en temps des opérations associées.

3.1 Aspects liés au matériel

La mémoire est un élément essentiel de tout système informatique. L'unité arithmétique et logique (UAL), au cœur du processeur, ne serait en effet d'aucune utilité si elle ne disposait pas d'un moyen pour mémoriser les résultats de ses calculs, afin de les réutiliser plus tard au cours de l'exécution du programme. Au plus proche de l'UAL se trouvent les registres internes du processeur, que l'on peut déjà considérer comme une certaine sorte de mémoire. Leur nombre est très limité, mais leur temps d'accès étant optimal, ils sont essentiels à la performance du processeur. En complément, on utilise donc d'autres types de mémoire, de tailles de plus en plus importantes, au prix de temps d'accès de plus en plus longs.

Cette *hiérarchie mémoire* dépend bien sûr du système considéré, mais comporte en général les niveaux suivants : les *registres* sont situés au cœur du processeur. La *mémoire cache* ou *antémémoire* est destinée à masquer les temps d'accès à la *mémoire centrale*, située à l'extérieur au processeur. La *mémoire de masse*, ou *mémoire de stockage*, est persistante, c'est à dire que les données qu'elle contient ne sont pas perdues en cas d'arrêt du système, même si elle n'a pas vocation à les conserver à très long terme, contrairement à la *mémoire d'archivage*.

Sur un ordinateur de bureau typique, la mémoire cache sera partie intégrante du processeur. Pour des raisons de performances, on trouvera même typiquement deux niveaux successifs de mémoire cache, appelés L1 et L2 (*Level 1* et *Level 2*). La mémoire centrale sera constituée des barrettes de mémoire vive installée sur la carte mère. C'est le disque dur de l'ordinateur qui sert de mémoire de masse, et l'archivage est en général effectué manuellement sur des supports externes, par exemple via des bandes magnétiques, ou un graveur de disques. Cette hiérarchie n'est pas tout le temps respectée. En effet, grâce à la présence dans le matériel d'une *unité de gestion mémoire* (MMU), un système d'exploitation classique exploitera typiquement plusieurs types de mémoire différents de façon transparente pour le programme, grâce à des mécanismes

de mémoire virtuelle et de pagination.

La *mémoire virtuelle* [CV65] est une technique permettant de dissocier les *adresses virtuelles* manipulées par le programme des *adresses physiques* où sont stockées réellement les données. L'unité de gestion mémoire (MMU) est un composant matériel chargé de traduire les adresses virtuelles en adresses physiques. Elle est en général utilisée conjointement avec des mécanismes de *pagination*, c'est à dire que l'ensemble de la mémoire est divisé en *pages* de taille fixe. Une table de correspondance est utilisée pour mémoriser quelle page de la mémoire physique contient chaque page de la mémoire virtuelle. Ces deux mécanismes permettent au système d'instaurer un *va-et-vient* (*swapping*) des pages entre différents emplacements physiques, et de reléguer par exemple sur le disque dur les pages les moins utilisées.

D'un autre côté, sur un système embarqué aux ressources limitées, on n'aura typiquement pas besoin d'archivage, et il n'y aura souvent pas de mémoire cache. En effet, la présence de mémoire cache rend les performances temporelles plus difficiles à prévoir, ce qui n'est pas souhaitable si le système est soumis à des contraintes temps-réel (cf section 2.5). La mémoire vive sera assez réduite, du fait de son coût important par rapport à la mémoire de mémoire de masse : par exemple, un baladeur mp3 ne disposera typiquement[1] que de quelques méga-octets de RAM pour plusieurs giga-octets de mémoire flash. De plus, les systèmes d'exploitation pour systèmes embarqués sont plus simples, souvent sans mémoire virtuelle, et doivent donc gérer avec soin les ressources mémoire.

3.2 Exploitation logicielle de la mémoire

3.2.1 Vocabulaire et définitions

On s'intéresse dans ce chapitres aux différentes techniques de gestion de la mémoire de travail. En général, on ne considère pas les registres comme en faisant partie, puisqu'ils sont manipulés directement par le processeur. L'espace mémoire proprement dit est en général appelé *le tas* (ou *heap memory*). Du point de vue logiciel, il se présente comme un vaste tableau de *mots* mémoire, identifiés chacun par son *adresse*.

L'unité de stockage dans cette mémoire est l'*objet*, c'est à dire un bloc de mots mémoire consécutifs. On appellera ces mots les *champs* de l'objet, et plutôt que d'y faire référence par leur rang (premier champ, second champ, etc.) on leur donnera souvent des *noms* plus significatifs selon le contexte. Chaque champ peut contenir soit une *référence* (un *pointeur*, c'est à dire l'adresse d'un autre objet), soit une valeur *scalaire* (c'est à dire un nombre entier, ou en virgule flottante, etc.).

Le tas peut donc se voir sous la forme d'un *graphe orienté*, dans lequel les sommets sont les objets et les arêtes sont les champs de type pointeur. Le programme accédera à ces données grâce à ses variables, globales et locales, qui forment les *racines* du graphe. En pratique, tant les variables globales que la pile d'exécution (qui contient les variables locales) sont bien sûr stockées dans la mémoire, mais on considérera ici qu'elles ne font pas partie du *tas*.

Par exemple la figure 3.1 représente un tas dans lequel pointent trois variables v_1, v_2 et v_3. La variable v_1 désigne l'objet o_1, qui lui-même possède trois champs. Son premier champ, nommé `left` pointe sur l'objet o_3, son deuxième champ est une valeur scalaire, non représentée, et son troisième champ nommé `right` pointe sur l'objet o_2. Le tas forme ainsi un graphe étiqueté, dans lequel les arêtes sont étiquetées par les noms des champs.

On utilise le terme d'objet *vivant* pour désigner les objets qui sont *transitivement accessibles* depuis les racines en suivant des arêtes dans le graphe du tas. A l'inverse, on dira d'un objet qu'il est *mort* (*garbage*) s'il n'existe aucun chemin de pointeurs depuis les racines jusqu'à lui.

[1] `http://ipodlinux.org/FAQ#How_much_RAM_is_inside_the_iPod.3F` (page consultée le 10/08/2007)

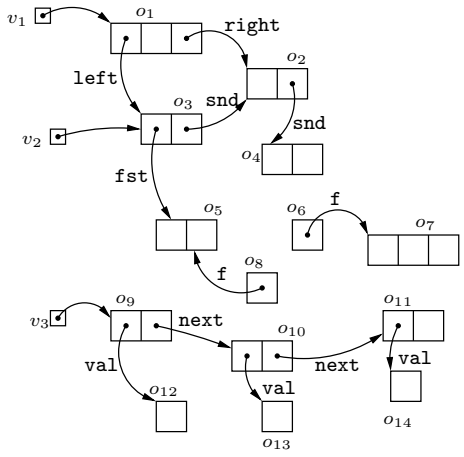


FIG. 3.1 – Le tas mémoire vu comme un graphe d'objets.

Dans la figure 3.1, les objets o_1 à o_5 et o_9 à o_{14} sont vivants. L'objet o_8, bien que connecté à o_5, n'est pas accessible depuis les racines en suivant des pointeurs, et il n'est pas vivant. Les objets o_6 et o_7 sont quant à eux complètement isolés des racines, et ne sont pas vivants non plus.

Cette terminologie n'est pas vraiment pertinente lorsque le programme s'autorise à faire de l'*arithmétique de pointeurs*. En effet, si l'on est capable de *reconstituer* l'adresse d'un objet devenu inaccessible et de créer un pointeur vers lui, on le rend de nouveau *vivant*. Cette technique, couramment employée lorsqu'on programme par exemple en C ou en C++, n'est pas disponible dans le langage Java. La notion d'objet *mort* y est donc alors plus intuitive, puisqu'un objet mort le reste définitivement.

On utilisera parfois la notion de *connexion faible* pour désigner un ensemble d'objets connectés dans le graphe non orienté correspondant au tas. Par exemple, on dira ainsi que les objets o_4 et o_8 sont (faiblement) connectés parce qu'ils sont reliés par un chemin continu de pointeurs, même s'ils ne sont pas accessibles l'un à partir de l'autre.

3.2.2 Gestion statique de la mémoire

3.2.2.1 Allocation statique

La technique de gestion de mémoire la plus simple, et la plus ancienne, est l'*allocation statique*. Elle consiste à fixer entièrement à l'avance l'adresse et la taille de chacune des données. Cette affectation peut être faite automatiquement, au moment de la compilation, ou entièrement par le programmeur, lors de l'écriture du code.

Cette approche peut paraître simpliste, mais elle présente des avantages et est toujours utilisée dans certains contextes. En effet, puisque tout y est fixé, elle permet de pratiquer des optimisations ou des vérifications approfondies sur le programme *avant* de l'exécuter. Ces traitements sont particulièrement intéressants lorsque l'on cherche à produire du logiciel embarqué fiable. Par exemple, on peut s'assurer que le programme ne manquera pas de mémoire lors de son exécution, puisque ses besoins en espace sont fixes, et sont déterminés dès la compilation. Il devient alors possible de se prémunir contre le manque de mémoire, en embarquant dans le système une quantité suffisante de mémoire vive. Le programme peut ainsi s'exécuter directement sur le matériel sans nécessiter la présence d'un environnement d'exécution particulier.

Discussion Si l'allocation statique ne suffit en général pas aux besoins des langages de programmation impératifs (à l'exception notable du Fortran [Jon96]), elle est très bien adaptée à d'autres contextes, en particulier dans l'embarqué temps réel, du fait de son caractère très prévisible.

Elle est utilisée, par exemple, par le code C généré par le compilateur Lustre (cf section 2.3.3). Un programme Lustre, écrit au départ sous la forme d'un système d'équations mutuellement récursives, est traduit en code séquentiel classique au moment de la compilation afin de s'exécuter efficacement sur la plate-forme cible. Le code généré utilise l'allocation statique, de façon à pouvoir être embarqué directement sur le matériel, sans le support d'un système d'exploitation.

3.2.2.2 Allocation en pile

L'allocation statique pure atteint rapidement ses limites, en particulier à cause des restrictions d'expressivité qu'elle impose aux langages de programmation. Entre autres, elle ne permet pas de récursivité, et toutes les tailles des objets doivent être connues à l'avance. Pour aller plus loin, les langages structurés ont introduit la notion de pile : à chaque activation de procédure, un nouveau *cadre de pile* (*stack frame*, parfois traduit par *bloc d'activation*) est créé sur la pile d'exécution, et il sera dépilé automatiquement au moment du retour de la procédure.

Cette technique permet d'améliorer grandement l'expressivité des langages de programmation. En effet, plusieurs activations de la même procédure peuvent cohabiter dans la pile, utilisant différents emplacements pour leurs variables locales, ce qui permet la récursivité. De plus, la taille des objets alloués dans la pile peut être variable, puisque les *cadres de piles* sont créés à la demande.

Discussion L'allocation en pile présente un comportement temporel complètement prévisible : créer un nouveau cadre de pile, ou le désallouer, consiste uniquement à décaler le pointeur de sommet de pile (on parle de *pointer bumping).* Il s'agit donc également d'une opération dont l'exécution est très rapide.

L'allocation en pile est utilisée implicitement dans tous les langages impératifs modernes, comme Java et les variantes du langage C. En général, ces langages permettent aussi l'allocation sur le tas, mais, dans un contexte embarqué, cette dernière n'est pas toujours disponible et le programmeur doit se restreindre à n'utiliser que la pile. On parle alors de *profils d'utilisation*, comme par exemple le profil Misra C [MIS98, Hat04], ou Ravenscar Ada [DB98], qui définissent informellement un sous-ensemble dédié à l'embarqué du langage correspondant. Dans les faits, la spécification JavaCard est également un exemple de langage à allocation en pile. Comme la mémoire n'est jamais libérée, un programmeur JavaCard n'a pas du tout intérêt à utiliser l'instruction `new` une fois passée la phase d'initialisation.

Certains langages de programmation dédiés à l'embarqué adoptent également l'allocation en pile comme seule technique de gestion mémoire (en plus bien sûr de l'allocation statique). Par exemple, on peut noter le langage nesC (cf section 2.3.1), une variante du langage C destinée à la programmation de logiciels pour réseaux de capteurs.

3.2.3 Allocation sur le tas

L'allocation des données sur la pile d'exécution contraint le programmeur à leur donner une durée de vie imbriquée, à l'image des appels de fonction. Pour plus de souplesse, la plupart des langages permettent d'allouer de la mémoire *dynamiquement*, c'est à dire à des points arbitraires de l'exécution du programme. En conséquence, pour éviter d'épuiser l'espace disponible, il devient nécessaire de *désallouer* (manuellement ou automatiquement) les blocs inutilisés, pour pouvoir les réutiliser par la suite.

On désigne sous le nom de *tas* la zone mémoire dans laquelle se font ces allocations. Le terme *gestion mémoire* se réfère souvent à la gestion du tas, car l'allocation statique et la gestion de la pile d'exécution ne posent pas de difficulté particulière. A l'inverse, la question de la gestion du tas est à l'origine d'un domaine de recherche très actif. Pour des plates-formes disposant d'une grande quantité de mémoire, il s'agit de proposer des techniques offrant de bonnes performances.

Dans le cadre des systèmes embarqués et temps-réel, l'allocation sur le tas n'est utilisée qu'avec parcimonie car il est souvent difficile d'en prévoir le comportement.

Dans le reste de ce chapitre, nous allons voir un aperçu de différentes approches qui s'attaquent au problème de la gestion de la mémoire dynamique, ainsi que leurs affinités avec l'embarqué et le temps-réel.

3.3 Gestion manuelle de la mémoire dynamique

Dans la plupart des langages impératifs, la mémoire dynamique est manipulée explicitement par le programme. L'approche traditionnelle consiste à offrir au programmeur des opérations primitives pour l'allocation et la désallocation, comme par exemple les fonctions `malloc()` et `free()` du langage C. Chaque fois que le programmeur a besoin d'un nouveau bloc mémoire de taille n, il appelle `malloc(`n`)` qui lui rend un pointeur p sur le bloc nouvellement alloué, et lorsqu'il n'en a plus besoin, il passe le pointeur p à la fonction `free()`.

Pointeurs invalides Cette apparente simplicité de programmation cache en réalité des difficultés considérables, la principale étant de s'assurer que le programme n'accède jamais à des blocs non alloués, ou précédemment désalloués. On parle de pointeurs invalides, ou *dangling pointers*, pour désigner des références à des zones mémoires où le programme n'a pas le droit d'accéder. Dans le cadre d'un système d'exploitation classique, l'accès invalide est en général intercepté par le système qui interrompt brutalement le programme. Les programmeurs C contournent souvent ce problème en ne désallouant pas du tout la mémoire, qui sera implicitement libérée en bloc à la terminaison du programme. Cette idée n'est pas cependant pas applicable lorsque le programme est destiné à s'exécuter pendant une très longue période, comme c'est typiquement le cas pour un système embarqué.

Cette grande difficulté de déterminer *quand* et *où* désallouer la mémoire est une des faiblesses principales de tous les langages qui utilisent cette technique. En effet, les erreurs d'exécution associées sont très difficiles à reproduire, ce qui complique beaucoup le travail de correction des programmes.

Fuites de mémoire Un autre inconvénient considérable de la gestion manuelle de la mémoire est la nécessité de s'assurer que les blocs devenus inutiles sont bien libérés au fur et à mesure. Si ce n'est pas le cas, ils s'accumulent et finissent par occuper tout l'espace disponible. Le système ne trouve alors plus de place où allouer de nouveaux blocs et rencontre une faute d'exécution. On appelle ce phénomène *fuite de mémoire* (ou *memory leak*).

D'un certain côté, on peut distinguer deux sortes de causes qui provoquent des fuites de mémoire : les fuites *logiques*, et les fuites *par inadvertance*.

Les premières sont dues à un nombre toujours croissant d'objets vivants, souvent à cause d'une *erreur de conception* du programme. Par exemple, une liste chaînée à laquelle on ne ferait que rajouter des éléments, sans jamais les désallouer, finira tôt ou tard par occuper toute la mémoire. Le programme n'est pas vraiment incorrect fonctionnellement parlant, car il s'exécuterait correctement si l'on disposait d'une quantité de mémoire suffisante. En pratique, il conduira quand même souvent à une erreur d'exécution, car l'espace mémoire se retrouvera entièrement occupé.

Les secondes sont dues à un nombre toujours croissant d'objets morts non désalloués, qui occupent donc toujours de l'espace. Si le programmeur modifie son programme pour retirer les maillons inutiles de sa liste chaînée, mais qu'il ne les désalloue pas, le problème n'est pas réglé.

D'un autre côté, cette distinction peut s'avérer assez artificielle selon les langages et les modèles de programmation considérés. En effet, en présence d'arithmétique de pointeurs, la notion d'objet *mort* est difficile à définir, puisqu'un objet peut revenir à la vie. Inversement, un

objet qui reste indéfiniment accessible (au sens du graphe du tas), mais que le programme ne va plus jamais utiliser, ne mérite pas vraiment le qualificatif de *vivant* [MFH95].

Dans tous les cas, le problème des fuites de mémoire est omniprésent dans les projets de développement logiciel, car il est la source de nombreuses erreurs d'exécution très difficiles à reproduire, donc à éliminer.

Fragmentation du tas – Gestionnaires mémoire Indépendamment des difficultés de programmation qu'elle introduit, la gestion manuelle de la mémoire dynamique n'est de toute façon pas adaptée à l'embarqué temps-réel. En effet, le modèle mémoire sous-jacent souffre d'un phénomène de *fragmentation* progressive qui mène à des comportement temporels impossibles à maîtriser. Le problème se manifeste lorsque l'exécution du programme se prolonge : au fur et à mesure des allocations et des désallocations, les zones libres et occupées s'éparpillent dans l'ensemble du tas, jusqu'à constituer un vrai gruyère. L'inconvénient de cette situation est que le système peut se retrouver dans l'incapacité de satisfaire une requête d'allocation alors que la quantité de mémoire disponible est bien présente, mais qu'il n'y a à cet instant aucun bloc contigu de taille suffisante.

Ce phénomène est illustré sur la figure 3.2. Le tas initial est entièrement libre en (a). Le programme demande ensuite une zone de taille égale 50% du tas, le gestionnaire mémoire lui alloue donc la première moitié de la mémoire (en grisé) et on obtient le tas (b). Le programme demande ensuite une zone de taille 25%, et on obtient le tas (c). Un problème de fragmentation apparaît lorsque le programme libère la première zone qu'il avait allouée. On obtient le tas (d), dans lequel l'espace libre (en blanc) est *fragmenté*. Bien que 75% de l'espace soit disponible, le gestionnaire mémoire ne peut pas satisfaire une demande d'allocation de cette taille, car il n'y a pas en mémoire de zone assez grande.

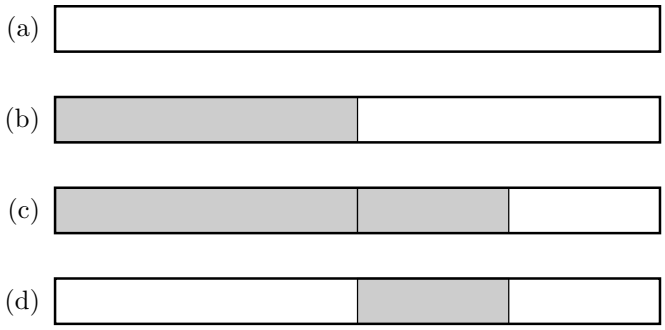


FIG. 3.2 – Fragmentation progressive de la mémoire au fur et à mesure des allocations et des libérations.

Pour lutter contre ce problème, les différents *gestionnaires mémoire* utilisent des stratégies heuristiques censées réduire la fragmentation. Par exemple, dans le choix du bloc libre à réserver pour satisfaire une allocation, on peut citer les stratégies *first-fit*, qui consiste à choisir le premier bloc assez grand, et *best-fit*, qui recherche le bloc le plus ajusté. La seconde est beaucoup moins rapide, mais provoque moins de fragmentation. Un autre moyen de lutte contre la fragmentation est de *recoller* des blocs libres adjacents, de façon à défragmenter le tas au moment des désallocations. Il existe ainsi une diversité très importante d'algorithmes de gestion mémoire [WJNB95].

Ces différentes techniques ont une efficacité très variable [BZM02], mais même les plus efficaces ne visent que la performance globale, sans souci de la *prévisibilité* des temps de réponse. Leur usage dans un contexte temps-réel est donc problématique, car il est en pratique impossible de prédire fidèlement le comportement temporel au pire cas de l'allocation.

3.4 Recyclage automatique de la mémoire

L'utilisation de mémoire dynamique offre un confort de programmation considérable, mais comme nous l'avons vu, la libération manuelle des zones inutilisées est souvent trop difficile. L'alternative consiste à confier entièrement la tâche de la gestion mémoire à l'environnement d'exécution. Le programmeur doit toujours réserver, explicitement ou non, la mémoire qu'il veut utiliser, mais il n'a plus à se préoccuper de la libérer, elle est récupérée automatiquement. Le terme anglais *garbage collection* (GC, parfois traduit par *glanage de cellules*, ou *ramassage de miettes*) désigne l'ensemble des techniques permettant d'automatiser cette détection des zones mémoires inutilisées (*garbage*) en vue de leur réutilisation [Jon96, Wil92].

Le concept (ainsi que le nom) de *ramasse-miettes* provient des premières implantations du langage Lisp, dans les années 60 [McC60]. En effet, le Lisp est un langage *orienté expressions*, qui ne fait pas de distinction entre les données et le code du programme, indifféremment représentés en mémoire par des listes chaînées. Entre autres innovations, le Lisp est un langage dit *fonctionnel*, c'est à dire que l'exécution d'un programme peut inclure la création à la volée de nouvelles fonctions, grâce à la présence d'un interpréteur dans l'environnement d'exécution. L'allocation y est donc implicite, et les durées de vie des différents objets ne suivent pas de scénario particulier. L'allocation en pile ou la désallocation manuelle ne sont donc pas utilisables dans ce contexte.

Pour résoudre le problème de la gestion mémoire, les auteurs du *LISP Programming System* ont donc décidé de décharger complètement le programmeur de cette tâche, et de la confier à l'environnement d'exécution. Les allocations et les désallocations se font de façon transparente. Lorsque le système arrive à court d'espace et ne peut plus satisfaire une requête d'allocation, il interrompt le programme, et parcourt la mémoire pour repérer et recycler les blocs inutilisés.

> *Nothing happens until the program runs out of free storage. When a free register is wanted, and there is none left on the free-storage list, a reclamation cycle starts.* [McC60].

3.4.1 Principe

L'objectif d'un ramasse miettes est de déterminer, parmi l'ensemble des objets présents en mémoire, lesquels sont encore vivants et lesquels sont devenus inaccessibles, de façon à *recycler* l'espace occupé par les objets morts. Ainsi, l'environnement d'exécution se réapproprie automatiquement la mémoire lorsque le programme ne l'utilise plus. Cette section présente le principe général des deux techniques principales utilisées à cet effet : le marquage-balayage, et le comptage de références. Il en existe bien sûr d'innombrables variantes et améliorations, comme nous le verrons plus tard, mais tous les ramasse-miettes utilisent une combinaison de ces deux techniques [BCR04]. La première «nettoie» périodiquement le tas, tout en prenant soin de conserver les objets vivants. La seconde cherche plutôt à détecter la mort des objets pour les recycler sur-le-champ.

3.4.1.1 Marquage - Balayage

Chronologiquement, la première [McC60] technique proposée pour caractériser les objets vivants est celle dite du marquage-balayage (ou *mark-and-sweep*). Elle découle directement de la définition récursive d'*objet vivant* que nous avons adoptée : un objet est vivant s'il est pointé par une racine ou par un autre objet vivant.

Lorsque le système n'arrive pas à satisfaire une requête d'allocation à cause du manque de mémoire, il suspend l'exécution du programme utilisateur et entame un cycle de collecte. Le ramasse-miettes commence par *parcourir* récursivement les objets accessibles depuis les racines, c'est à dire les variables globales et les pointeurs présents dans la pile d'exécution. Chaque fois qu'un objet est parcouru, il est *marqué* pour mémoriser qu'il est accessible. Lorsque l'algorithme ne découvre plus d'objets à marquer, c'est qu'il a parcouru l'ensemble des objets vivants. La

seconde phase de la collecte consiste alors à *balayer* l'ensemble du tas, en récupérant au passage tous les objets non marqués.

Inventaire des racines Pour pouvoir parcourir le graphe du tas, il est nécessaire d'en connaître les racines, c'est à dire les variables globales et locales du programme. Pour faire l'inventaire de ces dernières, le ramasse-miettes doit parcourir la pile d'exécution, et déterminer pour chaque mot s'il s'agit d'une valeur scalaire ou d'une référence.

Suivant le contexte, cette opération peut être d'une complexité très variable [Jon96]. Dans le contexte d'un langage interprété comme Java, la tâche est relativement simple. La pile d'appels est entièrement sous le contrôle de la machine virtuelle, il est donc possible de déterminer le type des valeurs qu'elle contient. A l'inverse, pour un langage compilé faiblement typé, comme le C, l'absence d'informations de typage complique considérablement l'opération de recherche de pointeurs. Il faut donc examiner avec prudence l'ensemble des variables de la pile, mais également les registres du processeur.

Marquage La phase de marquage suppose de parcourir récursivement l'ensemble des objets vivants, en commençant par les objets pointés directement par les racines, puis en suivant leurs champs de type pointeur, et ainsi de suite. Là encore il faut pouvoir différencier, à l'intérieur d'un objet, un pointeur d'une valeur scalaire, ce qui est plus ou moins difficile suivant la sophistication de l'environnement d'exécution et le langage de programmation considéré.

L'algorithme de parcours doit *marquer* les objets déjà atteints pour ne pas les traiter plusieurs fois. Chaque objet doit donc contenir un emplacement (au moins un bit) permettant d'enregistrer cette information.

Balayage Une fois que le marquage a parcouru tous les objets vivants, ceux qui restent sont réputés morts (*garbage*). Pour les désallouer, le ramasse-miettes *balaye* le tas d'un bout à l'autre, à la recherche d'objets dont le bit de marquage n'a pas été positionné. Au passage, il efface aussi la marque des objets vivants, de façon à préparer le prochain cycle de collecte.

Remarques La notion de ramasse-miettes, ainsi que l'algorithme de marquage-balayage, sont indépendants du modèle mémoire sous-jacent, et en particulier de la représentation des zones libres et occupées. *Désallouer* un objet signifie donc simplement «signaler au gestionnaire mémoire que cette zone est maintenant disponible pour l'allocation». Le gestionnaire mémoire doit donc faire face à la problématique de la fragmentation, et aux performances temporelles imprévisibles associées.

Temps de pause Pendant son parcours de la mémoire, le ramasse-miettes interrompt temporairement l'exécution du programme. Cette interruption, appelée *temps de pause*, est susceptible d'intervenir à *chaque* allocation, et son temps d'exécution au pire cas est très important car il inclut le pire cas du parcours de la pile, le pire cas du marquage et le pire cas du balayage. Du point de vue du programme, utiliser un tel ramasse-miettes pour gérer la mémoire implique donc un temps d'exécution au pire cas de l'allocation très long. Bien sûr, en pratique, la plupart des allocations seront exécutées en un temps beaucoup plus court, mais dans un contexte temps-réel, on ne retiendra que le pire cas, souvent insatisfaisant.

3.4.1.2 Comptage de références

A l'inverse du marquage-balayage, un ramasse-miettes à comptage de références [Col60] ne compte pas sur un parcours exhaustif du tas pour reconnaître les objets vivants, mais garde trace, à chaque modification de pointeur, de l'information d'accessibilité.

L'idée est de comptabiliser, pour chaque objet, le nombre d'objets vivants qui pointent directement sur lui. Ce *compteur de références*, un champ supplémentaire ajouté à chaque objet, est mis à jour automatiquement chaque fois qu'un pointeur, y compris les racines, est dirigé vers l'objet. Lorsque le compteur d'un objet tombe à zéro, cela signifie que le dernier pointeur vers lui vient d'être retiré. L'objet peut alors être désalloué, et donc le compteur de références de tous les objets qu'il pointait doit être décrémenté, et ainsi de suite. Cette technique permet de détecter plus précisément le moment où un objet cesse d'être accessible que par marquage-balayage, par contre elle nécessite un entier supplémentaire par objet, ce qui constitue un surcoût important.

De plus, cette désallocation est récursive, et donc risque de prendre un temps imprévisible. Les *temps de pause* provoqués par un ramasse-miettes à comptage de références sont donc associés non plus aux allocations mais aux écritures de pointeurs. Comme pour le ramasse-miettes à marquage-balayage, ils sont susceptibles d'intervenir sur *chaque* écriture de pointeurs, et de durer un temps imprévisible.

Barrières en écriture La mise à jour du compteur de références doit se faire automatiquement, sans que le programmeur n'ait à s'en préoccuper. Chaque écriture de pointeur, puisqu'elle modifie les relations d'accessibilité, doit donc s'accompagner de l'ajustement de *deux* compteurs : ceux des objets pointés respectivement avant et après la modification. Cette modification de la sémantique du langage est désignée sous le nom de *barrière en écriture* (*write-barrier*). Dans une machine virtuelle, ajouter une barrière signifie simplement modifier l'interprétation de l'instruction d'écriture de champ. Par contre, dans un langage compilé, il faut que le compilateur émette du code assembleur à chaque écriture pour réaliser la barrière.

L'utilisation de barrières permet de répartir naturellement la tâche de comptabilité tout au long de l'exécution. Cependant, elles ajoutent un surcoût important aux opérations mémoire : chaque écriture de pointeur est accompagnée de deux écritures de compteur. L'exécution de l'ensemble du programme peut s'en trouver considérablement ralentie. En conséquence, de nombreuses techniques ont été proposées pour réduire autant que possible le nombre de barrières ajoutées dans le programme. Par exemple, le *comptage de références différé* (*deferred reference counting*) est une technique d'optimisation qui consiste à retirer automatiquement certaines barrières lors de la compilation, typiquement celles liées aux variables locales, lorsque le compilateur est sûr qu'elles sont inutiles ou redondantes.

Collecte des structures cycliques La limitation principale du comptage de références est pourtant d'une autre nature : à lui seul, un ramasse-miettes simple à comptage de références est incapable de détecter les structures de données cycliques inaccessibles [McB63]. Alors qu'au premier abord, cette limitation peut paraître mineure, elle est d'une importance considérable en pratique, car de nombreuses structures de données usuelles produisent des graphes d'objets fortement connexes.

Par exemple, dans la figure 3.3, la liste doublement chaînée $o_2 - o_3 - o_4$ est initialement accessible seulement via l'objet o_1. Les compteurs de références sont représentés sur la gauche des objets : on suppose que l'objet o_1 est vivant, et qu'il est directement pointé par n objets. Le programme effectue une opération qui retire le lien de o_1 vers o_2 (par exemple en effaçant le pointeur, ou en lui affectant une nouvelle cible), rendant donc la liste $o_2 - o_3 - o_4$ inaccessible. À cause du pointeur venant de o_3, le compteur de références de o_2 ne tombe pas à 0 mais seulement à 1, et donc le ramasse-miettes ne se rend pas compte que la liste entière est maintenant inaccessible.

Cette limitation est un défaut majeur de la technique de comptage de références, car si un programme génère beaucoup de structures cycliques inaccessibles, elles ne seront jamais désallouées et la fuite de mémoire finira par provoquer une faute d'exécution. Pour cette raison, un ramasse-miettes à comptage de références n'est jamais utilisé seul, mais est toujours *secondé* par un autre ramasse-miettes, par exemple à marquage-balayage [RF02]. Le second ramasse-miettes est déclenché lorsque la mémoire est épuisée, de façon à récolter les cycles d'objets morts

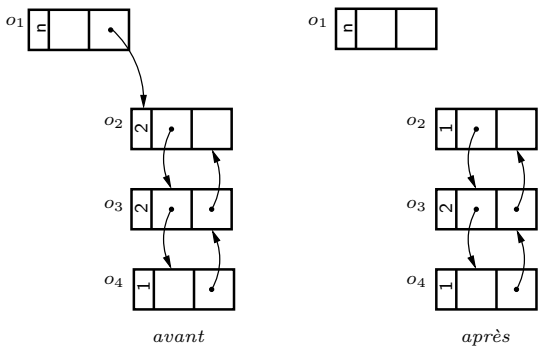


FIG. 3.3 – Comptage de références dans une structure cyclique : après l'effacement du pointeur, l'objet o_1 ne pointe plus sur o_2, mais le compteur de références de o_2 reste à 1.

que le comptage de références a omis.

3.4.2 Variantes réalistes

Les techniques abordées dans la section précédente ont été présentées jusqu'ici de façon assez simpliste, et ne sont pas représentatives des ramasse-miettes actuels. En particulier, elles ne sont efficaces ni en termes de comportement temporel, ni d'occupation de la mémoire. Dans cette section, on va s'intéresser à quelques améliorations que l'on peut apporter à ces techniques pour en améliorer les performances.

3.4.2.1 Ramasse-miettes concurrents et incrémentaux

Temps de pause Parmi les défauts les plus reprochés aux ramasse-miettes, a longtemps figuré l'importance excessive des *temps de pause* infligés au programme de l'utilisateur [Wil92]. En effet, les techniques de collecte naïves nécessitent d'interrompre l'exécution du programme pendant la totalité du cycle de collecte (*stop-the-world GC*). Par exemple, le ramasse-miettes à marquage-balayage présenté à la section 3.4.1.1 doit parcourir l'intégralité des objets vivants puis balayer l'ensemble du tas avant de rendre la main à l'application.

Cette faiblesse est particulièrement problématique dans un contexte temps-réel, par exemple lorsque le programme est soumis à des contraintes de temps de réponse : si le processeur est monopolisé par le ramasse-miettes, le système ne peut pas traiter une éventuelle requête. S'il abandonne au contraire le cycle de collecte, il devra le reprendre de zéro, et le problème se pose à nouveau. En effet, l'exécution du ramasse-miettes doit se faire de manière atomique.

Le ramasse-miettes et le corrupteur Pour pallier ce problème sont apparues les premières propositions de ramasse-miettes *concurrents*, c'est à dire exécutés en parallèle de l'application [DLM$^+$78]. Pour obtenir un parallélisme réel, il faut disposer de plusieurs processeurs, et réserver l'un d'entre eux à l'activité du ramasse-miettes. Cette possibilité est assez rare dans le monde des systèmes embarqués, où les ressources en puissance de calcul sont plus contraintes. Dans un contexte mono-processeur, le parallélisme est simulé par l'entrelacement des tâches, et on parle alors de ramasse-miettes *incrémental*.

Un ramasse-miettes incrémental n'effectue donc pas son cycle de collecte en une seule fois, et par conséquent il doit tenir compte des modifications que le programme utilisateur apporte entre-temps aux objets du tas. On peut ainsi parler de *corrupteur* (le terme anglais est *mutator*) pour désigner le programme applicatif par opposition au ramasse-miettes, car il *corrompt* le tas à l'insu de ce dernier.

Désallocation paresseuse L'idée d'entrelacer l'exécution du programme et du ramasse-miettes peut également s'appliquer au comptage de références : en effet, de par sa nature récursive, la tâche de désallocation associée à la mort d'un objet peut être à l'origine d'un temps de pause conséquent.

La désallocation paresseuse [Wei63] est une technique qui permet de limiter cette quantité de travail : à chaque fois qu'un objet devient inaccessible, il n'est pas désalloué directement mais simplement mémorisé sur une pile spécifique (la *free-stack*), sans s'intéresser à son contenu. Lors de la prochaine allocation, le système va piocher un objet libre sur le sommet de la *free-stack*, et c'est seulement à ce moment qu'il va parcourir ses champs et décrémenter les compteurs des des objets référencés. Si un compteur tombe alors à zéro, l'objet correspondant est à son tour empilé sur la *free-stack*.

Cette technique permet d'éviter les temps de pause dûs aux désallocations en cascade. Par exemple, si le programme retire le dernier pointeur vers un objet qui se trouve être la racine d'un arbre, la désallocation récursive pratiquée par un ramasse-miettes naïf à comptage de références va libérer l'objet, parcourir ses champs, décrémenter les compteurs des objets référencés, qui vont donc tomber à zéro, et ainsi de suite. La désallocation paresseuse va uniquement empiler la racine sur la *free-stack*, et les objets ne seront libérés qu'au fur et à mesure des allocations suivantes. Le travail de désallocation se trouve donc naturellement réparti tout au long de l'exécution de l'application.

3.4.2.2 Ramasse-miettes compactants

Les ramasse-miettes présentés jusqu'ici souffrent d'un problème déjà rencontré par les gestionnaires mémoire explicites, celui de la fragmentation. En effet, au fur et à mesure que des blocs de mémoire sont alloués et libérés, il devient de plus en plus coûteux de rechercher les blocs libres, et une allocation peut même échouer par manque d'une assez grande zone libre contiguë. Pour éviter ce phénomène, certains ramasse-miettes prennent le parti de *déplacer* les objets de façon à défragmenter le tas.

Ramasse-miettes à copie L'exemple le plus simple de ramasse-miettes compactant est celui dit à *demi-espaces* (*semi-space garbage collector* [FY69]). Le principe est de diviser le tas en deux moitiés, l'une contenant les données actives (le *fromspace*), et l'autre étant laissée «en jachère» (le *tospace*). L'allocation ne requiert plus la recherche d'un bloc libre, mais se fait *sur le tas*, c'est à dire simplement en décalant un curseur dans le *fromspace*. Chaque fois qu'un objet est alloué, on décale d'autant le curseur qui indique où commence l'espace libre, c'est à dire qui pointe au sommet du tas.

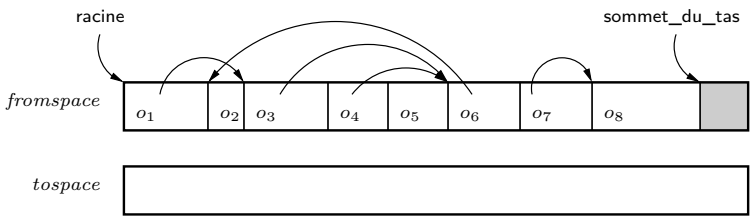


FIG. 3.4 – Ramasse-miettes à copie. Le programme demande l'allocation d'un objet de même taille que o_8, mais il n'y a plus assez d'espace libre (en grisé).

Le cycle de collecte se déclenche lorsque le *fromspace* est plein, et qu'il n'est plus possible d'y allouer un nouvel objet (cf. figure 3.4). Comme dans un ramasse-miettes à marquage-balayage, les objets vivants sont alors parcourus à partir des racines, mais à la place d'être marqués, ils sont *copiés* dans le *tospace*, en le remplissant au fur et à mesure (cf. figure 3.5). Cette opération

est délicate, car elle implique de modifier le contenu des objets et des variables locales actives pour rectifier la destination des pointeurs et préserver la cohérence du tas.

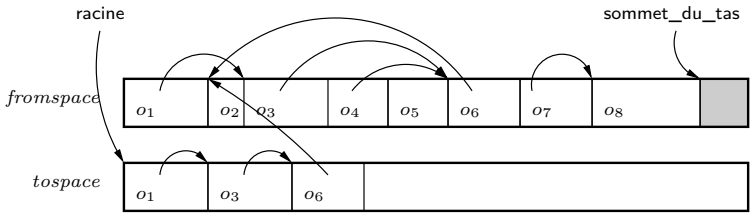


FIG. 3.5 – Ramasse-miettes à copie. Le ramasse-miettes est en train de transférer les objets vivants dans le *tospace*. L'objet o_6 a déjà été copié, mais tel quel : il pointe encore sur o_2 dans le *fromspace*. Lorsque o_2 aura été copié, ce pointeur sera rectifié.

Une fois que l'ensemble des objets a été copié, le rôle des deux demi-espaces est échangé (cf. figure 3.6). Tous les objets morts sont donc abandonnés sans précautions dans l'ancien *fromspace*. L'exécution du programme est relancée de façon transparente, les prochaines allocations se faisant au sommet du nouveau tas.

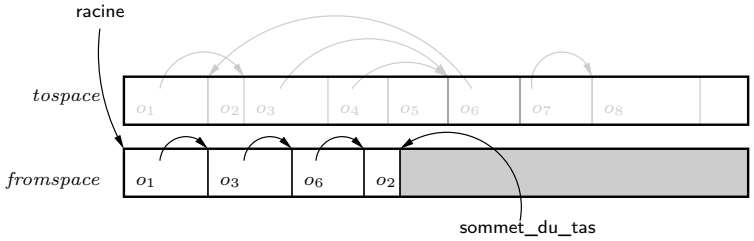


FIG. 3.6 – Ramasse-miettes à copie. Les deux demi-espaces ont été échangés. Les besoins d'allocation du programme peuvent maintenant être satisfaits (espace libre en grisé).

Ramasse-miettes à marquage-compactage La nécessité de diviser la mémoire en deux est parfois trop contraignante lorsque l'espace disponible est réduit, comme par exemple dans un contexte embarqué. La variante dite à *marquage-compactage* se soustrait à ce problème en déplaçant un à un les objets vivants à la base du tas.

Contrairement à la technique de copie, ce compactage doit dans un premier temps parcourir l'ensemble du tas pour déterminer les objets vivants, et ne pas les écraser lors de la copie des objets. Un cycle de collecte est illustré dans la figure 3.7. Au début du cycle *(a)*, le tas est entièrement occupé. Le ramasse-miettes parcourt alors le tas pour marquer les objets vivants *(b)*, car la défragmentation va ici s'opérer par balayage, et non pas récursivement comme avec un ramasse-miettes à demi-espaces. Le balayage *compacte* alors les objets vivants au début du tas *(c)*, tout en modifiant les champs des objets pour qu'ils pointent sur les mêmes objets qu'avant la collecte.

Ramasse-miettes générationnels Les différents types de ramasse-miettes à traçage présentés ci-dessus séparent, à chaque collecte, les objets vivants des objets morts. Pourtant, d'une collecte à l'autre, une bonne partie des objets vivants restent les mêmes, mais ils doivent cependant être parcourus, marqués ou copiés comme les autres. L'*hypothèse générationnelle* [Ung84], basée sur des observations empiriques, suppose même que «la plupart des objets meurent jeunes», et donc que les objets plus vieux ont toutes les chances de le devenir encore plus.

Il paraît donc sensé de s'attacher en priorité aux objets jeunes, puisque ce sont les plus susceptibles de se changer en espace libre lors des collectes. A cet effet, un ramasse-miettes *générationnel* distingue plusieurs zones différentes dans le tas : la *crèche* (ou *nursery*), où sont

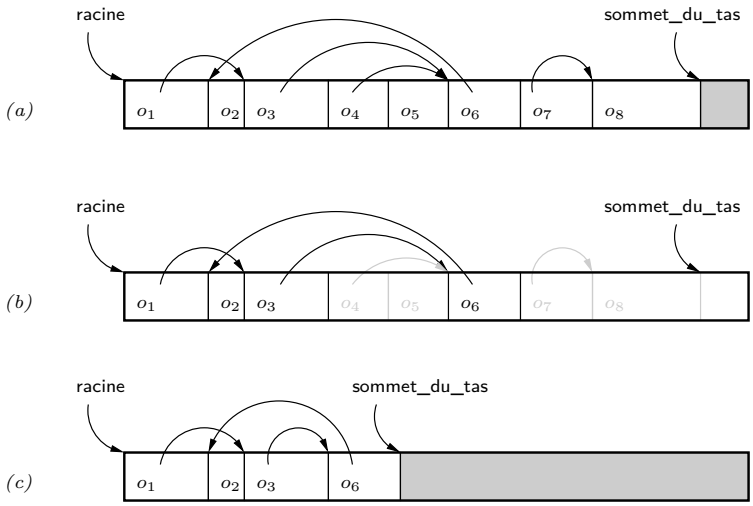


FIG. 3.7 – Ramasse-miettes à marquage-compactage. En *(a)*, le tas avant collecte ; en *(b)*, le tas après marquage, les objets morts (non marqués) sont représentés en gris ; en *(c)*, le tas après compactage.

alloués les nouveaux objets, et une *ancienne génération* (*old generation*, ou *mature space*), où se trouveront les objets réputés «vieux».

Les allocations se font dans la crèche, par un simple décalage de curseur. Lorsque l'espace est épuisé, le ramasse-miettes entreprend une *collecte mineure* : il parcourt la crèche en commençant par les racines, et déplace les objets vivants dans l'ancienne génération. Cette opération est appelée *titularisation* (*tenuring*) : si des objets ont survécu jusqu'à la collecte mineure, ils sont *titularisés* dans la zone principale.

La collecte des objets morts de la crèche requiert bien sûr un calcul d'accessibilité dans le graphe du tas tout entier, et donc le ramasse-miettes doit tenir compte des pointeurs entre les générations. Pour ne pas avoir à parcourir l'ensemble du tas à chaque collecte mineure, les pointeurs provenant de l'ancienne génération font l'objet d'un traitement spécial : ils sont considérés directement comme des racines lors du parcours de la crèche. Pour détecter ces pointeurs, un ramasse-miettes générationnel a besoin d'une barrière en écriture : à chaque fois que le programme crée un pointeur d'un «vieil» objet vers un «jeune» objet, l'environnement d'exécution copie ce pointeur dans une zone dédiée que le ramasse-miettes ajoutera à son ensemble de racines lors de la phase de marquage.

Par exemple, dans la figure 3.8, l'espace libre de crèche est épuisé, et le ramasse-miettes va entamer une collecte mineure. Les deux seules racines du tas sont les variables r1 et r2, mais la barrière en écriture a intercepté l'écriture dans o_5 du pointeur vers o_7, et a copié le pointeur dans les «racines de la crèche». Par contre, le pointeur de o_8 vers o_6 n'est pas intéressant : o_6 est déjà dans l'ancienne génération, et donc n'est pas affecté par les collectes mineures.

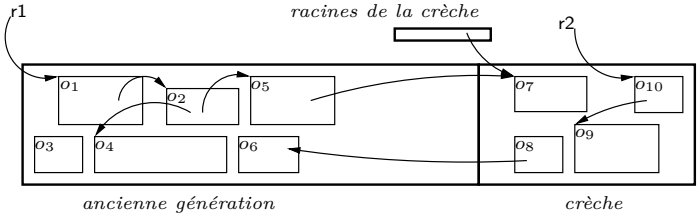


FIG. 3.8 – Ramasse-miettes générationnel. La crèche est entièrement occupée, et va subir une collecte mineure. Les objets survivants seront déplacés dans l'ancienne génération.

La crèche est collectée en utilisant une technique de copie : au fur et à mesure du parcours,

les objets vivants (o_7, o_9, o_{10}) sont déplacés dans l'ancienne génération. Le tas obtenu après la collecte est représenté sur la figure 3.9 : l'espace de la crèche est maintenant considéré comme libre (o_8 a été libéré implicitement lorsque le pointeur *sommet de la crèche* a été réinitialisé) et le programme utilisateur peut être relancé. On remarquera que o_3 et o_6 n'ont pas été collectés : ce sont des objets morts de l'ancienne génération, et il faudra attendre la prochaine collecte majeure, lors de laquelle l'ensemble du tas sera parcouru, pour s'apercevoir qu'ils sont inaccessibles.

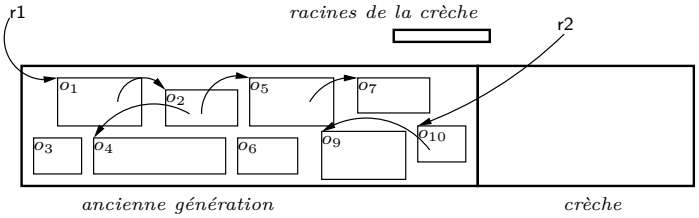


FIG. 3.9 – Ramasse-miettes générationnel après une collecte mineure. La crèche est entièrement disponible, et tous les objets restants sont situés dans l'ancienne génération.

Discussion L'objectif principal des techniques générationnelles est de diminuer les temps de pause. En effet, la taille réduite de la crèche permet de la collecter rapidement. La zone principale (l'ancienne génération), par contre, doit être gérée par un autre algorithme de ramasse-miettes, qui aura pour but de collecter les objets morts après leur titularisation. Selon l'hypothèse générationnelle, ces objets seront peu nombreux, et donc les *collectes majeures*, celles où tout le tas est examiné, resteront rares. Certains ramasse-miettes utilisent cependant plusieurs générations successives en amont de la zone principale afin de limiter au maximum les collectes majeures. D'autres travaux [JR06] tentent même de ne pas imposer de hiérarchie particulière entre les générations, mais plutôt de séparer grâce à des heuristiques les objets dans différentes *zones*, suivant leur date de mort présumée, afin de collecter chaque zone au moment le plus opportun.

Les différents ramasse-miettes compactants, incrémentaux ou non, présentent des avantages considérables en termes de performances sur le marquage-balayage simple, et sont aujourd'hui beaucoup plus utilisés. Cependant, leur obligation de déplacer les objets en limite l'usage à des environnements d'exécution contrôlés, tels que les machines virtuelles. En effet, dans un langage compilé faiblement typé, ou qui s'autorise l'arithmétique de pointeurs, il serait impossible de déplacer des données tout en préservant la cohérence de la mémoire.

3.4.2.3 Conclusion

À cause de la nature récursive des tâches qu'ils ont à accomplir, les algorithmes de ramasse-miettes présentent souvent des comportements temporels difficiles à prévoir. De nombreuses optimisations ont été proposées pour améliorer leurs performances, en particulier dans le but de réduire les temps de pause infligés à l'exécution. En particulier, des ramasse-miettes incrémentaux sont continuellement présentés comme *temps-réel* [Bro84, AEL88, Bak91, LPB98, SMB04], puisqu'ils permettent de *découper* le travail en fragments de durée fixe. Cependant, il s'agit le plus souvent d'algorithmes pour lesquels les temps de pause sont courts *la plupart du temps*. Les temps d'exécution au pire cas du ramasse-miettes, bien qu'ils ne se produisent presque jamais en pratique, sont trop importants pour être compatibles avec une politique d'ordonnancement temps-réel.

3.4.3 Ramasse-miettes «temps-réel»

Même s'ils utilisent de nombreuses optimisations pour améliorer leurs performances, les algorithmes de ramasse-miettes présentent souvent des comportement temporels difficiles à

prévoir. Dans un contexte temps-réel, leur interférence avec les performances du programme peut devenir problématique. En particulier, les temps de pause infligés à l'application sont très difficiles à quantifier à l'avance, et rendent inutiles les analyses de temps d'exécution nécessaires à l'ordonnancement.

Cependant, depuis quelques années, l'usage de langages à gestion mémoire automatique tels que Java ou C# s'est tellement répandu, y compris dans le monde de l'embarqué et du temps-réel, que plusieurs travaux ont été menés pour essayer de rendre possible l'utilisation d'un ramasse-miettes en présence de contraintes temps-réel.

Comptage de références temps-réel A priori, l'approche par comptage de références paraît plus appropriée, car elle est naturellement incrémentale, et libère la mémoire plus tôt que les techniques à traçage. Cependant, comme nous l'avons vu plus haut, une implantation naïve souffrirait de plusieurs défauts : la libération récursive des structures de données peut introduire des temps de pause importants et imprévisibles, les objets ne sont pas déplacés et donc le tas risque de se fragmenter peu à peu, ce qui rend impossible le calcul du temps d'exécution au pire cas de l'allocation, et les structures cycliques ne seront jamais collectées.

Ritzau [Rit03] propose d'éviter la libération récursive grâce au comptage de références différé [Wei63], et de régler le problème de la fragmentation en découpant les objets en blocs : la mémoire n'est plus organisée avec une granularité variable suivant la taille des objets, mais au contraire ce sont les objets qui sont répartis sur plusieurs blocs. Pour maintenir des performances acceptables, la taille des blocs doit être choisie empiriquement [RF02] de telle sorte que la majorité des objets tiennent sur un seul bloc, voire deux. Seuls les tableaux, de taille très variable, seront susceptibles d'occuper un grand nombre de blocs. Les allocations peuvent ainsi être réalisées en temps prévisible, puisque tous les blocs sont interchangeables.

Toutefois, le problème de la collecte des cycles n'est pas résolu de façon satisfaisante. Ritzau envisage plusieurs approches, parmi lesquelles la désallocation manuelle, ou l'utilisation d'un ramasse-miettes secondaire, mais les bénéfices du comptage de références en termes de garanties temps-réel sont alors perdus.

Traçage temps-réel Les limitations insurmontables du comptage de références ont donc conduit la communauté à s'intéresser plutôt aux ramasse-miettes à traçage. Il existe plusieurs algorithmes différents : Siebert [Sie99] utilise un mécanisme de marquage-balayage incrémental simple, alors que Bacon *et al.* [BCR03] proposent un système de *pages* entre lesquelles les objets sont copiés de temps à autre. Ces ramasse-miettes ont cependant de nombreuses caractéristiques communes. Il s'agit d'algorithmes incrémentaux, et la garantie des temps de réponse dépend d'un ordonnancement bien choisi entre l'exécution du programme et l'exécution du ramasse-miettes. La mémoire est organisée en blocs de taille fixe, comme dans le travail de Ritzau, de façon à empêcher la fragmentation du tas, et les objets sont répartis sur plusieurs blocs lorsque leur taille est trop importante [Sie00].

La principale limitation de ces travaux réside dans la question de l'ordonnancement. Pour que le système s'exécute correctement, deux conditions contradictoires doivent être satisfaites : d'une part, le ramasse-miettes doit être exécuté sur des périodes assez prévisibles pour que le programme utilisateur parvienne à satisfaire ses contraintes temps-réel, mais d'autre part, il doit disposer de suffisamment de temps d'exécution pour arriver à recycler la mémoire au même rythme que se font les allocations et désallocations. Si le ramasse-miettes n'est pas ordonnancé assez souvent, le système finira par être à court de mémoire libre car le ramasse-miettes ne recycle pas assez d'espace. S'il est ordonnancé trop souvent, le programme utilisateur n'arrivera pas à exécuter ses tâches en temps et en heure. Pour concilier les deux, il est nécessaire de pouvoir *prédire* quel sera le comportement mémoire du programme, et d'en tenir compte dans l'ordonnancement.

Bacon *et al.* [BCR03] proposent donc de calculer le *taux de ramassage des miettes* requis

(*garbage collector processing rate*) en fonction du taux d'allocation et du taux de mortalité des objets (*memory allocation rate* et *garbage allocation rate*). Siebert utilise des notations différentes [Sie98], basées sur la proportion de mémoire accessible dans le tas (*normalised reachable memory*), mais le principe est similaire. Ce modèle quantitatif de l'application est très difficile à obtenir rigoureusement. Mann *et al.* [MDLC05] proposent d'utiliser une analyse statique pour obtenir une sur-approximation du taux d'allocation, mais la détermination du taux de mortalité est une question beaucoup plus complexe [Det04]. En l'absence d'une modélisation fidèle et rigoureuse de l'application, l'utilisateur est condamné à exécuter le programme soit en sur-dimensionnant l'espace mémoire, soit en sous-utilisant le processeur.

3.4.4 Discussion

L'invention des ramasse-miettes est une étape capitale dans l'histoire de la programmation. Les bénéfices qu'ils apportent en termes de génie logiciel sont inestimables [Jon96]. Longtemps cantonnés à l'implantation de langages académiques, ils sont maintenant utilisés dans de nombreux contextes, notamment grâce à l'essor de langages grand public à ramasse-miettes comme Java ou C#. Même pour des langages moins appropriés, comme le C ou le C++, il existe des variantes des algorithmes présentés ci-dessus [BW88, Bar89] qui sont largement utilisés dans la pratique.

Leur usage dans le monde de l'embarqué et du temps-réel tarde cependant à se répandre, car ils souffrent d'une mauvaise réputation en terme de comportement temporel. Parmi les raisons de cette réputation, outre la performance [HB05], se trouve au premier plan le problème des temps de pause des ramasse-miettes atomiques (*stop-the-world*), dont les durées sont imprévisibles.

Pour la plupart, ces raisons ne tiennent plus aujourd'hui et de nombreux systèmes utilisent des ramasse-miettes. Une autre partie de ces raisons reste d'actualité toutefois, car aucune des adaptations au temps-réel dur n'est dépourvue de défauts : la proposition de Ritzau [Rit03], basée sur le comptage de références, ne collecte pas les structures cycliques. Les propositions de Siebert [Sie99] ou de Bacon *et al.* [BCR03] paraissent plus prometteuses, mais elles sont basées sur des algorithmes à copie, et consomment beaucoup d'espace. De plus, elles requièrent une modélisation quantitative des actions mémoire de l'application pour garantir un comportement temps-réel. Comme le note Detlefs dans son tour d'horizon [Det04] des ramasse-miettes temps-réel, les prochains progrès dans ce domaine sont à attendre du côté des analyses automatiques de programme. De même qu'un effort considérable est consacré aux calculs de temps d'exécution au pire cas, il y a maintenant des besoins importants en matière de calcul de taux d'allocation et de taille mémoire.

3.5 Gestion de la mémoire en régions

3.5.1 Principe

Pour terminer notre panorama des techniques de gestion mémoire, cette section présente la gestion mémoire en régions (*region-based memory management* [Tof98]). Le principe de cette approche est de grouper les objets de même durée de vie dans des *régions*, de façon à les traiter *en bloc* pour plus d'efficacité. Le modèle mémoire en régions n'est pas à comparer avec tel ou tel ramasse-miettes, mais plutôt avec les différentes techniques de lutte contre la fragmentation des gestionnaires mémoire classiques, c'est à dire qui ne déplacent pas les objets (on parle de *non-moving garbage collector* pour décrire les ramasse-miettes qui présentent cette propriété).

Les opérations primitives ne sont plus les mêmes que dans un gestionnaire mémoire traditionnel : l'allocation et la désallocation sont remplacées par la *création de région*, *l'allocation dans une région*, et la *destruction de région*.

Création de région Le rôle d'une *région* est de contenir un ensemble des objets destinés à être désalloués en même temps. Suivant les modèles, il peut s'agir d'une zone mémoire contiguë,

ou d'un ensemble de zones identifiées comme appartenant à la même région. Si l'implantation repose sur une zone mémoire unique, la taille de la région doit être connue au plus tard au moment de sa création, afin de réserver effectivement la mémoire dans le tas. Un modèle qui suppose des régions de taille variable permettra plus de souplesse dans la mise en œuvre, mais offrira des performances moindres du fait de la complexité des opérations.

Allocation dans une région Dans un modèle à régions, l'opération d'allocation requiert deux paramètres : non seulement la taille qui doit être réservée, mais aussi l'*identifiant de la région* dans laquelle l'allocation doit prendre place. L'idée est d'allouer physiquement les objets côte à côte à l'intérieur de la région, de façon à implanter l'opération d'allocation par un décalage de pointeur (*pointer bumping*), ce qui est traditionnellement impossible dans un modèle où les objets ne sont pas déplacés. Si les régions sont de taille fixe, la quantité d'objets alloués à l'intérieur ne peut dépasser une certaine borne, et l'opération d'allocation peut échouer. A l'inverse, dans un modèle où les régions ont une taille variable, cette situation provoquera l'extension de la région. L'allocation offre alors plus de souplesse, mais son temps d'exécution au pire cas est plus importante.

Destruction de région La désallocation individuelle d'objets, source de la fragmentation dans les gestionnaires mémoire traditionnels, est remplacée par la destruction *en bloc* de la région. Cette caractéristique, commune à tous les modèles en régions, est à la base du comportement temporel beaucoup plus prévisible des allocateurs en régions. Naturellement, il faut s'assurer que tous les objets situés dans une région sont bien morts au moment où celle-ci est détruite, pour ne pas créer de pointeurs invalides.

3.5.2 Applications

3.5.2.1 Langages fonctionnels

Les premiers travaux utilisant le modèle en régions sont ceux de Tofte et Talpin [TT94, TT97] autour d'une implantation sans ramasse-miettes du langage fonctionnel ML. Le langage ne permettant pas au programmeur de s'intéresser à la gestion mémoire, les différentes opérations primitives doivent être paramétrées automatiquement : où créer et où détruire des régions ? dans quelle région allouer les objets ? Tofte et Talpin proposent de pratiquer sur le programme une *analyse statique* lors de la compilation, afin de déterminer automatiquement ces paramètres. Cette *synthèse de régions* (*region inference*), basée sur une analyse de types, transforme le programme original en lui ajoutant des mots-clés spécifiques : la création et la destruction des régions sont déterminées par la formule `letregion` ρ `in` e `end`, et l'allocation est «placée» dans une région par l'expression e `in` ρ. Le compilateur ne cherche pas à calculer la taille des régions, qui seront implantées, à l'exécution, par des listes chaînées de pages.

3.5.2.2 Variantes en régions du langage C

RC Dans un langage impératif comme le C, il est beaucoup plus difficile d'automatiser l'usage des régions. Gay et Aiken [GA98] proposent donc d'offrir au programmeur la possibilité de les manipuler explicitement. A cet effet, ils ajoutent de nouvelles opérations primitives à la bibliothèque standard du langage C : `newregion()` et `deleteregion(reg)` permettent de créer et de détruire des régions, et l'allocation se fait à l'aide de la fonction `ralloc(size,reg)`. Dans un article ultérieur [GA01], ils optent pour une modification de la syntaxe du langage : ce qui offre la possibilité de pratiquer certaines vérifications de cohérence sur le programme au moment de la compilation, et d'ajouter des barrières autour des opérations sur les pointeurs. Les régions sont équipées d'un mécanisme de comptage de références, qui vérifie que l'usage de la mémoire par le programme est correct. Le système intercepte les opérations `deleteregion()`, et celles qui se rapportent à une région encore vivante produisent une erreur d'exécution.

Cyclone Dans le même esprit, le langage Cyclone [JMG+02], une variante fortement typée du langage C, propose au programmeur d'utiliser différents paradigmes de gestion mémoire : en complément des traditionnels `malloc()` et `free()`, il peut allouer ses objets en régions [GMJ+02], ou profiter de la présence d'un ramasse-miettes [BW88]. Il est également possible d'utiliser des *reaps* [SHM+06].

Le mot *reap*, introduit par Berger et Zorn [BZM02] pour décrire une nouvelle technique de gestion mémoire manuelle, provient de la contraction de *region* et *heap* : il s'agit d'une structure mémoire qui se comporte a priori comme une région, mais dans laquelle la désallocation individuelle reste possible. Un *reap* se comporte ensuite comme un tas classique, et les allocations suivantes se font en utilisant des stratégies traditionnelles de lutte contre la fragmentation. Le cas échéant, si le *reap* se retrouve plein, il recommence à se comporter comme une région. Les allocations se font donc à nouveau par décalage de pointeur, et de nouvelles pages sont ajoutées au *reap* pour augmenter sa taille. Cette combinaison des deux approches permet de bénéficier d'une grande partie des avantages des régions, tout en ne renonçant pas à la souplesse des techniques de gestion manuelle de mémoire dynamique. Cependant, il revient entièrement au programmeur d'assumer la tâche de gestion mémoire, ce qui n'est pas satisfaisant (cf section 3.3).

3.5.3 Gestion mémoire en régions pour Java

En Java, le programmeur n'a d'habitude pas à se préoccuper de la gestion mémoire. La plupart des machines virtuelles comportent des ramasse-miettes assez sophistiqués, qui recyclent la mémoire de façon transparente pour l'application. Il existe cependant plusieurs travaux qui explorent la possibilité d'utiliser des régions.

Synthèse de régions Cherem et Rugina [CR04] proposent d'adapter au langage Java la *synthèse de régions* de Tofte et Talpin. Dans un esprit analogue, Chin *et al.* [CCQR04] utilisent un système de types ad hoc. Des travaux antérieurs menés à Verimag [GNYZ04] se basent quant à eux sur une analyse d'échappement.

L'idée commune à ces approches est de procéder en deux temps. Dans un premier temps, le programme est enrichi automatiquement, grâce à une analyse statique (cf chapitre 4), avec des informations relatives aux régions. Il peut s'agir d'annotation de types [CCQR04], de la traduction du programme vers un langage à régions explicites [CR04], ou d'appels à un gestionnaire mémoire spécifique [GNYZ04]. Dans un second temps, le programme est exécuté par une machine virtuelle dotée d'un gestionnaire mémoire adapté. La gestion mémoire est automatique, en ce sens que les objets sont placés dans les régions sans intervention du programmeur. De même, celles-ci sont créées et détruites automatiquement grâce aux résultats de l'analyse statique.

L'approche adoptée dans ce travail est également basée sur une technique de synthèse de régions, nous allons donc y revenir plus en détail au cours de ce manuscrit (cf chapitre 5). Les algorithmes existants de synthèse de régions [CR04, CCQR04] souffrent en effet d'une limitation importante (cf section 3.5.5). Dans certains cas, les résultats obtenus par l'analyse conduisent le programme à placer presque tous les objets dans des régions qui ne sont jamais détruites. La mémoire occupée par ces régions n'est donc jamais désallouée, ce qui n'est pas acceptable dans le contexte d'un système embarqué, où le programme est destiné à s'exécuter pendant une longue période.

Allocateur en régions adaptatif Une autre approche intéressante est celle de Qian et Hendren [QH02], qui proposent d'utiliser un allocateur en régions pour faire de l'analyse d'échappement dynamique grâce à des barrières en écriture, et non pas grâce à de l'analyse statique. La machine virtuelle associe une région à chaque activation de méthode, et alloue par défaut tous les objets dans la région de la méthode courante. Lorsqu'une méthode se termine, sa région est détruite, libérant tous les objets qu'elle contient. Les régions sont implantées par des

listes chaînées de pages, ce qui permet d'augmenter facilement leur taille, ainsi que de détruire simplement une région en la concaténant à la liste des pages libres.

Pour éviter de désallouer des objets encore vivants, une barrière en écriture intercepte les opérations qui prolongent la durée de vie des objets, par exemple l'écriture de référence dans une variable globale. La machine virtuelle prend alors plusieurs mesures : tout d'abord, le site d'allocation qui a produit l'objet est marqué comme *non-local* dans une table descriptive globale, et les prochaines exécutions de ce même site placeront directement leurs objets dans le tas. Ensuite, la région qui contenait l'objet est concaténée à la liste de pages qui représente le tas. De cette façon, l'objet ne sera pas désalloué prématurément, mais sera à terme recyclé par le ramasse-miettes. Lors de la prochaine exécution de la méthode, les autres sites d'allocation continueront de placer leurs objets dans la région locale, et petit à petit les barrières en écriture se déclenchent de moins en moins souvent.

L'objectif de ce travail est de profiter des performances de l'allocation en région, proches de celles de l'allocation en pile, pour autant de sites d'allocation que possible. Les objets de faible durée de vie sont désalloués dès la fin de la méthode, épargnant donc également un travail substantiel au ramasse-miettes.

3.5.4 Gestion mémoire dans la RTSJ

Dans une machine virtuelle Java, la mémoire est traditionnellement gérée par un ramasse-miettes. Si cette caractéristique est généralement considérée comme des atouts du langage, nous avons vu qu'elle constitue plutôt un handicap dans le contexte des systèmes embarqués temps-réel, car elle rend impossibles en pratique les analyses de temps d'exécution. Dans son effort pour rendre Java plus «compatible» avec le temps-réel, la *Spécification Temps-Réel pour Java* [BBD+00], déjà évoquée plus haut (cf. section 2.6.1) propose un nouveau modèle pour programmer les tâches critiques de l'application, dans lequel en particulier les objets sont alloués par un gestionnaire de mémoire en régions.

Régions mémoire RTSJ Les régions de la RTSJ sont appelées *memory scopes*, et sont disponibles pour programmeur au travers de la classe `ScopedMemory` et ses dérivées. Chaque région est associée à une tâche : créée au moment où un processus entame l'exécution d'un `Runnable`, la région sera automatiquement détruite lors de la terminaison de la tâche. Par appel de méthodes, un même processus peut toutefois exécuter différentes sous-tâches, organisées en pile, pour lesquelles il disposera d'autant de régions, elles aussi organisées en pile. De là provient le terme de *Scoped Memory* : les durées de vie des différentes régions sont imbriquées, et elles seront détruites dans l'ordre inverse de leur création. La RTSJ offre deux types de régions, qui diffèrent par les garanties temporelles qu'elles offrent pour l'allocation.

Dans une région `LTMemory`, le temps pire cas d'exécution de l'instruction `new` est *linéaire* en la taille de l'objet, car Java initialise à zéro tous les mots qui composent l'objet. L'allocation ne se compose donc pas seulement du décalage de pointeur (opération de coût constant). Ces caractéristiques rendent déterministe le coût du `new`, ce qui permet l'analyse de temps d'exécution au pire cas des tâches qui s'exécutent dans une région `LTMemory`. Cependant, pour pouvoir garantir ce coût, la région doit être composée d'une seule zone de mémoire contiguë, et donc sa taille doit être connue au plus tard au moment de sa création.

Dans une région `VTMemory`, par contre, le temps d'exécution de `new` est autorisé à être *variable*, comme dans le tas traditionnel. Ces régions diffèrent du tas seulement en ce qu'elles sont libérées d'un seul coup à la terminaison d'un thread, sans passer par le ramasse-miettes. Il est cependant intéressant de travailler avec des `VTMemory` car elles ne nécessitent pas de connaître par avance la taille totale allouée.

Ce découpage clair des tâches en `Runnable`s, ainsi que la prévisibilité des opérations mémoire, a pour objectif de rendre possibles les analyses de temps d'exécution, et donc l'utilisation du langage Java pour la programmation temps-réel.

Processus RTSJ Les processus RTSJ, les `RealtimeThread`s, utilisent des régions pour ne plus subir d'interruption de la part du ramasse-miettes. A la différence de l'ordonnanceur Java Standard, celui de la RTSJ est préemptif, et utilise 28 niveaux de priorités fixes, chacune supérieure à la priorité du ramasse-miettes, elle-même supérieure aux 10 niveaux de priorité «indicatifs» des processus Java Standard. Ainsi, les tâches temps-réel de l'application cohabitent au sein de la machine virtuelle avec les tâches traditionnelles.

Un `RealtimeThread` est cependant limité à des contraintes temps-réel souple, car il peut toujours communiquer avec les tâches standard, ce qui peut introduire des synchronisations inattendues et finir par lui faire rater une échéance. Pour le temps-réel dur, la RTSJ propose le `NoHeapRealtimeThread`, un type de processus qui n'est pas du tout autorisé à manipuler des données situées dans le tas, ni même à y accéder. Les objets alloués par ce type de processus n'ont donc pas le droit de contenir des références vers les objets du tas.

Contraintes de programmation Les nouvelles possibilités de programmation offertes par la RTSJ ne sont pas sans contrepartie. En effet, la spécification impose des règles strictes quant à leur usage, en particulier à propos de la mémoire.

Du fait des *memory scopes*, il est interdit pour un objet du tas de pointer vers un objet d'une région, car elle pourrait disparaître intempestivement et rendre des pointeurs invalides. A l'inverse, un objet d'une région peut pointer vers un objet du tas, et les régions `VTMemory` associées à des `RealtimeThread`s sont examinées par le ramasse-miettes lors des collectes. Mais si une région est associée à un `NoHeapRealtimeThread`, de telles références sont interdites car le ramasse-miettes ne pourrait pas examiner cette région sans interrompre le calcul.

Ces règles sont résumées sur la figure 3.10. La RTSJ ne spécifie pas comment elles doivent être vérifiées, mais suggère de procéder soit par des analyses à la compilation, soit par des tests dynamiques (des barrières en lecture et en écriture) qui s'assurent que le programme se comporte correctement. Des exceptions spécifiques, par exemple l'`IllegalAssignmentError`, sont alors levées par la machine virtuelle pour interrompre brutalement le programme.

Un objet	peut pointer vers le tas	peut pointer vers une région
situé dans le tas	Oui	Interdit
situé dans une région (`RealtimeThread`)	Oui	Oui, si région inférieure
situé dans une région (`NoHeapRealtimeThread`)	Interdit	Oui, si région inférieure, *et* possédée par un `NoHeapRealtimeThread`

FIG. 3.10 – Les règles régissant les affectations de pointeurs dans la RTSJ

Dans un `RealtimeThread` ou un `NoHeapRealtimeThread`, l'instruction `new` alloue les objets dans la région en sommet de pile. Pour pouvoir allouer un objet dans l'une des régions inférieures, la RTSJ offre la primitive `getouterScope()`. Une fois la région désirée atteinte, grâce à des appels successifs à `getouterScope()`, on peut y allouer des objets par des primitives comme `ScopedMemory.newInstance()`.

La RTSJ remet donc à la charge du programmeur une grande partie de la tâche de gestion de la mémoire dynamique, et déterminer la durée de vie des objets est difficile. L'usage de l'API de régions peut rapidement s'avérer très complexe, en particulier du fait de la confusion introduite dans le code applicatif par le code de gestion mémoire.

Les inconvénients de la RTSJ Du fait de ces nombreuses restrictions, la RTSJ est considérée comme un modèle de programmation très difficile à utiliser manuellement [Wel04]. Le programmeur est obligé d'adopter de nouvelles habitudes dans la manière d'écrire son code car

le style traditionnel Java est incompatible avec les règles d'affectation de la RTSJ. De ce fait, la réutilisation de code est très délicate, et même la bibliothèque standard doit être entièrement réécrite si l'on veut en profiter dans le contexte d'une tâche temps-réel [Dau05].

Pizlo *et al.* [PFHV04] proposent un certain nombre de «patrons de conception» originaux, destinés à faciliter l'implantation en RTSJ de mécanismes classiques, comme le «producteur - consommateur». Par exemple, la figure 3.11 illustre la manière que proposent Pizlo *et al.* pour traduire une simple boucle de contrôle (a) en un programme RTSJ équivalent (b).

```
void runLoop()
{
  while ( true )
  {
    ... lire les entrées
    ... faire quelque chose
  }
}
```

(a)

```
class RunLoopIteration implements Runnable
{
  void run()
  {
    ... lire les entrées
    ... faire quelque chose
  }
}

void runLoop()
{
  ... fixer la taille de la région
  memory = new LTMemory( init_sz, max_sz);
  runLoop = new RunLoopIteration();
  while ( true ) memory.enter( runLoop );
}
```

(b)

FIG. 3.11 – Patron de conception *Scoped Run Loop* [PFHV04].

Cela dit, même avec l'aide de ces patrons de conception, la programmation RTSJ reste complexe car elle oblige le programmeur à se préoccuper en même temps des aspects fonctionnels et non fonctionnels du programme. Il faut raisonner sur les durées de vie de tous les objets pour savoir dans quelle région les placer, et dimensionner chacune de ces régions. Le code de gestion mémoire est ainsi mêlé au code applicatif, ce qui complique le développement, mais aussi la maintenance du programme.

Génération de code pour la RTSJ De nombreux travaux préfèrent donc générer le code RTSJ automatiquement à partir d'un programme écrit en Java Standard. Par exemple, Deters et Cytron [DC02] utilisent un profilage de l'application, exécutée sur plusieurs jeux de données d'entrée, pour mesurer les durées de vie de tous les objets. Le code est ensuite *instrumenté* automatiquement : chaque méthode de l'application originale est traduite en un `Runnable` auquel est associée une région, et les sites d'allocation sont modifiés de sorte à utiliser les primitives RTSJ pour placer l'objet alloué dans la région de bonne durée de vie. Cependant, les limitations de ce travail sont multiples : l'approche utilisée ne permet pas de déterminer la taille des régions, donc le programme RTSJ n'utilise que des régions `VTMemory`. De plus, il s'agit d'une analyse dynamique (cf chapitre 4), les mesures de durée de vie obtenues par le profilage ne sont donc pas valables pour n'importe quelle exécution. Une exécution du même programme dans un autre contexte est donc susceptible de se comporter différemment, en particulier en ce qui concerne la durée de vie des objets. La version RTSJ du programme risque ainsi de provoquer une faute d'exécution parce qu'elle ne respecte plus les règles d'affectation.

Au contraire plusieurs travaux [BSBR03, ZNV04] proposent de générer du code *correct par construction* à l'aide d'une «compilation» Java vers RTSJ. Il devient cependant nécessaire de demander au programmeur de décrire assez finement les relations d'accessibilité entre les objets qu'il manipule, par exemple par des annotations dans le texte du programme, car il est impossible pour le compilateur de déduire du programme ces informations avec suffisamment de précision. Là encore, le langage de programmation obtenu est en pratique largement différent du Java Standard, et se révèle donc trop difficile à utiliser.

3.5.5 Commentaire

La gestion de la mémoire en régions est une alternative intéressante aux gestionnaires mémoire classiques. Elle permet de bénéficier de certains avantages réservés habituellement aux ramasse-miettes (lutte contre la fragmentation, allocation efficace), tout en évitant les surcoûts associés à la collecte. Elle est aujourd'hui utilisée dans de nombreux contextes, comme par exemple dans l'implantation du compilateur GCC, ou dans le serveur web Apache [TBEH04]. Ces logiciels, bien que programmés dans des langages traditionnels, utilisent une bibliothèque d'allocation mémoire en régions.

Syndrome d'explosion des régions Les approches présentées dans cette section souffrent toutes d'un même problème, que nous nommerons *syndrome d'explosion des régions* : dans certaines situations, difficiles à prévoir, une ou plusieurs régions ont une durée de vie particulièrement longue, typiquement aussi longue que l'exécution de l'application, alors que le programme y alloue continuellement des données. Cette nouvelle sorte de *fuite mémoire* due aux régions, est remarquée par Berger *et al.* dans leur étude des différents gestionnaires mémoire [BZM02] et motive leur proposition des *reaps*. Effectivement, offrir au programmeur la possibilité de désallouer individuellement certains objets peut régler le problème : s'il identifie correctement l'origine de l'explosion, il peut ajouter dans le programme des instructions pour désallouer explicitement ces objets, et ainsi colmater la fuite de mémoire.

Cependant, si l'on veut profiter pleinement du comportement temporel des régions, la désallocation individuelle n'est pas possible, et rien ne permet alors de se prémunir contre le syndrome d'explosion des régions. Même dans des programmes à l'exécution assez courte, Berger *et al.* [BZM02] remarquent que l'utilisation manuelle de régions conduit à une utilisation mémoire considérable (*«Programs that use region allocators are especially draggy»*). Dans le cadre de la RTSJ, l'inextensibilité des régions révèle une autre facette du même problème [PFHV04]. Cherem et Rugina [CR04] observent également que leur synthèse de régions, pour certains programmes, mène à des régions de taille démesurée.

Certains travaux [BGY06, MDLC05] tentent donc de calculer par analyse statique une estimation de la mémoire allouée par le programme, mais sont en général contraints à des sur-approximations importantes.

3.6 Discussion

Nous avons vu dans ce chapitre un aperçu des différentes techniques de gestion de la mémoire dynamique. L'objectif de ce travail étant de permettre l'usage dans un contexte embarqué temps réel, nous avons donc étudié les caractéristiques de ces techniques vis-à-vis des contraintes propres à ce contexte.

La gestion manuelle classique de la mémoire, par exemple les primitives `malloc` et `free` offertes par le langage C, est très propice aux erreurs de programmation. Les fautes d'exécution non reproductibles qui en découlent sont très problématiques pour un système embarqué. De plus, le modèle mémoire associé souffre d'un problème de fragmentation, ce qui donne au gestionnaire mémoire un comportement temporel difficile à prédire.

L'utilisation d'un ramasse-miettes pour recycler la mémoire permet d'éliminer les erreurs de programmation ; cependant, les temps de pause non prévisibles associés en interdisent l'usage dans un contexte temps-réel. Des ramasse-miettes compatibles avec les contraintes du temps-réel ont été proposés, mais ils nécessitent de modéliser l'application d'une façon très précise, ce qui est trop difficile pour être utilisable en pratique.

Prenant le problème à l'envers, le modèle mémoire des régions permet de disposer d'opérations élémentaires au coût prévisible. C'est d'ailleurs le modèle adopté par la Spécification Temps-Réel pour Java (RTSJ). Cependant, l'utilisation manuelle d'un allocateur en régions pose de nouveau le problème de la difficulté de programmation.

La synthèse automatique de régions par analyse statique paraît donc prometteuse. Elle permet de profiter du comportement temporel prévisible des régions, tout en affranchissant le programmeur de la tâche de gestion mémoire. De même que lors de l'utilisation manuelle d'un allocateur en régions, l'occupation mémoire du programme a parfois tendance à augmenter irrémédiablement, ce que nous avons appelé le syndrome d'explosion des régions. Comme nous le verrons dans le chapitre 5, nous avons choisi d'attaquer ce problème grâce à une analyse statique simple permettant au programmeur de comprendre la cause de l'explosion et de corriger son programme. Auparavant, nous allons aborder dans le chapitre 4 le domaine de l'analyse statique, et faire un tour d'horizon des différentes analyses dédiées à la gestion mémoire.

Chapitre 4

L'analyse statique

4.1 Introduction

Il existe de nombreuses approches pour étudier les propriétés d'un programme. La plus naturelle consiste à exécuter le programme et à observer les résultats obtenus, le plus souvent grâce à un environnement d'exécution contrôlé. Les techniques de débogage, de profilage, et plus généralement les méthodes de test, entrent par exemple dans cette catégorie. Il s'agit d'une approche *existentielle*, car elle montre qu'«il existe telles données d'entrée pour lesquelles le programme se comporte de telle façon». Les observations ne seront cependant plus les mêmes si l'on change ces données, et les propriétés obtenues peuvent varier grandement. Un ensemble de données d'entrée bien choisi peut permettre de tirer des conclusions vraies «la plupart du temps», comme par exemple l'hypothèse générationnelle [HHDH02] communément admise dans le monde de la gestion mémoire, mais sans garantie de validité.

On oppose donc généralement ces analyses dites *dynamiques* aux analyses dites *statiques*, qui visent au contraire l'*universalité*, c'est à dire des résultats valables «pour n'importe quelle exécution». L'analyse statique se pratique donc directement sur le code source du programme (ou sur une représentation équivalente), sans l'exécuter à proprement parler.

En analyse de programme, la notion d'approximation est centrale, car il est souvent impossible d'obtenir des informations exactes. En effet, les problèmes considérés sont typiquement indécidables (cf section 4.2.1). Par exemple, déterminer si un programme termine, ou calculer quelle valeur exacte prendra une variable après l'exécution du programme, sont des questions indécidables dans le cas général. On recourt donc à des analyses *approchées*, pour simplifier le problème et le rendre décidable. Par exemple, si le problème original est de déterminer l'ensemble des valeurs qu'une variable peut prendre, l'analyse dynamique permet d'observer un sous-ensemble de ces valeurs lors d'exécution réelles, et l'analyse statique cherchera, au contraire, à en déterminer un sur-ensemble.

Les applications de l'analyse statique sont multiples, de l'optimisation à la vérification, en passant par l'aide à la compréhension de programme. Même une mesure très simple comme celle du nombre de lignes de code du programme, couramment utilisée en génie logiciel, peut être considérée comme de l'analyse statique. Cependant, des techniques d'analyse statique plus sophistiquées peuvent mettre en œuvre des théories mathématiques assez avancées : par exemple, un compilateur pratique couramment de complexes analyses de types sur le programme qu'il est en train de traiter.

Dans ce chapitre, nous allons tout d'abord donner un aperçu des techniques employées en analyse statique, basées en général sur des calculs de points fixes dans des treillis [NNH99]. Dans un second temps, nous nous pencherons plus particulièrement sur les analyses liées à la mémoire, de façon à déterminer dans quelle mesure les travaux existants permettent de résoudre le problème auquel nous nous intéressons dans cette thèse.

4.2 Techniques d'analyse

4.2.1 Indécidabilité et sur-approximations

L'analyse de programme est une affaire d'approximations. Les techniques d'analyse dynamique permettent, en observant l'exécution du programme, d'obtenir naturellement une sous-approximation de l'ensemble des comportement possibles. Dans une approche complémentaire, l'analyse statique cherche à en obtenir des sur-approximations. Par exemple, intéressons-nous aux valeurs que peut prendre la variable `z` à la fin du programme ci-dessous.

`read(x); if( x>0 ) { y = 1; } else { y = 2;` S `}; z = y;`

Intuitivement, selon la valeur de `x` obtenue en entrée, le programme choisira entre les deux branches du programme. On suppose que le fragment de code S ne contient pas d'affectation de la variable `y`. La valeur qui atteindra l'affectation `z = y` sera donc soit 1 soit 2.

Supposons qu'une certaine analyse statique affirme que seule la valeur 1 peut atteindre cette affectation. Cela paraît hasardeux, mais peut se révéler être correct si S s'avère ne jamais terminer lorsque $\texttt{x} \leq 0$ et $\texttt{y} = 2$. On remarquera que dans un tel cas, une analyse dynamique produira toujours le résultat 1, puisqu'aucune exécution de l'autre branche ne mènera à l'affectation de `z`.

Cependant, comme il est dans le cas général *indécidable* de déterminer statiquement les conditions de terminaison d'un programme, on ne s'attend pas à ce qu'une analyse statique cherche à détecter ce genre de phénomène. Au contraire, on se satisfera d'un résultat approché, du moment qu'il est *sûr*, c'est à dire qu'il contient au moins tous les comportement réels du programme. Par exemple ici, on pourrait imaginer une analyse qui donnerait pour `z` les valeurs 1, 2, et 42. Cette analyse serait toujours correcte, même si on préférera bien sûr un résultat plus précis à un résultat trop approché : répondre $\texttt{z} \in \mathbb{N}$ serait correct, mais pas d'une grande utilité.

Une analyse correcte calcule un sur-ensemble des comportements du programme, incluant éventuellement des comportements qu'on cherche à éviter. On qualifie ainsi souvent une analyse correcte de *pessimiste*, ou de *prudente* (*conservative analysis*).

4.2.2 Un exemple : l'analyse des définitions visibles

L'objectif d'une analyse statique est de déterminer ou de vérifier certaines propriétés d'un programme, ou des données qu'il manipule. Il s'agit par exemple de répondre à des questions du type «quel est le signe de cette variable en tel point du programme ?» (analyse de signes), ou «ce fragment de code est-il accessible, ou peut-il être éliminé ?» (analyse de code mort). Nous allons nous intéresser à l'analyse dite des définitions visibles (*reaching definitions*) en prenant comme point de départ le programme présenté à la figure 4.1. Ce programme calcule itérativement la factorielle de la variable `x` et rend son résultat dans la variable `z`.

```
1  y = x;
2  z = 1;
3  while ( y > 1 )
4  {
5    z = z * y;
6    y = y - 1;
7  }
8  y=0;
```

FIG. 4.1 – Calcul de factorielle.

L'objectif de l'analyse des définitions visibles est de déterminer, en chaque point du programme et pour chacune des variables, quelles sont les définitions (ou affectations) visibles depuis ce point. Une définition v `=` e de la variable v est dite *visible* depuis un point p du programme s'il existe un chemin d'exécution allant de v `=` e à p le long duquel aucune instruction ne modifie la valeur de v, c'est à dire ne redéfinit la variable v.

Par exemple, la définition de y ligne 1 est visible depuis la ligne 2, mais la définition de z ligne 2 n'est pas visible depuis la ligne 6 à cause de la redéfinition de z à la ligne 5. En général, on distingue d'ailleurs deux points de programme pour désigner l'entrée et la sortie d'une même instruction. On notera ainsi $\mathsf{RD}_{entry}(l)$ pour désigner l'ensemble des définitions visibles depuis l'entrée de la ligne l, et $\mathsf{RD}_{exit}(l)$ pour l'ensemble des définitions visibles depuis la sortie de la ligne l. Par exemple, l'affectation de la ligne 2 est visible jusqu'à l'entrée de la ligne 5, ce qu'on notera $(\mathtt{z}, 2) \in \mathsf{RD}_{entry}(5)$, mais pas à sa sortie, ce qu'on notera $(\mathtt{z}, 2) \notin \mathsf{RD}_{exit}(5)$. On notera aussi $(\mathtt{x}, ?) \in \mathsf{RD}_{entry}(1)$ pour représenter le fait que l'analyse ne sait pas localiser d'où vient la valeur de x à la ligne 1. Avec ces notations, les définitions visibles en chaque point du programme sont données à la figure 4.2.

l	$\mathsf{RD}_{entry}(l)$	$\mathsf{RD}_{exit}(l)$
1	$\{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\}$	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\}$
2	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},?)\}$	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{z},2)\}$
3	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},5)\}$	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},5)\}$
5	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},5)\}$	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},6),(\mathtt{z},5)\}$
6	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},6),(\mathtt{z},5)\}$	$\{(\mathtt{x},?),(\mathtt{y},6),(\mathtt{z},5)\}$
8	$\{(\mathtt{x},?),(\mathtt{y},1),(\mathtt{y},6),(\mathtt{z},2),(\mathtt{z},5)\}$	$\{(\mathtt{x},?),(\mathtt{y},8),(\mathtt{z},2),(\mathtt{z},5)\}$

FIG. 4.2 – Définitions visibles dans le programme de calcul de factorielle.

Si l'on rajoute par exemple la définition $(\mathtt{z}, 2)$ aux définitions visibles depuis la ligne 5, on a toujours une approximation correcte, car elle forme toujours un sur-ensemble des comportements du programme. Par contre, retirer $(\mathtt{z}, 2)$ des définitions visibles depuis la ligne 8 serait incorrect, car les résultats de l'analyse ne rendraient plus compte de tous les comportements du programme : si $\mathtt{x} \leq 1$, l'affectation de la ligne 2 est bien visible depuis la ligne 8. Une analyse statique cherche en général à produire la sur-approximation la plus proche possible de la réalité.

4.2.3 Analyses de flot de données

Les analyses de *flot de données* forment une part importante du domaine de l'analyse statique. Dans cette section, on utilise l'exemple de l'analyse des définition visibles pour introduire différentes notions qui vont nous être utiles dans toute la suite : graphe de flot de contrôle, systèmes d'équations de flot de données, et résolution itérative de ces systèmes.

Graphe de flot de contrôle Le graphe de flot de contrôle est une représentation du programme dans laquelle la relation de succession entre les instructions est rendue explicite : chaque instruction constitue un sommet du graphe, et des arêtes décrivent les chemins que peut suivre le contrôle dans le programme [All70]. Le graphe de flot de contrôle de notre programme est donné sur la figure 4.3.

Suivant les langages de programmation considérés, déterminer statiquement les successeurs d'une instruction est un problème plus ou moins complexe. Dans un langage impératif simple comme celui utilisé dans l'exemple, la construction du graphe de flot de contrôle de chaque méthode est une opération syntaxique, facile à automatiser. Dans des langages plus réalistes, en particulier en présence d'instructions d'appel, il s'agit d'un problème plus difficile.

Vocabulaire Une instruction possédant plusieurs successeurs dans le graphe de flot de contrôle en souvent appelée *point de branchement*. A l'inverse, une instruction possédant plusieurs prédécesseurs est appelée *point de jonction*. On regroupe parfois en un seul sommet les instructions qui se suivent de façon séquentielle, d'un point de jonction au point de branchement suivant, pour former un *bloc de base*.

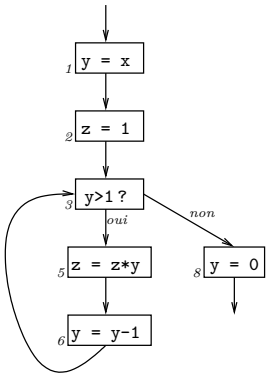


FIG. 4.3 – Le graphe de flot de contrôle du programme de calcul de factorielle.

4.2.3.1 Formulation sous forme d'équations

Une analyse de flot de données, comme celle des définitions visibles, s'exprime souvent de façon naturelle sous la forme d'un système d'équations mutuellement récursives. En effet, ce genre d'analyse procède typiquement par propagation : le résultat en un point du programme dépend des résultats aux points voisins, et ainsi de proche en proche. Par exemple, il est facile d'exprimer pour chaque instruction quelles sont les définitions visibles au point *juste après* l'instruction en fonction de celles visibles *juste avant*. Pour le programme de calcul de factorielle, on obtient le système d'équations suivant :

$$\begin{aligned}
\mathsf{RD}_{exit}(1) &= (\mathsf{RD}_{entry}(1) \setminus \{(\mathtt{y}, l) \,|\, l \in \mathbb{N}\}) \cup \{(\mathtt{y}, 1)\} \\
\mathsf{RD}_{exit}(2) &= (\mathsf{RD}_{entry}(2) \setminus \{(\mathtt{z}, l) \,|\, l \in \mathbb{N}\}) \cup \{(\mathtt{z}, 2)\} \\
\mathsf{RD}_{exit}(3) &= \mathsf{RD}_{entry}(3) \\
\mathsf{RD}_{exit}(5) &= (\mathsf{RD}_{entry}(5) \setminus \{(\mathtt{z}, l) \,|\, l \in \mathbb{N}\}) \cup \{(\mathtt{z}, 5)\} \\
\mathsf{RD}_{exit}(6) &= (\mathsf{RD}_{entry}(6) \setminus \{(\mathtt{y}, l) \,|\, l \in \mathbb{N}\}) \cup \{(\mathtt{y}, 6)\} \\
\mathsf{RD}_{exit}(8) &= (\mathsf{RD}_{entry}(8) \setminus \{(\mathtt{y}, l) \,|\, l \in \mathbb{N}\}) \cup \{(\mathtt{y}, 8)\}
\end{aligned}$$

Le principe est de remarquer que, pour chaque instruction i, les définitions visibles après i sont les mêmes que celles visibles avant i, sauf si i est elle-même une définition. Dans ce cas, si v est la variable redéfinie, i a *masqué* les autres définitions de v, et en devient l'unique définition visible.

De même, l'ensemble des définitions visibles depuis l'entrée d'une instruction i s'exprime naturellement comme l'union des ensembles des définitions visibles depuis la sortie des instructions qui précèdent i dans le graphe de flot de contrôle. Pour notre programme, cela revient à écrire :

$$\begin{aligned}
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(6) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(6) &= \mathsf{RD}_{exit}(5) \\
\mathsf{RD}_{entry}(8) &= \mathsf{RD}_{exit}(3)
\end{aligned}$$

Le cas de la première instruction est particulier. Dans le cadre d'une analyse inter-procédurale, si le calcul de factorielle est contenu dans une procédure, il faudra tenir compte des points d'où peut provenir le contrôle avant d'arriver ici. On suppose pour l'instant que le programme commence bien à la ligne 1, et on utilise par exemple la notation suivante pour exprimer qu'aucune définition n'est visible depuis ce point-là :

$$\mathsf{RD}_{entry}(1) = \{(v, ?) \,|\, v \text{ est une variable du programme}\}$$

ce qui se traduit, pour le programme de calcul de factorielle, par l'équation :

$$\mathsf{RD}_{entry}(1) = \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\}$$

Remarque On parle d'analyse *en avant* (dans le sens du flot de contrôle) lorsque les propriétés pour chaque point du programme sont calculées, comme ici, en fonction de celles des *prédécesseurs* de ce point. A l'inverse, les équations d'une analyse *en arrière* propageraient l'information dans le sens inverse. Par ailleurs, on distingue les analyses *may*, qui caractérisent ce qui peut se passer sur l'*un* des chemins d'exécution, des analyses *must*, qui s'attachent à des propriétés valables le long de *tous* les chemins. L'analyse des définitions visibles est une analyse *may* en avant, alors que par exemple l'analyse des expressions inévitables (*very busy expressions*) est une analyse *must* en arrière.

Plus petite solution Le système d'équations construit ci-dessus définit douze ensembles $\mathsf{RD}_{entry}(1), \mathsf{RD}_{exit}(1), \cdots, \mathsf{RD}_{entry}(8), \mathsf{RD}_{exit}(8)$. Si on note en raccourci $\overrightarrow{\mathsf{RD}}$ pour désigner ce douze-uplet, on peut considérer ce système comme une unique équation $\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$, où F est une certaine fonction qui synthétise toutes les équations précédentes. L'analyse des définitions visibles revient à trouver une solution à cette équation. Si l'on ne restreint pas a priori l'ensemble des définitions possibles, l'ensemble de départ et l'ensemble d'arrivée de F sont, tous les deux $(\mathcal{P}(Var \times \mathbb{N}))^{12}$, où Var est l'ensemble des noms de variables, et $\mathbb{N}$ l'ensemble des numéros de ligne possibles (on choisit de coder $(v, ?)$ par $(v, 0)$, puisque notre programme ne comporte pas de ligne 0). Cependant, on remarque que les seules valeurs intéressantes sont celles des variables présentes dans le programme, et que les seuls numéros de ligne intéressants sont ceux correspondant à des instructions du programme. En notant respectivement $Variable$ et $Label$ ces deux ensembles (finis), F s'écrit donc :

$$F : (\mathcal{P}(Variable \times Label))^{12} \longrightarrow (\mathcal{P}(Variable \times Label))^{12}.$$

Cet ensemble $(\mathcal{P}(Variable \times Label))^{12}$ peut être muni d'un ordre partiel $\sqsubseteq$ (cf section 4.2.3.3) en posant :

$$\overrightarrow{\mathsf{RD}} \sqsubseteq \overrightarrow{\mathsf{RD}'} \text{ ssi } \left(\mathsf{RD}_{entry}(1) \subseteq \mathsf{RD}'_{entry}(1)\right) \wedge \cdots \wedge \left(\mathsf{RD}_{exit}(8) \subseteq \mathsf{RD}'_{exit}(8)\right).$$

Le plus petit élément de cet ensemble est $\overrightarrow{\varnothing} = (\varnothing, \cdots, \varnothing)$. Par ailleurs, en raison de la forme des équations qui la composent, la fonction F s'avère être monotone croissante vis-à-vis de cet ordre partiel, c'est à dire que :

$$\overrightarrow{\mathsf{RD}} \sqsubseteq \overrightarrow{\mathsf{RD}'} \Longrightarrow F(\overrightarrow{\mathsf{RD}}) \sqsubseteq F(\overrightarrow{\mathsf{RD}'}).$$

Dans ces conditions, la suite des $F^n(\overrightarrow{\varnothing})$ pour $n \in \mathbb{N}$ est une suite croissante : comme $\overrightarrow{\varnothing} \sqsubseteq F(\overrightarrow{\varnothing})$, pour chaque $n > 1$ on a $F^n(\overrightarrow{\varnothing}) \sqsubseteq F^{n+1}(\overrightarrow{\varnothing})$. L'ensemble $(\mathcal{P}(Variable \times Label))^{12}$ étant fini, cette suite ne peut pas être strictement croissante : il existe donc un n pour lequel $F^n(\overrightarrow{\varnothing}) = F^{n+1}(\overrightarrow{\varnothing})$. Cette valeur est donc un point fixe de la fonction F, c'est à dire une solution du système d'équations original.

En réalité, nous avons par ce calcul itératif obtenu la *plus petite solution* du système d'équations. En effet, supposons qu'il existe une autre solution $\overrightarrow{\mathsf{RD}}$. Par définition, on a $\overrightarrow{\varnothing} \sqsubseteq \overrightarrow{\mathsf{RD}}$, et par induction on vérifie que $F^n(\overrightarrow{\varnothing}) \sqsubseteq \overrightarrow{\mathsf{RD}}$. On verra dans la section 4.2.3.3 que ce raisonnement est un cas particulier du théorème de Kleene.

$F^n(\overrightarrow{\varnothing})$ est la plus petite solution, c'est à dire celle qui contient le moins de définitions visibles, et donc qui décrit le plus précisément le comportement du programme. Ajouter d'autres définitions donnerait une valeur qui serait également un point fixe de F, et donc une sur-approximation correcte, mais moins précise, de la réalité.

4.2.3.2 Formulation sous forme de contraintes

Une alternative fréquente à la formulation sous forme de système d'équations est l'utilisation d'inégalités, ou d'inclusions, ou d'une autre forme de contraintes entre les différentes valeurs des

propriétés étudiées. Dans le cas de l'analyse des définitions visibles, il s'agit donc d'exprimer la même propagation, tout en rendant peut-être plus clair le fait que la solution recherchée est une sur-approximation du résultat.

Si l'on cherche à modéliser l'effet des instructions sur la visibilité des définitions, on obtient les contraintes suivantes :

$$\begin{array}{lcl}
\mathsf{RD}_{exit}(1) & \supseteq & \mathsf{RD}_{entry}(1) \setminus \{(\mathtt{y}, l) \,|\, l \in \mathbb{N}\} \\
\mathsf{RD}_{exit}(1) & \supseteq & \{(\mathtt{y}, 1)\} \\
\mathsf{RD}_{exit}(2) & \supseteq & \mathsf{RD}_{entry}(2) \setminus \{(\mathtt{z}, l) \,|\, l \in \mathbb{N}\} \\
\mathsf{RD}_{exit}(2) & \supseteq & \{(\mathtt{z}, 2)\} \\
\mathsf{RD}_{exit}(3) & \supseteq & \mathsf{RD}_{entry}(3) \\
\mathsf{RD}_{exit}(5) & \supseteq & \mathsf{RD}_{entry}(5) \setminus \{(\mathtt{z}, l) \,|\, l \in \mathbb{N}\} \\
\mathsf{RD}_{exit}(5) & \supseteq & \{(\mathtt{z}, 5)\} \\
\mathsf{RD}_{exit}(6) & \supseteq & \mathsf{RD}_{entry}(6) \setminus \{(\mathtt{y}, l) \,|\, l \in \mathbb{N}\} \\
\mathsf{RD}_{exit}(6) & \supseteq & \{(\mathtt{y}, 6)\} \\
\mathsf{RD}_{exit}(8) & \supseteq & \mathsf{RD}_{entry}(8) \setminus \{(\mathtt{y}, l) \,|\, l \in \mathbb{N}\} \\
\mathsf{RD}_{exit}(8) & \supseteq & \{(\mathtt{y}, 8)\}
\end{array}$$

De même, en exprimant comment cette visibilité se propage le long du graphe de flot de contrôle, on obtient une contrainte pour chaque arête :

$$\begin{array}{lcl}
\mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(6) \\
\mathsf{RD}_{entry}(5) & \supseteq & \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(6) & \supseteq & \mathsf{RD}_{exit}(5) \\
\mathsf{RD}_{entry}(8) & \supseteq & \mathsf{RD}_{exit}(3)
\end{array}$$

De même que dans la formulation par équations, une contrainte spécifique indique qu'au début du programme, les définitions précédentes des variables sont indéterminées :

$$\mathsf{RD}_{entry}(1) \supseteq \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\}$$

Puisque chacune de ces inclusions est non-stricte, une solution $\overrightarrow{\mathsf{RD}}$ du système d'équations $\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$ est aussi une solution du système d'inéquations $\overrightarrow{\mathsf{RD}} \sqsupseteq F(\overrightarrow{\mathsf{RD}})$, et ce pour la *même* fonction F.

D'ailleurs, ces deux formulations de l'analyse des définitions visibles admettent la même *plus petite solution*. En effet, dans la formulation $\overrightarrow{\mathsf{RD}} = F(\overrightarrow{\mathsf{RD}})$, la solution est obtenue itérativement en calculant les $F^n(\overrightarrow{\varnothing})$ jusqu'à une valeur de n pour laquelle $F^n(\overrightarrow{\varnothing}) = F^{n+1}(\overrightarrow{\varnothing})$. Si par ailleurs $\overrightarrow{\mathsf{RD}}$ est une autre solution de $\overrightarrow{\mathsf{RD}} \sqsupseteq F(\overrightarrow{\mathsf{RD}})$, alors par définition $\overrightarrow{\varnothing} \sqsubseteq \overrightarrow{\mathsf{RD}}$, et donc par induction on a $F^n(\overrightarrow{\varnothing}) \sqsubseteq \overrightarrow{\mathsf{RD}}$. Or $F^n(\overrightarrow{\varnothing})$, en tant que solution du système d'équations, est aussi une solution du système de contraintes. Comme elle est plus petite que n'importe quelle solution $\overrightarrow{\mathsf{RD}}$, il s'agit bien de la plus petite solution.

4.2.3.3 Rappels théoriques et résolution de l'analyse

Cette formulation d'une analyse sous forme de calcul de point fixe d'une fonction monotone croissante, et sa résolution par calcul itératif est une approche classique d'implantation des analyses statiques [NNH99]. Dans cette section, après quelques rappels théoriques sur la théorie des treillis, nous verrons comment faire l'analyse des définitions visibles pour n'importe quel programme, c'est à dire comment construire le système de contraintes ou d'équations dont on cherche le plus petit point fixe.

Définition 4.4 (Relation d'ordre). Soit E un ensemble, et $\mathcal{R}$ une relation binaire sur E, c'est à dire $\mathcal{R} \subseteq E \times E$. On dit que $\mathcal{R}$ est *relation d'ordre dans* E si elle est réflexive, transitive, et antisymétrique.

Définition 4.5 (Ensemble partiellement ordonné). Une relation d'ordre $\sqsubseteq$ dans E est dite *totale* si tous les éléments de E sont comparables deux à deux, c'est à dire si $\forall x, y \in E, x \sqsubseteq y \wedge y \sqsubseteq x$. Dans le cas contraire, elle est dite *partielle*, et on dit que l'ensemble E est *partiellement ordonné*.

Définition 4.6 (Majorants, Minorants). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné, et $F \subseteq E$ une partie de E, On définit :

$\text{Majorants}(F) = \{x \in E \mid \forall y \in F, y \sqsubseteq x\}$ l'ensemble des majorants de F, et
$\text{Minorants}(F) = \{x \in E \mid \forall y \in F, x \sqsubseteq y\}$ l'ensemble des minorants de F.

Définition 4.7 (Plus grand élément, plus petit élément). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné, et $F \subseteq E$ une partie de E.
S'il existe, on appelle *plus grand élément de* F un élément $x \in E$ tel que $x \in F \cap \text{Majorants}(F)$.
S'il existe, on appelle *plus petit élément de* F un élément $y \in E$ tel que $y \in F \cap \text{Minorants}(F)$.

Propriété 4.8. Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné, et $F \subseteq E$ une partie de E. S'ils existent, le plus grand élément de F et le plus petit élément de F sont uniques, c'est à dire :

$\forall x, x' \in F \cap \text{Majorants}(F), \quad x = x'$, et
$\forall y, y' \in F \cap \text{Minorants}(F), \quad y = y'$.

Démonstration. Soient $x, x' \in F \cap \text{Majorants}(F)$. Par définition d'un majorant, $\forall z \in E$, on a $z \sqsubseteq x$ et $z \sqsubseteq x'$. En particulier, on a $x \sqsubseteq x'$ et '$x' \sqsubseteq x$ et donc $x = x'$. La démonstration pour le plus petit élément est similaire. □

Dans l'exemple des définitions visibles, le plus petit élément de l'ensemble partiellement ordonné $(\mathcal{P}(Variable \times Label))^{12}$ est $\overrightarrow{\varnothing} = (\varnothing, \cdots, \varnothing)$.

Définition 4.9 (Borne supérieure, borne inférieure). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné. Une partie F de E admet une *borne supérieure* si et seulement si $\text{Majorants}(F)$ admet un plus petit élément, qu'on notera alors $\sqcup F$. Respectivement, on dira que F admet une *borne inférieure* si et seulement si $\text{Minorants}(F)$ admet un plus grand élément, qu'on notera alors $\sqcup F$.

Définition 4.10 (Treillis). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné. On dit que E est un *treillis* si pour toute paire d'éléments $x, y \in E$, l'ensemble E contient la borne supérieure $\sqcup\{x, y\}$ ainsi que la borne inférieure $\sqcap\{x, y\}$.

Cette définition peut également se lire comme celle d'une opération binaire entre les éléments du treillis. On parle alors de l'opération de *jonction* (*join*) pour désigner $x \sqcup y$, et de *jointure* (*meet*) pour désigner $x \sqcap y$. Par exemple, dans l'analyse des définitions visibles, l'opérateur de jonction est l'union composante par composante :

$$\overrightarrow{\mathsf{RD}} \sqcup \overrightarrow{\mathsf{RD}'} = (\mathsf{RD}_{entry}(1) \cup \mathsf{RD}'_{entry}(1)), \cdots, (\mathsf{RD}_{exit}(8) \cup \mathsf{RD}'_{exit}(8)).$$

Définition 4.11 (Treillis complet). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné. On dit que E est un *treillis complet* si toute partie F de E admet une borne supérieure et une borne inférieure.

Remarque Un treillis complet E admet donc un plus grand élément $\sqcup E$ qu'on note en général $\top$. De même, il admet un plus petit élément $\sqcap E$ qu'on note en général $\bot$.

Par exemple, l'ensemble $\mathcal{P}(A)$ des parties d'un ensemble A, ordonné par la relation d'inclusion, forme un treillis complet dont le plus petit élément est $\bot = \varnothing$ et le plus grand élément est $\top = A$. De plus, le produit cartésien de deux treillis forme lui-même un treillis, ordonné par la relation induite composante par composante. Ainsi, dans l'analyse des définitions visibles, l'ensemble $(\mathcal{P}(Variable \times Label))^{12}$ est un treillis complet.

Définition 4.12 (Monotonie). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné. Une fonction f de $(E, \sqsubseteq)$ dans $(E, \sqsubseteq)$ est dite *monotone croissante* si $\forall x, y \in E,\ x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

Définition 4.13 (Chaîne ascendante). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné. Une *chaîne ascendante* sur E est une suite finie ou infinie $\{x_i\}_{\in \mathbb{N}}$ d'éléments de E telle que $\forall i,\ x_i \sqsubseteq x_{i+1}$.

Définition 4.14 (Continuité). Soit $(E, \sqsubseteq)$ un ensemble partiellement ordonné. Une fonction monotone croissante f de E dans E est dite *continue* si pour toute chaîne ascendante F d'éléments de E on a :

$$f(\sqcup F) = \sqcup\{f(x) \mid x \in F\}.$$

Propriété 4.15 (Théorème de Kleene). Soit $(E, \sqsubseteq)$ un treillis complet, et f une fonction monotone croissante continue de E dans E. Alors f admet un plus petit point fixe $\mathrm{lfp}(f)$, et

$$\mathrm{lfp}(f) = \sqcup\{f^n(\bot) \mid n \geq 0\}$$

Démonstration. Soit $a = \sqcup\{f^n(\bot) \mid n \geq 0\}$, et montrons que a est le plus petit point fixe de f.

Comme f est continue, on a $f(a) = f(\sqcup\{f^n(\bot) \mid n \geq 0\}) = \sqcup\{f^{n+1}(\bot) \mid n \geq 0\} = \sqcup\{f^n(\bot) \mid n \geq 1\}$. Comme $\bot$ est le plus petit élément de E, alors $\sqcup\{f^n(\bot) \mid n \geq 1\} = \sqcup(\{f^n(\bot) \mid n \geq 1\} \cup \bot) = \sqcup\{f^n(\bot) \mid n \geq 0\} = a$. Donc a est bien un point fixe de f.

Soit x un second point fixe de f. $\bot \sqsubseteq x$, donc par récurrence, $\forall n \geq 0,\ f^n(\bot) \sqsubseteq f^n(x) = x$, ce qui implique $\sqcup\{f^n(\bot) \mid n \geq 0\} \sqsubseteq x$, donc a est bien le plus petit point fixe de f. □

Ce théorème est essentiel, car il permet de résoudre un problème d'analyse statique en l'exprimant sous la forme d'un système d'équations ou d'inéquations, et en calculant itérativement les images du plus petit élément $\bot$ du treillis. Dans l'exemple des définitions visibles, il s'agissait de calculer $F^n(\overrightarrow{\varnothing})$ pour $n \geq 0$ jusqu'à obtenir un point fixe. Cependant, pour s'assurer de la terminaison d'un tel calcul, il faut vérifier que le treillis considéré est de *hauteur* finie, c'est à dire qu'il n'admet pas de chaîne infinie strictement ascendante. S'il est de hauteur infinie, alors un simple calcul itératif risque de ne pas être suffisant pour atteindre le point fixe en un temps borné. Le treillis de l'analyse des définitions visibles étant de taille finie, il est également de hauteur finie.

Dans la section suivante, nous allons voir comment construire le système de contraintes ou d'équations décrivant le programme.

4.2.3.4 Construction du système de contraintes

Un algorithme d'analyse statique procède généralement en deux phases. En effet, avant de résoudre itérativement le systèmes de contraintes décrivant les propriétés recherchées, il faut le construire de telle sorte qu'il corresponde au programme auquel on s'intéresse. Dans cette section, on utilise l'exemple de l'analyse des définitions visibles pour introduire une notation, celle des schémas de règles d'inférence, que nous utiliserons dans les chapitres suivants pour décrire nos algorithmes d'analyse statique.

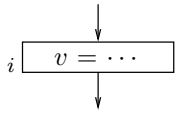
Définition 4.16 (Règle d'inférence). Une règle d'inférence est une fonction qui prend un certain nombre de formules et rend une formule. Ses arguments sont appelés *prémisses* et sa valeur la *conclusion*. On les note en général de la façon suivante :

$$\frac{\begin{array}{c}\textit{prémisse } 1\\ \textit{prémisse } 2\\ \cdots\\ \textit{prémisse } n\end{array}}{\textit{conclusion}}$$

Les règles d'inférences sont une notation très courante, en particulier dans le domaine de la logique. Nous les utiliserons ici pour exprimer des *schémas de règles* se lisant «si prémisse 1, et prémisse 2, ... et prémisse n, alors conclusion», dans lesquelles les prémisses et la conclusion contiennent des variables universelles.

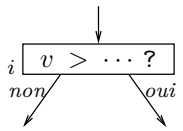
Par exemple, pour décrire la construction du système de contraintes associé à l'analyse des définitions visibles, on utilisera le schéma de règles donné ci-dessous. Les prémisses de ces règles sont des variables universelles représentant les éléments du graphe de flot de contrôle, et les conclusions sont des contraintes composant le système d'inéquations. Construire le système d'inéquations consiste à *appliquer* le plus possible de ces règles au programme étudié.

Affectation Cette règle s'applique pour toutes les instructions d'affectation du programme. Son effet est d'ajouter au système une contrainte reliant $\mathsf{RD}_{exit}(i)$ et $\mathsf{RD}_{entry}(i)$ pour chaque instruction i qui définit une variable v.



$$\mathsf{RD}_{exit}(i) \supseteq (\mathsf{RD}_{entry}(i) \setminus \{(v,l) \mid l \in \mathbb{N}\}) \cup \{(v,i)\}$$

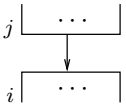
Branchement conditionnel Cette règle exprime le fait qu'une instruction de branchement ne modifie pas la visibilité des définitions.



$$\mathsf{RD}_{exit}(i) \supseteq \mathsf{RD}_{entry}(i)$$

Notre exemple ne comporte que des instruction de définition ou de branchement. Dans le cas général, il faudrait éventuellement prévoir une règle pour chaque type d'instruction, de façon à modéliser l'effet de toutes les instructions sur la propriété recherchée.

Propagation Cette règle exprime la propagation de la visibilité des définitions entre des instructions successives. Pour chaque arête dans le graphe de flot de contrôle, reliant une instruction j à une instruction i, elle ajoute une contrainte reliant $\mathsf{RD}_{entry}(i)$ et $\mathsf{RD}_{exit}(j)$.



$$\mathsf{RD}_{entry}(i) \supseteq \mathsf{RD}_{exit}(j)$$

Dans notre exemple des définitions visibles, l'application de cette règle générera les deux contraintes portant sur $\mathsf{RD}_{entry}(3)$, puisque deux arêtes mènent à l'instruction 3. De même, elle propagera $\mathsf{RD}_{exit}(3)$ vers les instructions 5 et 8.

Initialisation Le schéma de règles contient également une règle permettant d'exprimer le fait qu'à l'entrée du programme, les définitions précédentes des variables sont indéterminées. Pour chaque variable v figurant dans le programme, cette règle ajoute donc une contrainte de la forme $\mathsf{RD}_{entry}(1) \supseteq (v, ?)$.

$$\frac{v \in Variable}{\mathsf{RD}_{entry}(1) \supseteq (v, ?)}$$

Dans notre programme de calcul de factorielle, $Variable = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, et cette règle générera les mêmes contraintes que dans l'exemple.

Cette technique permet d'exprimer l'analyse des définitions visibles pour n'importe quel programme. En effet, l'application de ce schéma de règles aux éléments d'un graphe de flot de contrôle produira un système de contraintes définissant une nouvelle équation de point fixe. La résolution itérative de cette équation permet de déterminer les valeurs de RD en tous les points du programme.

Nous utiliserons dans les chapitres suivants des notations semblables pour définir les algorithmes d'analyse statique. Les prémisses représentent les éléments analysés, typiquement les instructions du programme considéré, et les conclusions représentent les contraintes modélisant ces instructions dans le treillis des propriétés.

4.2.4 Variantes : sensibilité au flot, au contexte

Les formulations présentées ci-dessus, sous forme d'équations et de contraintes ensemblistes, sont très utilisées pour exprimer les analyses dites *flot de données* : des propriétés sont calculées pour chaque point du programme, et l'enchaînement des instructions dans le graphe de flot de contrôle permet de propager de proche en proche l'effet des instructions sur ces propriétés. Cependant dans l'aperçu que nous avons donné jusqu'ici de ces techniques, nous n'avons utilisé comme illustration qu'un programme très simple. Lorsque l'on s'intéresse à un langage plus complexe, les bases théoriques de l'analyse restent les mêmes (treillis, calcul de point fixe), mais la formulation de l'analyse doit tenir compte des fonctionnalités du langage.

Par exemple, si le langage auquel on s'intéresse permet les appels de fonctions, il faut en tenir compte dans l'analyse, en se donnant une représentation de ces appels dans le graphe de flot de contrôle. Pour ne pas faire des sur-approximations trop grossières, on doit disposer d'un moyen de *combiner* les résultats obtenus dans les différentes fonctions, on parle alors d'analyse *inter-procédurale*.

Pour cela il faut pouvoir déterminer, à chaque site d'appel, quelle est la fonction, ou les fonctions, qui peuvent être invoquées à l'exécution. Obtenir ce *graphe d'appel statique* peut d'ailleurs s'avérer plus ou moins difficile en fonction du langage considéré. En Java, le graphe d'appel statique est en général construit grâce à une analyse *points-to* (cf section 4.3.1).

Sensibilité au contexte Si le comportement d'une fonction dépend évidemment de celui des fonctions qu'elle appelle, il dépend aussi souvent beaucoup de son propre contexte d'appel. Une analyse inter-procédurale dite *insensible au contexte* (*context-insensitive*), va produire pour chaque fonction un seul résultat, qui sera valable pour toutes les exécutions de la fonction. Pour plus de précision, les analyses dites *sensibles au contexte* (*context-sensitive*) calculent, au contraire un résultat différent pour chacun des contextes d'appel de la fonction [SP81]. La notion de contexte peut se référer à la chaîne d'appel complète menant à la fonction courante, ou à une sous-partie de cette chaîne. On parle alors d'analyse k-CFA, où k est la longueur de la sous-chaîne considérée [Shi88]. Elle peut également, suivant les cas, exprimer d'autres formes de contextes d'appels. Par exemple, dans le cas de langages orientés objets, on parle d'analyse *object-sensitive* pour désigner une analyse sensible à provenance de l'objet `this` [MRR05].

Sensibilité au flot On parle d'analyse *sensible au flot* (*flow-sensitive*) pour désigner les algorithmes d'analyse qui calculent un résultat différent en chaque point du programme. Certaines analyses dites *sensibles au chemin* (*path-sensitive*) enregistrent même l'ensemble du chemin parcouru dans le graphe de flot de contrôle jusqu'au point de programme courant.

Cependant, dans certains cas l'ordre des différentes instructions n'a pas d'incidence sur les propriétés auxquelles on s'intéresse. Il devient alors intéressant de ne calculer qu'un seul résultat pour tout le programme, ou pour toute la fonction, de façon à réduire le coût de l'analyse. Puisqu'elles manipulent moins de données, les analyses *insensibles au flot* (*flow-insensitive*) sont en effet bien plus efficaces, même si la granularité plus importante des résultats est parfois source d'imprécision.

Forme à affectation unique – SSA Une technique assez courante pour lutter contre cette imprécision est la transformation du code sous forme dite *à affectation unique* (*static single assignment*, ou SSA) [AWZ88]. Dans la forme SSA, chaque variable ne peut être définie (au même sens que dans l'analyse des définitions visibles) que par une seule instruction. La transformation d'un programme sous forme SSA [CFR+89] implique donc un renommage des variables affectées plusieurs fois. Ces variables sont ainsi *coupées en morceaux* : à chacune des définitions de la variable v, on remplace v par une nouvelle variable v_i. Aux points de jonction du graphe de flot de contrôle, on introduit un nouveau type d'instruction, l'affectation par une φ-*expression*, pour réconcilier les définitions visibles dans les différents prédécesseurs.

Programme d'origine Forme SSA

FIG. 4.17 – Insertion d'une φ-expression à un point de jonction du graphe de flot de contrôle.

Par exemple dans la figure 4.17, le programme d'origine comporte dans chacune des branches de la jonction une définition de la variable `v`. La transformation en SSA renomme donc `v` en deux nouvelles variables $\mathtt{v}_1$ et $\mathtt{v}_2$. À la jonction, `v` est encore renommée en $\mathtt{v}_3$. Pour maintenir la cohérence, on introduit donc une instruction $\mathtt{v}_3 = \varphi(\mathtt{v}_1, \mathtt{v}_2)$ dont le rôle est de *sélectionner* l'une ou l'autre des deux valeurs, selon la provenance du flot de contrôle.

La figure 4.18 donne le résultat de la traduction en SSA du programme de calcul de factorielle. Les variables `y` et `z` sont renommées en $\mathtt{y}_1 \cdots \mathtt{y}_4$ et $\mathtt{z}_1 \cdots \mathtt{z}_3$, et deux φ-expressions sont introduites au point de jonction correspondant à la boucle `while`.

Les bénéfices de la transformation en forme SSA sont multiples, le plus évident d'entre eux est peut-être la simplification considérable des analyses de flot de données : de fait, sur du code SSA, une analyse insensible au flot aura la même précision qu'une analyse sensible au flot. Depuis plusieurs années, le compilateur GCC utilise une représentation intermédiaire sous forme SSA pour améliorer l'efficacité des optimisations qu'il pratique sur le programme [Pop06].

4.2.5 Interprétation abstraite

Les analyses présentées jusqu'ici s'expriment en général assez facilement, sous forme de calcul de point fixe dans des treillis assez simples. S'assurer de la correction de telle ou telle analyse n'est cependant pas une tâche facile, surtout lorsqu'on s'intéresse à des propriétés de plus en plus complexes. De plus, certaines propriétés s'expriment dans des treillis de hauteur infinie, la stratégie du calcul itératif se retrouve donc trop limitée pour résoudre les analyses de ces propriétés.

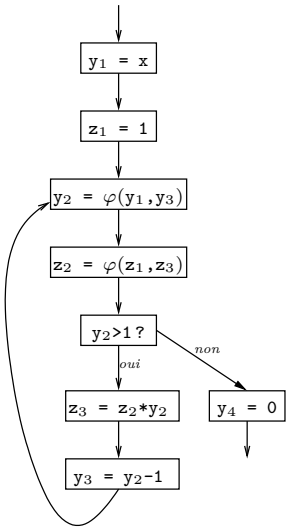


FIG. 4.18 – Traduction sous forme SSA du programme de calcul de factorielle.

Basée entre autres sur des théories qui permettent de comparer différents treillis entre eux, l'*interprétation abstraite* [CC77] est une discipline qui permet de manipuler des analyses statiques comme des objets mathématiques à part entière. Par des abstractions successives d'une certaine sémantique formelle du langage, elle permet ainsi d'obtenir différentes analyses statiques *correctes par construction* [CC79].

L'interprétation abstraite est beaucoup utilisée dans le domaine des analyses numériques [Min02, BCC+03], où l'on cherche à caractériser les valeurs prises par les variables numériques d'un programme, et éventuellement les relations entre ces variables. En effet, dans ce contexte il est naturel de chercher à déterminer quelles sont les relations d'approximation entre différentes analyses, puisqu'elles traitent toutes de propriétés similaires. Cependant, il y a d'autres champs d'application de l'analyse statique pour lesquels l'interprétation abstraite n'apporte pas grand-chose, notamment si on connaît des moyens de résoudre l'analyse sans avoir à faire d'approximation. Par exemple, des analyses de flot de données comme l'analyse des sous-expressions communes, ou des définitions visibles, utilisées pour l'optimisation dans les compilateurs ne sont habituellement pas considérées en termes d'interprétation abstraite.

4.2.6 Conclusion

Ce rapide aperçu du domaine de l'analyse statique n'est bien sûr pas du tout exhaustif. Il s'agit en effet d'un domaine très vaste, et l'objectif de cette section est seulement d'en poser quelques bases de façon à positionner notre travail. Il existe de nombreuses autres techniques d'analyse statique, par exemple celles basées sur le typage, qui sont également très utilisées.

Notons cependant que ces différents types de techniques (analyse de flot de données, analyse de flot de contrôle, interprétation abstraite, systèmes de types, etc.) malgré des différences considérables dans la manière de formuler le problème, ont des objectifs similaires. Dans tous les cas, il s'agit de produire une sur-approximation *correcte* d'une certaine propriété du programme, le plus souvent sous la forme d'un élément de treillis [NNH99].

4.3 Les analyses de pointeurs

Nous avons présenté dans la section précédente différentes techniques d'analyse statique, et remarqué que les théories sous-jacentes étaient souvent similaires. Cependant, les analyses présentées jusqu'ici s'intéressent essentiellement à des propriétés qualitatives du programme

(analyse des définitions visibles), ou aux valeurs numériques prises par les variables (interprétation abstraite).

On désigne sous le nom d'*analyses de pointeurs* les analyses statiques qui s'intéressent à des propriétés liées aux objets manipulés par le programme, par exemple à leur provenance, ou à leur durée de vie. Nous nous intéressons dans ce travail à des questions liées à la gestion mémoire, en particulier dans le contexte du langage Java. Il est donc nécessaire de faire un tour d'horizon des différentes problématiques étudiées dans ce domaine.

4.3.1 Analyses «points-to»

L'une des questions les plus évidentes qui se posent lors de l'analyse d'un programme à pointeurs est «sur quels objets ce pointeur peut-il pointer ?» Les analyses statiques qui répondent à ce type de questions sont appelées *points-to analysis* dans la littérature. L'abstraction la plus répandue pour représenter l'ensemble des objets en question consiste à ne retenir que les *sites d'allocation* d'où proviennent ces objets, c'est à dire les points de programme où ils ont été alloués. Malgré de nombreux travaux à ce sujet, la question reste un thème de recherche très actif [Hin01].

Suivant le compromis désiré entre performance et précision, il existe plusieurs approches de ce problème. Certains travaux adaptent les techniques d'analyse de flot de données pour propager des *ensembles de sites d'allocation* (*points-to sets*) dans le programme, soit *en avant* [And94], soit de façon bidirectionnelle [Ste96]. Pour plus de précision, certains algorithmes utilisent une analyse sensible au contexte [NKH04].

Dans les langages orientés objet, une autre approche classique est de se baser uniquement sur la hiérarchie des classes [DGC95, BS96, SHR+00], ce qui est très efficace car l'algorithme n'a plus besoin d'analyser le code proprement dit, mais les résultats obtenus sont en général moins précis.

Construction du graphe d'appel statique L'une des applications principales des analyses «points-to» dans les langages orientés objet est la construction du *graphe d'appel statique* du programme. En effet, contrairement à ce qui se passe dans un langage impératif classique, de nombreuses fonctions peuvent a priori être invoquées à chaque site d'appel virtuel, et ce n'est qu'au moment de l'exécution que le système, connaissant le type des objets, décidera quelle fonction utiliser. On parle de *résolution dynamique* (*dynamic dispatch*) pour désigner ce mécanisme. Par exemple, en Java, les conteneurs offerts par la bibliothèque standard appellent fréquemment la méthode `Object.hashcode()` pour déterminer si deux objets qu'ils contiennent sont différents. Mais si l'objet en question est d'une classe qui redéfinit la méthode `hashcode()`, le système invoquera la méthode redéfinie, et non la méthode d'origine.

La construction du graphe d'appel nécessite donc une connaissance des différents types possibles des objets utilisés à chaque site d'appel. Les analyses «points-to» par analyse de flot de données permettent de répondre à cette question, mais doivent à leur tour disposer d'un graphe d'appel. Pour cette raison, des analyses rapides basées sur le typage sont souvent utilisées en première approximation, et le graphe d'appel est ensuite *raffiné* par une analyse plus précise [LH03].

4.3.2 Analyses de forme

L'objectif d'une analyse de forme (*shape analysis* [SRW02]) est de déterminer statiquement certaines propriétés à propos de la *forme* des structures de données manipulées par le programme. Il s'agit par exemple de répondre à la question «la liste chaînée que désigne cette variable est-elle acyclique ?». De même qu'à l'exécution on se représente en général le tas comme un graphe, une analyse de forme calcule une abstraction statique du tas elle aussi sous forme de graphe. Pour

cela, il faut en général modéliser plusieurs objets en un seul sommet du graphe abstrait, de façon à limiter la taille des résultats.

En effet, comme le programme peut allouer durant son exécution un nombre arbitraire d'objets, il est essentiel pour l'analyse de s'en donner une représentation compacte. Pour des programmes manipulant des structure régulières, comme les listes, il peut être intéressant par exemple de *borner* la taille des structures représentées [JM81, Str92]. Une autre approche, plus similaire aux analyses *points-to*, est d'abstraire par un seul sommet de graphe tous les objets alloués au même endroit du programme [JM82, CWZ90]. On parle alors de modélisation par le site d'allocation (*allocation site model*). D'autres travaux utilisent des expressions symboliques pour représenter les *chemins d'accès* qui mènent au même objet [LH88].

Il existe un grand nombre d'algorithmes d'analyse de forme dans la littérature, car les applications de ces techniques sont très variées. En ce qui concerne la gestion mémoire, on peut notamment citer des travaux récents [CR06, GMF06] qui cherchent à caractériser la durée de vie des objets grâce à une analyse de forme sensible au flot. L'idée est d'insérer des instructions de désallocation explicite dans le programme aux points où l'on est certain que tel ou tel objet sera mort.

4.3.3 Analyses d'échappement

On dit qu'un objet *s'échappe* du contexte dans lequel il a été alloué lorsqu'il devient accessible depuis l'extérieur de ce contexte [HS02]. Par exemple, en Java, un objet retourné par une méthode s'échappe de cette méthode, puisqu'il sera accessible depuis une variable locale de la méthode appelante. Une *analyse d'échappement* a pour objectif de déterminer statiquement les sites d'allocation dont les objets peuvent s'échapper à l'exécution.

Cette information est utile en particulier dans une optique d'optimisation de programmes. Par exemple, si un objet ne s'échappe pas du *thread* dans lequel il a été alloué, toutes les opérations de synchronisation liées à cet objet peuvent être ignorées sans risque [ACSE99, BH99, Ruf00]. L'autre application principale de l'analyse d'échappement est la transformation de programme pour l'allocation en pile : si un objet ne s'échappe pas de la méthode qui l'alloue, le système peut allouer l'objet sur la pile d'exécution plutôt que dans le tas. Ainsi, il sera automatiquement désalloué lors du retour de la méthode, allégeant ainsi le travail du ramasse-miettes.

De nombreux algorithmes d'analyse d'échappement ont été proposés pour Java [GS00, SR01, Bla03, CGS+03], avec en général le double objectif de l'allocation en pile et de l'élimination des synchronisations implicites. La quantité d'objets alloués en pile par ces techniques est très variable d'un programme à l'autre. Pour certains programmes, une analyse d'échappement précise peut parvenir à allouer la grande majorité des objets dans la pile, et donc à décharger largement le ramasse-miettes. Cependant, les objets restants sont toujours alloués dans le tas, et donc la présence d'un ramasse-miettes reste nécessaire.

4.3.4 Algorithmes de synthèse de régions

Pour aller plus loin que l'allocation en pile, on peut utiliser un gestionnaire mémoire en régions. En effet, l'allocation en pile peut être vue comme un cas particulier d'allocation en régions, où une nouvelle région est associée automatiquement à chaque activation de fonction.

Tofte et Talpin [TT94, TT97] proposent une *synthèse de régions* (*region inference*) pour la langage ML, basée sur une analyse de types. Le programme est augmenté d'annotations qui sont traduites à la compilation en des primitives d'utilisation de régions. Pour Java, plusieurs travaux explorent une approche similaire. Chin *et al.* [CCQR04] utilisent également un système de types, alors que Cherem et Rugina [CR04] se basent sur une analyse *points-to* inter-procédurale. Dans les deux cas, les résultats expérimentaux obtenus à l'exécution sont très inégaux. Pour une proportion considérable des programmes, la synthèse de régions produit une sur-approximation trop grossière des durées de vie, ce qui amène le système à placer une grande quantité des données

allouées dans une région qui n'est jamais détruite. Ce problème, que nous déjà rencontré dans la section 3.5.5 (cf page 54), et appelé *syndrome d'explosion des régions*, n'est pas uniquement dû à l'imprécision des analyses : même en utilisant manuellement un allocateur en régions, il est difficile pour un programmeur de ne pas provoquer le même genre de phénomène [BZM02].

Comme rien ne permet de *prédire* ce syndrome avant l'exécution, ces approches ne sont pas satisfaisantes pour la problématique qui nous occupe : dans un contexte embarqué, on veut pouvoir s'assurer dès la phase de développement que le système ne manquera pas de ressources une fois déployé dans son environnement.

4.4 Discussion

Le domaine de l'analyse de programmes est très vaste, et de nombreux travaux appliquent des analyses statiques à des problématiques de gestion mémoire. Cependant, aucune des approches présentées dans ce chapitre ne permet de résoudre de façon satisfaisante le problème des temps de pause auquel nous nous intéressons dans cette thèse.

Par exemple, les approches basées sur la désallocation explicite ne traitent qu'une partie des objets, et doivent donc cohabiter avec un ramasse-miettes. Pour que la désallocation ait un sens, il faut utiliser un modèle mémoire dans lequel les objets ne sont jamais déplacés. En effet dans un modèle mémoire *compactant*, comme celui d'un ramasse-miettes à copie, désallouer explicitement des objets n'est d'aucun intérêt. Les problèmes liés à la fragmentation empêchent donc l'usage de ce genre de techniques dans un contexte temps-réel.

De même, les approches basées sur des analyses d'échappement ne résolvent que partiellement le problème. Pour une certaine part, variable, de la mémoire, elles permettent d'utiliser des mécanismes d'allocation en pile aux performances prévisibles. Mais il suffit qu'un site d'allocation dans le programme soit jugé s'*échappant* pour que les garanties liées à l'allocation en pile ne tiennent plus. En effet, si le programme est contraint de placer certains de ses objets dans le tas, alors le système doit comporter un ramasse-miettes pour les désallouer, et subir des temps de pause inopinés.

Seule l'utilisation exclusive d'un gestionnaire mémoire en régions permet de profiter d'opérations mémoire ayant un comportement temporel complètement prévisible. Les analyses statiques visant la synthèse de régions sont donc très prometteuses, mais leur sensibilité au *syndrome d'explosion des régions* est problématique. En effet, un tel risque de fuite de mémoire interdit d'embarquer un programme sur un système aux ressources limitées.

Deuxième partie

Contribution

Chapitre 5

Présentation générale de l'approche

5.1 Introduction

Pour permettre l'utilisation de Java dans le monde des systèmes embarqués et temps-réel, le problème de la gestion mémoire reste rédhibitoire. On peut distinguer dans ce problèmes deux questions distinctes, même si elles ne sont pas complètement indépendantes : celle de la *quantité* de mémoire utilisée par le programme, et celle de *l'impact temporel* du gestionnaire mémoire sur le programme. La première est plutôt liée aux notions de système *embarqué*, et la seconde à celles de système *temps-réel*.

Les approches existantes Les différentes approches présentées dans les chapitres précédents attaquent chacune ce problème sous un angle différent, cependant aucune d'entre elles ne le résout de façon satisfaisante. Les travaux qui reposent sur un changement de langage, comme ceux basés sur la RTSJ, sont très délicats à mettre en œuvre car ils impliquent un effort de développement considérable. Il serait préférable pour le développeur de continuer à programmer selon ses habitudes, et donc de recourir à des outils plus ou moins automatiques. Les algorithmes d'analyse d'échappement (cf section 4.3.3) ne cherchent pas à allouer tous les objets en pile, et doivent donc recourir à un ramasse-miettes pour traiter le reste de la mémoire. Le comportement temporel au pire cas du programme est donc conditionné par celui du ramasse-miettes en question. D'autre part, les approches basées sur la synthèse de régions (cf section 4.3.4) souffrent de fuites de mémoire intempestives, qui sont détectées seulement lors de l'exécution du programme. Il paraît donc hasardeux de les employer dans le contexte des systèmes embarqués dans lesquels la mémoire est une ressource critique. Les ramasse-miettes temps-réel (cf section 3.4.3) quant à eux requièrent un modèle du comportement mémoire de l'application. Cette modélisation fine de l'application est très difficile à obtenir pour le développeur. De plus, une approximation trop grossière oblige le système à s'exécuter dans de mauvaises conditions, c'est à dire en consacrant au ramasse-miettes la majeure partie du temps d'exécution.

A notre avis, une des limitations majeures de ces approches est leur intervention trop tardive dans le cycle de vie du programme. En effet, elles supposent un scénario de développement assez rigide, dans lequel elles s'insèrent à un moment où le programme est déjà écrit. D'abord le développeur écrit son programme, ensuite une phase d'analyse (automatique ou manuelle) examine le code pour en déduire certains résultats, et finalement le programme est exécuté dans un environnement ad hoc, où certaines propriétés sont garanties.

Notre approche Nous proposons d'attaquer le problème en le *remontant* plus en amont dans le cycle de développement. En effet, le code n'est pas écrit d'un seul jet ; à l'inverse le programmeur fait sans arrêt des allers-retours : écrire un peu de code, le compiler, puis le corriger, etc. Il s'agit donc d'*intégrer* l'analyse de pointeurs dans ce cycle de développement, au niveau du compilateur, de façon à *assister* le programmeur tout au long de sa tâche. L'idée est de lui offrir

non pas des outils entièrement automatiques, puisque nous avons vu que les problèmes rencontrés sont trop difficiles pour les analyses de pointeurs existantes, mais des outils *semi*-automatiques, lui fournissant des résultats intelligibles et exploitables.

Nous avons adopté la synthèse de régions comme technique principale, afin de bénéficier du comportement temporel attrayant du modèle mémoire associé. Toutefois, il faut résoudre de façon satisfaisante les problèmes liés : d'une part le syndrome d'explosion des régions risque d'introduire des fuites de mémoire dans l'exécution de programmes qui n'en comportent pas à l'origine. D'autre part, il faut s'assurer que le système de gestion mémoire utilisé à l'exécution offre bien des opérations dont les coûts sont prévisibles.

Pour le premier problème, nous proposons d'*impliquer* le programmeur dans le processus d'analyse. Il faut donc que l'algorithme d'analyse soit assez simple pour être compris par le développeur, et que les résultats obtenus soient facilement intelligibles. Pour le second, nous allons proposer une implantation astucieuse du modèle mémoire en régions, dans laquelle toutes les opérations mémoire auront un coût d'exécution constant.

5.2 Synthèse de régions par analyse statique

Déterminer statiquement la durée de vie des objets alloués par le programme est un problème indécidable (cf section 4.2.1), et aucun algorithme d'analyse statique ne peut espérer y parvenir en toute généralité. On s'intéresse donc à des analyses qui cherchent à produire une sur-approximation de ces durées de vie. Les différentes analyses existantes (cf section 4.3) représentent ainsi différents compromis entre la précision des résultats et le coût de l'algorithme.

Les analyses d'échappement, même pour les plus sophistiquées d'entre elles, attaquent la question avec beaucoup de conservatisme : pour chaque site d'allocation, soit elles parviennent à déterminer qu'il produira *uniquement* des objets très éphémères, soit elles répondent «je ne sais pas», ce qui n'est pas facilement exploitable. À l'inverse, les algorithmes de synthèse de régions cherchent plutôt à calculer des *relations* entre les durées de vie de différents objets : si l'on arrive à déterminer statiquement que la durée de vie de deux objets va être similaire, alors on indique au système de les placer tous les deux dans la même région. Le manque de précision se manifeste ici d'une autre façon, par le *syndrome d'explosion des régions*.

L'hypothèse générationnelle Puisque nous avons décidé d'impliquer le programmeur dans le processus d'analyse, notre algorithme de synthèse de régions doit produire des résultats simples. L'algorithme que nous proposons, baptisé *analyse d'interférence de pointeurs* est donc assez simpliste, même si on verra qu'en pratique (cf section 8.3) il mène à un comportement mémoire sensiblement équivalent aux algorithmes existants de synthèse de régions [CR04, CCQR04], avec un moindre coût d'analyse. L'idée est de grouper chaque structure de données (c'est à dire un ensemble d'objets connectés) dans une région. Une variante récente de l'hypothèse générationnelle [HHDH02] nous indique en effet que la relation de connexion a une corrélation très forte avec la durée de vie : «*in most cases, the probability that linked objects die at the same time is much higher than the probability of any two objects dying at the same time*».

Principe Plutôt que d'essayer de détecter statiquement des relations complexes de durée de vie entre les objets, nous proposons donc d'adopter la stratégie d'allocation en régions la plus simple qui soit : «deux objets connectés vont dans la même région». L'objectif de notre analyse statique est donc uniquement de prévoir les connexions entre objets. Le principe de l'analyse est le suivant : pour chaque méthode m du programme, l'algorithme partitionne l'ensemble des variables locales de m en regroupant celles qui *interfèrent*, c'est à dire qui risquent de pointer sur des objets connectés. Nous désignerons cette relation d'interférence par le symbole $\sim_m$, et les classes d'équivalence de $\sim_m$ par le terme de *famille*.

Cette analyse se déroule en deux phases. Pendant la première phase, intra-procédurale, les familles sont construites à partir des instructions de manipulation de pointeurs : dans chaque méthode, les instructions de la forme `v=u`, `v.f=u` ou `v=u.f` montrent que les variables `u` et `v` *interfèrent*, c'est à dire que les objets sur qui elles pointeront à l'exécution seront à coup sûr connectés. L'analyse en déduit donc `u` $\sim_m$ `v`, puis fait la clôture transitive de $\sim_m$ pour obtenir une relation d'équivalence. Pendant la seconde phase, inter-procédurale, l'analyse reproduit sur les familles l'effet des appels de méthodes : à chaque site d'appel, l'interférence des paramètres formels (dans la méthode appelée) est reportée sur les arguments effectifs. Par exemple, si une méthode m appelle une méthode m' avec deux arguments `x` et `y`, et que les paramètres formels correspondants `p` et `q` vérifient `p` $\sim_{m'}$ `q`, l'analyse en déduit `x` $\sim_m$ `y`.

Stratégie d'allocation La stratégie d'allocation utilise les familles ainsi calculées pour placer les objets dans les régions à l'exécution. Lors de l'allocation d'un nouvel objet à un site d'allocation de la forme `x = new C` dans la méthode m, on cherche une autre variable locale de m qui soit dans la même famille que `x`. S'il en existe une, par exemple `y`, alors le nouvel objet doit être placé dans la même région que l'objet pointé par `y`. Dans le cas contraire, on est en train d'allouer le premier objet d'une nouvelle structure de données, on peut donc créer une nouvelle région et y placer l'objet.

Cette technique est illustrée sur la figure 5.1. Ce programme crée tout d'abord une liste, puis alloue les deux objets o_1 et o_2, et ajoute ensuite o_1 à la liste. Pour plus de clarté, l'allocation d'un objet et l'appel du constructeur de la classe correspondante ont été explicités séparément, comme c'est le cas dans le *bytecode* (cf section 6.2).

```
main()
{
  ArrayList list = new ArrayList;
  list.<init>(10);

  Object o1=new Object;
  Object o2=new Object;

  list.add(o1);
}
```

```
class ArrayList
{
  Object[] data;
  int size;

  <init>(int capacity)
  {
    this.size = 0;
    tmp = new Object[capacity];
    this.data = tmp;
  }

  void add(Object o)
  {
    this.data[this.size] = o;
    this.size ++;
  }
}
```

FIG. 5.1 – Illustration de la stratégie d'allocation sur un programme simple.

Sur cet exemple, l'analyse d'interférence de pointeurs calcule tout d'abord `this` $\sim_{\texttt{<init>}}$ `tmp` et `this` $\sim_{\texttt{add}}$ `o`. La phase inter-procédurale propage ensuite cette seconde interférence dans `main` sur les deux arguments `list` et `o1`. Lors de l'exécution, l'objet liste et o_1 seront donc alloués dans la même région. L'analyse remarque cependant que o_2 ne sera jamais connecté avec quiconque (la variable `o2` est seule dans sa famille), et il pourra donc être alloué dans une autre région.

Discussion L'intérêt de cette approche est qu'elle impose des coûts très faibles à la machine virtuelle et chacune des opérations liées à la gestion mémoire peut être réalisée en temps constant. En implantant les régions sous la forme d'une liste chaînée de pages mémoire de taille uniforme, la création, la destruction, ainsi que l'extension d'une région sont réalisées en quelques manipulations de pointeurs.

La création d'une région se limite à la sélection d'une page libre. Cette opération est réalisée en retirant la première page de la liste des pages libres (*free-list*), et s'exécute en temps constant.

L'allocation d'un objet est toujours faite dans la dernière page de la région, par décalage de pointeur. Si cette page ne dispose plus de suffisamment d'espace, la région est étendue par l'adjonction d'une nouvelle page. Au pire cas, chacune de ces opérations s'exécute donc en un temps fixe et succinct.

De plus, l'opération la plus complexe, la recherche d'une autre variable locale dans la famille de celle qu'on est en train d'allouer, s'exécute en un temps prévisible, puisqu'au pire cas il faut examiner toutes les autres variables locales de la méthode, qui sont en nombre connu. La création des régions est provoquée à la volée par l'allocation du premier objet de la structure de données correspondante. La destruction, quant à elle, peut intervenir dès que plus aucune variable locale ne pointe dans une région. Cette propriété peut être détectée de différentes manières, notamment par un comptage de références associé aux variables locales, ou par un examen périodique de la pile d'exécution. Pour ne pas imposer de barrières en écriture ni avoir à interrompre l'exécution pour parcourir la pile, nous avons associé la création et la destruction des régions à celle des cadres de pile contenant les variables locales elles-mêmes.

Ainsi, notre approche de gestion mémoire en régions permet d'exécuter le programme sans avoir à faire de parcours du tas ou de la pile, ni à rajouter de barrières en écriture aux opérations sur les pointeurs, tout en profitant du comportement temporel favorable offert par l'allocation par décalage de pointeur.

5.3 Interaction avec le programmeur

Explosion des régions Comme toutes les approches basées sur la gestion mémoire en régions, notre proposition risque cependant de conduire à un important surcoût en espace pour certains programmes, ce que nous avons baptisé le *syndrome d'explosion des régions*. Le problème se révèle lorsque le cycle de vie des objets et celui des régions correspondantes sont trop différents : à cause de l'impossibilité de désallouer individuellement les objets, il peut arriver qu'un nombre sans cesse croissant d'objets *morts* s'accumulent dans une certaine région, mais qu'elle ne puisse pourtant pas être détruite car elle contient encore des objets vivants.

Ce problème est illustré sur la figure 5.2. Dans ce programme, la méthode `main` crée tout d'abord un objet `root` qui aura une longue durée de vie, puis entre dans une boucle infinie où elle appelle la méthode `bla`. À chaque itération, cette méthode alloue un nouvel objet `foo`, qu'elle chaîne à son paramètre `this`. Pour ce programme, notre politique d'allocation placera tous les objets `foo` dans une même région, puisqu'ils seront chacun connectés à `root` pendant un moment. Comme la région en question ne sera jamais détruite (puisque la variable `root` pointe directement dedans), toutes les instances de `foo` vont s'y accumuler, occupant ainsi l'ensemble de la mémoire.

```
main()
{
  RefObject root=new RefObject();

  while(true)
  {
    root.bla();
  }
}
```

```
class RefObject
{
  Object f;

  bla()
  {
    Object foo=new Object;
    this.f=foo;
  }
}
```

FIG. 5.2 – Un programme qui provoque une explosion de région.

Dans ce programme, les objets `root` et `foo`, connectés par un pointeur, n'ont pas du tout une durée de vie similaire. Le premier est utilisé pendant toute l'exécution du programme, alors que le second ne vit que pendant une itération de la boucle. Ceci traduit le fait que le programme de

la figure 5.2 ne respecte pas notre *hypothèse générationnelle* (présentée page 76).

Détection par analyse statique À notre connaissance, ce genre de problème affecte toutes les approches qui utilisent la gestion mémoire en régions, tant celles basées sur des transformations automatiques du programme [CR04, CCQR04] que sur une utilisation manuelle de l'allocateur [BZM02]. En effet, pour l'éviter, il faudrait être capable de placer sans exception des objets de durées de vie différentes dans des régions distinctes, et donc de déterminer avec exactitude ces durées de vie à l'avance.

Nous proposons donc de chercher à *prévoir* le problème dès la phase d'analyse, de façon à faire remonter l'information au programmeur sans qu'il ait à exécuter le programme. Pour cela, nous proposons une seconde analyse statique, qui exploite les résultats de l'analyse d'interférence de pointeurs en les combinant au graphe d'appel statique, pour y rechercher des situations «à risque», c'est à dire des situations où un objet à longue durée de vie est connecté, à l'intérieur d'une boucle, à des instances successives d'un autre objet à durée de vie courte.

L'algorithme d'analyse est encore une fois assez simple. Il consiste à reconstruire, toujours statiquement, l'ensemble des sites d'allocation qui vont placer leurs objets dans la même région, et à déterminer ceux risquant de s'accumuler. Nous désignerons un tel ensemble sous le nom de *tribu*, par extension des *familles* locales à chaque méthode. Intuitivement, une tribu est un groupe de familles, reliées entre elles par des correspondances argument-paramètre, dont tous les objets seront placés dans la même région. L'idée est de remarquer que les explosions de région ne sont en général pas provoquées par une seule méthode, mais par l'interaction de plusieurs méthodes. Un objet qui appartient à une certaine famille dans une certaine méthode peut se retrouver appartenir à une autre famille dans une méthode appelée, puisqu'il aura «changé de nom» lors du passage de paramètre.

Cette idée est illustrée dans la figure 5.3. Dans ce programme, la méthode `main` alloue deux objets, `root1` et `root2`, avant d'entrer dans une boucle infinie où elle appelle différentes méthodes sur ces objets. La méthode `bla`, identique à celle de la figure 5.2, alloue un nouvel objet qu'elle chaîne à son paramètre. La méthode `bli` par contre, alloue un objet temporaire `bar` qui n'interfère avec aucune autre variable locale. Enfin, la méthode `blo` accède à un champ de son paramètre, donc la variable `baz` interfère avec `this`.

```
1  main()
2  {
3
4    RefObject root1=new RefObject();
5
6    root1.bla();
7
8    RefObject root2=new RefObject();
9
10   while(true)
11   {
12     root2.bla();
13     root2.bli();
14
15     root1.blo();
16   }
17 }
```

```
20 class RefObject
21 {
22   Object f;
23
24   bla()
25   {
26     Object foo=new Object;
27     this.f=foo;
28   }
29
30   bli()
31   {
32     Object bar=new Object;
33   }
34
35   blo()
36   {
37     Object baz=this.f;
38     this.bli();
39   }
40 }
```

Fig. 5.3 – Un programme qui possède différentes formes de tribus. Les tribus proprement dites sont données dans la figure 5.4

Dans la méthode `main`, les deux objets `root1` et `root2` sont seuls dans leur famille respective,

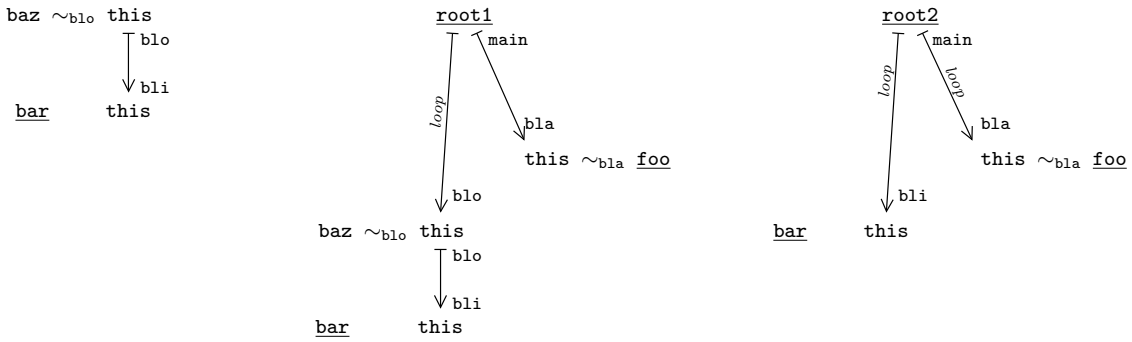


FIG. 5.4 – Les *tribus* du programme de la figure 5.3.

car aucune instruction ne les fait interférer. Mais si on s'intéresse aux «autres noms» que porteront ces objets dans les méthodes appelées, on peut construire les tribus présentées sur la figure 5.4. Les variables soulignées (`root1`, `root2`, `bar`, et `foo`) correspondent aux sites d'allocation du programme. Les flèches représentent les passages de paramètres lors des appels. Les appels situés dans une boucle sont repérés par la mention *loop*.

Chaque tribu va correspondre à une région, dans laquelle se retrouveront tous les objets produits aux sites d'allocation mentionnés dans les familles. En fonction de leur forme, on peut déterminer les risques d'explosion de région.

Dans la figure 5.4, les tribus correspondant aux familles de `bar` et de `root1` représentent des régions dans lesquelles le nombre d'objets maximum est borné, car elles ne comportent pas de site d'allocation exécuté dans une boucle. On peut y trouver soit des appels dans une boucle, mais qui ne mènent pas à une allocation (comme l'appel `root1.blo()`), soit des appels qui mènent à une allocation mais hors d'une boucle (appel `root1.bla()`). Il faut noter que les appels à la méthode `bli` provoquent certes une allocation, mais dans une famille qui n'interfère pas avec les paramètres formels, l'objet `bar` sera donc systématiquement alloué dans une région distincte, détruite lorsque la méthode `bli` se termine. L'exécution répétée de cette méthode, quel que soit le contexte d'appel, ne provoquera pas d'explosion de région.

Au contraire, la tribu de `root2` risque de provoquer une explosion de région à cause du motif

$$\underline{\texttt{allocation}} \cdots \xmapsto{\quad loop \quad} \cdots \underline{\texttt{allocation}}$$

En effet, comme dans l'exemple de la figure 5.2, chaque exécution de la méthode `bla` provoque l'allocation d'un objet dans la région de `root2` qui n'est jamais détruite.

Notre outil *avertit* le programmeur lorsqu'il trouve des tribus qui comportent des motifs problématiques. De cette façon, il peut comprendre quelle méthode ou quelle combinaison de méthodes est à l'origine du risque d'explosion. Par exemple, pour ce programme, on obtiendrait l'avertissement suivant :

```
example.java:26: Possible memory leak
calling context: example.java:12:    root2.bla()
      Object foo=new Object;
                 ^
```

En intervenant tôt dans le cycle de développement, cette approche permet de guider le programmeur : il peut par exemple rectifier son code pour supprimer le problème, ou décider que l'avertissement est une fausse alerte, par exemple parce qu'il sait par ailleurs borner la boucle problématique. Bien sûr, certains cas resteront insolubles : si la logique de l'application requiert réellement une structure de données durable *et* changeante, dont les objets ne respectent pas du tout notre hypothèse à propos des durées de vie, le programme ne pourra pas être exécuté en régions. Notre approche permet cependant de dépister ces programmes à l'avance, et réciproquement de garantir l'absence de problème pour les autres programmes.

Organisation des chapitres suivants

Nous avons présenté dans ce chapitre les idées sur lesquelles repose ce travail. Nous proposons d'utiliser un gestionnaire mémoire en régions, afin d'avoir des opérations mémoire dont les temps d'exécution sont prévisibles. Pour placer les objets dans les régions, nous proposons un algorithme d'analyse statique qui prévoit les connexions entre les objets. Cette technique ne donne pas des résultats satisfaisants si le programme ne respecte pas l'hypothèse générationnelle, c'est à dire connecte des objets de durées de vie très différentes. Pour détecter ces complications, nous proposons une seconde analyse statique qui avertit le programmeur des situations à risque.

Le chapitre suivant établit la syntaxe du langage analysé, et propose de reformuler chaque méthode du programme sous la forme d'un graphe. En effet, il est plus naturel de raisonner sur les objets manipulés par le programme en se basant sur une représentation où les relations de pointage entre les objets sont explicites, plutôt que sur le code source du programme.

Le chapitre 7 donne une formulation plus rigoureuse de l'analyse d'interférence de pointeurs présentée ici, de la politique associée d'allocation en régions, ainsi que de l'analyse d'anticipation des régions. Nous y donnons aussi une preuve que des objets connectés sont correctement placés dans une même région au moment de leur allocation.

Le chapitre 8 présente notre implantation de l'analyse statique ainsi que du gestionnaire mémoire, et détaille en particulier les techniques mises en œuvre pour obtenir des opérations mémoire s'exécutant en temps prévisible.

Chapitre 6

Représentations du programme

Avant d'être exécuté, un programme Java est traduit depuis le langage source vers un langage binaire, un bytecode, destiné à être interprété par la machine virtuelle. Le langage source Java offre beaucoup de sucre syntaxique comme les structures de contrôle ou la concaténation implicite de chaînes de caractères qui facilitent la tâche du programmeur mais qui rendent plus difficile l'analyse automatique de programmes. Le bytecode Java par contre, a une structure syntaxique plus simple, mais sa nature de langage à pile le rend plus difficile à lire à l'œil nu. La machine virtuelle Java est en effet une machine à pile et non pas une machine à registres. Son jeu d'instructions opère donc uniquement sur des opérandes stockées dans la pile d'exécution.

Dans un souci de clarté, nous utilisons ici une syntaxe équivalente au bytecode mais basée sur une machine à registres. Cette syntaxe est inspirée de celle du langage Jimple [VRH98], la représentation intermédiaire utilisée par l'infrastructure d'analyse statique SOOT [VRCG+99], avec laquelle nous avons implanté les analyses statiques présentées ici (cf chapitre 8). Le code Jimple est construit à partir du bytecode, les opérations complexes y sont donc décomposées. Par exemple, une expression Java de la forme `x=y.f`$_1$`.f`$_2$, lors de la compilation en bytecode, est séparée en plusieurs instructions GETFIELD successives. Le programme Jimple correspondant contiendra ainsi `t=y.f`$_1$ `; x=t.f`$_2$. La figure 6.3 page 87 illustre cette différence entre les langages.

Le langage est cependant simplifié ici afin de faciliter le discours, nous avons par exemple supprimé les types de données numériques, les exceptions, ainsi qu'un certain nombre de caractéristiques du langage d'origine, comme la présence de méthodes natives, la réflexivité et l'introspection, ou le chargement dynamique de classes. Pour des raisons de lisibilité, le code des exemples sera donné dans une syntaxe proche du Java, notamment en ce qui concerne les structures de contrôle `while` et `for`. Le langage étudié ne comporte en réalité que des sauts conditionnels, car il est reconstruit par SOOT à partir du bytecode.

6.1 Notations mathématiques

$X = \{x_0, x_1, \ldots, x_k\}$ représente l'ensemble des éléments $x_0, x_1, \ldots, x_k$ supposés distincts, et $\varnothing$ représente l'ensemble vide. La notation $\mathcal{P}(X)$ représente l'ensemble des parties de X, c'est à dire $\mathcal{P}(X) = \{Y | Y \subseteq X\}$. Le produit cartésien $X \times Y$ est l'ensemble des paires $\langle x, y \rangle$ telles que $x \in X$ et $y \in Y$.

L'ensemble $X^\star$ est l'ensemble des séquences finies ou infinies d'éléments de X. Une liste (finie) sera ainsi notée $[x_o, x_1, \ldots, x_k]$, et une suite (infinie) $[x_i]_{i \in \mathbb{N}}$. La notation $l(i)$ désigne l'élément de rang i dans la liste l. La liste $x :: l$ est la liste obtenue en ajoutant x en tête de la liste l, et la liste vide sera notée $[\,]$.

La notation $A \longrightarrow B$ désigne l'ensemble des fonctions partielles de l'ensemble A vers l'ensemble B. La fonction $f = \{a_i \mapsto b_i\}_{i \in I}$ est la fonction partielle f telle que $\forall i \in I, f(a_i) = b_i$, et f n'est pas définie ailleurs. En particulier, la fonction nulle $\{\}$ n'est définie pour aucun élément de A. Le

domaine de f, c'est à dire l'ensemble des éléments de A pour lesquels $f \in A \longrightarrow B$ est définie, est noté $dom(f)$. Pour une fonction partielle $f \in A \longrightarrow B$, et deux éléments $a \in A, b \in B$, la fonction $g = f[a \mapsto b]$ est la fonction partielle de A vers B pour laquelle $g(a) = b$, et $g(x) = f(x)$ pour les autres éléments $x \in dom(f) \setminus \{a\}$.

6.2 Syntaxe du langage

Structure Un programme est constitué d'un ensemble fini de classes $Class$, et d'un ensemble fini de méthodes $Method$. Chaque classe $C \in Class$ est dotée d'un certain ensemble de méthodes $methods(C) \subseteq Method$ et de champs $fields(C) \subseteq Fields$. Certains de ces champs sont dits *statiques*, ils se rapporteront donc à une valeur globale, alors que les champs d'instance auront une valeur différenciée pour chaque objet.

Chaque méthode possède une liste de variables locales $variables(m) = [v_0, \ldots, v_{n_v-1}]$, parmi lesquelles on distingue ses paramètres formels $parameters(m) = [p_0, \ldots, p_{n_p-1}]$, et son corps est constitué d'une liste d'instructions. Le premier paramètre p_0 est parfois noté `this`, par analogie avec la syntaxe du langage source, et on utilisera souvent des noms de variables explicites par souci de lisibilité. De plus, les variables de chaque méthode sont supposées différentes : la variable v_0 d'une méthode m_1 n'est pas la même que la variable v_0 d'une méthode m_2, etc.

Dans chaque classe C, chaque méthode de $methods(C)$ porte un nom $n \in MethodName$ unique. Plusieurs méthodes de classes différentes peuvent cependant porter un nom identique, en cas de redéfinition. Lorsqu'une classe C' hérite d'une classe C, elle peut ainsi *redéfinir* une certaine méthode de C en déclarant une méthode de même nom. Pour simplifier, nous utilisons ici le terme de *nom* pour désigner la notion de *signature* d'une méthode Java. Par exemple, dans la classe `java.io.PrintStream`, on distinguera comme deux noms différents `println:(Ljava/lang/Object;)V` et `println:(Ljava/lang/String;)V`. En effet, si dans le source Java ces deux méthodes semblent s'appeler toutes les deux `println`, il ne s'agit que de sucre syntaxique et le type des arguments permet de les différencier de façon unique. Par contre, la méthode `hashCode:()I` de la classe `Object` sera typiquement *redéfinie* par des méthodes de même nom dans d'autres classes héritant de `Object`.

Les notation utilisées pour représenter le programme sont résumées dans la figure 6.1.

$$\begin{array}{rcl} C & \in & Class = \{C_0, C_1, ...\} \\ m & \in & Method = \{m_0, m_1, ...\} \\ n & \in & MethodName = \{\text{"main"}, \text{"<init>"}, ...\} \\ f & \in & Field = \{f_0, f_1, ...\} \cup \{[*]\} \\ v, p, v_{ret} & \in & Variable = \{v_0, v_1, ...\} \cup \{v_{ret}\} \cup \{p_0, p_1, ...\} \\ lb & \in & Label = Method \times \mathbb{N} \end{array}$$

FIG. 6.1 – Notations utilisées pour les programmes.

Instructions Les différents types d'instructions qui peuvent composer une méthode sont donnés dans la figure 6.2. La syntaxe adoptée est proche de celle du langage source Java, mais le jeu d'instructions est similaire à celui du bytecode : les expressions complexes ont été décomposées, les structures de contrôle de haut niveau ont été traduites en sauts conditionnels vers des labels explicites. De plus, on ne s'intéresse pas aux types de données scalaires (valeurs numériques, booléennes), ni aux instructions associées. Les programmes présentés dans les exemples comporteront souvent des valeurs scalaires, mais elles seront ignorées par l'analyse. En particulier, on utilisera parfois la notation $[*]$ pour désigner l'ensemble des cases d'un tableau, que l'analyse ne différencie pas entre elles. À cette particularité près, les tableaux seront essentiellement considérés comme des objets.

Type d'instruction	Syntaxe	Signification
COPY	$v_1 = v_2$	copie une variable locale dans une autre
NEW	$v =$ `new` C	alloue un nouvel objet de classe C ; tous les champs du nouvel objet sont initialisés à `null`
NEWARRAY	$v =$ `new` $C[k]$	alloue un nouveau tableau de références à des objets de classe C ; toutes les cases du tableau sont initialisées à `null`
NULLIFY	$v =$ `null`	affecte la valeur `null` à la variable v ; représente toutes les affectations à une valeur constante
PUTFIELD	$v_1.f = v_2$	stocke la valeur de v_2 dans le champ f de l'objet pointé par la variable v_1
PUTSTATIC	$C.f = v$	stocke la valeur de v dans le champ statique $C.f$
GETFIELD	$v_1 = v_2.f$	affecte à v_1 la valeur du champ f de l'objet pointé par la variable v_2
GETSTATIC	$v = C.f$	affecte à v la valeur du champ statique $C.f$
IF	`if(`$\cdots$`) goto` i	saut conditionnel vers le label i de la méthode courante ; on ne s'intéresse pas à la condition elle-même, qui n'a pas d'effet sur la mémoire
INVOKE	$v_r = v_0.n(v_1, ..., v_j)$	appelle la méthode de nom n de l'objet v_0, avec comme paramètres formels les arguments $v_0, ..., v_j$
RETURN	`return` v_{ret}	retourne à la méthode appelante avec comme résultat la valeur de v_{ret}
NOP	`nop`	aucun effet ; représente toutes les instructions qui ne manipulent pas de pointeurs

FIG. 6.2 – Les instructions du langage.

Exécution L'exécution du programme commence par la première instruction d'une méthode particulière $m_{main} \in Method$. Chaque méthode possède une liste d'instructions $body(m) = [s_0, s_1..., s_{n_s-1}]$. Chacune d'entre elles est identifiée par un *label* de la forme $\langle m, i \rangle$, où $i \in \mathbb{N}$ est le rang dans la liste des instructions. Nous utiliserons ainsi les notations $labels(m) \subseteq m \times \mathbb{N}$ pour désigner l'ensemble des labels des instructions de m, et $instr(lb)$ pour désigner l'instruction située au label lb. Par commodité, si un label lb est de la forme $\langle m, i \rangle$, la notation $next(lb)$ désignera le label $\langle m, i + 1 \rangle$.

La sémantique des instructions, donnée ci-dessous de façon informelle, est conforme à celle de Java. Une sémantique opérationnelle de ce langage est donnée à la section 7.2.1 page 100.

La plupart des instructions sont exécutées séquentiellement, dans l'ordre des labels. L'instruction IF permet de modifier cet ordre en renvoyant le flot de contrôle vers une autre instruction de la même méthode. Comme on ne s'intéresse qu'aux aspects liés à la mémoire, la condition proprement dite n'est pas significative. En effet, dans le bytecode Java dont est tiré notre langage, les sauts conditionnels n'ont pas d'effets de bord.

L'instruction d'appel représente dans le cas général les appels de méthodes virtuelles Java : le flot de contrôle est transféré à la méthode m de nom n dans la classe C de l'objet v_0. Grâce à l'héritage, plusieurs classes peuvent déclarer des méthodes de même nom, et la méthode effectivement appelée n'est déterminée qu'au moment de l'exécution. On parle de *résolution dynamique* (*dynamic dispatch*) pour décrire ce mécanisme. Le passage de paramètres se fait par valeur : dans la méthode appelée, les paramètres sont initialisés avec la valeur des arguments au moment de l'appel. On suppose que toutes les méthodes de nom n ont les mêmes paramètres formels (ce qui est une simplification de la notion de *signature de méthode* dans la machine virtuelle Java). Lorsque la méthode appelée termine, elle exécute l'instruction `return` v_{ret}, ce qui renvoie le flot de contrôle dans la méthode appelante et met à jour la valeur de la variable

résultat. Pour simplifier l'analyse, on suppose que seule la variable v_{ret} peut être utilisée comme opérande de l'instruction `return`. Cela ne restreint pas l'expressivité du langage, car on peut écrire l'équivalent de `return` v en deux instructions v_{ret} `=` v `;` `return` v_{ret}.

Graphe de flot de contrôle Le graphe de flot de contrôle de chaque méthode est assez simple à construire. Chaque instruction en constitue un sommet, et une arête relie l'instruction $instr(lb)$ à son successeur $instr(next(lb))$. L'instruction de saut conditionnel IF constitue un point de branchement, et une arête supplémentaire relie chaque IF à l'instruction située au label de destination $\langle m, i \rangle$.

La figure 6.3 donne un exemple de graphe de flot de contrôle pour un programme simple. Elle illustre également les différentes représentations intermédiaires : la figure (a) présente le programme d'origine, en Java. La figure (b) montre le bytecode généré par le compilateur Java. La figure (c) présente la représentation intermédiaire sur laquelle nous allons travailler, le Jimple. Le graphe de flot de contrôle (d) est construit à partir du Jimple. Chaque instruction est suivie d'une arête menant à son successeur, et les instructions de branchement ont une arête supplémentaire vers la destination du saut.

6.3 Analyse de forme locale

6.3.1 Motivation

Dans ce travail on s'intéresse à la mémoire dynamique, et donc en particulier aux relations de référencement entre les différents objets de la mémoire, via des pointeurs. Ces notions se formulent naturellement en termes de graphes : chemins, accessibilité, connexité, présence de cycles, etc. De façon à faciliter le raisonnement, et la formulation des analyses statiques, nous proposons donc de ne pas se baser sur la syntaxe textuelle du programme, mais d'adopter une autre représentation intermédiaire qui colle de plus près à ces notions. La transformation présentée dans cette section est une sorte d'analyse de forme *générique*, dans le sens où elle n'est pas orientée vers une analyse statique en particulier. Il s'agit au contraire de *reformuler* le programme sous forme de graphes.

Cette approche est similaire à celle de Sălcianu et Rinard, qui pratiquent sur le programme une analyse de pointeurs sophistiquée pour construire un graphe par méthode. Ces graphes sont ensuite exploités par d'autres algorithmes, par exemple pour obtenir une analyse d'échappement [SR01]. Des travaux précédents menés à Verimag [GNYZ04] utilisaient ces graphes comme base d'une synthèse de régions. La limitation principale de cette approche est la grande complexité de l'analyse de pointeurs de Sălcianu et Rinard, qui implique des temps d'analyse très importants. Notre objectif est au contraire de proposer une analyse statique qui s'intègre dans le cycle de développement sans faire obstacle au programmeur, elle doit donc s'exécuter très efficacement. L'analyse de forme locale présentée ici est en pratique beaucoup plus rapide que l'analyse de Sălcianu et Rinard, au prix d'une précision moins importante. Une comparaison plus détaillée de ces deux approches est donnée à la section 6.5.

6.3.2 Définitions

Pour chaque méthode m, l'analyse va construire un graphe de forme locale $LPTG(m)$ (pour *local points-to graph*) représentant les objets que la méthode m manipule à l'exécution. Un exemple de programme et les graphes de forme locale de ses méthodes sont donnés dans la figure 6.4. Au cours de cette section, nous allons détailler les règles de constructions de ces graphes.

L'objectif de cette analyse est de produire, le plus efficacement possible, une représentation de la portion du tas que m manie directement. En particulier, elle construit le graphe $LPTG(m)$ en se basant uniquement sur les variables et les instructions de m. Les interactions entre les

```
public RefObject makelist(int x)
{
  RefObject list=null;
  for(int i=0;i<x;i++)
  {
    RefObject head=new RefObject();
    head.f = list;
    list = head;
  }
  return list;
}
```

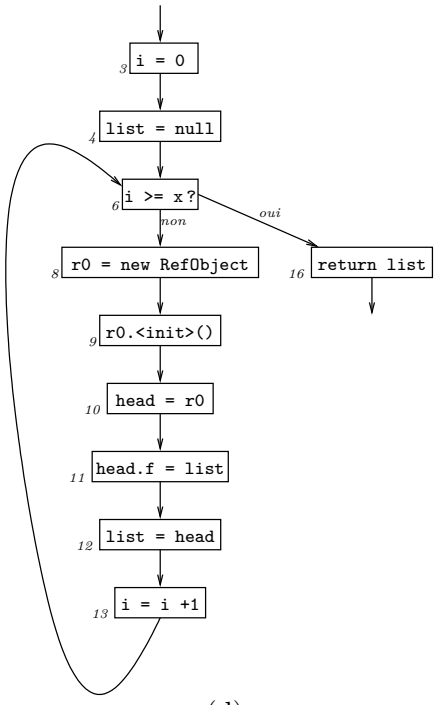
(a)

```
public RefObject makelist(int);
Code:
  0:  aconst_null
  1:  astore_2
  2:  iconst_0
  3:  istore_3
  4:  iload_3
  5:  iload_1
  6:  if_icmpge 33
  9:  new #2; //class RefObject
  12: dup
  13: invokespecial #3; //Method RefObject."<init>":()V
  16: astore 4
  18: aload 4
  20: aload_2
  21: putfield #8; //Field RefObject.f:Ljava/lang/Object;
  24: aload 4
  26: astore_2
  27: iinc 3, 1
  30: goto 4
  33: aload_2
  34: areturn
```

(b)

```
1  public RefObject makelist(int x)
2  {
3    int i=0;
4    RefObject list=null, head, r0;
5
6    if i >= x goto 16;
7
8    r0 = new RefObject;
9    r0.<init>();
10   head = r0;
11   head.f = list;
12   list = head;
13   i = i + 1;
14   goto 6;
15
16   return list;
17 }
```

(c)



(d)

FIG. 6.3 – Un programme de construction de liste chaînée, en Java (a), en bytecode (b), et en Jimple (c). Le graphe de flot de contrôle (d) est construit à partir du Jimple.

différentes méthodes ne seront pas incorporées dans cette première analyse, mais seront considérées séparément dans la section 6.4.

Sommets Pour chaque méthode m du programme, le graphe $LPTG(m)$ contient deux types de sommets, les sommet «objets» et les sommets «noms». Les sommets «noms» représentent les points d'entrée que m utilise pour accéder aux objets du tas : variables locales v, paramètres formels p, et noms de classe C seront notés par exemple $\boxed{v}$, $\boxed{p}$, ou $\boxed{C}$.

Un sommet «objet» représente au contraire un ou plusieurs objets manipulés par m, par exemple un objet alloué par m, ou obtenu comme paramètre formel, etc. Le graphe de forme locale est construit par une analyse statique, il faut donc choisir une manière de représenter statiquement les objets manipulés à l'exécution. Comme de nombreux travaux du domaine [JM82, CWZ90, CR04, Sal06] nous adoptons une modélisation basée sur les sites d'allocation : tous les objets issus d'un même site d'allocation étiqueté lb seront abstraits en un seul sommet (lb). Les divers sommets «objets» seront ainsi étiquetés de façon à décrire les objets correspondants, même si leurs étiquettes ne sont pas restreintes aux labels identifiant les instructions du programme. On notera par exemple (C) pour représenter la classe C, et (p_i) pour représenter l'objet pointé par un paramètre formel p_i de la méthode.

Arêtes Les arêtes du graphe $LPTG(m)$ représentent les relations de référencement entre les différents sommets. Une arête entre un sommet «nom» et un sommet «objet», de la forme $\boxed{} \rightarrow ()$ signifie intuitivement «cette variable peut pointer sur cet objet». D'autre part, une arête entre deux sommets «objets», de la forme $() \xrightarrow{f} ()$ signifie intuitivement «cet objet peut pointer sur cet autre objet par le champ f». Pour plus de précision on différenciera deux types d'arêtes entre les sommets «objets». Les arêtes simples, comme ci-dessus, représenteront les références *découvertes* par m, et on utilisera des arêtes doubles de la forme $() \xRightarrow{f} ()$ pour noter les références *créées* par m. Par exemple, dans la méthode `add` de la figure 6.4 page 92, la référence entre `this` et `this.data` est représentée par une arête simple, car elle n'est pas créée par `add`, alors que la référence de `this.data` vers `o`, créée par l'affectation de la ligne 21, est représentée par une arête double.

6.3.3 Construction du graphe

Pour chaque méthode m du programme, le graphe $LPTG(m)$ est construit par une analyse intra-procédurale insensible au flot. Le graphe est initialement vide, et les règles de construction qui ajoutent les sommets et les arêtes sont données ci-dessous de façon informelle. La figure 6.4 page 92, commentée à la section 6.3.4, donne un exemple de programme avec plusieurs méthodes et leurs graphes de forme locale.

6.3.3.1 Déclarations

Le langage que nous considérons ne comprend pas de *déclarations* explicites des variables utilisées. Les sommets «noms» du graphe représentent cependant chacune des variables et des paramètres formels utilisés dans la méthode.

Paramètres formels Le graphe $LPTG(m)$ comporte un sommet $\boxed{p_i}$ pour chacun des paramètres p_i de $parameters(m)$. Pour représenter l'objet pointé par ce paramètre, l'analyse ajoute un sommet (p_i) pointé par le sommet nom : $\boxed{p_i} \rightarrow (p_i)$.

$$\frac{p_i \in parameters(m)}{\boxed{p_i} \rightarrow (p_i) \ \in LPTG(m)}$$

Variables locales Le graphe $LPTG(m)$ comporte un sommet $\boxed{v_i}$ pour chacune des variables v_i de $variables(m) \setminus parameters(m)$. Initialement, aucune arête n'est issue de ce sommet, ce sera l'analyse des instructions où intervient v_i qui en ajoutera.

$$\frac{v_i \in variables(m) \setminus parameters(m)}{\boxed{v_i} \in LPTG(m)}$$

Variables globales De même, pour chacun des noms de classe C qui figurent dans les instructions du corps de m (notés ci-dessous $globals(m)$), l'analyse ajoute un couple $\boxed{C} \rightarrow (C)$ pour représenter le nom et l'objet correspondant.

$$\frac{C \in globals(m)}{\boxed{C} \rightarrow (C) \in LPTG(m)}$$

6.3.3.2 Instructions

L'algorithme de construction du graphe $LPTG(m)$ traite ensuite toutes les instructions du corps de m pour créer le reste du graphe. Selon le type d'instruction, l'analyse rajoute des arêtes et éventuellement de nouveaux sommets.

NEW, NEWARRAY Pour chaque allocation de la forme $lb : v = \texttt{new C}$ ou $lb : v = \texttt{new C[*]}$, l'analyse ajoute un sommet «objet» étiqueté par lb et pointé par v : $\boxed{v} \rightarrow (lb)$.

$$\frac{instr(lb) = v = \texttt{new C} \quad lb \in labels(m)}{\boxed{v} \rightarrow (lb) \in LPTG(m)} \qquad \frac{instr(lb) = v = \texttt{new C[*]} \quad lb \in labels(m)}{\boxed{v} \rightarrow (lb) \in LPTG(m)}$$

NULLIFY Pour chaque instruction de la forme $v = \texttt{null}$, l'analyse ajoute un sommet $(\texttt{null})$ et une arrête $\boxed{v} \rightarrow (\texttt{null})$. Il y a au plus un sommet $(\texttt{null})$ dans le graphe $LPTG(m)$, qui représente toutes les affectations à `null` figurant dans la méthode.

$$\frac{instr(lb) = v = \texttt{null} \quad lb \in labels(m)}{\boxed{v} \rightarrow (\texttt{null}) \in LPTG(m)}$$

COPY Pour chaque affectation de la forme $v_1 = v_2$, et pour chaque sommet objet o pointé par v_2, l'analyse ajoute une arête $\boxed{v_1} \rightarrow (o)$.

$$\frac{instr(lb) = v_1 = v_2 \quad lb \in labels(m) \quad \boxed{v_2} \rightarrow (o) \in LPTG(m)}{\boxed{v_1} \rightarrow (o) \in LPTG(m)}$$

Si le sommet $\boxed{v_2}$ avait des arêtes vers plusieurs objets o, elles sont toutes copiées. Par exemple, si le graphe comporte les arêtes $\boxed{v_2} \rightarrow (o_1) \quad (o_2)$, l'analyse de l'instruction $v_1 = v_2$ ajoutera les arêtes $\boxed{v_2} \rightarrow (o_1) \quad (o_2) \leftarrow \boxed{v_1}$.

PUTFIELD, PUTSTATIC Pour chaque instruction de la forme $x.f = v$ (où x est une variable, ou un paramètre, ou un nom de classe), l'analyse ajoute une arête double étiquetée par f entre les sommets objets pointés par x et ceux pointés par v. Par exemple, si le graphe comporte $\boxed{x} \to (o_1)$ et $\boxed{v} \to (o_2)$, l'algorithme ajoute une arête $(o_1) \overset{f}{\Rightarrow} (o_2)$ entre les deux.

$$\frac{\begin{array}{l} instr(lb) = \ x.f = v \\ lb \in labels(m) \\ \boxed{x} \to (o_1) \in LPTG(m) \\ \boxed{v} \to (o_2) \in LPTG(m) \end{array}}{(o_1) \overset{f}{\Rightarrow} (o_2) \in LPTG(m)}$$

GETFIELD, GETSTATIC Pour chaque expression de chargement $x.f$ figurant dans une instruction de la forme $v = x.f$ (où x est une variable, un paramètre, ou un nom de classe), l'analyse ajoute un sommet «objet» $(x.f)$, ainsi qu'une arête vers celui-ci pour chacun des sommets «objets» pointés par x. On obtient donc par exemple la forme suivante : $\boxed{x} \to (o) \overset{f}{\to} (x.f)$. Ensuite, pour chacun des sommets (o') tels que $(o) \overset{f}{\to} (o')$ ou $(o) \overset{f}{\Rightarrow} (o')$, y compris pour le sommet $(x.f)$, l'algorithme ajoute une arête $\boxed{v} \to (o')$ pour représenter le fait que la variable v pointe sur l'objet en question.

$$\frac{\begin{array}{l} instr(lb) = \ v = x.f \\ lb \in labels(m) \\ \boxed{x} \to (o) \in LPTG(m) \end{array}}{(o) \overset{f}{\to} (x.f) \in LPTG(m)}$$

$$\frac{\begin{array}{l} instr(lb) = \ v = x.f \\ lb \in labels(m) \\ \boxed{x} \to (o) \overset{f}{\to} (o') \in LPTG(m) \end{array}}{\boxed{v} \to (o') \in LPTG(m)} \qquad \frac{\begin{array}{l} instr(lb) = \ v = x.f \\ lb \in labels(m) \\ \boxed{x} \to (o) \overset{f}{\Rightarrow} (o') \in LPTG(m) \end{array}}{\boxed{v} \to (o') \in LPTG(m)}$$

INVOKE Pour chaque site d'appel de la forme $lb : v_r = v_0.n(v_1, ..., v_j)$, l'analyse ajoute un sommet «objet» étiqueté par lb et pointé par v_r : $\boxed{v_r} \to (lb)$. Ce sommet représente l'objet retourné par la méthode n. Comme pour l'instruction NEW, l'analyse utilise un seul sommet, étiqueté par le site d'appel, comme abstraction de tous les objets obtenus à ce site d'appel.

$$\frac{\begin{array}{l} instr(lb) = \ v_r = v_0.n(v_1, ..., v_j) \\ lb \in labels(m) \end{array}}{\boxed{v_r} \to (lb) \in LPTG(m)}$$

IF, NOP, RETURN Les autres instructions du langage n'ont pas d'effet sur la mémoire, du moins pas qui soit visible depuis m. Elles n'ont donc pas d'effet sur la construction du graphe de forme locale.

6.3.4 Exemple

La figure 6.4 page 92 présente un exemple de programme ainsi que les graphes de forme locale associés à chaque méthode. Pour plus de clarté, la syntaxe utilisée n'est pas tout à fait celle du langage présenté, mais la différence n'est pas significative pour notre analyse. En particulier, la présence des structures de contrôle n'a pas d'influence sur les graphes obtenus puisque l'algorithme

de construction est insensible au flot. De plus, l'exemple utilise des numéros de lignes en guise de *labels*, et certaines instructions portent sur des valeurs entières, afin de rendre le programme plus compréhensible.

Méthode main La méthode `main` alloue successivement une liste `list` et deux objets `o1` et `o2`, puis ajoute `o1` à la liste. Son graphe de forme locale est donc uniquement constitué des trois sommets «noms» représentant les variables locales, et des sommets objets correspondant aux sites d'allocation.

Le reste du programme est une classe `ArrayList` qui encapsule un tableau pour offrir une interface de liste. Dans une implantation réaliste, ses champs seraient privés, ainsi que la méthode `extend`, mais nous ne nous intéressons pas ici à ces aspects.

Méthode <init> Le graphe de forme locale du constructeur `<init>` est très simple. Le sommet `this` représente l'objet proprement dit, et `tmp` est une variable locale qui pointe sur le tableau alloué à la ligne 10. L'arête double `this` $\xrightarrow{\texttt{data}}$ 10 est construite par l'analyse pour modéliser l'instruction PUTFIELD `this.data=tmp` de la ligne 11. Les autres instructions sont ignorées car elles portent sur des entiers.

Méthode add Le graphe de la méthode `add` comporte deux couples `this` → `this` et `o` → `o` pour représenter les paramètres formels, ainsi qu'un sommet `t` pour représenter la variable intermédiaire `t`. L'analyse ajoute un sommet `this.data` pour représenter l'objet obtenu par l'instruction GETFIELD de ligne 20, et une arête entre ce sommet et le sommet `o` pour représenter l'affectation de la ligne 21.

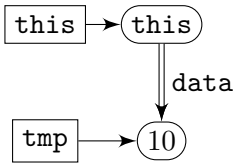
De tels graphes à la structure très simple sont obtenus en pratique pour une grande majorité des méthodes dans un programme, en particulier pour les méthodes de la bibliothèque standard. En matière d'analyse statique, la difficulté liée à un programme Java provient en effet plutôt de la combinaison de très nombreuses petites méthodes plutôt que de méthodes intrinsèquement complexes.

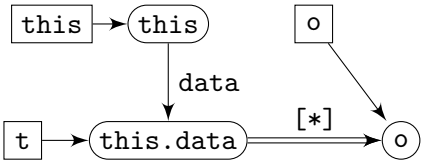
Méthode extend Le graphe de la méthode `extend` est plus intéressant. Cette méthode, appelée par la méthode `add` lorsque la liste est entièrement occupée, alloue un nouveau tableau `data` et y recopie un à un les éléments de l'ancien tableau. L'exemple comporte une structure de contrôle `for` pour plus de lisibilité, qui représente un saut conditionnel dans la syntaxe que nous avons adoptée. Le graphe $LPTG$(`extend`) contient un sommet 29 qui représente l'allocation du nouveau tableau. Les éléments `t1.[*]` de la liste sont pointés par la variable `e` avant d'être stockés dans ce tableau.

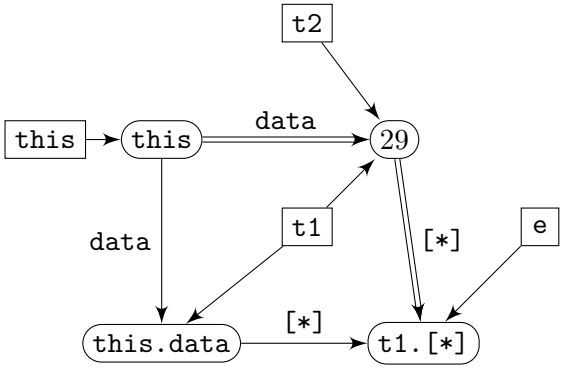
Dans l'analyse de cette méthode, l'insensibilité au flot de notre algorithme provoque une imprécision : l'arête `t1` → 29 ne représente pas une relation de pointage réelle. Sa présence est due à la règle GETFIELD, appliquée sur l'instruction `t1=this.data` *après* la construction de l'arête `this` $\xrightarrow{\texttt{data}}$ 29.

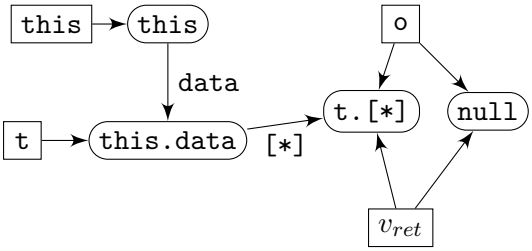
Méthode get Le graphe de la méthode `get` comporte quatre sommets «noms» pour représenter le paramètre `this` et les trois variables locales. Comme pour les autres méthodes, notre analyse ignore les variables entières et les instructions qui ne manipulent pas de pointeurs. Le sommet `this.data` représente le tableau, et le sommet `t.[*]` représente l'objet `o` obtenu à la ligne 47. À cause de l'affectation de la ligne 43, la variable `o` peut également pointer sur `null`. Les deux `return o` des lignes 44 et 48 sont des raccourcis de notation pour v_{ret}`=o ; return` v_{ret}, c'est donc la règle associée à l'instruction COPY qui s'applique pour construire les deux arcs de v_{ret} vers `t.[*]` et `null`.

```
1  class ArrayList
2  {
3    Object data[];
4    int capacity;
5    int size;
6
7    <init>(int capacity)
8    {
9      this.size=0;
10     Object[] tmp=new Object[capacity];
11     this.data=tmp;
12     this.capacity=capacity;
13   }
14
15   void add(Object o)
16   {
17     if(this.size == this.capacity)
18       this.extend();
19
20     Object[] t = this.data;
21     t[this.size] = o;
22     this.size++;
23   }
24
25   void extend()
26   {
27     this.capacity*=2;
28     Object[] t1 = this.data;
29     Object[] t2 = new Object[capacity];
30     for(int i=0;i<this.size;i++)
31     {
32       Object e=t1[i];
33       t2[i]=e;
34     }
35     this.data=t2;
36   }
37
38   Object get(int index)
39   {
40     Object o;
41     if( index >= this.size )
42     {
43       o = null;
44       return o;
45     }
46     Object [] t = this.data:
47     o = t[index];
48     return o;
49   }
50
51   main()
52   {
53     ArrayList list = new ArrayList;
54     list.<init>(10);
55
56     Object o1=new Object;
57     Object o2=new Object;
58
59     list.add(o1);
60   }
61 }
62
```









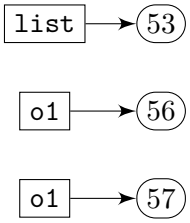


FIG. 6.4 – Un exemple de programme et les graphes de forme locale associés à chaque méthode.

6.4 Graphe d'appel détaillé

La section précédente présente une analyse qui traduit chaque méthode en un graphe représentant une abstraction des objets qu'elle manipule. Pour pouvoir combiner les informations contenues dans ces graphes, et obtenir des renseignements sur le comportement global du programme, nous allons également utiliser le graphe d'appel du programme.

Construction du graphe d'appel On oppose traditionnellement (cf section 4.1) le graphe d'appel *dynamique* au graphe d'appel *statique*. Dans les deux cas, les sommets du graphe d'appel sont les méthodes du programme, et une arête relie m à m' si m appelle m'. Un graphe d'appel dynamique peut être construit facilement en ajoutant des arêtes au fur et à mesure de l'observation d'une exécution. Cependant, comme toujours en analyse de programmes, on obtiendra seulement un sous-ensemble des comportements possibles, et une nouvelle exécution produira éventuellement un graphe différent. L'analyse statique, au contraire, nous donnera une sur-approximation de ces comportements. La construction du graphe d'appel statique d'un programme est un thème de recherche très actif, déjà évoqué dans la section 4.3.1 page 69. Nous supposerons ainsi que le graphe d'appel statique est construit à l'avance. En pratique, nous avons utilisé SPARK [LH03], le module de construction de graphe d'appel de l'infrastructure SOOT. On dispose donc d'une fonction $CallGraph$ qui nous indique, pour chaque instruction d'appel du programme étiquetée lb, un sur-ensemble $CallGraph(lb) \subseteq Method$ des méthodes qui peuvent être invoquées à l'exécution en ce point.

Graphe d'appel détaillé L'objectif de ce chapitre est de donner une représentation du programme qui permette d'exprimer des analyses statiques en se détachant le plus possible de la syntaxe textuelle du programme. Nous allons donc enrichir le graphe d'appel statique du programme avec des informations plus détaillées sur les interactions entre les méthodes.

On se donne en particulier une fonction qui nous indique, pour chaque site d'appel du programme, la correspondance entre les arguments et les paramètres formels de la méthode appelée. Pour chaque site d'appel, situé au label lb et de la forme $v_r = v_0.n(v_1, \ldots, v_j)$, on pose $arguments(lb) = \{(v_0, p_0), \ldots, (v_j, p_j), (v_r, v_{ret})\}$. Chacune des paires (v_i, p_i) représente un couple argument-paramètre dans l'appel, et la paire (v_r, v_{ret}) représente le passage de la valeur de retour. Les arguments $v_0, \ldots, v_j$ sont des variables locales de la méthode appelante. Statiquement, on ne sait pas nécessairement quelle méthode de nom n va être invoquée. Cependant toutes les méthodes de nom n ont la même liste de paramètres formels, et on suppose l'usage d'une variable unique v_{ret} comme valeur de retour, la correspondance obtenue est donc valable pour chacune d'entre elles.

Ces fonctions sont des reformulations de renseignements déjà disponibles immédiatement dans le texte du programme. Leur intérêt réside dans le fait qu'elles permettent, une fois combinées au graphe d'appel statique, d'obtenir une information exhaustive sur la manière dont les différentes méthodes du programme se passent les objets entre elles. Ce *graphe d'appel détaillé* va nous permettre de *combiner* par analyse inter-procédurale des informations obtenues à partir des graphes de forme locale et nous offre donc un cadre d'analyse statique plus propice aux analyses de pointeurs que les représentations classiques du programme, comme le code source, ou le graphe de flot de contrôle.

6.5 Discussion

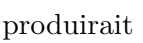
La motivation de cette analyse de forme locale est de servir de base à d'autres analyses de pointeurs. Par exemple, notre synthèse de régions est formulée dans le chapitre suivant en se basant sur ces graphes de forme locale. En effet, une analyse de pointeurs s'intéresse par nature

aux objets que le programme manipule et à leurs relations, ce qui se traduit très naturellement en termes de graphes et d'accessibilité. Chaque méthode est ici modélisée par son graphe de forme locale, et les interactions entre les méthodes sont représentées par le graphe d'appel détaillé. Il nous paraît ainsi plus naturel de manipuler le programme sous forme de graphes.

L'analyse *points-to* de Sălcianu et Rinard Une approche comparable est adoptée par Sălcianu et Rinard. Ils utilisent une même analyse de pointeurs [Sal06] comme base de plusieurs autres travaux. Les graphes obtenus permettent de pratiquer, sans avoir à revenir au texte du programme, une analyse d'échappement [WR99, SR01], une analyse déterminant les possibilités de *préallocation* statique des objets [GSR03], ainsi qu'une analyse de pureté [SR05] (une méthode est dite *pure* si elle n'a pas d'effet visible sur le tas). Dans chacun de ces travaux, l'analyse se passe en deux phases. Dans un premier temps, une analyse de forme inter-procédurale [Sal06] construit un *points-to graph* pour chaque méthode. Ce graphe, basé sur la modélisation par le site d'allocation, représente une abstraction de tous les objets manipulés par cette méthode ainsi que par toutes les méthodes qu'elle appelle. Les sommets correspondant à des objets créés pendant l'exécution de la méthode (ou d'une méthode appelée) sont appelés «sommets intérieurs», et sont abstraits par un seul sommet représentant leur site d'allocation. Les autres objets sont abstraits par des «sommets extérieurs» représentant l'instruction qui les a rendus accessibles.

Ces graphes sont obtenus grâce à une analyse inter-procédurale de flot de données *en avant*, très précise. Cela permet d'obtenir une relation d'abstraction claire entre les graphes *points-to* et les objets manipulés à l'exécution. Cependant cette analyse est très coûteuse en pratique, et elle a besoin d'examiner le programme tout entier pour construire les graphes de chaque méthode. En effet, le graphe d'une certaine méthode m dépend très fortement de celui des méthodes m' qu'elle appelle. À peu de chose près, en pratique le graphe de m va contenir celui de chacune des m'. Cette redondance est le prix à payer pour obtenir une abstraction dont on peut prouver certaines propriétés par rapport aux objets concrets, mais elle rend l'analyse très exigeante en matière de ressources. Non seulement son temps d'exécution est considérable, puisque sa composante inter-procédurale est essentielle, mais ses besoins en espace sont également très importants, à cause de la grande redondance des graphes.

Comparaison Notre approche est quelque part similaire à celle de Sălcianu et Rinard [Sal06] dans le sens où nous adoptons une représentation des méthodes sous forme de graphe au préalable de l'analyse, mais notre représentation n'a pas du tout la même signification formelle. Afin de rendre nos analyses suffisamment rapides pour être utilisables interactivement par le programmeur, nous avons opté pour une analyse plus légère et moins orientée vers une certaine interprétation. Les graphes de forme locale sont ainsi construits pour chaque méthode par une analyse syntaxique intra-procédurale et insensible au flot, contrairement à l'analyse inter-procédurale sensible au flot de Sălcianu et Rinard. Les résultats obtenus sont certes moins précis, mais l'analyse est beaucoup plus rapide.

Par exemple, dans la figure 6.4 page 92, l'analyse de Sălcianu et Rinard [Sal06] produirait des graphes plus complexes que la nôtre : le graphe de `add` comporterait une copie de celui de `extend`, et le graphe de `main` contiendrait une copie de celui de `add`. Pour chaque méthode, l'information obtenue serait ainsi exhaustive, mais les résultats seraient plus coûteux à représenter. De plus, leur analyse ne produirait pas l'équivalent de l'arête 

dans la méthode `extend`. En effet, étant sensible au flot, elle produirait en chaque point de la méthode un graphe différent. A l'instruction `t1=this.data`, le champ `data` ne pointe pas encore sur le nouvel objet. Cette information revient cependant assez cher en pratique puisqu'il faut calculer environ 10 graphes différents pour la méthode `extend` si l'on veut pratiquer une analyse sensible au flot.

Pour cette raison, malgré la similarité des deux approches, il n'est pas possible de proposer une analyse inter-procédurale qui combine nos graphes de forme locale pour obtenir les graphes

de Sălcianu et Rinard. En effet, la sensibilité au flot de leur analyse les autorise à *supprimer* certains arcs dans leurs graphes, ce qu'aucun mécanisme ne permet dans notre cas.

Forme à affectation unique – SSA En pratique cependant, nous avons observé assez peu de cas dans lesquels l'insensibilité au flot de notre analyse de forme locale causait de grandes imprécisions, car la plupart du temps les méthodes sont assez courtes et assez simples. La complexité du programme provient surtout de la taille du graphe d'appel. Dans la section 6.3.3, on a formulé les règles de construction des graphes au pluriel (*tous* les sommets, etc) mais en pratique les graphes obtenus sont assez simples, et comportent rarement plus d'un sommet objet pour chaque variable locale. Nous avons également comparé les graphes avec ceux obtenus sur une variante à affectation unique du langage d'entrée [Uma06]. En effet, la mise sous forme SSA d'un programme (cf section 4.2.4) donne à une analyse insensible au flot une précision proche de celle d'une analyse sensible au flot sur le programme d'origine. Cependant, les graphes de forme locale obtenus étaient sensiblement équivalents à ceux construits sur la forme standard du langage.

Conclusion Grâce à cette représentation du programme sous forme d'une collection de graphes, nous allons pouvoir nous intéresser à des analyses plus globales, en propageant le long du graphe d'appel certaines propriétés bien choisies des graphes de forme locale. Nous obtiendrons ainsi des résultats spécifiques, que nous aurions certainement pu calculer sur les graphes *points-to* de l'analyse de Sălcianu et Rinard, mais au prix d'une analyse beaucoup plus coûteuse.

Or notre approche suppose une intégration de l'analyse statique dans le cycle de développement, les algorithmes utilisés doivent donc être rapides, de façon à ne pas gêner le travail du développeur. L'analyse de forme locale présentée dans ce chapitre est très simple, et s'exécute très efficacement (cf section 8.3). Bien que n'ayant pas de signification formelle car construits par un algorithme intra-procédural, les graphes obtenus peuvent déjà être intéressants pour le développeur, car ils constituent une autre manière de visualiser son programme.

Chapitre 7

Analyse de régions

Dans ce chapitre, nous allons détailler les divers aspects de notre approche présentée dans le chapitre 5. Pour profiter des coûts temporels prévisibles du modèle mémoire en régions, nous proposons de grouper chaque structure de données dans une même région. Pour automatiser ce placement, nous allons donc présenter une analyse statique calculant une sur-approximation de la connectivité des objets. Lors de l'exécution, notre stratégie d'allocation utilisera les résultats de cette analyse pour placer les objets de telle manière que des objets connectés se trouvent toujours dans une même région.

La section 7.1 présente en détail l'analyse d'interférence de pointeurs. La section 7.2 détaille la politique de placement des objets, de création et de destruction des régions. La section 7.3 s'intéresse à montrer que cette politique a un comportement correct, c'est à dire ne détruit pas de région contenant des objets vivants. La section 7.4 présente notre analyse d'anticipation des régions, qui cherche à déterminer si le gestionnaire mémoire va effectivement recycler l'espace ou risque de placer trop d'objets dans une même région. Enfin, la section 7.5 présente quelques extensions de notre approche.

7.1 Analyse d'interférence de pointeurs

L'objectif de l'analyse d'interférence de pointeurs est de déterminer statiquement quelles seront les connexions entre les objets (cf section 3.2.1). Il s'agit donc d'obtenir une sur-approximation de la relation de connexion faible $\leftrightarrow$:

Définition 7.1 (objets connectés). Deux objets o_1 et o_2 du tas sont dits *connectés* s'ils appartiennent à une même composante faiblement connexe dans le tas. On notera alors $o_1 \leftrightarrow o_2$.

Nous appellerons cette approximation statique *interférence*, et nous noterons $v_1 \sim v_2$ l'interférence entre v_1 et v_2. Comme nous l'avons vu dans le chapitre 5, notre analyse va calculer cette approximation sous la forme d'une relation d'équivalence $\sim$ entre les variables du programme. Pour être plus précis, il s'agit de calculer, dans chaque méthode, une partition des variables locales de cette méthode. La relation $\sim$ peut donc être vue comme l'ensemble des relations $\sim_m$ pour toutes les méthodes du programme.

7.1.1 Hypothèse de séparation des paramètres

Pour des raisons de performances, notre analyse est insensible au contexte, c'est à dire que chacune des méthodes est analysée sans tenir compte de ses contextes d'appels. Cependant, pour que l'analyse soit correcte, les résultats doivent être valables quelle que soit la façon dont la méthode est appelée. Le *pire cas* est un contexte d'appel dans lequel tous les arguments seraient connectés, et donc a priori la seule hypothèse *correcte* sur les paramètres est de supposer $p \sim p'$ pour chaque couple de paramètres formels de chaque méthode.

Cette hypothèse est cependant très pessimiste, et conduirait probablement l'analyse à conclure que tous les objets risquent d'être connectés, et doivent donc être placés dans la même région. Ce résultat, bien que correct, ne serait pas intéressant en termes de gestion mémoire, puisqu'aucun objet ne serait jamais désalloué.

Nous adoptons plutôt une hypothèse de *séparation maximale* des paramètres (*maximal unaliasing* [SR01]). L'analyse est ainsi menée en supposant que les paramètres formels de la méthode pointent sur des objets non connectés, et calcule un résultat correct *aux paramètres formels près*. La relation $\sim$ construite par notre analyse n'est donc pas une sur-approximation de la relation $\leftrightarrow$. Cependant, comme nous le verrons à la section 7.3, la stratégie d'allocation que nous proposons prend en compte cette caractéristique lors du groupement des structures de données en région, et garantit bien la propriété que deux objets connectés sont placés dans la même région.

7.1.2 Règles d'analyse

L'analyse d'interférence de pointeurs se passe en deux phases. Dans un premier temps, une analyse intra-procédurale examine les graphes de forme locale pour y détecter les interférences. Dans un second temps, une analyse inter-procédurale propage les interférences depuis les méthodes appelées vers les méthodes appelantes.

7.1.2.1 Analyse intra-procédurale

L'analyse intra-procédurale *résume* les graphes de forme locale pour ne garder que les informations concernant l'interférence. Dans chaque méthode m, elle construit une partition $\sim_m$ de l'ensemble des variables locales en *familles*. Initialement, chaque variable locale forme une famille. L'analyse rassemble alors toutes les variables connectées dans le graphe de forme locale en appliquant la règle suivante :

$$\frac{\boxed{v} \in LPTG(m) \quad \boxed{v'} \in LPTG(m) \quad \boxed{v} \longleftrightarrow \boxed{v'}}{v \sim_m v'}$$

La notation $\longleftrightarrow$ dans la règle ci-dessus désigne la connexion faible dans le graphe $LPTG(m)$. Deux sommets du graphe sont dits faiblement connectés s'il existe un chemin non orienté de l'un à l'autre.

Par exemple, dans le programme de la figure 6.4 page 92, les familles ainsi obtenues sont :

`this` $\sim_{\texttt{<init>}}$ `tmp`
`this` $\sim_{\texttt{add}}$ `t` $\sim_{\texttt{add}}$ `o`
`this` $\sim_{\texttt{extend}}$ `t1` $\sim_{\texttt{extend}}$ `t2` $\sim_{\texttt{extend}}$ `o`
`this` $\sim_{\texttt{get}}$ `t` $\sim_{\texttt{get}}$ `o` $\sim_{\texttt{get}}$ v_{ret}
`list` (dans `main`)
`o1` (dans `main`)
`o2` (dans `main`)

Localement, l'analyse a correctement identifié les variables qui pointent dans la même structure de données. En effet, dans les méthodes de `Arraylist`, toutes les variables locales et les paramètres seront vraiment en interférence à l'exécution, ce qui est reflété par les graphes de forme locale qui forment à chaque fois une seule composante faiblement connexe.

Par contre, dans la méthode `main`, les variables locales ne semblent pas pour l'instant interférer.

7.1.2.2 Analyse inter-procédurale

L'analyse inter-procédurale *propage* ensuite les informations obtenues dans la première phase, en appliquant à chaque site d'appel étiqueté lb la règle suivante :

$$\frac{\begin{array}{c} m' \in CallGraph(lb) \\ lb \in labels(m) \\ \{(v,p),(v',p')\} \subseteq arguments(lb) \\ p \sim_{m'} p' \end{array}}{v \sim_m v'}$$

Il faut noter que l'ensemble $arguments(lb)$ contient également une paire (v_r, v_{ret}), et donc une interférence dans la méthode appelée entre un paramètre formel et la valeur de retour sera bien propagée dans la méthode appelante sous la forme d'une interférence entre l'argument correspondant et la variable résultat.

Par exemple, dans le programme de la figure 6.4, rappelé en partie ci-dessous, cette règle s'applique à l'appel de la ligne 59, où la méthode `main` appelle la méthode `add` avec pour arguments `list` et `o1`. Dans la méthode `add`, l'analyse sait que `this` et `o` interfèrent, et elle en déduit $\texttt{list} \sim_{\texttt{main}} \texttt{o1}$.

```
51  main()
52  {
53    ArrayList list = new ArrayList;
54    list.<init>(10);
55
56    Object o1=new Object;
57    Object o2=new Object;
58
59    list.add(o1);
60  }
```

7.1.2.3 Terminaison

Ce schéma de règles (cf section 4.2.3.3) décrit notre analyse d'interférence de pointeurs par la construction d'un ensemble de contraintes sur la forme de la relation $\sim_m$. Cet ensemble de contraintes définit une équation qui sera résolue par un calcul itératif de point fixe. Le treillis dans lequel la propriété considérée est calculée est le produit cartésien, méthode par méthode, de l'ensemble des partitions des variables de m :

$$\bigtimes_{m \in Method} \prod(variables(m))$$

On munit l'ensemble des partitions des variables $\prod(variables(m))$ de la relation d'ordre «est plus fine que». Cette relation d'ordre se définit ainsi : si π_1 et π_2 sont des partitions d'un ensemble E, on dit que π_1 *est plus fine que* π_2 lorsque toute partie A_i de π_1 est incluse dans une certaine partie B_j de π_2. L'ensemble $\prod(variables(m))$ ainsi ordonné est un treillis complet, et donc le produit cartésien pour toutes les méthodes m de ces ensembles, ordonné par la relation induite composante par composante, est également un treillis complet.

Avec ces définitions, les règles d'analyse que nous proposons forment bien une fonction monotone croissante, ce qui montre la terminaison de l'algorithme de calcul itératif. En pratique, du fait du nombre de variables locales dans chaque méthode, la hauteur du treillis est assez réduite et l'algorithme converge très rapidement (cf section 8.3).

7.2 Stratégie d'allocation en régions

Pour décrire plus rigoureusement la stratégie d'allocation présentée dans le chapitre 5, nous allons d'abord nous donner une sémantique du langage considéré. Nous présentons ensuite les

modifications de cette sémantique qu'implique l'allocation en régions, ainsi que les règles de création et de destruction de régions, et la politique de placement.

7.2.1 Sémantique du langage

7.2.1.1 État du programme

À chaque instant de l'exécution, l'état du programme est caractérisé par l'état de la pile et la configuration du tas. On note cet état par un triplet $\sigma = (S, H, TY) \in \Sigma$, dans lequel $S \in Stack$ représente la pile d'exécution, $H \in Heap$ représente les références entre les objets dans le tas, et $TY \in Type$ représente le type de ces objets. Ces notations sont résumées dans la figure 7.2, en utilisant les mêmes conventions pour représenter le programme que dans la figure 6.1 page 84.

La pile La pile d'exécution $S \in Stack$ est une liste de cadres de pile, ou *stack frames* :

$$S = [\langle V_{n-1}, lb_{n-1}\rangle, \langle V_{n-2}, lb_{n-2}\rangle, \ldots, \langle V_0, lb_0\rangle]$$

Chaque cadre de pile $SF = \langle V, lb\rangle \in StackFrame$ est une paire formée d'une d'interprétation des variables $V : Variable \longrightarrow Object$, et d'un *label* qui indique le pointeur d'instruction courant. La fonction V est une fonction partielle : elle n'a de sens que pour les variables locales figurant dans $variables(m)$, où m est la méthode du label lb.

$$\begin{array}{rcl}
\sigma & \in & \Sigma = Stack \times Heap \times Type \\
o & \in & Object = \{o_{null}, o_0, o_1, \cdots\} \\
H & \in & Heap = Object \longrightarrow Field \longrightarrow Object \\
S & \in & Stack = StackFrame^\star \\
SF & \in & StackFrame = Value \times Label \\
V & \in & Value = Variable \longrightarrow Object \\
TY & \in & Type = Object \longrightarrow Class
\end{array}$$

FIG. 7.2 – Notations utilisées pour représenter l'état du programme à l'exécution.

Les objets du tas Lors de l'exécution, l'instruction `new` provoque l'allocation d'un nouvel objet. Comme le programme peut ne pas terminer, il peut créer un nombre infini d'objets. Dans la machine virtuelle, les contraintes physiques liées à la mémoire amèneront probablement le système à réutiliser l'espace au cours du temps, mais en termes de sémantique chaque objet est distinct des autres. L'ensemble $Object = \{o_{null}, o_0, o_1, \ldots\}$ contient ainsi une infinité d'éléments. Chaque allocation retournera un nouvel objet $o \in Object$ différent de tous ceux alloués jusqu'ici. L'objet particulier o_{null} sera utilisé pour représenter les pointeurs nuls.

Le tas $H : Object \longrightarrow Field \longrightarrow Object$ associe à chaque objet o_1 et à chaque champ f, un objet $o_2 = H(o_1)(f)$. Il s'agit d'une double fonction partielle : évaluer $H(o)$ n'a pas de sens si l'objet o n'est pas encore alloué, et $H(o)(f)$ n'a pas de sens si o ne possède pas de champ f. Considérer H comme une fonction permet de mettre l'accent sur le fait qu'un champ d'un objet ne peut pointer que sur un seul un objet à la fois. Cependant, nous verrons parfois le tas comme un graphe étiqueté, en notant $\langle o_1, f, o_2\rangle \in H$ pour $H(o_1)(f) = o_2$.

Enfin, la fonction de typage $TY : Object \longrightarrow Class$ indique la classe de chaque objet. Il s'agit également d'une fonction partielle, qui n'est pas définie pour les objets non encore alloués.

7.2.1.2 Exécution

Une trace d'exécution $\mathcal{T}$ est une séquence potentiellement infinie d'états. $\mathcal{T} = [\sigma_k]_{k\in\mathbb{N}}$ commence par l'état initial $\sigma_0 = ([\langle\{\}, \langle m_{main}, 0\rangle\rangle], \{\}, \{\})$, et se poursuit grâce à $\Rightarrow$ qui est la

relation de transition définie par la sémantique : $\sigma_k \Rightarrow \sigma_{k+1}$. Les types de transitions possibles en fonction des différentes instructions sont présentés ci-dessous, sous la forme d'un schéma de règles.

COPY La transition associée à l'instruction COPY modifie la valeur des variables locales dans la méthode de sommet de pile : après l'exécution de l'instruction, la variable v_1 pointe sur le même objet que la variable v_2.

$$\frac{\begin{array}{l} instr(lb) = \ v_1 \texttt{ = } v_2 \\ V' = V[v_1 \mapsto V(v_2)] \end{array}}{(\langle V, lb\rangle :: S, H, TY) \Rightarrow (\langle V', next(lb)\rangle :: S, H, TY)}$$

NULLIFY L'instruction d'affectation à `null` est similaire à la copie de variable : la variable v est affectée à o_{null}.

$$\frac{\begin{array}{l} instr(lb) = \ v \texttt{ = null} \\ V' = V[v \mapsto o_{null}] \end{array}}{(\langle V, lb\rangle :: S, H, TY) \Rightarrow (\langle V', next(lb)\rangle :: S, H, TY)}$$

NEW L'instruction `new` alloue un *nouvel* objet $o \in Object$, c'est à dire un objet différent de tous ceux rencontrés jusqu'à présent dans l'exécution. Elle affecte v à cet objet, et initialise tous ses champs à `null`. La fonction TY est aussi mise à jour pour refléter la classe du nouvel objet.

$$\frac{\begin{array}{l} instr(lb) = \ v \texttt{ = new C} \\ V' = V[v \mapsto o] \quad \text{où } o \text{ est un nouvel objet.} \\ H' = H[o \mapsto \{f \mapsto o_{null}\}_{f \in fields(\texttt{C})}] \\ TY' = TY[o \mapsto \texttt{C}] \end{array}}{(\langle V, lb\rangle :: S, H, TY) \Rightarrow (\langle V', next(lb)\rangle :: S, H', TY')}$$

GETFIELD L'instruction de chargement affecte à v_1 l'objet pointé par le champ f de l'objet pointé par v_2. On fait l'hypothèse que l'objet pointé par v_2 n'est pas `null`.

$$\frac{\begin{array}{l} instr(lb) = \ v_1 \texttt{ = } v_2.f \\ V(v_2) \neq o_{null} \\ V' = V[v_1 \mapsto H(V(v_2))(f)] \end{array}}{(\langle V, lb\rangle :: S, H, TY) \Rightarrow (\langle V', next(lb)\rangle :: S, H, TY)}$$

PUTFIELD L'instruction PUTFIELD modifie la valeur du champ f de l'objet pointé par v_1 : après l'exécution de l'instruction, ce champ pointe sur le même objet que v_2. Les autres champs de cet objet ainsi que les champs des autres objets ne sont pas modifiés.

$$\frac{\begin{array}{l} instr(lb) = \ v_1.f \texttt{ = } v_2 \\ H' = H[V(v_1) \mapsto H(V(v_1))[f \mapsto V(v_2)]] \end{array}}{(\langle V, lb\rangle :: S, H, TY) \Rightarrow (\langle V, next(lb)\rangle :: S, H', TY)}$$

IF Si la condition est vraie, l'instruction `if` modifie le flot de contrôle : au lieu de continuer avec $next(lb)$, l'exécution se poursuit par l'instruction située au label i.

$$\frac{\begin{array}{l} instr(lb) = \ \texttt{if(}\cdots\texttt{) goto } i \\ lb = \langle m, j\rangle \\ lb' = \begin{cases} \langle m, i\rangle & \text{si la condition est vraie} \\ next(lb) & \text{sinon} \end{cases} \end{array}}{(\langle V, lb\rangle :: S, H, TY) \Rightarrow (\langle V, lb'\rangle :: S, H, TY)}$$

INVOKE L'instruction d'appel crée un nouveau cadre de pile $\langle V', lb' \rangle$ dans lequel le programme va exécuter la méthode choisie. Ce choix est déterminé par la classe C de l'objet v_0 au moment de l'appel. On se donne pour cela une fonction auxiliaire *lookup* chargée de résoudre les appels virtuels, définie de la façon suivante : si la classe C comporte une méthode nommée n, alors $lookup(C, n)$ renvoie cette méthode ; sinon, elle renvoie $lookup(C', n)$ où C' est la classe dont hérite C. Le compteur ordinal du nouveau cadre de pile est donc initialisé à $\langle lookup(C, n), 0 \rangle$. Dans la méthode appelée, les paramètres formels sont initialisés avec les valeurs des arguments, c'est à dire $p_i \mapsto V(v_i)$ pour chaque i entre 0 et j.

$$\frac{\begin{array}{l} instr(lb) = \ v_r \texttt{ = } v_0.n(v_1, \cdots, v_j) \\ V' = \{p_i \mapsto V(v_i)\}_{0 \leq i \leq j} \\ C = TY(V(v_1)) \\ lb' = \langle lookup(C, n), 0 \rangle \end{array}}{(\langle V, lb \rangle :: S, H, TY) \Rightarrow (\langle V', lb' \rangle :: \langle V, lb \rangle :: S, H, TY)}$$

RETURN L'instruction `return` dépile le cadre d'exécution situé au sommet de la pile, et met à jour la variable résultat dans la méthode appelante.

$$\frac{\begin{array}{l} instr(lb) = \texttt{return } v_{ret} \\ instr(lb') = \ v_r \texttt{ = } v_0.n(v_1, \cdots, v_j) \\ V'' = V'[v_r \mapsto V(v_{ret})] \\ lb'' = next(lb') \end{array}}{(\langle V, lb \rangle :: \langle V', lb' \rangle :: S, H, TY) \Rightarrow (\langle V'', lb'' \rangle :: S, H, TY)}$$

Il se peut que la pile d'exécution ne comporte pas de cadre d'exécution $\langle V', lb' \rangle$, ce qui signifie que toutes les méthodes sont terminées. Le programme suit alors la transition suivante, et termine son exécution. La trace $\mathcal{T}$ est alors de longueur finie.

$$\frac{instr(lb) = \texttt{return } v}{([\langle V, lb \rangle], H, TY) \Rightarrow ([\,], H, TY)}$$

7.2.2 Allocation en régions

Pour décrire formellement notre politique d'allocation, nous allons enrichir la sémantique du langage : chaque objet, au moment où il est alloué, est placé dans une certaine région. D'une façon similaire à l'ensemble *Object*, on se donne donc un ensemble infini $Region = \{r_0, r_1, \ldots\}$, dans lequel on pourra choisir un nouvel élément à chaque fois qu'il faudra créer une région. On ajoute donc à l'état d'exécution σ une composante $R : Object \longrightarrow Region$ qui indique dans quelle région est placé chaque objet. De plus, chaque cadre de pile SF comporte maintenant un identifiant de région $r \in Region$ dont nous verrons l'utilité ci-dessous.

7.2.2.1 Sémantique en régions

Il nous faut maintenant définir à nouveau la sémantique de notre langage pour y faire figurer ces ajouts. Pour chaque transition $\sigma \Rightarrow \sigma'$ à l'exception de celles produites par l'instruction `new`, on définit $\sigma'.R = \sigma.R$, ce qui représente le fait que les objets ne changent pas de région une fois qu'ils ont été alloués.

Le principe de notre politique d'allocation est de grouper chaque structure de données dans une région, en utilisant les résultats de l'analyse statique, c'est à dire la relation $\sim$.

Pour placer l'objet o alloué par un site d'allocation de la forme v `= new` C, on cherche un autre membre v' de la même famille que la variable v, et on place o dans la même région $R(V(v'))$. Si $v \sim v_{ret}$, on ne peut pas faire exactement la même chose, car v_{ret} n'a pas nécessairement

déjà une valeur. C'est à ça que sert l'identifiant de région ajouté dans chaque cadre de pile : à indiquer dans quelle région placer les objets qui vont être retournés.

S'il n'existe pas de v', c'est soit parce que la variable v est seule dans sa famille, soit parce qu'aucune autre variable de la famille n'a reçu de valeur pour l'instant (ce que représente $dom(V)$), ce qui signifie que o est le premier objet d'une structure de données. On crée alors une nouvelle région r pour recevoir o.

Il faut noter que si plusieurs variables v' satisfont le critère de sélection, alors les objets correspondants sont tous déjà dans la même région (il s'agit de la propriété 7.9 (2) page 106) et le choix de la variable v' n'a pas d'impact sur la région obtenue. C'est la fonction *choose_region*, définie ci-dessous qui va ainsi se charger de décider dans quelle région placer un objet.

Définition 7.3 (choix d'une région). On définit la fonction *choose_region* qui implante notre politique d'allocation. Étant donnés une variable v, un cadre de pile $\langle V, lb, r_{ret}\rangle$, et une correspondance $R : Object \to Region$, elle choisit la région dans laquelle placer l'objet pointé par v :

$$choose\_region : \begin{cases} Variable \times StackFrame \times (Object \to Region) \longrightarrow Region \\ \\ (v, \langle V, lb, r_{ret}\rangle, R) \quad \longmapsto \quad \begin{cases} \text{si } v \sim v_{ret}\text{, alors } r_{ret} \\ \text{sinon, si } \exists\, v' \in dom(V) \text{ t.q. } v \sim v' \\ \text{alors } R(V(v')) \\ \text{sinon, une nouvelle région } r \in Region \end{cases} \end{cases}$$

NEW La transition associée à l'instruction `new` est alors la suivante :

$$\frac{\begin{array}{l} instr(lb) = \ v\texttt{ = new C} \\ V' = V[v \mapsto o] \quad \text{où } o \text{ est un nouvel objet.} \\ H' = H[o \mapsto \{f \mapsto o_{null}\}_{f \in fields(\texttt{C})}] \\ TY' = TY[o \mapsto \texttt{C}] \\ R' = R\,[o \longmapsto choose\_region(v, \langle V, lb, r\rangle, R)] \end{array}}{(\langle V, lb, r\rangle :: S, H, TY, R) \Rightarrow (\langle V', next(lb), r\rangle :: S, H', TY', R')}$$

Exemple Ce mécanisme est illustré dans la figure 7.4 (a). Dans ce programme, l'analyse d'interférence de pointeurs calcule `x` $\sim_{m1}$ `y` et `p` $\sim_{m2}$ `q`. Lors de l'exécution, la première instruction exécutée est `x = new`. Comme `y` n'a pas encore de valeur, *choose_region* crée une nouvelle région pour y placer `x`. Le programme appelle ensuite `m2`, qui alloue un second objet `q`. À ce moment-là, *choose_region* examine la famille de `q`, et constate que `p` pointe déjà sur un objet. L'objet `q` est donc alloué dans la même région $R(V(p))$.

Remarque Comme on le verra à la section 8.2.2.2, il n'est pas strictement nécessaire de procéder de façon aussi systématique. Le choix de la région peut ainsi être implanté de façon plus efficace, même si le mécanisme reste essentiellement le même.

INVOKE Le recours à la fonction *choose_region* est également nécessaire lors de l'exécution d'une instruction d'appel. En effet, il peut arriver que la méthode qu'on appelle alloue un objet et nous le rende par valeur de retour, afin qu'il soit ensuite connecté à une structure de données, comme illustré sur la figure 7.4 (b). Contrairement au premier programme, la méthode `m2` ne dispose pas d'un paramètre formel pointant dans la structure, ce qui lui aurait permis de déterminer dans quelle région placer l'objet `q`. Puisqu'il faut cependant permettre ce genre de programmes, on a ajouté à chaque cadre de pile un identifiant de région pour permettre à la

(a)

```
m1()
{
  x = new;
  m2(x);
  y = x.f ;
}
```

```
m2(p)
{
  q = new;
  p.f = q;
}
```

(b)

```
m1()
{
  x = new;
  y=m2();
  x.f = y;
}
```

```
m2()
{
  q = new;
  return q;
}
```

FIG. 7.4 – Illustration de la sémantique en régions. Dans les deux programmes, l'objet `q` doit être placé dans la région de `x`.

méthode de savoir où elle doit placer les objets qu'elle va retourner. La transition associée à l'instruction d'appel est alors la suivante :

$$\frac{\begin{array}{l} instr(lb) = \ v_r = v_0.n(v_1, \cdots, v_j) \\ V' = \{p_i \mapsto V(v_i)\}_{0 \leq i \leq j} \\ C = TY(V(v_1)) \\ lb' = \langle lookup(C, n), 0 \rangle \\ r' = choose\_region(v_r, \langle V, lb, r \rangle, R) \end{array}}{(\langle V, lb, r \rangle :: S, H, TY, R) \Rightarrow (\langle V', lb', r' \rangle :: \langle V, lb, r \rangle :: S, H, TY, R)}$$

Dans notre exemple, au moment de l'appel de `m2`, *choose_region* va choisir la région de `x` pour r'. Ainsi, lors de l'allocation de `q`, l'interférence $\mathtt{q} \sim_{m2} v_{ret}$ provoquera le choix de cette même région pour placer le nouvel objet.

7.2.3 Stratégie de destruction des régions

La politique d'allocation ci-dessus groupe les objets par régions, et crée de nouvelles régions lorsqu'une nouvelle structure de données est allouée. Notre gestionnaire mémoire doit cependant désallouer les objets à un moment ou à un autre.

Intuitivement, un objet est vivant soit parce qu'il est directement pointé par une variable locale quelque part dans la pile, soit parce qu'il est pointé par un champ d'un autre objet vivant. On se donne ainsi une définition formelle de la notion d'*objet vivant*.

Définition 7.5 (Objet vivant). Le prédicat $live : \Sigma \longrightarrow \mathcal{P}(Object)$ est défini comme le plus petit point fixe du système de contraintes suivant :

$$\frac{\sigma = (S_1 :: \langle V[v \mapsto o], lb, r \rangle :: S_2, H, TY, R)}{o \in live(\sigma)} \qquad \frac{\begin{array}{l} \sigma = (S, H, TY, R) \\ \langle o_1, f, o_2 \rangle \in H \\ o_1 \in live(\sigma) \end{array}}{o_2 \in live(\sigma)}$$

Cette définition nous permet donc de formuler la notion de *région vivante*. Une région vivante est une région qui contient au moins un objet vivant.

Définition 7.6 (Région vivante). Le prédicat $live\_regions : \Sigma \longrightarrow \mathcal{P}(Region)$ est défini comme le plus petit point fixe de :

$$\frac{o \in live(\sigma)}{R(o) \in live\_regions(\sigma)}$$

Il s'agit donc maintenant pour le gestionnaire mémoire de détruire les régions au moment où elles meurent, ou le plus tôt possible ensuite. Mais l'ensemble $live\_regions(\sigma)$ n'est pas facile à déterminer à la volée, car sa définition repose sur celle de $live(\sigma)$, qui nécessite un parcours de l'ensemble des objets.

On se donne donc une seconde définition, celle de *région active*. Une région active est une région qui contient un objet directement pointé par une variable de la pile d'exécution.

Définition 7.7 (Région active). Le prédicat $active\_regions : \Sigma \longrightarrow \mathcal{P}(Region)$ est défini comme le plus petit point fixe de :

$$\frac{\sigma = (S_1 :: \langle V[v \mapsto o], lb, r\rangle :: S_2, H, TY, R)}{R(o) \in active\_regions(\sigma)}$$

Il est plus facile pour le système de calculer $active\_regions(\sigma)$ pendant l'exécution, car il suffit d'examiner les variables du programme, par exemple grâce à des barrières en écriture (cf chapitre 3), ou par examen ponctuel de la pile. Une fois que plus aucune variable ne pointe dans une région, elle n'est plus *active*.

On se propose donc ici de détruire une région *sitôt qu'elle est devenue inactive*. Il reste cependant à prouver que cette stratégie ne va pas mener à la désallocation d'objets vivants, et c'est ce que montre la preuve présentée dans la section suivante.

7.3 Validité de la politique d'allocation

7.3.1 Détection de la mort des régions

La politique de gestion mémoire proposée dans les sections précédentes consiste en deux parties. D'une part, une politique d'allocation nous indique dans quelle région placer chaque objet au moment de son allocation, d'autre part une politique de désallocation nous indique à quel moment détruire les régions. Le gestionnaire mémoire ne doit cependant pas détruire de région contenant des objets vivants, sans quoi il mènerait le programme à une faute d'exécution. Nous allons donc prouver que la politique de placement proposée respecte bien la propriété suivante :

Propriété 7.8. Sur une trace d'exécution $\{\sigma_k\}_{k\in\mathbb{N}}$, on a pour chaque état σ_k l'égalité :

$$live\_regions(\sigma_k) = active\_regions(\sigma_k).$$

Démonstration. Au vu des définitions, l'inclusion $active\_regions(\sigma_k) \subseteq live\_regions(\sigma_k)$ est immédiate : si un objet o est pointé par une variable, il est vivant, donc sa région est aussi vivante.

Pour montrer l'inclusion réciproque, on s'intéresse donc à une région vivante $r \in live\_regions(\sigma_k)$ et on montre qu'elle est *active*. Soit $(S, H, TY, R) = \sigma_k,\ o \in Object$ tel que $o \in live(\sigma_k) \wedge R(o) = r$. Selon la définition 7.5, l'objet o soit est pointé par une variable de la pile, soit est pointé par un autre objet vivant o'.

- Dans le premier cas, par définition, r est une région active ;
- Dans le second cas, la propriété 7.9 (1) ci-dessous nous indique que $R(o') = R(o)$, et on peut donc reporter le raisonnement sur o', et ainsi de suite. Le nombre d'objets total étant fini, on peut aller de cette façon jusqu'à un certain objet $o^{(n)}$ directement pointé par une variable locale de la pile.
  Il existe donc un chemin de H passant par les objets $o^{(n)}, \ldots, o', o$, le long duquel on a l'égalité $R(o^{(n)}) = \cdots = R(o') = R(o) = r$. La région $r = R(o^{(n)})$ est donc une région active.

□

7.3.2 Regroupement des structures de données en régions

Propriété 7.9. Sur une trace d'exécution $\{\sigma_k\}_{k\in\mathbb{N}}$, on a pour chaque état $\sigma_k = (S_k, H_k, TY_k, R_k)$, et pour chaque cadre de pile $SF_k = \langle V_k, lb_k, r_k\rangle$ de S_k les propriétés suivantes :

(1)	$\forall o_1, o_2 \in \mathit{Object} \setminus o_{null}, \forall f \in \mathit{Field},$	si	$\langle o_1, f, o_2\rangle \in H_k$	alors $R_k(o_1) = R_k(o_2)$
(2)	$\forall v_1, v_2 \in \mathit{dom}(V_k),$	si	$v_1 \sim v_2$	alors $R_k(V_k(v_1)) = R_k(V_k(v_2))$
(3)	$\forall v_3 \in \mathit{dom}(V_k),$	si	$v_3 \sim v_{ret}$	alors $R_k(V_k(v_3)) = r_k$

Démonstration. Ces trois propriétés se prouvent conjointement par induction sur la longueur de la trace d'exécution. Elles sont immédiatement vérifiées par l'état initial σ_0 dans lequel le tas ne contient aucun objet, et dans lequel aucune variable n'a encore de valeur.

Soit donc un préfixe de trace $\sigma_0 \Rightarrow \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_{k-1}$ durant lequel les trois propriétés sont vérifiées, et σ_k l'état tel que $\sigma_{k-1} \Rightarrow \sigma_k$. On va vérifier que les trois propriétés sont également vraies en σ_k quelle que soit l'instruction $\mathcal{I}$ exécutée entre les deux. Soit $(S_k, H_k, TY_k, R_k) = \sigma_k$, et $(S_{k-1}, H_{k-1}, TY_{k-1}, R_{k-1}) = \sigma_{k-1}$.

COPY On suppose que $\mathcal{I}$ est une copie de variable de la forme $v = v'$.

7.9 (1) Selon la sémantique de l'instruction, $R_k = R_{k-1}$ et $H_k = H_{k-1}$. Par hypothèse d'induction la propriété 7.9 (1) est vraie en σ_{k-1}, elle est donc également vraie en σ_k.

7.9 (2) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (2) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'affectation $v = v'$. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}.
Si $v \notin \{v_1, v_2\}$, alors $V_k|_{\{v_1,v_2\}} = V_{k-1}|_{\{v_1,v_2\}}$, et la propriété reste vraie par hypothèse d'induction. Par contre, si $v = v_1$ (ou v_2, par symétrie), alors après l'affectation on a $V_k(v_1) = V_k(v')$. Or par transitivité de la relation d'équivalence $\sim$, la variable v_2 vérifie également $v_2 \sim v'$, d'où $R_{k-1}(V_{k-1}(v_2)) = R_{k-1}(V_{k-1}(v'))$.
Comme $R_k = R_{k-1}$, on a donc $R_k(V_k(v_1)) = R_k(V_k(v_2))$, et la propriété 7.9 (2) est vraie en σ_k.

7.9 (3) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (3) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'affectation $v = v'$. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}.
Si v_3 n'est pas la variable v, alors sa valeur n'est pas modifiée et la propriété reste vraie par hypothèse d'induction. Par contre, si v_3 est v, alors après l'affectation on a $V_k(v_3) = V_{k-1}(v') = V_k(v')$. Par transitivité de la relation d'équivalence $\sim$, la variable v' vérifie également $v' \sim v_{ret}$, et donc par hypothèse d'induction $R_{k-1}(V_{k-1}(v')) = r_{k-1}$.
Comme $R_k = R_{k-1}$ et $r_k = r_{k-1}$, alors $R_k(V_k(v_3)) = r_k$, et la propriété 7.9 (3) est vraie en σ_k.

NEW On suppose que $\mathcal{I}$ est une instruction d'allocation de la forme $v =$ `new C`.

7.9 (1) Selon la sémantique de l'instruction, R_k et R_{k-1} coïncident sauf pour le nouvel objet o alloué par $\mathcal{I}$. De plus, si $\langle o_1, f, o_2\rangle \in H_k$, alors $\langle o_1, f, o_2\rangle \in H_{k-1}$, et par hypothèse d'induction la propriété 7.9 (1) reste vraie en σ_k.

7.9 (2) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (2) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'allocation $v =$ `new C`. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}.

Si $v \notin \{v_1, v_2\}$, alors $V_k|_{\{v_1,v_2\}} = V_{k-1}|_{\{v_1,v_2\}}$, et la propriété reste vraie par hypothèse d'induction. Par contre, si $v = v_1$ (ou v_2, par symétrie), alors après l'allocation, v pointe sur un nouvel objet o dont la région $R_k(o)$ est déterminée par $choose\_region(v, SF_{k-1}, r_{k-1})$:

- Si $v \sim v_{ret}$, alors $R_k(o) = r_{k-1}$. Or par transitivité de $\sim$ on a également $v_2 \sim v_{ret}$. Par la propriété 7.9 (3), on a également $R_{k-1}(V_{k-1}(v_2)) = r_{k-1}$. Donc $R_k(V_k(v_1)) = R_k(V_k(v_2))$, et la propriété 7.9 (2) est vraie en σ_k.
- Sinon, alors v_2 est une variable v' valable pour $choose\_region$, et donc $R_k(o) = R_{k-1}(V_{k-1}(v_2)) = R_k(V_k(v_2))$. Il faut noter que, par hypothèse d'induction, toutes les variables de la famille de v pointent sur des objets de la même région, donc le choix de l'une ou l'autre variable n'influe pas sur la région obtenue.

7.9 (3) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (3) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'allocation v `= new C`. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}. Si v_3 n'est pas la variable v, alors sa valeur n'est pas modifiée et la propriété reste vraie par hypothèse d'induction. Par contre, si v_3 est v, alors après l'allocation, v_3 pointe sur un nouvel objet o dont la région $R_k(o)$ est déterminée par $choose\_region(v, SF_{k-1}, r_{k-1})$, où $v = v_3 \sim v_{ret}$. Donc $choose\_region$ choisit r_{k-1}.
Comme $r_k = r_{k-1}$, alors $R_k(V_k(v_3)) = r_k$, et la propriété 7.9 (3) est vraie en σ_k.

GETFIELD On suppose que $\mathcal{I}$ est une instruction de chargement de la forme $v = v'.f$.

7.9 (1) Selon la sémantique de l'instruction, $R_k = R_{k-1}$ et $H_k = H_{k-1}$. Par hypothèse d'induction la propriété 7.9 (1) est vraie en σ_{k-1}, elle est donc également vraie en σ_k.

7.9 (2) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (2) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'affectation $v = v'.f$. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}. Si $v \notin \{v_1, v_2\}$, alors $V_k|_{\{v_1,v_2\}} = V_{k-1}|_{\{v_1,v_2\}}$, et la propriété reste vraie par hypothèse d'induction. Par contre, si $v = v_1$ (ou v_2, par symétrie), alors après l'affectation, on a $V_k(v_1) = H_{k-1}(V_{k-1}(v'))(f)$. Par hypothèse d'induction (propriété 7.9 (1)), les deux objets sont dans la même région, c'est à dire $R_k(V_k(v_1)) = R_{k-1}(V_{k-1}(v'))$. Par transitivité de la relation d'équivalence $\sim$ on a également $v_2 \sim v'$, et donc par hypothèse d'induction $R_{k-1}(V_{k-1}(v_2)) = R_{k-1}(V_{k-1}(v'))$. On a donc bien $R_k(V_k(v_1)) = R_{k-1}(V_{k-1}(v_2))$.
Comme $R_k = R_{k-1}$, on a donc $R_k(V_k(v_1)) = R_k(V_k(v_2))$, et la propriété 7.9 (2) est vraie en σ_k.

7.9 (3) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (3) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'affectation $v = v'.f$. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}. Si v_3 n'est pas la variable v, alors sa valeur n'est pas modifiée et la propriété reste vraie par hypothèse d'induction. Par contre, si v_3 est v, alors après l'affectation, on a $V_k(v_3) = H_{k-1}(V_{k-1}(v'))(f)$. Par hypothèse d'induction (propriété 7.9 (1)), les deux objets sont dans la même région, c'est à dire $R(V_k(v_3)) = R_{k-1}(V_{k-1}(v'))$. Par transitivité de la relation d'équivalence $\sim$ on a également $v' \sim v_{ret}$, et donc par hypothèse d'induction $R_{k-1}(V_{k-1}(v')) = r_{k-1}$.
Comme $r_k = r_{k-1}$, alors $R_k(V_k(v_3)) = r_k$, et la propriété 7.9 (3) est vraie en σ_k.

PUTFIELD On suppose que $\mathcal{I}$ est une affectation de champ de la forme $v.f = v'$.

7.9 (1) Soit une arête $\langle o_1, f, o_2\rangle \in H_k$. Si elle existait déjà dans H_{k-1}, alors par hypothèse d'induction o_1 et o_2 sont dans la même région, $R_{k-1}(o_1) = R_{k-1}(o_2)$, et comme $R_k = R_{k-1}$ la propriété 7.9 (1) est vraie en σ_k.
Sinon, c'est que l'instruction $\mathcal{I}$ est responsable de la création de cette arête, avec $V_k(v) = o_1$ et $V_k(v') = o_2$.
La règle intra-procédurale de l'analyse d'interférence de pointeurs (cf page 98) nous assure que $v \sim v'$. En effet, une instruction de la forme $v.f = v'$ implique la présence d'une arête entre $\boxed{\text{v}}$ et $\boxed{\text{v'}}$ dans le graphe de forme locale de la méthode (cf la règle associée à PUTFIELD page 90). Par hypothèse d'induction (propriété 7.9 (2)), $R_{k-1}(V_{k-1}(v)) = R_{k-1}(V_{k-1}(v'))$. Comme $R_k = R_{k-1}$ et $V_k = V_k$ on a donc $R_k(V_k(v_1)) = R_k(V_k(v_2))$, soit,$R_k(o_1) = R_k(o_2)$, et la propriété 7.9 (2) est vraie en σ_k.

7.9 (2), 7.9 (3) L'instruction PUTFIELD ne modifie pas la valeur des variables locales, les propriétés 7.9 (2) et 7.9 (3) restent donc vraies par hypothèse d'induction de σ_{k-1} à σ_k.

IF On suppose que $\mathcal{I}$ est un saut conditionnel de la forme `if(···) goto` i.

7.9 (1) Selon la sémantique de l'instruction, $R_k = R_{k-1}$, et $H_k = H_{k-1}$. Par hypothèse d'induction la propriété 7.9 (1) est vraie en σ_{k-1}, elle est donc également vraie en σ_k.

7.9 (2), 7.9 (3) L'instruction IF ne modifie pas la valeur des variables locales, les propriétés 7.9 (2) et 7.9 (3) restent donc vraies par hypothèse d'induction de σ_{k-1} à σ_k.

INVOKE On suppose que $\mathcal{I}$ est une instruction d'appel de la forme $v_r = v_0.n(v_1, \ldots, v_j)$.

7.9 (1) Selon la sémantique de l'instruction, $R_k = R_{k-1}$, et $H_k = H_{k-1}$. Par hypothèse d'induction la propriété 7.9 (1) est vraie en σ_{k-1}, elle est donc également vraie en σ_k.

7.9 (2) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (2) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, et les seules variables à avoir une valeur dans V_k sont les paramètres formels de la méthode m appelée. Les variables v_1 et v_2 sont donc des paramètres p_1 et p_2, et on a $p_2 \sim_m p_2$. Soient donc a_1 et a_2 les arguments correspondants dans la méthode appelante m'. La règle inter-procédurale de l'analyse d'interférence de pointeurs (cf page 99) nous indique qu'ils vérifient également $a_1 \sim_{m'} a_2$. Par hypothèse d'induction, les objets correspondants sont donc dans la même région en σ_{k-1}, et le sont toujours en σ_k. On a donc $R_k(V_k(v_1)) = R_k(V_k(v_2))$, et la propriété 7.9 (2) est vraie en σ_k.

7.9 (3) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (3) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, et les seules variables à avoir une valeur dans V_k sont les paramètres formels de la méthode m appelée.
La variable v_3 est donc un certain p, et on a $p \sim_m v_{ret}$. Soit a l'argument correspondant dans la méthode m' appelante. De même que ci-dessus, on sait que dans cette dernière on a $a \sim_{m'} v_r$. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}
- Si on a également $a \sim_{m'} v_{ret}$, alors par hypothèse d'induction $R_{k-1}(V_{k-1}(a)) = r_{k-1}$. Par transitivité de la relation $\sim$, on aura également $v_r \sim_{m'} v_{ret}$, et donc au moment de la création de SF_k, la fonction *choose_region* aura choisi r_{k-1}, donc $r_k = r_{k-1}$ et on a bien $R_k(V_k(p)) = R_{k-1}(V_{k-1}(a)) = r_{k-1} = r_k$.
- Sinon, a est en tous cas une variable v' valable pour *choose_region*, et donc au moment de la création de SF_k, on a bien $r_k = R_{k-1}(V_{k-1}(a)) = R_k(V_k(p))$.

Dans les deux cas, on a $R_k(V_k(v_3)) = r_k$, et la propriété 7.9 (3) est vraie en σ_k.

RETURN On suppose que $\mathcal{I}$ est une instruction de la forme `return` v_r.

7.9 (1) Selon la sémantique de l'instruction, $R_k = R_{k-1}$, et $H_k = H_{k-1}$. Par hypothèse d'induction la propriété 7.9 (1) est vraie en σ_{k-1}, elle est donc également vraie en σ_k.

7.9 (2) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (2) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'instruction `return`. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}. Si $v_r \notin \{v_1, v_2\}$, alors leur valeur n'est pas modifiée par le `return` et la propriété reste vraie par hypothèse d'induction. Par contre, si $v_r = v_1$ (ou v_2 par symétrie), montrons que $R_k(V_k(v_r)) = R_k(V_k(v_2))$. Par hypothèse d'induction, on sait que $R_{k-1}(V_{k-1}(v_{ret})) = r_{k-1}$, il reste donc à montrer que $r_{k-1} = R(V_k(v_2))$. Au moment de l'appel correspondant, r_{k-1} a été fixée par un $r_{k-1} = \mathit{choose\_region}(v_r, \cdots, r_k)$.
- Si on a $v_2 \sim_m v_{ret}$, alors par transitivité de $\sim_m$ on a également $v_1 = v_r \sim_m v_{ret}$, et donc *choose_region* a choisi $r_{k-1} = r_k$. Comme par hypothèse d'induction, $R(V_k(v_2)) = r_k$, on a bien $r_{k-1} = R_k(V_k(v_r)) = R_k(V_k(v_1)) = R_k(V_k(v_2)) = r_k$.
- Sinon, v_2 est en tous cas une variable v' valable pour *choose_region*, qui a donc choisi $r_{k-1} = R(V_k(v_2))$.

Dans les deux cas on a $R_k(V_k(v_1)) = R_k(V_k(v_2))$, et la propriété 7.9 (2) est vraie en σ_k.

7.9 (3) Si SF_k n'est pas le cadre de sommet de pile de S_k, alors soit $i \in \mathbb{N}$ tel que $SF_k = S_k(i)$. Selon la sémantique de l'instruction, on a également $SF_k = S_{k-1}(i)$. Or par hypothèse d'induction la propriété 7.9 (3) est vraie en σ_{k-1}. Elle est donc également vraie en σ_k.
Sinon, c'est que SF_k est le cadre de sommet de pile de S_k, celui qui est modifié par l'instruction `return`. Soit $SF_{k-1} = (V_{k-1}, lb_{k-1}, r_{k-1})$ le cadre de sommet de pile de S_{k-1}. Si v_3 n'est pas la variable v_r, alors sa valeur n'est pas modifiée par le `return` et la propriété reste vraie par hypothèse d'induction. Par contre, si $v_3 = v_r$, montrons que $R(V_k(v_r)) = r_k$. Selon la sémantique de l'instruction, $V_k(v_r) = V_{k-1}(v_{ret})$, or par hypothèse d'induction, on sait que $R_{k-1}(V_{k-1}(v_{ret})) = r_{k-1}$. On doit donc encore montrer que $r_{k-1} = r_k$. Au moment de l'appel correspondant, r_{k-1} a été fixée par un $r_{k-1} = \mathit{choose\_region}(v_r, \cdots, r_k)$, qui a nécessairement choisi r_k puisque $v_r \sim v_{ret}$. On a donc $R_k(V_k(v_3)) = r_k$, et la propriété 7.9 (3) est vraie en σ_k.

□

7.4 Analyse de comportement

7.4.1 Introduction

La section précédente donne une formulation rigoureuse de notre analyse, ainsi qu'une preuve de son comportement correct : chaque structure de données se trouve bien isolée dans une région (propriété 7.9), ce qui permet de détruire cette dernière dès lors qu'aucune variable locale ne pointe directement dedans (propriété 7.8). Ainsi, le système n'a à aucun moment besoin d'effectuer un parcours de la mémoire pour déterminer quels objets désallouer, ce qui permet au programme de s'exécuter sans temps de pause imprévisible.

Cependant, comme nous l'avons noté à la section 5.3, notre politique simple de placement des objets peut mener dans certains cas à ce que nous avons appelé l'*explosion* d'une région. En effet, lorsque le programme utilise une structure de données dont les éléments ont une durée de vie très différente, les objets de courte durée de vie s'accumulent dans la même région que les autres, sans pouvoir être désalloués avant la destruction du reste de la région.

Ce phénomène est inhérent à la gestion de mémoire en régions, et se manifeste tant dans les approches automatiques, similaires à la nôtre [CR04, CCQR04], que dans les approches manuelles [BZM02]. Cependant, la grande simplicité de notre algorithme de synthèse de régions et de la politique de placement des objets associée nous permettent de le *prévoir* dès la phase d'analyse de façon à avertir le programmeur.

7.4.2 Anticipation statique des régions

La stratégie de placement des objets est basée strictement sur les familles. Au sein d'une même méthode, tous les sites d'allocation correspondant à des variables locales d'une même famille placeront leurs objets dans la même région.

Sites d'allocation d'une variable, d'une famille Soit une famille $f \subseteq variables(m)$. On définit l'ensemble $sites(f) = \bigcup_{v \in f} sites(v)$, où $sites(v)$ désigne l'ensemble des sites d'allocation sur lesquels pointe v dans son graphe de forme locale :

$$\frac{\begin{array}{l} v \in variables(m) \\ \boxed{v} \longrightarrow (lb) \in LPTG(m) \\ instr(lb) = \ v\texttt{ = new C} \end{array}}{lb \in sites(v)}$$

Lorsque plusieurs méthodes interagissent, le graphe d'appel détaillé nous permet de relier les différentes familles qui participeront à une même structure de données. On peut ainsi déterminer l'ensemble des sites d'allocation qui placeront leurs objets dans une région en parcourant alternativement le graphe d'appel détaillé et les familles.

Tribu En partant d'une famille $f \subseteq variables(m)$, on construit la *tribu* associée à f comme un graphe dont les sommets sont des variables locales du programme et dont les arcs sont de deux sortes :

- un arc «horizontal» $v \sim v'$ relie deux variables d'une même méthode si et seulement si elles sont de la même famille
- un arc «vertical» $a \longmapsto p$ relie deux variables de méthodes différentes si et seulement si elles sont argument et paramètre dans un appel : $(a, p) \in arguments(lb)$, où $a \in variables(m), p \in variables(m'), lb \in labels(m),$ et $m' \in CallGraph(lb)$.

Sites d'allocation d'une tribu Si la famille de base f ne comporte pas de paramètre formel, alors toutes les variables locales qui peuvent pointer dans la structure de données sont celles de la tribu t associée à f. On peut donc retrouver l'ensemble des sites d'allocation qui vont placer leurs objets dans la région correspondante : $sites(t) = \bigcup_{v \in t} sites(v)$

Exemple La figure 7.10 comporte deux programmes très simples destinés à illustrer la notion de tribu, et son utilisation pour détecter le syndrome d'explosion des régions. Dans les deux programmes, les familles sont identiques : `x` (dans `m1`) d'une part, et `p` $\sim_{\texttt{m2}}$ `q` d'autre part. La tribu construite sur la famille de `x` est `x` $\longmapsto$ `p` $\sim$ `q`, et l'ensemble de sites d'allocation correspondant est $\{3, 12\}$.

Dans les deux programmes de la figure 7.10, l'allocation à la ligne 3 provoque la création d'une région, qui sera choisie par l'allocation à la ligne 12. La différence entre les deux programmes est la boucle dans le programme (b), à cause de laquelle la méthode `m2` est appelée un nombre inconnu de fois, causant l'allocation d'un nombre *a priori* inconnu d'objets `q`. Ils seront tous placés dans la même région, provoquant ainsi une explosion, et occuperont toute la mémoire si la boucle est exécutée trop longtemps.

Il nous faut une manière de différencier automatiquement ces deux programmes. C'est justement la présence de la boucle qui va nous le permettre.

Analyse de fuites La construction des tribus permet de déterminer l'ensemble des sites d'allocation qui, dans un certain contexte d'exécution, vont placer leur objets dans la même région. En combinant cette information avec les boucles du graphe de flot de contrôle, on peut détecter les tribus qui vont comporter un nombre *a priori* inconnu d'objets. On augmente donc nos tribus des annotations suivantes :

(a)

```
1  m1()
2  {
3    x = new;
4    m2(x);
5  }
```

```
10 m2(p)
11 {
12   q = new;
13   p.f = q;
14 }
```

(b)

```
1  m1()
2  {
3    x = new;
4    while( ... )
5    {
6      m2(x);
7    }
8  }
```

```
10 m2(p)
11 {
12   q = new;
13   p.f = q;
14 }
```

FIG. 7.10 – Illustration de la notion de tribu.

- les variables v pour lesquelles $sites(v)$ n'est pas vide sont celles qui correspondent à l'allocation d'un objet ; on les représente donc soulignées : $\underline{v}$
- les sites d'appel situés dans une boucle[1] sont susceptibles de provoquer l'exécution d'un site d'allocation un nombre *a priori* inconnu de fois ; on les représente donc $\stackrel{loop}{\longmapsto}$
- de même, les sites d'allocation directement situés dans une boucle risquent de provoquer un nombre inconnu d'allocations ; on les représente $\underline{v}^{loop}$

Avec ces notations, on peut détecter les risques de fuite de mémoire, c'est à dire les régions susceptibles d'exploser, en examinant les tribus à la recherche des situations suivantes :

- un site d'allocation dans une boucle $\underline{v}^{loop}$
- un chemin de la forme $v_1 \cdots \stackrel{loop}{\longmapsto} \cdots \underline{v_2}$
- un cycle contenant un site d'allocation.

Le premier cas correspond à un site d'allocation situé dans une boucle de contrôle d'une méthode. Chaque exécution du corps de boucle peut provoquer l'allocation d'un objet dans la région, et donc causer une fuite. Le second cas est similaire, et correspond à un site d'allocation situé dans une méthode appelée dans une boucle, via un certain chemin d'appel, comme celui de la figure 7.10 (b) : $\mathtt{x} \stackrel{loop}{\longmapsto} \mathtt{p} \sim \underline{\mathtt{q}}$. Le troisième cas caractérise simplement un site d'allocation situé dans une méthode récursive ; un cycle dans une tribu correspond à un cycle dans le graphe d'appel du programme.

Chacune de ces situations représente un scénario dans lequel un site d'allocation risque d'être exécuté un nombre arbitraire de fois, tout en plaçant les objets alloués dans la même région. Notre outil les signale donc au programmeur, qui peut ensuite examiner le code pour comprendre la cause de l'explosion.

A l'inverse, le reste des tribus correspond à des portions du code où le nombre total d'objets de la région ne peut pas augmenter arbitrairement : s'il n'y a pas de récursivité, et qu'aucun site d'allocation n'est situé dans une boucle (modulo les appels), alors on peut imaginer *déplier* le graphe de la tribu en un arbre, dans lequel chacun des sites d'allocation ne serait exécuté qu'au plus une fois, comme par exemple dans la figure 7.10 (a).

Interaction avec le programmeur Nous avons choisi de faire remonter au programmeur un avertissement lorsque l'outil d'analyse détecte une tribu pour laquelle un nombre *a priori* inconnu d'objets risquent de s'accumuler dans la même région. Par exemple, dans le programme de la figure 7.10 (b), l'outil affiche le message suivant :

[1] Une instruction est située dans une boucle si, dans le graphe de flot de contrôle de la méthode, elle est accessible à partir d'elle-même.

```
example.java:12: Possible memory leak
calling context: example.java:6:    m2(x)
  q = new ;
      ^
```

Nous n'avons pas cherché à analyser automatiquement plus en détail les situations en question, estimant que le programmeur était le plus compétent pour décider de la marche à suivre. Par exemple, il peut s'agir d'une boucle pour laquelle il sait, par une certaine hypothèse, qu'elle ne peut s'exécuter qu'un certain nombre de fois, et donc la région ne risque pas d'exploser. Il peut aussi s'agir de l'allocation d'une structure utile à l'application. Par exemple, la construction récursive d'une liste chaînée sera considérée par notre analyse comme une fuite de mémoire potentielle, mais le développeur choisira d'ignorer l'avertissement, sachant que les objets alloués ne sont pas une fuite mémoire.

Discussion Il faut remarquer de toute façon que cette analyse détecte uniquement les explosions de région liées à notre politique d'allocation, et pas l'ensemble des fuites mémoire possibles. Par exemple, un programme qui contiendrait une récursivité infinie sans aucun site d'allocation ne serait pas repéré par cette analyse. Pourtant, il provoquerait lui aussi une erreur d'exécution après avoir épuisé la mémoire réservée à la pile d'exécution.

Les expérimentations que nous avons menées, présentées dans le chapitre suivant, montrent que l'analyse statique détecte effectivement ces deux sortes de situations. Pour certains avertissements, liés à la phase d'initialisation du programme, il n'est pas nécessaire de modifier quoi que ce soit. Par contre, lorsque l'analyse indique un risque d'explosion de mémoire pendant la phase principale de l'exécution, les sites d'allocation en question produisent bien des objets qui s'accumulent dans une région durable.

7.5 Extensions

Cette section présente plusieurs extensions de la politique d'allocation présentée dans les sections précédentes. À chaque fois, l'objectif est de traiter plus de programmes que ne le permet l'approche simple vue jusqu'ici. La première extension porte sur la gestion des variables globales, que nous avons laissée de côté depuis le début de ce chapitre par souci de clarté. En effet, il s'agit d'un aspect qui ne change pas beaucoup de choses en pratique, et qui fait partie de notre approche depuis le début.

Les points suivants s'intéressent à combiner la politique d'allocation en régions que nous proposons avec d'autres modèles mémoire, et ont au contraire été inspirés par nos résultats expérimentaux (cf chapitre 8). Par exemple, utiliser une analyse d'échappement en parallèle avec l'analyse d'interférence de pointeurs permettrait d'allouer en pile certains objets auxiliaires, réduisant ainsi les risques d'explosion de région. Nous avons également étudié la possibilité de combiner l'allocation en régions avec un ramasse-miettes à comptage de références, toujours dans une optique de recycler le plus possible d'espace avant de requérir l'intervention du programmeur.

7.5.1 Variables globales et mémoire immortelle

Pour clarifier le discours, nous avons présenté jusqu'ici une version légèrement simplifiée de l'analyse d'interférence de pointeurs, de la politique d'allocation en régions, et de l'analyse d'anticipation des régions. En particulier, nous avons omis les instructions GETSTATIC et PUTSTATIC, qui permettent d'accéder aux champs de classe. Notre implantation permet bien entendu leur usage, et nous détaillons dans cette section comment ces deux instructions sont prises en compte par l'analyse statique et par le gestionnaire mémoire.

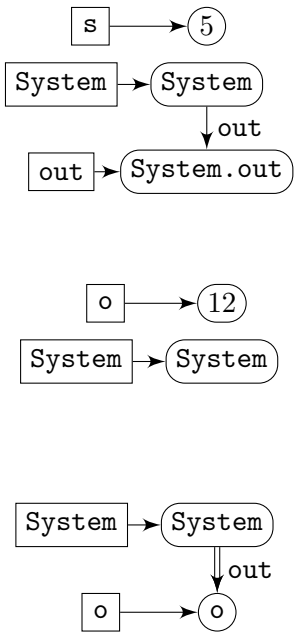
Analyse d'interférence de pointeurs Dans l'analyse d'interférence de pointeurs, les sommets représentant des noms de classe $\boxed{C}$ sont traités de façon similaire aux variables locales, et intégrés

aux familles. Dans l'analyse inter-procédurale, on considère les noms de classe comme des variables que les méthodes se passent systématiquement en argument.

Exemple Dans la figure 7.11 (a), le programme commence par appeler une méthode `setup` qui change le flux de sortie standard. Ensuite, il alloue une chaîne de caractères `s` et appelle le traditionnel `System.out.println` pour afficher le message. Les graphes de forme locale sont donnés dans la figure 7.11 (b). Dans la méthode `setOut` de la classe `System`, l'analyse détecte l'interférence `System` $\sim$ `o` car les deux sommets noms correspondants sont connectés dans le graphe de forme locale. Cette interférence est propagée dans la méthode `setup`, comme si `System` était un nom de variable locale. Dans la méthode `main`, il y a également une interférence `System` $\sim$ `out`, mais l'appel à `println` ne fait pas interférer `out` et `s`. Nous n'avons pas représenté ici le code de `println`, qui est par ailleurs assez complexe, mais il ne fait pas interférer ses deux arguments.

```
1  main()
2  {
3    setup();
4
5    String s = new String("Hello world !");
6    PrintStream out = System.out
7    out.println(s)
8  }
9
10 setup()
11 {
12   PrintStream o = new PrintStream(...);
13   System.setOut(o);
14 }
15
16 class System
17 {
18   static PrintStream out;
19
20   static setOut(PrintStream o)
21   {
22       System.out = o;
23   }
24
25   ...
26 }
```

(a)



(b)

FIG. 7.11 – Un programme faisant usage des champs de classe, et les graphes de forme locale associés.

Placement des objets La sémantique de notre langage telle que nous l'avons donnée dans la section 7.2.1 ne permet pas de rendre compte de l'exécution de ce programme. En effet, les champs de classe comme `System.out` sont utilisés via les instructions GETSTATIC et PUTSTATIC auxquelles nous n'avons pas donné de sémantique. En pratique, le programme ne commence pas par exécuter la méthode `main` en partant d'un tas ne contenant aucun objet. Comme on le verra plus en détail dans le chapitre 8, le processus de démarrage de la machine virtuelle est complexe, et le chargement du programme provoque déjà l'exécution de certaines méthodes, notamment les initialisateurs statiques (méthodes `<clinit>`). Par exemple, pendant cette phase, la méthode `System.<clinit>` alloue le premier `PrintStream` qui servira de flux de sortie standard, et l'affecte au champ `System.out`, bien avant l'exécution de `main`.

Dans notre implantation, nous considérons que tous les objets créés pendant cette phase de

démarrage, notamment les objets représentant les classes, sont immortels. Tous ces objets forment une région particulière, qui ne sera jamais désallouée, que nous appellerons *région immortelle*. Une approche similaire est adoptée par la Spécification Temps-Réel pour Java (cf section 2.6.1) qui définit une région particulière appelée *Immortal Memory*.

La stratégie de placement des objets considère cette région comme n'importe quelle autre. À l'exécution, le programme de la figure 7.11 placera donc l'objet alloué ligne 12 dans la région immortelle, puisque la variable `o` est en interférence avec le nom de classe `System`, qui «pointe» sur l'objet (`System`) situé dans la région immortelle. Par contre, la chaîne de caractères allouée ligne 5 sera placée dans une nouvelle région, puisque la variable `s` n'interfère pas avec les autres variables de `main` ni avec des noms de classe.

Analyse de régions L'algorithme qui construit les tribus considère également les noms de classe comme une variable locale. L'analyse que nous avons donnée dans la section 7.4 est donc également capable de détecter une éventuelle explosion de la région immortelle. En pratique cependant, nous n'avons pas rencontré de programme présentant une potentielle explosion de la région immortelle. Le cas échéant, il se serait agi d'un style de programmation assez inélégant, reposant sur des variables globales pour stocker l'information nécessaire à l'application.

7.5.2 Combinaison avec un allocateur en pile

Lorsque le programme présente un risque de fuite de mémoire, l'analyse de région présentée à la section 7.4 permet de déterminer quels sont les sites d'allocation incriminés. Dans certaines situations, ces sites d'allocation allouent uniquement des objets *auxiliaires*, comme des itérateurs. Le cas de ces objets est particulier : il pointent effectivement dans la structure de données principale, puisque leur utilité est d'en permettre le parcours, mais ils ne sont jamais référencés par un autre objet.

A cause de cette connexion, la politique d'allocation en régions décide de les placer dans la même région que la structure de données en question, au risque de provoquer une fuite de mémoire si un nombre a priori inconnu d'itérateurs est ainsi utilisé. Le programmeur n'a pas de moyen satisfaisant pour régler ce problème, car l'allocation d'un itérateur à chaque parcours de structure est une pratique encouragée en Java.

Analyse d'échappement En pratiquant sur le programme une analyse d'échappement en parallèle avec l'analyse d'interférence de pointeurs, une extension de notre politique d'allocation permettrait de combiner l'allocation en régions avec l'allocation en pile (cf section 4.3.3). Ainsi, les objets auxiliaires comme les itérateurs, ou les `StringBuffer` qui sont utilisés pour concaténer les chaînes de caractères, pourraient être alloués en pile, car ils s'échappent rarement de la méthode dans laquelle ils sont alloués.

Politique d'allocation Cependant, cette extension ne respecterait pas l'invariant que nous avons adopté pour notre modèle mémoire (cf propriété 7.9 page 106), puisque des objets connectés se retrouveraient placés à des endroits différents. Il faudrait donc modifier considérablement la politique d'allocation, notamment pour savoir où placer les objets connectés uniquement à l'objet auxiliaire. Cette situation est illustrée dans le programme de la figure 7.12, qui fait une simple concaténation de chaînes. Un `StringBuffer` est doté d'un tableau de caractères dans lequel il va recopier les éléments des chaînes que l'on veut concaténer. Si le `StringBuffer` de la ligne 6 est alloué sur la pile dans la méthode `main`, au moment de l'exécution de son constructeur, la politique d'allocation ne saura pas dans quelle région placer le tableau alloué ligne 18.

Discussion Les graphes de forme locale constituent un cadre attrayant pour construire des analyses de pointeurs, et permettraient de concevoir commodément une telle analyse d'échappe-

```
1  main()
2  {
3    s1 = new String("Hello, ");
4    s2 = new String("world !");
5
6    sb = new StringBuffer(42);
7
8    sb.append(s1);
9    sb.append(s2);
10 }
```

```
11 class StringBuffer()
12 {
13   char value[];
14   int count;
15
16   <init>(int size)
17   {
18     tmp = new char[size];
19     this.value = tmp;
20     count = 0;
21   }
22
23   void append( String s )
24   {
25     t1 = s.value;
26     t2 = this.value;
27     for( i=0 ; i< t1.length ; i++, count++)
28     {
29       t2[count]=t1[i];
30     )
31   }
32 }
```

FIG. 7.12 – Un programme concaténant deux chaînes de caractères à l'aide d'un `StringBuffer`.

ment. Il faudrait ensuite modifier la politique d'allocation en régions pour qu'elle tienne compte des objets alloués en pile, y compris pour placer des objets comme le tableau alloué ligne 18. Nous n'avons pas mené d'expérimentations dans cette voie, car nous avons privilégié la piste plus générale présentée dans la section ci-dessous.

7.5.3 Combinaison avec un ramasse-miettes à comptage de références

Pour construire un gestionnaire mémoire au comportement temporel prévisible, nous nous sommes basés sur le modèle mémoire des régions. Cependant, nous avons vu dans le chapitre 3 que les ramasse-miettes à comptage de références présentaient également un comportement temporel attrayant. Nous avons donc étudié la possibilité de combiner la gestion mémoire en régions avec un tel ramasse-miettes [Ber07].

Principe L'idée est de profiter du comportement temporel prévisible de ces deux techniques de gestion mémoire, tout en évitant leurs défauts respectifs, liés au comportement mémoire. D'une part, un ramasse-miettes à comptage de références est incapable de recycler les structures de données isolées contenant des chemins cycliques. D'autre part, la synthèse de régions a tendance pour certains programmes à placer trop de données dans une même région durable, dont la taille augmente indéfiniment. Dans le cas où ces données ont une forme acyclique, il paraît pertinent de les placer dans une région spécifique, gérée par un ramasse-miettes à comptage de références.

Exemple Dans le programme de la figure 7.10 (b) page 111, le programme utilise une structure de données constituée d'un objet `x` de longue durée de vie et d'un objet `q` référencé par `x`. À chaque itération de la boucle principale, un nouvel objet `q` est alloué et remplace le précédent qui devient inaccessible. Avec la politique d'allocation en régions, tous les exemplaires de `q` sont placés dans la même région et occupent de plus en plus d'espace. Mais comme la structure en question est toujours acyclique, l'utilisation d'un ramasse-miettes à comptage de références permettrait dans ce programme de désallouer `q` à chaque fois qu'il devient inaccessible, recyclant ainsi l'espace au fur et à mesure.

Modification de la politique d'allocation Nous n'avons pas cherché à automatiser l'utilisation du ramasse-miettes à comptage de références, vu le surcoût important qu'il ajoute à toutes les opérations mémoire à cause des barrières en écriture. Au contraire, nous avons opté pour l'idée d'offrir au programmeur la possibilité de l'utiliser à la demande, pour des sites d'allocation bien choisis. Nous avons ainsi ajouté manuellement un nouveau membre spécial dans les familles des sites d'allocation en question. Par exemple, dans le programme de la figure 7.10 (b), la famille de `q` devient `p` $\sim$ `q` $\sim$ GC. Nous avons écrit une version modifiée du gestionnaire mémoire pour qu'il place en conséquence les objets de ces familles dans une région spéciale, gérée par un ramasse-miettes à comptage de références.

Discussion Grâce à cette technique, certains programmes comportant des explosions de région peuvent s'exécuter sans encombrer toute la mémoire, car les objets qui devraient s'accumuler dans les régions sont recyclés dès leur mort par le ramasse-miettes. Le comportement temporel de ces programmes reste prévisible, car les temps de pause associés aux opérations de comptage de références sont importants mais connus. Néanmoins, cette technique comporte quelques limites. Comme avec l'allocation en pile, l'invariant choisi pour notre modèle mémoire n'est plus vérifié. En effet, certains objets situés dans une région pointent vers des objets situés dans la zone associée au ramasse-miettes, et inversement. La validité de la politique de destruction des régions est alors remise en question. De plus, comme le comptage de références ne permet pas de détecter la mort de structures cycliques, la zone gérée par le ramasse-miettes peut finir par s'encombrer elle aussi, à moins de s'assurer de l'absence de cycles. La combinaison entre allocation en régions et comptage de références est donc une piste intéressante, mais plusieurs problématiques qu'elle soulève doivent encore être éclaircies.

7.6 Conclusion

Au cours de ce chapitre, nous avons abordé dans une formulation théorique les divers éléments qui constituent notre approche. L'analyse d'interférence de pointeurs examine le graphe de forme locale de chaque méthode pour y déterminer une sur-approximation des connexions entre les objets. À l'exécution, le gestionnaire mémoire utilisera ces résultats pour créer les régions et y grouper les objets appartenant à une même structure de données.

Nous avons vu que cette stratégie de placement était correcte, c'est à dire qu'elle ne peut pas détruire de région contenant des objets vivants. Au contraire, pour certains programmes, elle peut même mener à l'explosion d'une région, c'est à dire l'allocation continuelle dans la même région d'objets de courte durée de vie, risquant d'occuper à terme l'ensemble de la mémoire. L'analyse d'anticipation des régions détecte statiquement ce phénomène, de façon à avertir le programmeur des endroits problématiques dans le programme.

Dans le chapitre suivant, nous allons voir différents points plus pratiques permettant de confirmer par l'expérience l'intérêt de notre approche. En particulier, on s'intéressera à l'implantation du modèle mémoire en régions, qui permet de réaliser toutes les opérations mémoire en temps prévisible. On étudiera également comment se comportent à l'exécution les régions en termes d'occupation mémoire dans différents programmes.

Chapitre 8

Validation expérimentale

Dans ce chapitre, nous allons voir les divers aspects qui permettent la validation expérimentale de notre approche, présentée dans les chapitres précédents. Dans un premier temps, les sections 8.1 et 8.2 s'intéressent à l'implantation des algorithmes proposés. L'analyse statique a été réalisée à l'aide de la plate-forme Soot. D'autre part, le gestionnaire mémoire en régions a été implanté dans la machine virtuelle de l'environnement Jits.

Dans un second temps, la section 8.3 présente les expérimentations que nous avons menées et les résultats obtenus.

8.1 Implantation de l'analyse statique

Pour implanter les algorithmes décrits dans les chapitres 6 et 7, nous avons utilisé l'infrastructure Soot [VRCG+99], dédiée à l'analyse statique et à l'optimisation de programmes Java. Comme illustré sur la figure 8.1, plusieurs représentations intermédiaires sont utilisées. A chaque étape, des analyses statiques et des transformations sont pratiquées sur le code, et il est possible d'insérer de nouveaux algorithmes dans ce processus.

Dans un premier temps, Soot traduit le bytecode Java en une première représentation intermédiaire nommée Baf, qui est proche du bytecode mais où chaque instruction est explicitement typée. Il s'agit cependant encore d'un langage destiné à une machine à pile, et certaines optimisations ne se prêtent pas bien à ce genre de langage. Le code est donc traduit dans un langage à trois adresses appelé Jimple, dans lequel les instructions portent sur des variables temporaires (des *pseudo-registres*) et non plus sur la pile. La syntaxe de Jimple est en apparence assez proche de celle de Java, mais il s'agit toujours d'un langage non structuré et de bas niveau. En effet, les structures de contrôle ne sont pas reconstruites, et restent exprimées sous la forme de sauts conditionnels. De plus, les expressions complexes sont décomposées en expressions élémentaires, comme nous l'avons vu au chapitre 6.

La syntaxe que nous avons adoptée dans les chapitres précédents pour exprimer les analyses est ainsi très fortement inspirée de celle de Jimple. Il s'agit de la représentation intermédiaire la plus utilisée par Soot, qui offre de nombreux algorithmes d'analyse et d'optimisation. Par exemple, le module Spark [LH03] construit un graphe d'appel statique du programme en utilisant la représentation intermédiaire Jimple. L'utilisateur peut cependant ajouter de nouvelles analyses dans ce processus. C'est de cette façon que nous avons implanté les analyses statiques présentées dans cette thèse.

Nous n'avons pas exploité cette fonctionnalité, mais Soot permet également de pratiquer des optimisations sur les programmes. Après la phase d'analyse et transformation, le code Jimple est donc traduit de nouveau en bytecode, en passant par une représentation intermédiaire nommée Grimp dans laquelle les expressions complexes ont été agrégées.

Dans l'objectif de rester toujours au plus près des manières habituelles de programmer en Java, notre approche, contrairement à celles de Cherem et Rugina [CR04] et de Chin *et al.*

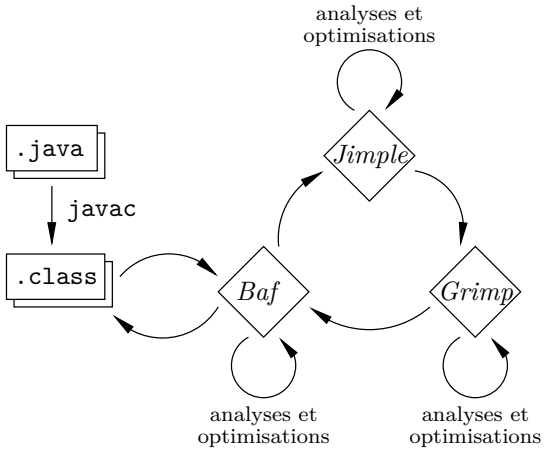


FIG. 8.1 – Les trois représentations intermédiaires utilisées par SOOT.

[CCQR04], ne suppose pas de modifier le langage exécuté. Notre analyse ne modifie donc pas le programme, mais enregistre séparément les résultats de l'analyse dans un format qui sera repris par la machine virtuelle au moment du chargement du programme.

8.2 Gestionnaire mémoire

8.2.1 Intégration dans JITS

Pour implanter notre système de gestion mémoire en régions, nous avons modifié la machine virtuelle du projet JITS (cf section 2.4.4). Cette section détaille les différentes problématiques associées à cette intégration.

Pré-chargement et migration système L'approche proposée par JITS décompose la construction du logiciel embarqué en plusieurs phases [Cou06]. Pour permettre l'embarquement du système (application, bibliothèque standard et machine virtuelle Java) sur des plates-formes aux ressources très limitées, une partie de l'exécution est déportée en amont sur une station de travail classique. La machine virtuelle est ainsi amorcée et démarrée *in vitro* sur une station de travail aux ressources abondantes.

À un moment arbitraire de l'exécution, par exemple juste avant l'appel de la fonction `main` de l'application, l'exécution est interrompue et JITS prend une *photographie* complète de l'état du système, et enregistre l'ensemble des objets du tas dans une représentation appelée l'*objectspool*. Le système est ensuite *migré* sur la plate-forme embarquée, où l'exécution sera reprise de façon transparente. Ainsi, tout le démarrage de la machine virtuelle et le chargement de l'application auront déjà été effectués et ne seront pas à la charge du système embarqué. Cette propriété est très intéressante non seulement car elle économise les ressources du système, mais également car elle lui permet de s'exécuter dans un environnement où il n'a plus la possibilité de faire ce chargement de classes.

Pour permette de capturer non seulement l'état de l'application mais également l'état interne de la machine virtuelle, toutes ses structures de données (y compris les classes, le code des méthodes, et la pile d'exécution) sont implantées sous la forme d'objets Java. En effet, la machine virtuelle de JITS est presque entièrement écrite en Java ; seuls l'interpréteur d'instructions, le système de gestion mémoire, et quelques méthodes de très bas niveau, sont implantés par du code natif. Ces éléments ne possèdent pas d'état interne, la capture du tas permet donc de représenter l'ensemble de l'état d'exécution du système.

Afin d'associer à chaque méthode les informations représentant les résultats de l'analyse d'interférence de pointeurs, nous avons complété le processus de chargement de classes de JITS par

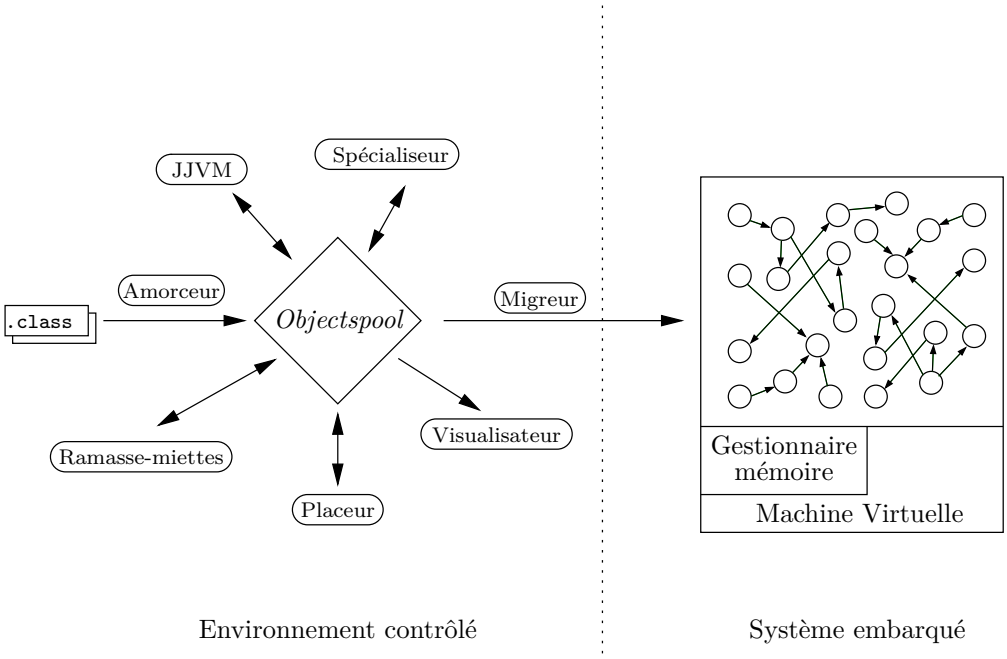


FIG. 8.2 – Architecture de JITS, centrée autour de l'*objectspool*.

une phase qui consulte les résultats précédemment enregistrés et les ajoute aux objets décrivant la méthode.

Spécialisation tardive Pour reprendre l'exécution sur la plate-forme cible, il n'est pas nécessaire de transférer l'ensemble de l'*objectspool* capturé. En effet, certains objets ne seront plus jamais d'aucune utilité au système. Par exemple, les objets morts ne seront plus jamais accessibles, ce n'est donc pas la peine de les conserver. JITS offre ainsi un outil *ramasse-miettes* permettant de débarrasser l'*objectspool* des objets inaccessibles pour ne pas les transférer sur le système embarqué. Plus sophistiqué, l'outil *spécialiseur* analyse l'*objectspool* pour y éliminer les objets accessibles mais définitivement inutiles. Par exemple, les objets représentant les méthodes `<clinit>` ainsi que leur code peuvent être supprimés puisque les initialisations de classes ne sont exécutées qu'une seule fois. Une autre outil permet de factoriser des données constantes [RD04] définies par les différentes classes. En effet, chaque classe du programme possède une table appelée *constant pool* dans laquelle sont stockées toutes les données constantes utilisées par les méthodes de la classe. Lorsque certaines de ces données sont égales d'une classe à l'autre, il est inutile de garder deux exemplaires différents du même objet constant. Le *compacteur* permet ainsi d'éliminer les références redondantes pour réduire l'empreinte mémoire du système.

Divers outils de *configuration* permettent au contraire d'ajouter au tas des informations qui seront utiles lors de la migration. Par exemple, le *placeur mémoire* analyse le tas pour déterminer quels objets ne seront plus jamais modifiés, de façon à les placer dans la mémoire morte sur le système embarqué [MCG05], bien plus abondante que la mémoire vive. L'ensemble de ces outils est représenté sur la figure 8.2. En réalité, l'amorçage du système est lui aussi réalisé par un outil appelé *amorceur*, qui génère un premier *objectspool* à partir d'un ensemble de classes non chargées. De même une machine virtuelle, la *JJVM*, permet d'exécuter le système décrit par l'*objectspool* pour faire progresser son état.

Avant d'être transféré sur le système embarqué, le logiciel s'exécute donc dans un environnement virtuel. Notamment, aucune limite n'est imposée sur la taille de l'*objectspool*. Au moment de la migration, les différents objets sont placés sur les différentes mémoires disponibles (ROM, RAM, mémoire flash, etc.), chacune gérée par un sous-système spécifique.

L'architecture retenue par JITS élimine ainsi du système réellement exécuté une partie

considérable des allocations, ce qui est très profitable à notre approche.

Le système de gestion mémoire de JITS L'objectif de JITS est de fournir un environnement d'exécution Java standard sur des architectures matérielles aux ressources très contraintes, en particulier au niveau de la mémoire. Pour tirer parti des différents types de mémoires disponibles sur ces architectures, le système de gestion mémoire applique une politique de *placement mémoire* pour répartir les objets [MG07]. Notamment, tous les objets représentant les données constantes utiles à la machine virtuelle, comme les descripteurs de classes, de méthodes et les tableaux de bytecode sont placés dès l'origine dans la ROM.

En fonction des caractéristiques des différentes mémoires, différents ramasse-miettes peuvent être employés sur chacune d'entre elles [GM07]. Par exemple, on peut employer conjointement des algorithmes de marquage-balayage, marquage-compactage, et de copie par demi-espaces (cf section 3.4).

Notre travail se situe lui aussi au niveau du système de gestion mémoire, mais suppose de désactiver le ramasse-miettes pour ne pas infliger au programme de l'application des pauses dans son exécution. Afin de mener à bien nos expérimentations, nous avons donc modifié légèrement l'interface entre l'interpréteur d'instructions et le système de gestion mémoire, de façon à pouvoir implanter notre allocateur en régions *à la place* d'un ramasse-miettes. Certains aspects du fonctionnement de la machine virtuelle se retrouvent donc altérés, ne respectant plus tout à fait la spécification Java. Par exemple, l'appel à `System.gc()` n'a plus d'effet, puisque le système ne comporte plus de ramasse-miettes. De même, les algorithmes adoptés pour la manipulation des régions en temps constant ne permettent pas de respecter la spécification quant aux destructeurs d'objets (méthodes `finalize`). Ces restrictions paraissent malgré tout secondaires devant le bénéfice apporté par la gestion mémoire en régions.

8.2.2 Implantation du gestionnaire mémoire

Au cours de l'exécution, le gestionnaire mémoire est chargé de mettre en œuvre la politique d'allocation (cf section 7.2.2). Il doit donc pouvoir réaliser les différentes opérations nécessaires, comme la création et la destruction de région, et l'allocation d'un objet dans une certaine région. Nous adoptons un modèle où la taille des régions n'est pas fixée à l'avance, il faut donc également pouvoir augmenter à la demande la taille d'une région. Pour permettre la prévisibilité du comportement temporel du programme, toutes ces opérations sont implantées de façon que leur temps d'exécution au pire cas soit constant pour un programme donné.

8.2.2.1 Modèle mémoire

Nous avons adopté un modèle mémoire à base de pages mémoire de taille fixe, organisées en listes chaînées. L'ensemble de la mémoire disponible pour l'allocateur est donc divisé en pages, alignées sur des adresses multiples de la taille de page. Toutes les pages sont initialement chaînées ensemble pour former la *free-list*. Chaque région r est également représentée par une liste de pages, et identifiée par un pointeur sur sa première page. Cependant, il ne s'agit alors pas d'une liste simplement chaînée, car certaines opérations ne pourraient pas être réalisées en temps fixe. Chaque page p dispose, en plus du pointeur $p.nextpage$ vers la suivante, d'un pointeur $p.firstpage$ vers la première page de la liste, ce qui permet au système d'identifier la région en temps constant. Les allocations se font à chaque fois dans la dernière page de la région. La première page d'une région dispose donc d'un pointeur supplémentaire $p.lastpage$ vers cette dernière page. Ce modèle mémoire est résumé sur la figure 8.3, et les différentes opérations du gestionnaire mémoire sont détaillées dans les paragraphes suivants.

Initialisation L'initialisation du gestionnaire mémoire consiste à découper l'espace disponible en pages de taille fixe, et à les chaîner entre elles par leur champ *nextpage*. La taille des pages

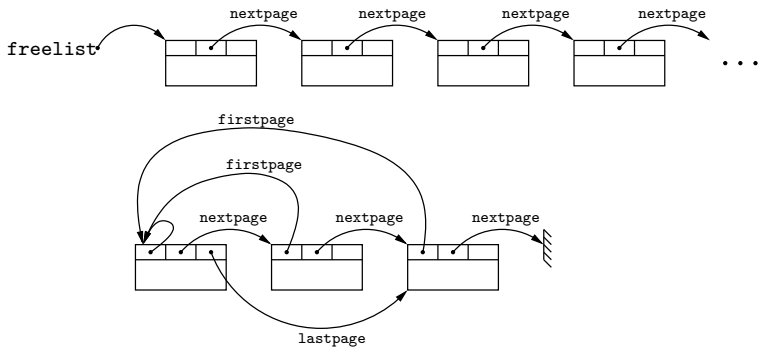


FIG. 8.3 – Le modèle mémoire en régions.

doit être fixée à ce moment-là. Dans nos expérimentations, nous avons choisi une taille de page de 512 octets, de façon à pouvoir contenir les plus gros objets de nos programmes d'essais. Dans le cas général, les programmes peuvent allouer des tableaux arbitrairement grands, donc aucune taille de page ne peut suffire à contenir tous les objets possibles. Ce problème est courant dans les systèmes de gestion mémoire temps-réel. L'approche classique [Sie00, RF02, BCR03] consiste à *diviser* les grands tableaux en morceaux au moment de l'allocation, et à intercepter les accès pour déterminer l'emplacement réel de chaque case. Comme nos programmes d'essais ne comportaient pas de grands tableaux, nous n'avons pas reproduit ce mécanisme.

L'espace disponible pour le gestionnaire mémoire en régions ne représente pas la totalité du tas. En effet, au moment de l'initialisation, de nombreux objets sont déjà présents en mémoire à cause du pré-chargement du programme : les objets représentant les classes, les donnes globales constantes, le code des méthodes, etc. L'ensemble de ces objets, typiquement placés en ROM sur le système embarqué, sont considérés par notre gestionnaire mémoire comme faisant partie de la région immortelle, et ignorés par la politique de placement des objets.

8.2.2.2 Opérations mémoire en temps prévisible

Création d'une région Lorsque le gestionnaire mémoire ne trouve pas de région dans laquelle placer un objet, il en crée une nouvelle. Pour cela, il retire la première page p de la *free-list* qui devient la première page de la région. Les champs $p.firstpage$ et $p.lastpage$ sont initialisés à p, et le champ $p.nextpage$ est initialisé à *null* puisque la région ne contient pour l'instant qu'une seule page. L'allocation se déroule alors normalement, comme décrit ci-dessous.

Examen des variables locales pour l'allocation Lors de l'exécution d'un site d'allocation `v=new`, le gestionnaire mémoire cherche une autre variable locale `v'` dans la même famille que `v`, et place le nouvel objet o dans la même région que l'objet o' référencé par `v'`. Cette opération nécessite, au pire cas, d'examiner toutes les variables locales de la méthode. Son temps d'exécution au pire cas est donc constant pour un programme donné.

En pratique, nous avons simplifié légèrement la politique d'allocation par rapport à la règle présentée page 103. Plutôt que d'examiner une à une toutes les variables locales de la méthode, le gestionnaire mémoire ne s'intéresse qu'aux paramètres formels. L'ensemble des autres variables locales de la méthode sont confondues en une seule référence, placée parallèlement à r_{ret} dans le cadre de pile. Cette modification est due entre autres au fait que la machine virtuelle qui exécute le programme sous forme de bytecode est une machine à pile. En conséquence, les différentes variables locales présentes dans le code analysé ne subsistent pas toutes au moment de l'exécution.

De plus, il n'est pas nécessaire d'attendre l'exécution pour examiner les familles. L'association entre chaque site d'allocation et l'un ou l'autre des paramètres formels de sa méthode peut être calculée à l'avance. Nous avons implanté ce calcul dans un outil JITS, de façon à l'exécuter

avant la migration, résuisant ainsi considérablement le travail du gestionnaire mémoire lors de l'exécution du système final.

Examen des variables locales pour la destruction de région Dans notre politique d'allocation, les variables du programme servent aussi à déterminer quelles régions peuvent être détruites (cf définition 7.7 page 105). Nous avons également simplifié cette règle en ne cherchant pas à distinguer les différentes variables locales à l'intérieur d'un même cadre de pile, mais en associant chaque région à un certain cadre. Chaque retour de méthode provoque donc éventuellement une destruction de région. Le temps nécessaire pour déterminer si une région est associée ou pas au cadre de pile courant est constant puisqu'il s'agit d'un simple accès mémoire.

Il est à noter qu'à l'exception d'un seul programme de notre banc de test (le programme TSP, voir page 127), cette simplification n'a pas d'impact observable sur le comportement du gestionnaire mémoire.

Identifiant de région r_{ret} La politique d'allocation suppose la présence d'un identifiant de région r_{ret} dans chaque cadre de pile (cf définition 7.3 page 103) . Nous avons modifié la structure de la pile d'exécution de JITS pour y rajouter cet identifiant. De plus, chaque instruction d'appel doit initialiser cet identifiant dans le cadre de pile de la méthode appelée. Comme décrit dans la section 7.2.2, nous avons écrit une seule implantation de *choose_region*, utilisée à la fois par les sites d'allocation et les sites d'appel.

Accès à la région d'un objet Pour déterminer la région d'un objet, le gestionnaire mémoire arrondit son adresse au plus proche multiple de 512 de façon à obtenir la page p de l'objet, puis résout $p.firstpage$. Les pages sont alignées sur les adresses multiples d'une puissance de 2, un masquage de bits sur l'adresse de l'objet permet donc de déterminer en temps constant l'adresse de la page. Les trois pointeurs *nextpage*, $firstpage$ et *lastpage* sont situés dans les trois premiers mots de la région. Seul le reste de l'espace est dédié au stockage des objets.

Allocation d'un objet Les allocations sont réalisées en temps constant par décalage de pointeur. L'en-tête de chaque page, outre les trois pointeurs mentionnés ci-dessus, contient donc également un curseur qui indique le niveau de remplissage de la page. Pour allouer un objet dans une région, le gestionnaire mémoire accède à la dernière page de cette région ($r.lastpage$), mémorise l'emplacement pointé par le curseur, *décale* ce dernier de la taille de l'objet alloué, puis retourne l'emplacement mémorisé. Si l'espace disponible dans la page est insuffisant, le gestionnaire mémoire étend la région avec une nouvelle page comme indiqué ci-dessous, puis réitère l'allocation dans cette nouvelle page. Les allocations se font ainsi au pire cas en temps constant. Une politique qui examinerait à chaque allocation l'ensemble des pages de la région pour chercher une page comportant assez d'espace libre mènerait à une fragmentation interne moins importante puisqu'elle exploiterait mieux l'espace. Cependant, le temps au pire cas de l'allocation ne serait plus constant.

Extension d'une région Pour ajouter une nouvelle page à une région r, le gestionnaire mémoire retire la page p en tête de la *free-list*, puis l'ajoute à la fin de la région. La nouvelle page est initialisée avec $p.firstpage = r$ et $p.nextpage = null$. Pour l'insérer en fin de liste, le gestionnaire mémoire accède à la page $p' = r.lastpage$ et met à jour $p'.nextpage = p$. Ensuite, il fait pointer $r.lastpage$ sur p, et la région a de nouveau la forme présentée dans la figure 8.3. Ainsi, la dernière page de la région d'un objet o est toujours accessible en temps constant par l'expression $getpage(o).firstpage.lastpage$.

Destruction de région Détruire une région revient à la concaténer avec la *free-list*. Cette opération de concaténation en tête peut être réalisée en temps constant grâce au champ *lastpage*.

Le gestionnaire mémoire met simplement à jour le pointeur *r.lastpage.nextpage* avec l'adresse de la première page de la *free-list*, puis fait pointer la *free-list* sur r. Lorsque des pages sont considérées comme libres, les pointeurs $firstpage$ et *lastpage* de leur en-tête n'est pas significatif, mais le chaînage est conservé.

8.2.2.3 Discussion

Le modèle mémoire adopté pour l'implantation des régions permet de réaliser l'ensemble des opérations du gestionnaire mémoire en temps constant. Ainsi, il devient possible d'évaluer les temps d'exécution au pire cas d'un fragment de code même en présence d'allocation mémoire dynamique, puisque l'exécution n'est plus dérangée par des temps de pause imprévisibles.

8.3 Résultats expérimentaux

Cette section présente les expérimentations que nous avons réalisées avec les outils décrits ci-dessus, ainsi que les résultats obtenus. En particulier, nous avons étudié l'impact de notre algorithme de synthèse de régions sur le comportement mémoire des programmes.

8.3.1 Protocole expérimental

Pour évaluer l'effet de l'utilisation de régions, nous avons comparé deux exécutions pour chaque programme, l'une avec un ramasse-miettes, et l'autre avec notre gestionnaire mémoire en régions. Nous avons utilisé pour cela la machine virtuelle de Jits, et le ramasse-miettes par défaut, du type marquage-compactage (cf section 3.4.2.2). Nous avons exclu de nos mesures les phases d'exécution correspondant au démarrage de la machine virtuelle ainsi qu'au chargement de l'application. En effet, nous nous intéressons dans ce travail à la gestion de la mémoire dynamique dans les programmes applicatifs et non à la gestion de la mémoire dans la machine virtuelle. Les courbes présentées ci-dessous représentent donc uniquement l'exécution à partir du début de la méthode `main` du programme utilisateur. Comme la machine virtuelle n'alloue pas de mémoire en dehors de la phase de démarrage, elle n'interfère pas avec le comportement du programme.

Il faut noter que les temps présentés sur les figures de la 8.3.2.2 sont mesurés en nombre d'instructions interprétées par la machine virtuelle, et non pas en secondes. De cette façon, les courbes obtenues pour les deux versions du programme sont en phase, et la comparaison des deux exécutions est plus aisée. Les collectes du ramasse-miettes paraissent ainsi instantanées, puisqu'aucune instruction de l'application n'est interprétée pendant le temps de pause. Bien sûr, ce n'est pas le cas dans la réalité. A l'inverse, notre gestionnaire mémoire est conçu pour être prévisible. En particulier, les opérations élémentaires sur les régions sont toutes implantées de façon à s'exécuter en temps constant. L'exécution du programme n'est donc pas interrompue par des temps de pause imprévisibles.

8.3.2 Exemples de petite taille

Pour étudier le comportement de notre allocateur en régions, nous avons choisi d'appliquer notre approche à la suite de programmes de test JOlden [CR95, CM01], également utilisée dans les travaux existants de synthèse de régions [CR04, CCQR04]. Il s'agit de dix programmes Java constitués chacun d'essentiellement une seule tâche algorithmique, comme un parcours d'arbre, ou des opérations sur de grands tableaux. De plus, ce ne sont pas des programmes destinés à l'embarqué, et ils utilisent donc des manières de programmer typiques du langage Java, comme l'usage intensif d'objets temporaires ou le polymorphisme. Par exemple, ils font souvent des affichages à l'écran en concaténant des chaînes de caractères avec l'opérateur `+`, ce que le compilateur traduit par la création et l'utilisation d'un objet `StringBuffer`. Si ce genre d'affichage ne paraît pas très réaliste pour un système embarqué, des manipulations de chaînes de

caractères sont également assez courantes lors des communications réseau, et nous ne les avons donc pas désactivées lors de nos expériences.

La taille des différents programmes, ainsi que les temps d'analyse correspondants, sont indiqués sur la figure 8.4. Les colonnes 2 et 3 se rapportent la taille de l'application elle-même, alors que les colonnes 4 et 5 incluent l'ensemble des méthodes accessibles à partir de la méthode `main` dans le graphe d'appel statique, y compris celles de la bibliothèque standard. Le programme Voronoi, qui utilise plus de classes dans la hiérarchie `java.util` que les autres programmes, nécessite l'analyse d'une quantité de code nettement plus importante.

Les colonnes LPTA, PIA, et TA représentent respectivement les temps passés à exécuter l'analyse de forme locale, l'analyse d'interférence de pointeurs, et l'analyse d'anticipation des régions. Ces expériences ont été menées sur un Pentium IV 3.2GHz doté de 2Go de RAM, à l'aide de Soot 2.2.2. Les temps d'analyse obtenus sont assez faibles, et dépendent en pratique plutôt des variations de charge de la machine utilisée que de la taille des programmes traités.

Programme	Application		Total		Temps d'analyse (secondes)			
	Méthodes	Instructions	Méthodes	Instructions	LPTA	PIA	TA	Total
BH	70	1469	802	12473	0.429	1.16	0.557	2.15
BiSort	15	363	747	11367	0.237	1.22	0.962	2.42
Em3d	22	527	754	11531	0.187	1.11	0.348	1.65
Health	29	727	761	11737	0.157	1.22	0.902	2.28
MST	36	544	768	11548	0.24	1.15	0.855	2.25
Perimeter	45	658	777	11662	0.143	1.49	0.961	2.59
Power	32	1112	764	12116	0.149	0.903	0.639	1.69
TreeAdd	12	222	744	11226	0.15	1.01	0.868	2.03
TSP	18	533	750	11537	0.305	1.035	0.983	2.32
Voronoi	73	1168	1406	20903	0.307	1.384	0.656	2.35

Fig. 8.4 – Temps d'analyse pour les programmes de la suite JOlden.

8.3.2.1 Présentation des programmes

BH Le programme BH est une simulation du problème des n corps : on connaît la masse ainsi que la position et la vitesse initiales d'un ensemble de n corps célestes qui interagissent gravitationnellement, et on cherche à déterminer leurs trajectoires au cours du temps. Le programme est basé sur l'algorithme de Barnes-Hut, c'est à dire il divise l'espace sous la forme d'un arbre 8-aire incomplet. Les sommets de cet arbre, appelés *cellules*, représentent chacun un certain cube contenant soit du vide, auquel cas ils n'ont pas de successeur dans l'arbre, soit un ou plusieurs corps, auquel cas ils sont divisés en 8 sous-cellules, et ainsi de suite. Chaque pas de la simulation consiste à calculer la masse et la position du centre de gravité de chaque cellule, ainsi que l'attraction des cellules les unes sur les autres.

Le programme commence par allouer la structure de données qui représente l'arbre 8-aire, constituée d'objets `Cell` et `Body`. Chaque `Cell` possède un champ de type tableau pouvant contenir ses sous-cellules ou des `Body` proprement dits. La position en trois dimensions de chaque cellule est représentée par un objet de type `MathVector`, constitué uniquement de trois nombres à virgule flottante. Chaque corps possède lui aussi un `MathVector` pour représenter sa position, un autre pour sa vitesse, ainsi que *deux* pour représenter l'accélération.

À chaque pas de la simulation, la masse totale ainsi que la position des cellules sont recalculées, puis leur attraction sur chaque corps est évaluée pour obtenir la nouvelle accélération des corps. L'effet de cette accélération est reporté sur la vitesse, et celui de la vitesse sur la position des corps, puis la structure des cellules est réajustée pour correspondre à la nouvelle répartition des corps. Tous ces calculs provoquent l'allocation d'une quantité considérable de mémoire, majoritairement sous la forme d'objets de type `MathVector`.

Dans ce programme, notre analyse d'anticipation des régions détecte plusieurs problèmes potentiels. Dans la méthode `createTestData`, qui construit la structure de données, de nombreux objets `Body`, ainsi que les `MathVector` sont alloués dans une boucle. Cet avertissement n'est pas significatif, car il correspond à une allocation nécessaire à l'initialisation du programme. Par contre, dans la boucle de simulation, l'analyse détecte également un risque d'explosion de région. De nombreux `MathVector` sont alloués au cours du calcul, et ils seront placés à l'exécution dans la même région que la structure de données principale. De même, des itérateurs sont alloués pour parcourir la structure de données et ils sont en interférence avec elle. Enfin, le programme réajuste la forme de l'arbre pour tenir compte des mouvements des corps, et peut donc allouer au cours de l'exécution de la boucle un nombre inconnu d'objets de type `Cell`. L'analyse statique signale au programmeur l'ensemble des sites d'allocation en question, indiquant ainsi à quels endroits il faudrait modifier le code pour corriger le comportement du programme.

BiSort Le programme BiSort est une implantation d'un algorithme de tri appelé *tri bitonique*, destiné aux architectures d'exécution parallèles. Par exemple, si l'on dispose de deux processeurs, le principe est de diviser la séquence à trier en deux moitiés, et de laisser chacun trier sa moitié. Les deux processeurs s'envoient ensuite respectivement leurs deux demi-listes, et procèdent, en parallèle, à une fusion locale des valeurs. Le premier processeur ne garde que les plus petites valeurs, et le second ne garde que les plus grandes. Une fois cette fusion effectuée, ils disposent chacun d'une moitié de la séquence triée.

L'implantation de cet algorithme dans le programme BiSort est séquentielle, mais suit le principe présenté ci-dessus. La séquence à trier est représentée par un arbre binaire complet d'objets de type `Value`, initialisés avec des valeurs aléatoires. Chaque sous-arbre est ensuite trié récursivement avant d'être fusionné avec son voisin. Toutes ces opérations sont effectuées en déplaçant des sommets dans l'arbre, mais ne provoquent pas d'allocation de mémoire après la phase d'initialisation.

Le programme affiche ensuite la séquence triée, puis la trie de nouveau dans l'ordre inverse, et l'affiche une seconde fois. Les affichages entraînent beaucoup d'allocation mémoire à cause des concaténations de chaînes de caractères, qui utilisent autant d'objets de type `StringBuffer`.

Dans ce programme, notre analyse d'anticipation des régions ne détecte aucun risque d'explosion de région, à l'exception de la méthode qui crée la structure de données lors de l'initialisation.

Em3d Le programme Em3d simule la propagation d'ondes électromagnétiques dans un certain objet en trois dimensions. L'espace dans lequel se propagent ces ondes est représenté sous la forme d'un graphe bipartite constitué de sommets `H` et de sommets `E`. Chaque sommet `E` admet un certain nombre de sommets `H` *voisins* et inversement. La distance entre chaque paire de sommets est modélisée par un nombre à virgule flottante associé à chaque arête du graphe. Cette structure est initialisée aléatoirement au début de l'exécution, puis n'est pas modifiée.

À chaque pas de la simulation, le champ électrique en chaque sommet `E` est recalculé à partir des valeurs des champs magnétiques des sommets `H` voisins, et vice versa. Le programme utilise ainsi deux itérateurs pour parcourir les listes chaînées qui rassemblent les deux types de sommets. En dehors de ces deux allocations par itération, le programme n'alloue plus de mémoire après la phase d'initialisation, mais notre analyse d'anticipation des régions détecte bien ces deux risques d'explosion des régions.

Health Le programme Health est censé simuler le système de gestion des hôpitaux Colombien. La structure de données principale est un arbre 4-aire constitué d'objets de type `Village`, chacun doté d'un objet de type `Hospital`. Chacun de ces hôpitaux peut accueillir un certain nombre de `Patients`, stockés dans une liste chaînée. Les hôpitaux des villages situés plus près de la racine ont une capacité d'accueil plus importante.

A chaque itération de la boucle principale, le programme parcourt l'arbre qui représente les villages, et chaque patient est soit traité, soit laissé en attente, soit transmis à l'hôpital du village parent. Par ailleurs, de nouveaux patients apparaissent aléatoirement de temps à autre dans les villages.

Ce programme alloue une quantité considérable de mémoire tout au long de son exécution, en particulier sous la forme d'objets temporaires. Il utilise des itérateurs pour le parcours de l'arbre, ainsi que l'examen des patients de chaque hôpital. Par ailleurs, de nouveaux `Patient`s sont alloués tout au long de la simulation. Ils sont placés au cours du temps dans différentes listes chaînées formées de `ListNode`s. Chaque transfert de patient depuis une liste vers une autre provoque ainsi l'allocation d'un nouveau `ListNode`.

L'analyse d'anticipation des régions détecte de nombreux problèmes potentiels dans ce programme. La création récursive de la structure de données principale, comme dans les autres programmes, est considérée comme un risque, mais un rapide examen du code montre que tous les objets alloués dans cette phase sont utiles à l'application. Par contre, l'analyse détecte également de nombreux problèmes situés dans les méthodes correspondant à la phase principale de calcul : le parcours des listes chaînées provoque l'allocation de plusieurs itérateurs, l'ajout dans ces listes provoque l'allocation de `ListNodes`, et le calcul alloue même de nouvelles listes chaînées pour représenter les transferts de patients. De plus, le programme alloue continuellement de nouveaux `Patients` au cours de la simulation. Tous ces sites d'allocation appartiennent à la même tribu, et risquent de produire un nombre inconnu d'objets qui seront tous placés dans la région de la structure principale.

MST Le programme MST calcule l'arbre couvrant de poids minimal dans un graphe. La classe `Graph` ne possède qu'un seul champ, un tableau dans lequel sont stockés tous les sommets, de type `Vertex`. Le graphe est initialisé avec des distances aléatoires entre chaque paire de sommets, ce que le programme représente par une table de hachage `neighbors` associée à chaque sommet. Dans cette table, les clés sont les autres sommets, et les valeurs associées sont les distances.

L'algorithme parcourt ensuite plusieurs fois l'ensemble de ces sommets pour les insérer dans une liste par ordre de distance à l'origine. L'algorithme est assez complexe, mais il n'alloue presque pas de mémoire. La seule allocation se situe dans une méthode qui doit renvoyer deux informations à la fois comme valeur de retour : un `Vertex` et une valeur entière. Le programme définit donc une classe spécifique `Return` à deux champs pour encapsuler ces deux informations. L'analyse d'anticipation des régions détecte cette allocation qui risque de provoquer une explosion de région. En effet, chaque objet `Return` est connecté avec la structure de données principale, puisqu'il pointe sur un `Vertex`, et sera donc placé dans la même région.

Perimeter Ce programme calcule le périmètre d'une zone contiguë de même couleur dans une image matricielle (*bitmap*) en noir et blanc. L'image est représentée non pas comme un tableau à deux dimensions mais sous la forme d'un arbre 4-aire (*quadtree*). Chaque sommet de cet arbre correspond à une certaine zone rectangulaire dans l'image. Si elle n'est pas de couleur unie, le sommet possède quatre fils correspondant chacun à un quart de la zone, et ainsi de suite. Les sommets internes sont ainsi des objets de type `GreyNode`, et les feuilles de l'arbre sont des objets type `WhiteNode` ou `BlackNode` suivant la couleur des points de la zone qu'ils représentent.

Après la création de la structure de données, le programme fait le calcul de périmètre proprement dit en parcourant récursivement l'arbre qui décrit l'image. Cette phase de l'exécution ne provoque aucune allocation mémoire.

Power Le programme Power résout un problème d'équilibrage des prix dans un modèle de réseau de distribution d'énergie. Le réseau est construit en forme d'arbre autour d'une centrale électrique `Root`, et est constitué de sommets de types `Lateral`, eux-mêmes contenant des `Branch`es. Les feuilles de l'arbre sont des objets de type `Leaf` représentant les consommateurs d'énergie.

Après la phase d'initialisation qui crée la structure de données, le programme entre dans une boucle de calcul pour chercher la répartition idéale des prix de vente de l'énergie à chaque niveau. Les prix sont alors ajustés le long du réseau, en propageant les demandes (sous la forme d'objets de type `Demand`) depuis les consommateurs jusqu'aux entités situées plus haut dans la hiérarchie. Le programme s'exécute ainsi jusqu'à stabilisation du système, allouant de nombreux objets temporaires pour faciliter les calculs.

Dans ce programme, l'analyse d'anticipation des régions n'indique pas de risque d'explosion de région, en dehors de la création initiale de la structure de données.

TreeAdd Ce programme est un simple parcours d'arbre. Durant une phase d'initialisation, il crée récursivement un arbre binaire équilibré, dans lequel chaque sommet reçoit une valeur entière. Dans un second temps, le programme effectue un parcours récursif de cet arbre pour faire l'addition de toutes les valeurs des sommets.

L'exécution de ce programme ne provoque aucune allocation de mémoire après la phase d'initialisation. L'analyse d'anticipation des régions ne détecte donc que la création de l'arbre comme cause possible d'explosion de région.

TSP Le programme TSP résout le problème du voyageur de commerce. Il s'agit de trouver le chemin le plus court possible passant par un certain nombre de villes, chacune repérée par sa position sur un plan. Le programme utilise une structure de données en forme d'arbre binaire dans lequel chaque sommet représente une ville.

Le principe de l'algorithme est basé sur l'heuristique *diviser pour régner*. L'arbre est ainsi divisé récursivement en deux sous-arbres jusqu'à obtenir un problème plus petit qu'une certaine taille fixe. Une approximation du chemin le plus court est alors obtenue dans chaque sous-arbre par la méthode du plus proche voisin. Une fois qu'un chemin est obtenu dans les deux moitiés de l'arbre, le programme fusionne les deux chemins en utilisant une autre heuristique pour obtenir une solution au problème de départ.

Après la phase d'allocation de la structure, ce programme ne fait plus d'allocation avant l'affichage des résultats. Toutes les manipulations sur les chemins se font en manipulant la structure de données. En effet, chaque objet représentant une ville possède également un champ de type pointeur permettant de construire des listes chaînées pour représenter les chemins.

Outre la création de la structure de données, l'analyse d'anticipation des régions détecte dans ce programme une allocation située dans une boucle lors de l'affichage des résultats. Il s'agit de l'allocation d'un `StringBuffer` utilisé pour concaténer des chaînes de caractères avant de les afficher. Dans ce programme, nous avons modifié légèrement le code à l'endroit indiqué par l'analyse afin d'isoler cette allocation de sorte qu'elle soit placée dans une région différente. Avec cette modification, le programme ne comporte plus de risque d'explosion de région.

Voronoi Ce programme génère un ensemble de points $p_1, \ldots, p_n$ placés aléatoirement en deux dimensions, puis calcule une partition de Voronoi du plan par rapport à ces points. Il s'agit de construire autour de chacun de ces points p_i, une *cellule* constituée de l'ensemble des points du plan plus proches de p_i que d'un autre p_j. Les points p_i sont représentés par des objets de type `Vertex`, organisés en un arbre binaire.

L'algorithme est basé sur une triangulation de Delaunay, qui est la construction duale du diagramme de Voronoi. Cette opération divise le plan de proche en proche en triangles dont les sommets sont les p_i. Cette triangulation est implantée ici par un algorithme récursif de type *diviser pour régner* : l'ensemble des points est divisé au hasard jusqu'à ne plus contenir que trois points, qui constituent alors un triangle. Le programme représente les côtés de ces triangles par des objets de type `Edge`. Ces triangulations sont ensuite fusionnées deux à deux jusqu'à obtenir une solution pour l'ensemble du plan. Enfin, le programme construit la division de Voronoi correspondante, par des calculs sur les coordonnées des points et des arêtes.

Ce programme alloue continuellement de la mémoire pour représenter les nouvelles arêtes créées dans la triangulation. Comme pour le programme Health, l'analyse d'anticipation des régions détecte dans le programme Voronoi un nombre très important de sites d'allocation problématiques.

8.3.2.2 Comportement mémoire à l'exécution

Comme indiqué ci-dessus, nous avons comparé pour chaque programme l'occupation de la mémoire au cours de deux exécutions. Dans un premier temps (trait pointillé), le programme est exécuté avec un ramasse-miettes. Les traits verticaux correspondent ainsi aux collectes de la mémoire lorsque l'espace est épuisé. Dans un second temps, le programme est exécuté avec le gestionnaire mémoire en régions (trait plein), et le ramasse-miettes est désactivé.

Power Après la création initiale de la structure, le programme Power entre dans une boucle de calcul qui alloue une importante quantité de mémoire. La courbe pointillée comporte de nombreuses collectes de la part du ramasse-miettes, ce qui indique que les objets de type `Demand` alloués par la simulation ont une durée de vie assez courte.

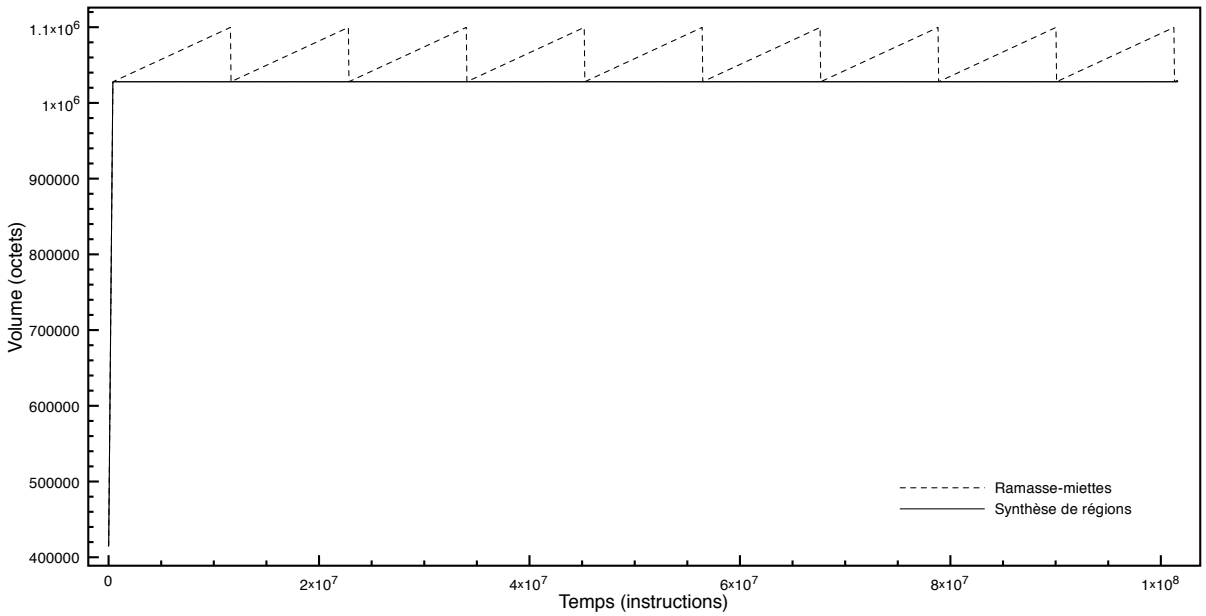


Fig. 8.5 – Occupation mémoire du programme Power.

La version en régions de ce programme présente une occupation mémoire qui semble constante. La figure 8.6, sur laquelle l'échelle est plus réduite, montre que ce n'est pas tout à fait le cas et que la mémoire est régulièrement désallouée au fur et à mesure de l'exécution. À chaque pas de la boucle, une région est créée et les objets temporaires y sont placés. Cette région est détruite avant la prochaine itération, libérant ainsi régulièrement la mémoire. La version du programme utilisant le ramasse-miettes alloue les mêmes objets, mais ils ne sont désalloués que lorsque l'espace est épuisé et qu'un cycle de collecte est déclenché. L'exécution est alors interrompue pendant un temps imprévisible. Au contraire, le programme utilisant les régions n'est pas gêné par des temps de pause grâce au comportement temporel prévisible du gestionnaire mémoire en régions.

BiSort Dans ce programme également, l'utilisation des régions permet de libérer efficacement l'ensemble de la mémoire allouée par le programme. Pendant la première phase d'exécution, correspondant à la création de la structure de données principale, les deux courbes sont identiques.

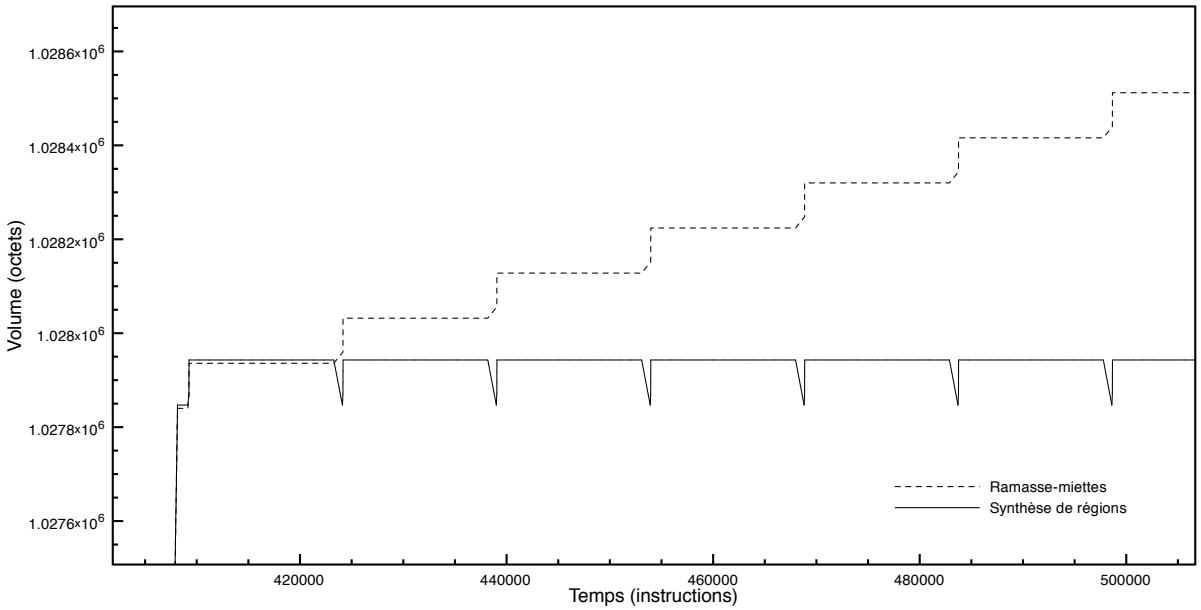


FIG. 8.6 – Occupation mémoire du programme Power (détail).

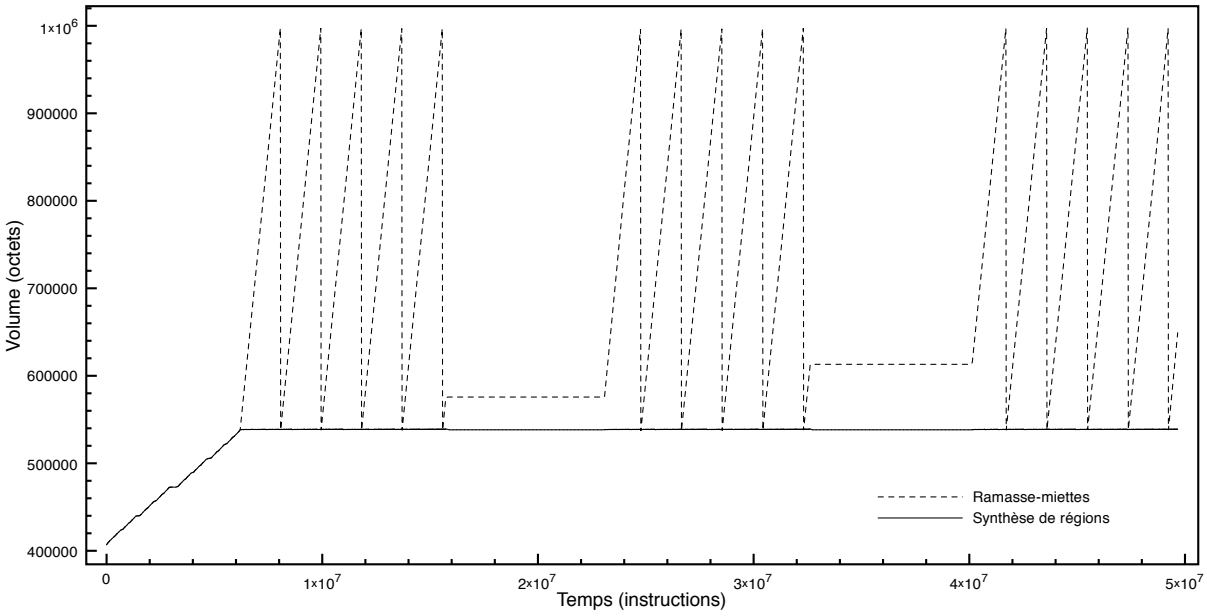


FIG. 8.7 – Occupation mémoire du programme BiSort.

On distingue ensuite plusieurs phases d'exécution pendant lesquelles les deux versions du programme se comportent différemment. Tout d'abord, le programme affiche sur la sortie standard les valeurs contenues dans la structure de données. Cette opération provoque l'allocation d'une grande quantité de mémoire temporaire, surtout sous la forme de `StringBuffer`s et d'objets associés. Le ramasse-miettes recycle tous ces objets une fois que l'espace est entièrement occupé, alors que le gestionnaire mémoire en régions les place dans des régions de faible durée de vie qui sont détruites au plus tôt. L'occupation mémoire est ainsi bien plus uniforme.

Cependant, pendant les phases où le programme n'alloue pas de mémoire, le niveau d'occupation reste constant pour les deux versions du programme. La différence entre les deux courbes n'est alors pas significative, car elle dépend uniquement de la différence d'occupation à la fin de la phase précédente.

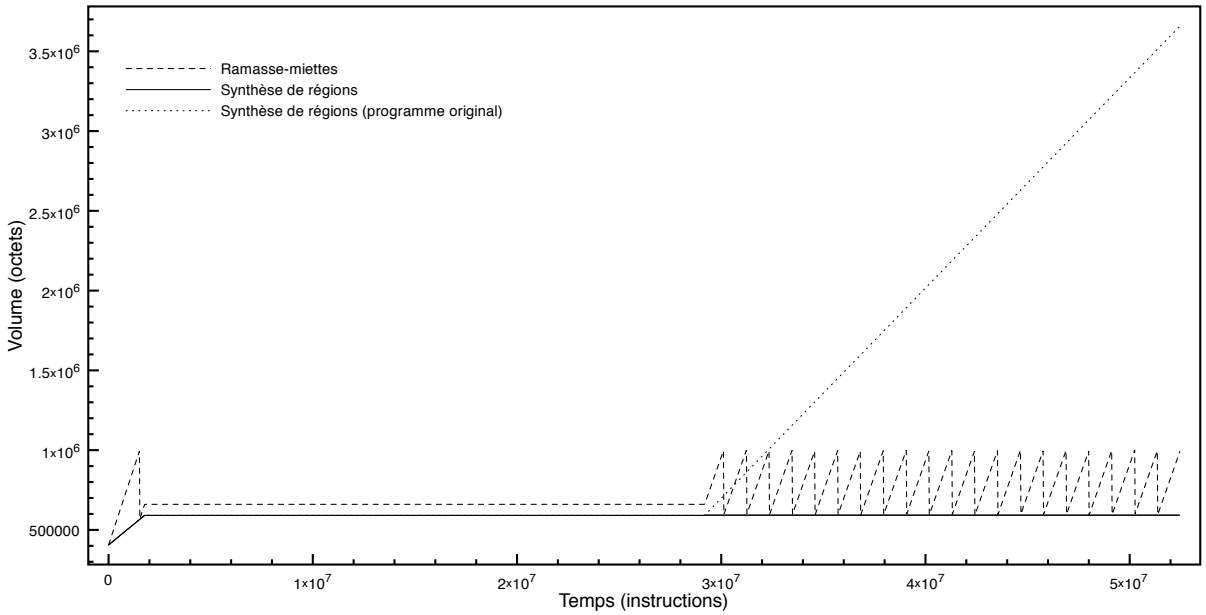


FIG. 8.8 – Occupation mémoire du programme TSP.

TSP On distingue trois phases dans l'exécution de ce programme. Dans un premier temps, la structure de données principale est construite. Cette construction provoque l'allocation d'une quantité importante d'objets temporaires, ce qui force le ramasse-miettes à faire une première collecte de la mémoire pendant cette phase. L'occupation de la mémoire croît plus lentement pour la version en régions car ces objets sont placés dans des régions spécifiques qui sont détruites au plus tôt.

Pendant la phase de calcul, le programme n'alloue pas de mémoire, et les niveaux d'occupation restent constants pour les deux versions du programme. La dernière partie de l'exécution correspond à l'affichage des résultats. Cette phase alloue une quantité importante de mémoire, sous la forme de `StringBuffer`s et d'objets associés. Dans la version originale du programme, cet affichage se situait dans une boucle de la méthode `main`, de telle sorte que tous les objets s'accumulaient dans la même région (courbe en points, sur la figure 8.8). Nous avons modifié légèrement le code pour isoler cet affichage dans une région. L'occupation mémoire est alors globalement constante (courbe en train plein).

Perimeter Dans ce programme, les deux courbes sont encore une fois identiques, comme le montre la figure 8.9. La première phase de l'exécution correspond à la création de l'image, et la seconde au calcul proprement dit. Ce calcul s'exécute sans allouer de mémoire, les deux courbes sont donc horizontales.

TreeAdd Dans ce programme, deux exécutions également ont un profil d'occupation mémoire identique. Comme pour le programme Perimeter, on distingue sur la courbe les deux phases de l'exécution. Dans un premier temps, le programme crée récursivement l'arbre sur lequel il va faire l'addition. Dans un second temps, cet arbre est parcouru et les valeurs présentes sur chaque sommet sont additionnées.

Cette seconde phase ne provoque aucune allocation, donc les deux courbes sont horizontales pendant cette partie de l'allocation.

Em3d Les deux courbes représentant l'occupation mémoire du programme Em3d sont confondues. En effet, ce programme ne provoque pas de collecte du ramasse-miettes, et il n'y a pas non plus de destruction de région. Cependant, il faut ici distinguer les différentes phases de l'exécution,

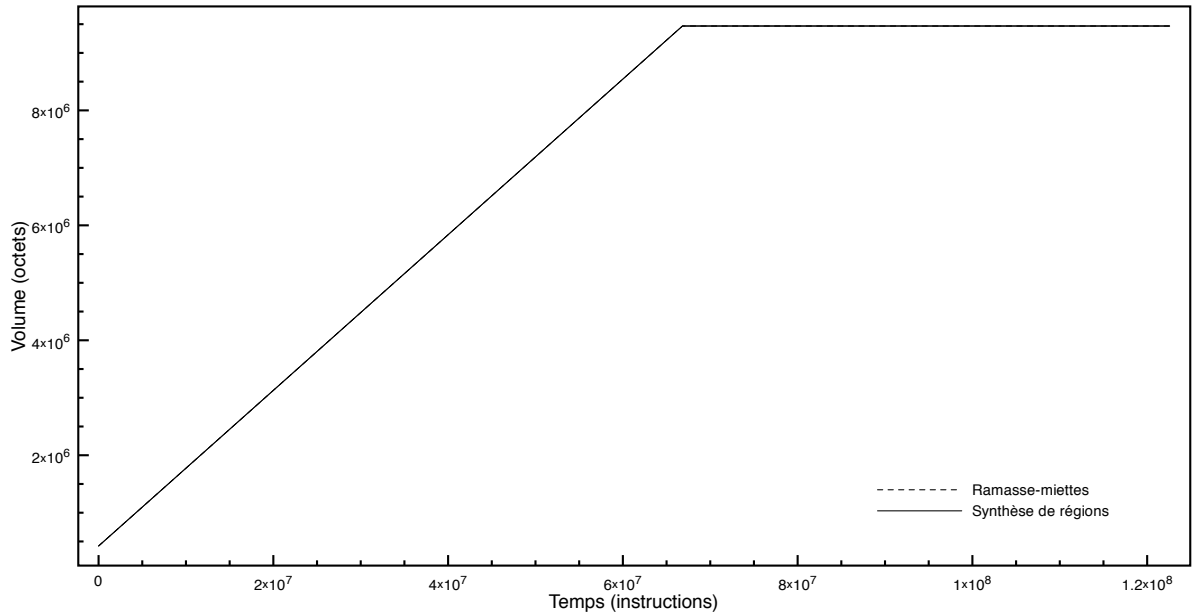


FIG. 8.9 – Occupation mémoire du programme Perimeter

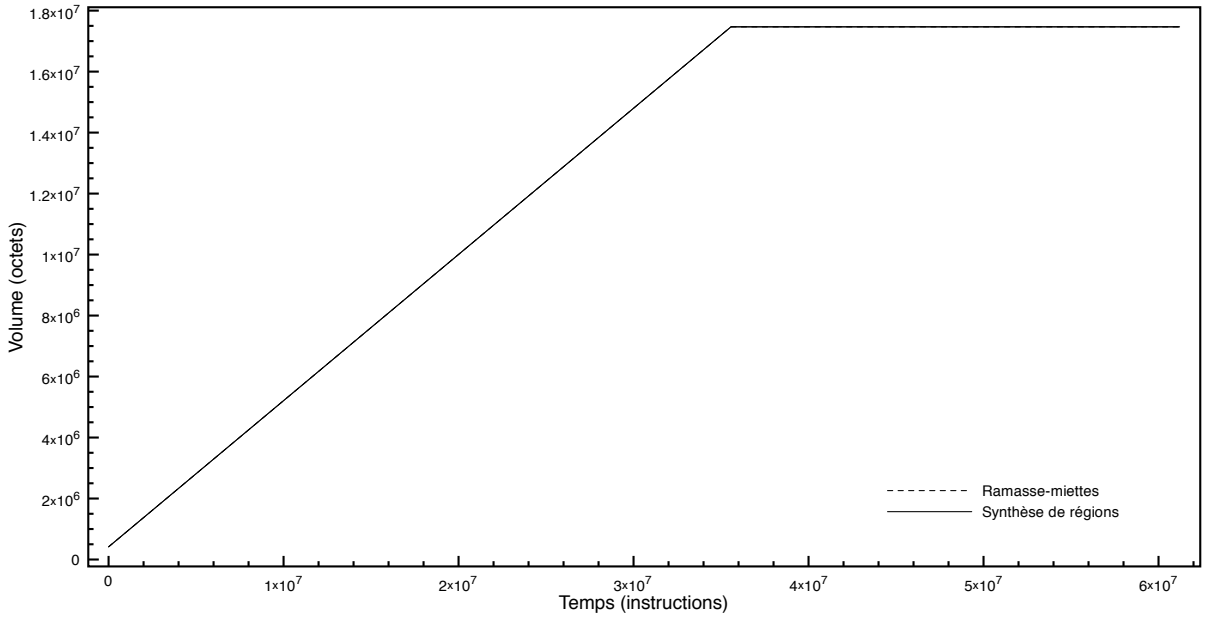


FIG. 8.10 – Occupation mémoire du programme TreeAdd.

qui correspondent respectivement à l'initialisation de la structure de données principale, puis à la simulation proprement dite.

Pendant cette seconde phase, l'occupation mémoire représentée sur la figure 8.11 semble constante. Néanmoins la figure 8.12, sur laquelle l'échelle verticale est plus réduite, confirme les résultats obtenus par l'analyse statique : la phase de simulation provoque bien une allocation régulière de mémoire, et conduit donc à une occupation croissante de l'espace. Si l'exécution se prolongeait, le ramasse-miettes serait déclenché à un certain moment pour recycler l'espace inutilisé, alors que la version en régions ne libérerait jamais de mémoire.

MST Les résultats obtenus pour ce programme sont similaires à ceux obtenus pour le programme Em3d. Les deux exécutions présentent un comportement mémoire semblable, car le programme

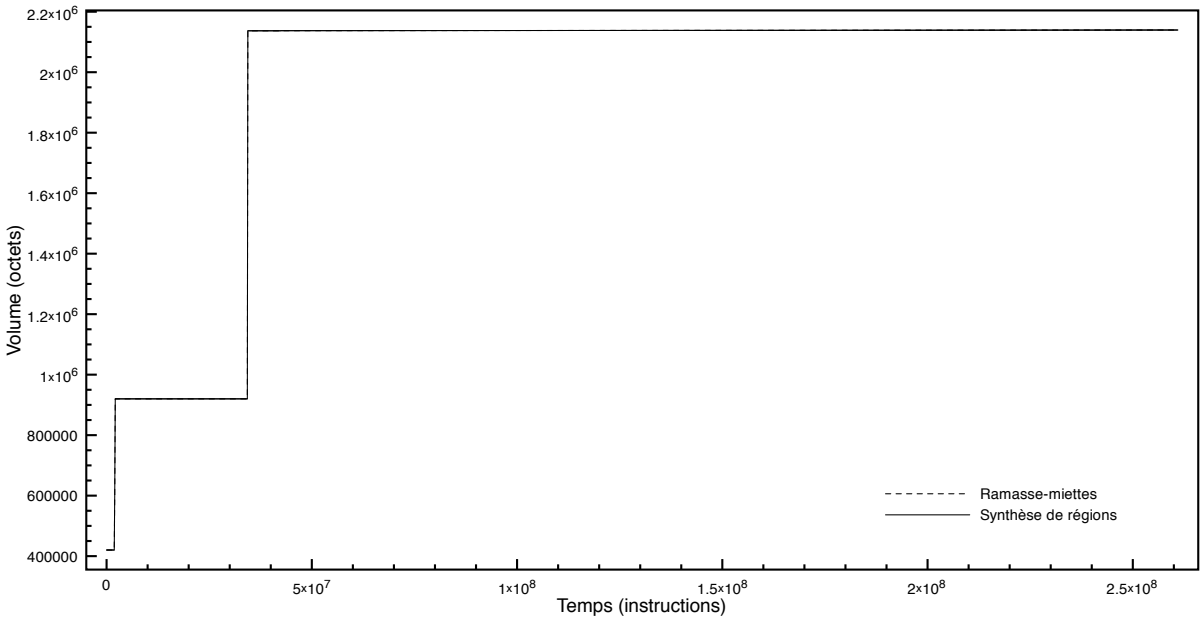


FIG. 8.11 – Occupation mémoire du programme Em3d.

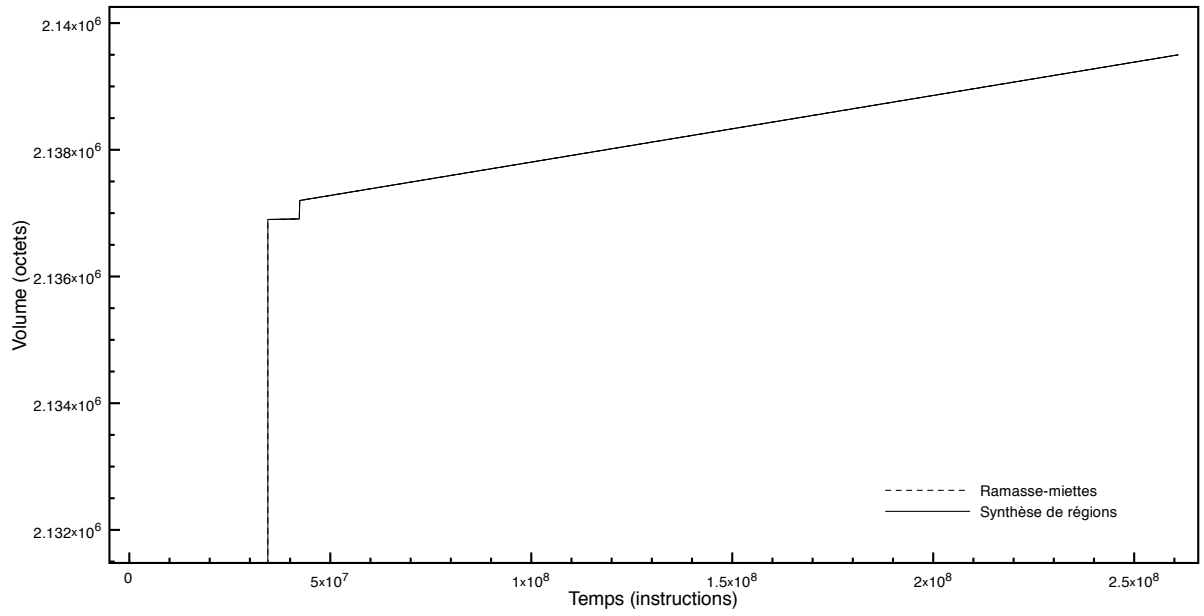


FIG. 8.12 – Occupation mémoire du programme Em3d (détail).

ne déclenche aucune collecte du ramasse-miettes. Cependant, la phase de calcul alloue ici aussi de la mémoire, et la courbe correspondante n'est pas tout à fait horizontale.

Les objets qui s'accumulent dans la région de la structure de données principale sont les objets `Return` utilisés par le programme pour renvoyer simultanément une référence et une valeur entière.

BH Dans ce programme, le risque d'explosion de région détecté par l'analyse statique se manifeste clairement à l'exécution. La simulation s'exécute globalement en espace constant, comme le montrent les niveaux d'occupation mémoire atteints après chaque collecte. Pourtant, lorsque le programme est exécuté en régions, il occupe de plus en plus de mémoire, jusqu'à l'épuiser complètement si l'exécution devait se prolonger.

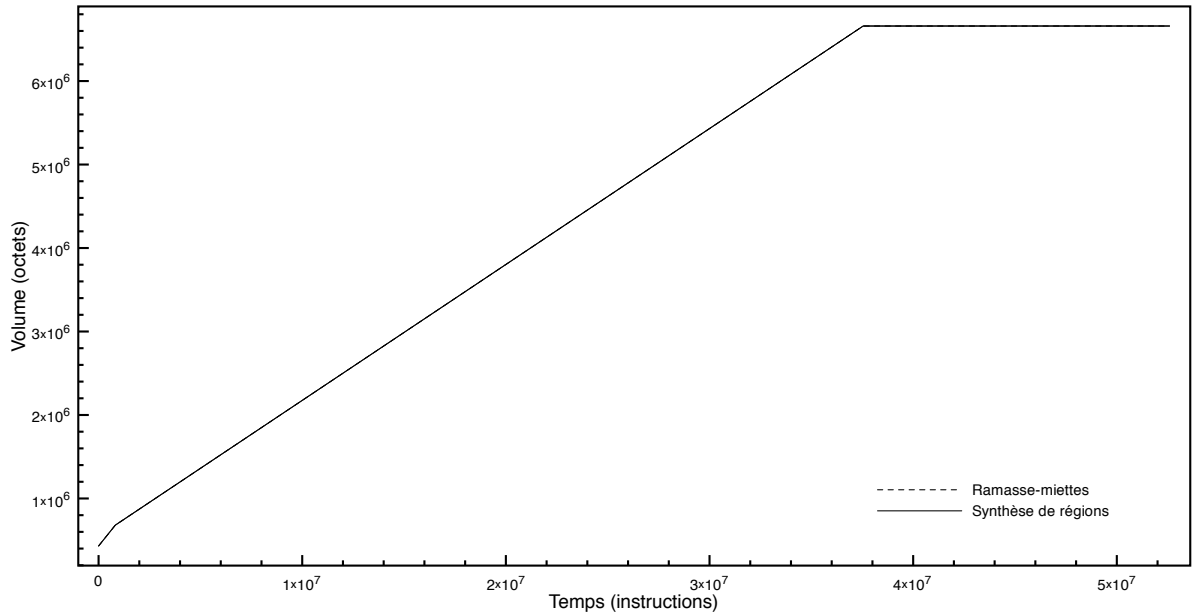


FIG. 8.13 – Occupation mémoire du programme MST

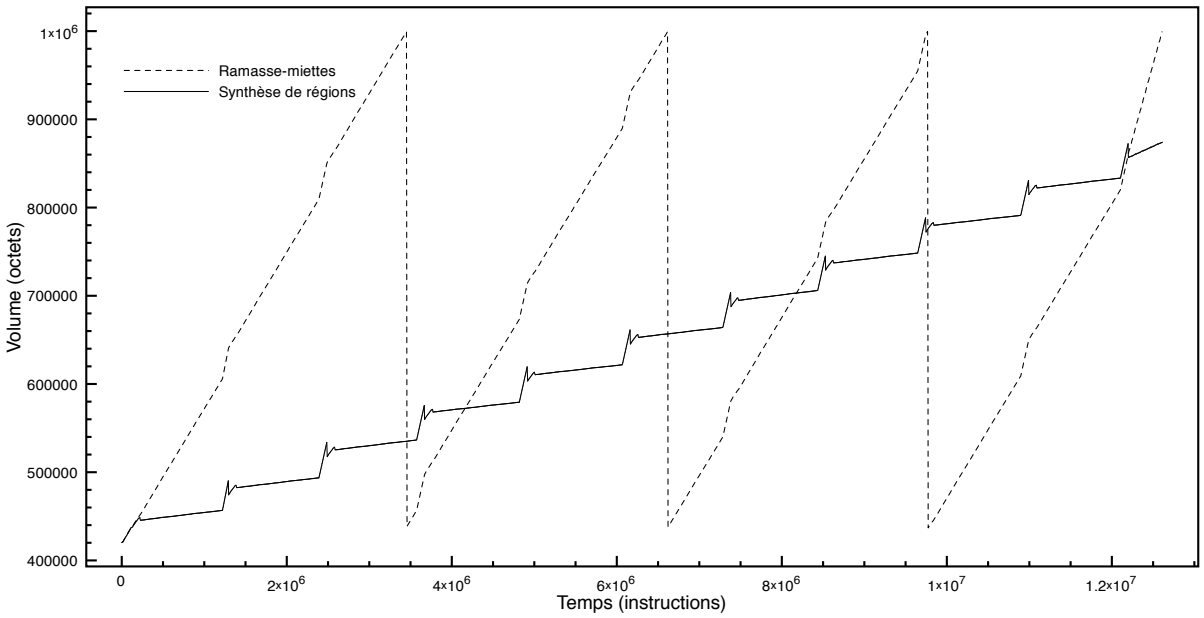


FIG. 8.14 – Occupation mémoire du programme BH.

La courbe des régions a une pente moins importante que la courbe du ramasse-miettes, ce qui signifie qu'une certaine partie de la mémoire allouée est rapidement libérée lors de la destruction de régions. Le reste de la mémoire est cependant alloué dans la région de la structure principale, et s'accumule au fur et à mesure de l'exécution.

Health Dans ce programme, l'explosion de région prévue par l'analyse statique se produit de façon spectaculaire, comme le montre la figure 8.15. Alors que le ramasse-miettes est souvent déclenché et recycle à chaque fois une certaine partie de l'espace, le gestionnaire mémoire en régions place tous les objets alloués par la simulation dans la région de la structure de données principale.

Contrairement aux autres programmes rencontrés jusqu'ici, aucune destruction de région ne

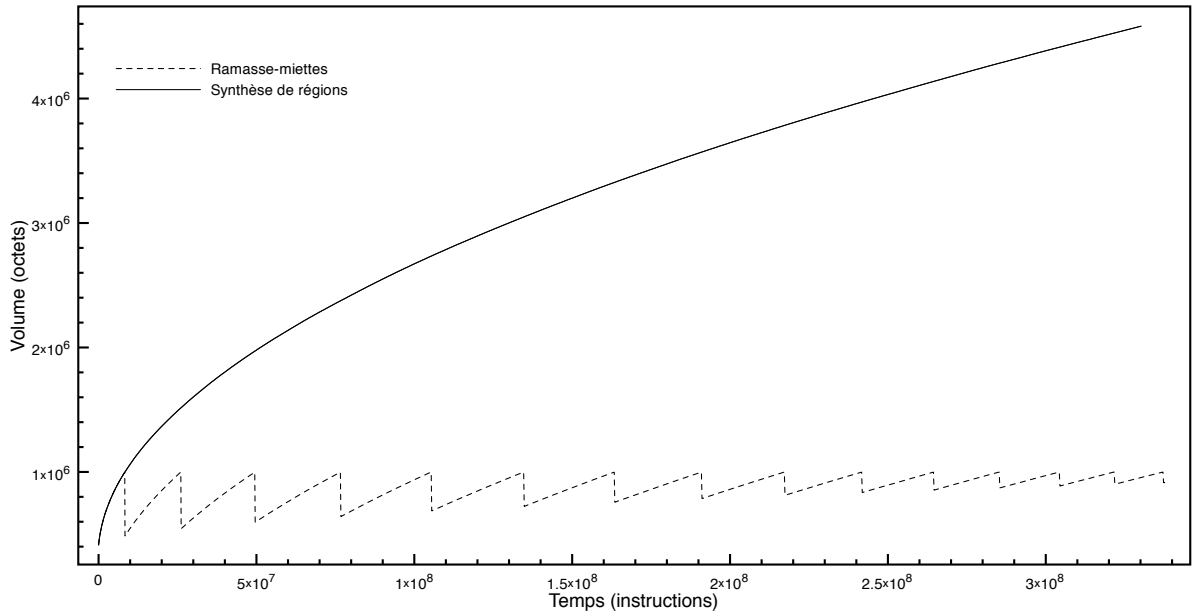


FIG. 8.15 – Occupation mémoire du programme Health.

vient ralentir le phénomène. Ce problème est dû principalement au fait que le programme Health ne respecte pas du tout notre *hypothèse générationnelle* (cf section 5.2) : les objets alloués par le programme ont des durées de vie très différentes, bien qu'ils soient tous connectés à un moment avec la structure de données principale. La gestion mémoire en régions ne convient donc pas bien à ce genre de programme, car il n'est pas possible de séparer par durée de vie les objets dans des régions différentes.

Voronoi Conformément aux résultats de l'analyse d'anticipation des régions, ce programme provoque également une importante explosion de région, représentée sur la figure 8.16. Après la phase d'initialisation de la structure de données principale, pendant laquelle les deux courbes sont identiques, le programme continue à allouer une grande quantité de mémoire. Alors qu'une certaine partie de cette mémoire est recyclée par le ramasse-miettes, le gestionnaire mémoire en régions place tous les objets dans la même région, puisqu'ils sont connectés à un moment ou à un autre.

Comme pour le programme Health, ce comportement est dû au fait que le programme Voronoi ne respecte pas du tout notre hypothèse quant aux durées de vie des objets. Tout au long de l'exécution, de nouveaux objets sont attachés à la structure de données principale, d'autres en sont éliminés, et ils ont finalement des durées de vie très différentes malgré leur connexion. Cet usage assez irrégulier de la mémoire peut se vérifier sur la courbe obtenue avec le ramasse-miettes.

8.3.2.3 Discussion

Phases d'exécution On peut remarquer que tous ces programmes ont une structure assez similaire. Dans une phase d'initialisation, une structure de données importante est allouée, qui servira de base au reste du calcul. Cette structure est vivante tout au long de l'exécution, et se trouve naturellement placée dans une même région. Dans un second temps, le programme exécute son algorithme principal, en se basant sur cette structure de données. Pour certains programmes, l'exécution se termine par une dernière phase pendant laquelle les résultats sont affichés.

Du point de vue de l'utilisation de Java dans la programmation de systèmes embarqués, la phase qui nous intéresse le plus est la phase principale, la plus significative de la tâche du système, et la plus susceptible d'être soumise à des contraintes temps-réel. Ce fractionnement

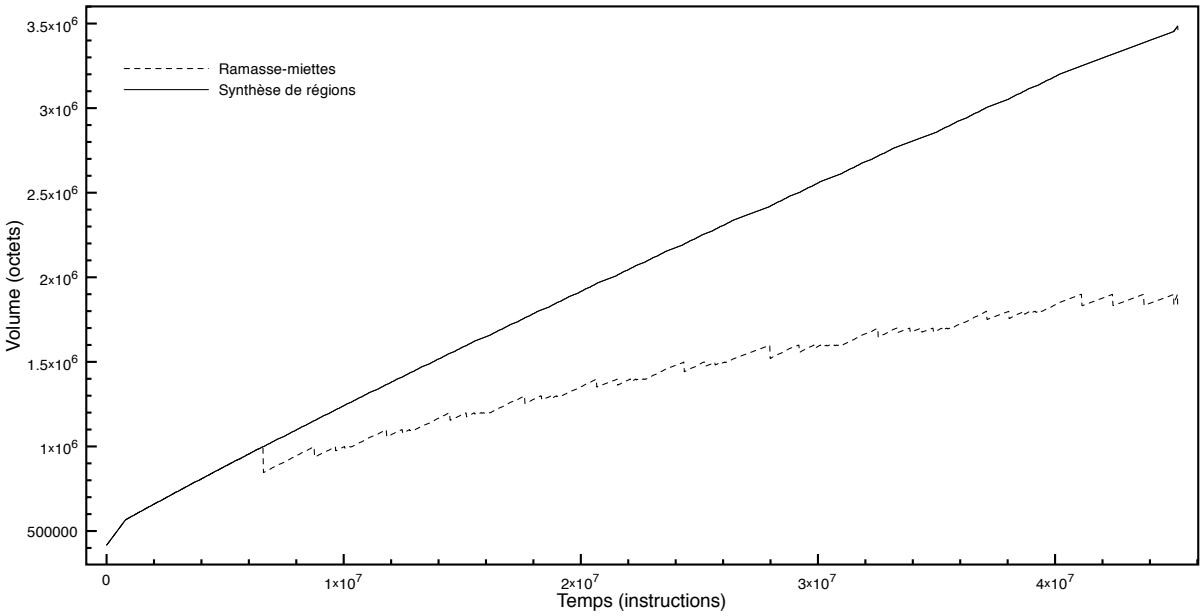


FIG. 8.16 – Occupation mémoire du programme Voronoi.

en phases d'exécution est d'ailleurs le même que celui préconisé par le profil Ravenscar Java [KWK02]. Pendant la phase d'*initialisation*, l'application est autorisée à allouer des structures de données dans le tas. Pendant la phase *mission*, par contre, elle doit se contenter d'utiliser les régions RTSJ (*Scoped Memory*).

Les avertissements annoncés par l'analyse statique correspondent donc effectivement à des allocations, mais le programmeur peut les ignorer sans risque car ces allocations sont situées dans la phase d'initialisation de l'application.

Comportement mémoire Lorsque la phase principale n'alloue pas de mémoire (comme dans les programmes TreeAdd et Perimeter), notre approche n'a pas d'impact sur le comportement du programme car le gestionnaire mémoire n'est pas sollicité. Dans d'autres programmes, comme Power, Bisort, ou TSP, la synthèse de régions désalloue la mémoire dès qu'elle devient inutile, et permet de se passer du ramasse-miettes. De plus, le comportement temporel de notre gestionnaire mémoire donne alors au programme une exécution plus prévisible, qui n'est plus interrompue par les temps de pause associés aux collectes. Dans l'ensemble de ces cas *favorables*, la gestion mémoire en régions permet ainsi de rendre les temps d'exécution des opérations mémoire prévisibles, ce qui est nécessaire dans une optique temps-réel.

Dans les programmes Em3d, MST, et BH, la synthèse de régions ne parvient pas à libérer toute la mémoire allouée par le calcul. Au cours de l'exécution, un certain nombre de sites d'allocation produisent des objets qui sont placés dans la même région que la structure de données principale, et ne sont donc pas désalloués. Dans ces cas plus *défavorables*, le code nécessiterait d'être modifié pour pouvoir être exécuté sans ramasse-miettes. L'analyse d'anticipation des régions indique au programmeur quels sont les endroits du code d'où proviennent les objets en question. Dans le programmes Em3d et dans MST, on peut néanmoins remarquer que ces objets qui s'accumulent dans la mémoire sont des objets temporaires, par exemple des itérateurs. On pourrait donc combiner notre approche avec une analyse d'échappement pour allouer ce genre d'objets sur la pile, bien qu'ils soient connectés au reste de la structure (cf section 7.5.2). Dans le programme BH, il y a en plus des objets qui sont référencés un certain temps par la structure de données principale, et donc cette technique ne suffirait pas. Cependant, on peut ici envisager de combiner l'allocation en régions avec un ramasse-miettes à comptage de références pour recycler ces objets, comme le montre la figure 8.20 page 139.

Il reste encore certains cas *incurables* comme les programmes Health ou Voronoi, dans lesquels les régions n'ont pas du tout l'effet escompté. Le programme alloue continuellement de la mémoire, ajoutant ou retirant des objets dans la structure de données principale, dont la région n'est jamais détruite. La connexion entre des objets n'a alors pas de corrélation forte avec leur durée de vie. Pour ce genre de programmes, qui ne respectent pas du tout notre *hypothèse générationnelle* (cf section 5.2), la gestion mémoire en régions est de toute façon mal adaptée. Même avec une analyse plus sophistiquée, Cherem et Rugina [CR04] obtiennent des résultats similaires aux nôtres pour ces programmes.

8.3.3 Étude de cas : le décodeur MP3 JLayer

Pour valider les observations présentées ci-dessus, nous avons mis en pratique notre approche sur une application plus réaliste. Nous avons choisi pour cela le décodeur MP3 JLayer. Il s'agit d'un programme écrit en Java Standard disponible librement sur Internet[1]. Les traitements multimédia sont en général des tâches assez complexes, de plus en plus présentes dans les systèmes embarqués. L'utilisation de Java pour l'implantation des algorithmes associés apporte des bénéfices importants au cycle de développement de ces systèmes, notamment grâce à la facilité de développement et à la portabilité du code. De plus, la lecture d'un fichier MP3 est une tâche temps-réel souple, qui ne supporterait pas bien les saccades imposées à l'exécution par les cycles de collecte d'un ramasse-miettes. À ce titre, le décodeur MP3 nous a paru une application pertinente pour évaluer notre approche.

L'application elle-même consiste en 329 méthodes et 16122 instructions, et l'ensemble du code analysé comporte 1252 méthodes et 29828 instructions. Pour ce programme, l'ensemble de l'analyse statique prend environ 4 secondes.

Structure du programme et analyse statique Bien que d'une taille nettement plus importante que les programmes de la suite JOlden, le décodeur MP3 JLayer est également organisé en deux phases successives, initialisation puis décodage. Cependant, la structure du code ne reflète pas exactement les phases d'exécution. Si le décodage du fichier MP3 est basé effectivement sur une boucle principale dans laquelle chaque trame est décodée séquentiellement, la phase d'initialisation n'est pas clairement séparée de la boucle. Cette structure est reprise, de façon très simplifiée, sur la figure 8.17.

Le programme alloue un objet `Player` et lui demande de décoder un certain fichier. L'initialisation des structures de données ne se fait pas dans le constructeur, mais dans la méthode de lecture, car elles dépendent de certains détails du fichier. De plus, certains de ces objets sont alloués *à l'intérieur* de la boucle, comme l'objet `output`. Cette manière de programmer conduit l'analyse d'anticipation des régions à soupçonner de nombreux risques d'explosion de région. Cependant, un examen manuel du code nous a permis de vérifier que les sites d'allocation en question ne seraient exécutés qu'une seule fois. En effet, lors des itérations suivantes de la boucle, la condition `output == null` ne sera plus vérifiée.

Comportement mémoire à l'exécution La figure 8.18 montre le profil d'occupation mémoire du programme lors du décodage d'un court fichier MP3. On distingue la phase d'initialisation pendant laquelle une grande quantité de mémoire est allouée, puis la phase de décodage elle-même.

La courbe correspondant à l'exécution avec le ramasse-miettes croit régulièrement, mais les cycles de collecte ramènent l'occupation mémoire à un niveau constant. La courbe en trait plein représente l'exécution avec le gestionnaire mémoire en régions. On distingue des petits pics réguliers qui correspondent au décodage de chaque trame.

La figure 8.19 est un agrandissement de la précédente, située juste au moment de la première collecte du ramasse-miettes. On y distingue plus clairement la mémoire alloué par le décodage

[1] `http://www.javazoom.net/javalayer/javalayer.html`

```
main()
{
  player = new Player();

  player.play("file.mp3");
}

class Player
{
  play(String filename)
  {
    OutputBuffer output = null;

    File file = new File(filename);
    Decoder decoder = new Decoder(file);
    Bitstream stream = new Bitstream(file);

    while( true )
    {
       if( output == null )
       {
         output = new OutputBuffer();
       }

       Frame frame=stream.readFrame();
       if( frame == null )
          break;

       decoder.decodeFrame(frame,output);

       ...
  }
}
```

FIG. 8.17 – Structure simplifiée du lecteur MP3 JLayer.

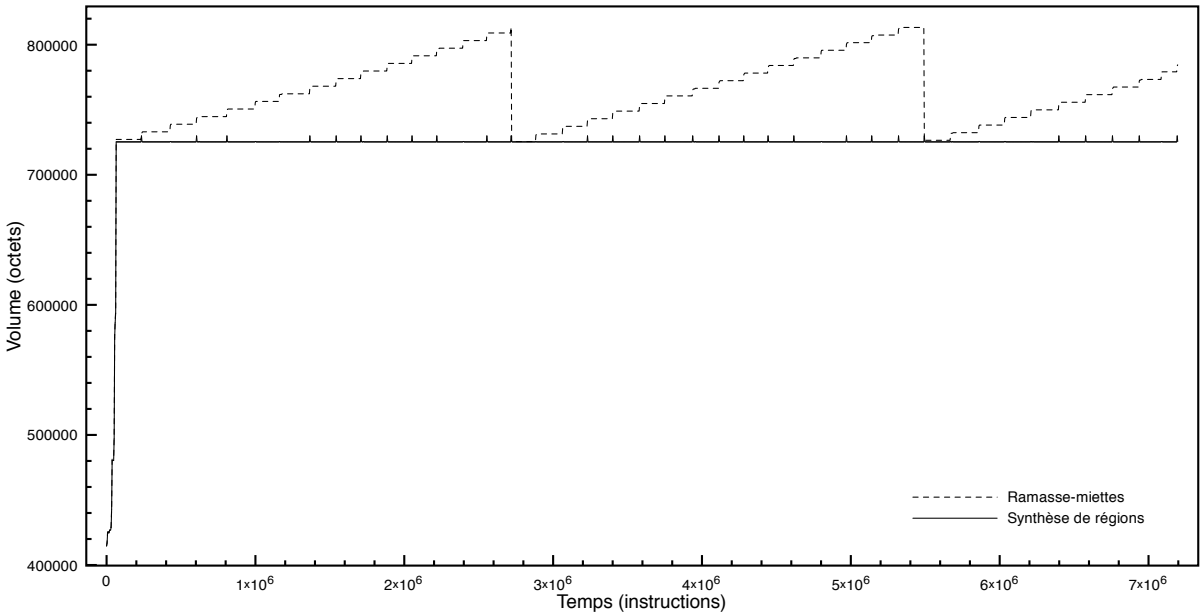


FIG. 8.18 – Occupation mémoire du décodeur MP3 JLayer.

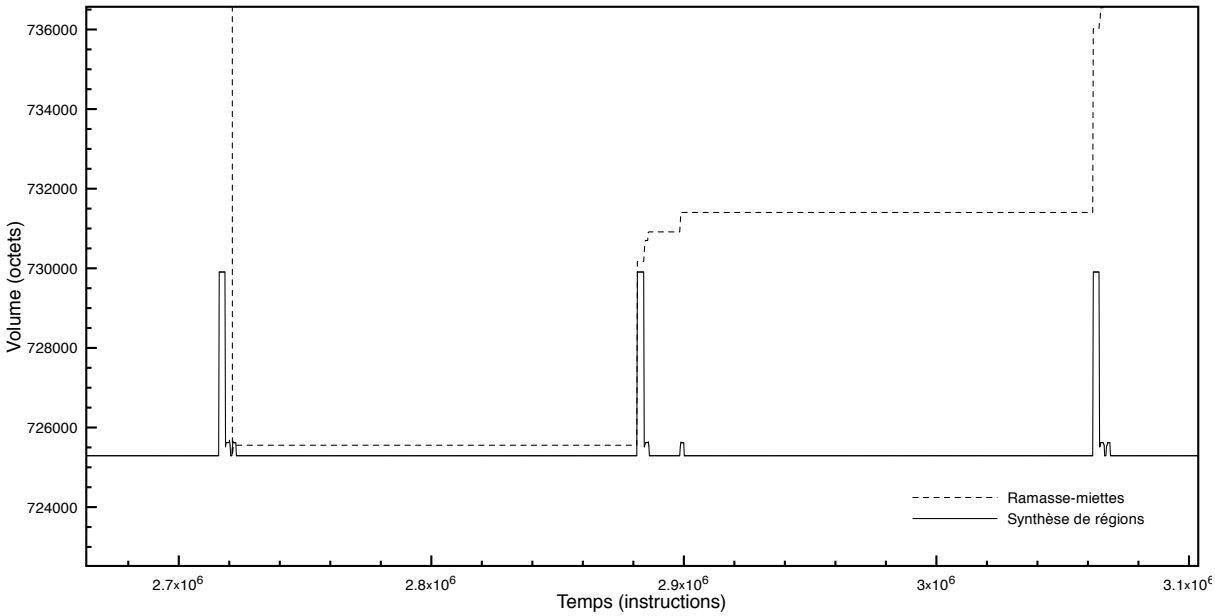


FIG. 8.19 – Occupation mémoire du décodeur MP3 JLayer (détail).

de chaque trame, et libérée au plus tôt par le gestionnaire mémoire en régions. La version du programme qui utilise le ramasse-miettes alloue la même quantité de mémoire, mais elle n'est pas désallouée avant la prochaine collecte.

L'utilisation des régions s'avère donc concluante pour ce programme. L'analyse d'anticipation des régions donne plusieurs avertissements à propos de certains sites d'allocation, mais un examen du code nous permet de conclure qu'il s'agit de structures de données qui ne sont allouées qu'à la première itération de la boucle principale, et ne risquent pas de provoquer d'explosion de région. En effet, l'exécution du programme avec le gestionnaire mémoire en régions se comporte bien de la façon attendue. La mémoire est désallouée dès que possible, et le comportement temporel du gestionnaire mémoire en régions nous garantit l'absence de temps de pause imprévisibles dans l'exécution.

8.4 Combinaison avec un ramasse-miettes à comptage de références

Comme nous l'avons vu dans la section 7.5.3, l'analyse d'interférence de pointeurs a tendance à grouper dans une même régions des objets qui pourraient être efficacement recyclés par un ramasse-miettes à comptage de références. Pour explorer cette voie, nous avons implanté dans la machine virtuelle de Jits un tel ramasse-miettes, et modifié le gestionnaire mémoire pour pouvoir allouer séparément certains objets, qui ne sont alors plus gérés par le système des régions mais uniquement par le ramasse-miettes [Ber07].

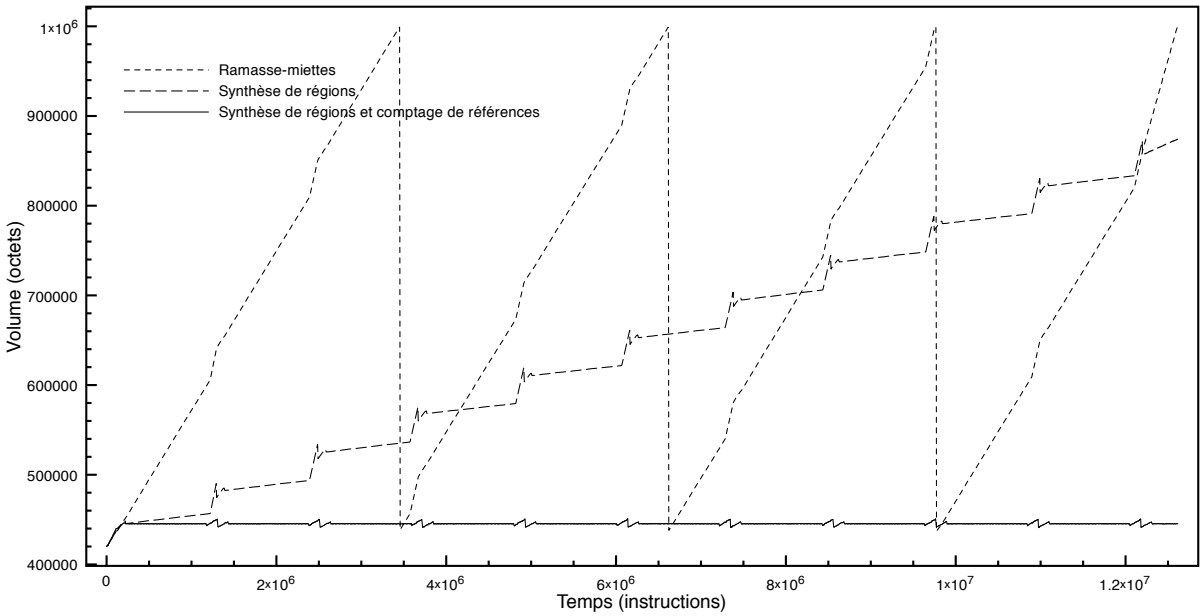


Fig. 8.20 – Occupation mémoire du programme BH.

Par exemple, pour le programme BH, dans lequel une partie des objets alloués par le calcul s'accumulaient dans une région durable, l'utilisation de cette technique a permis de recycler l'espace au fur et à mesure, et ainsi d'éviter l'explosion de région. Ces résultats sont illustrés sur la figure 8.20. Les deux courbes pointillées représentent respectivement l'exécution du programme avec le ramasse-miettes par défaut et avec le gestionnaire mémoire en régions. La courbe en trait plein représente l'occupation mémoire mesurée lors de l'exécution du programme avec un gestionnaire mémoire combinant allocation en régions pour la plupart des objets, et comptage de références pour les sites d'allocation identifiés comme problématiques par l'analyse d'anticipation des régions.

Ces résultats sont encourageants, car ce gestionnaire mémoire mixte permet pour ce programmer de recycler l'ensemble de la mémoire allouée par le calcul, évitant ainsi l'explosion de région. Cependant, la combinaison de ces deux techniques de gestion mémoire remet en question l'invariant associé aux régions. Des objets situés dans une région pointent vers des objets situés dans la zone gérée par le ramasse-miettes, et inversement. Le gestionnaire mémoire en régions n'est alors plus assuré de détruire uniquement des régions ne contenant aucun objet vivant. De plus, un ramasse-miettes à comptage de référence ne peut pas détecter la mort d'une structure de données contenant un chemin cyclique, il faut donc s'assurer que les objets confiés au ramasse-miettes ne peuvent pas former de tels chemins.

Il reste donc à éclaircir les questions soulevées ci-dessus si l'on veut pouvoir profiter pleinement des bénéfices apportés par la combinaison d'un gestionnaire mémoire en régions et d'un ramasse-miettes à comptage de références.

8.5 Conclusion

Nous avons vu dans ce chapitre les différents aspects de notre approche liés à l'implantation. Réalisée à l'aide de l'infrastructure Soot, l'analyse statique s'exécute assez rapidement pour être intégrée sans problème dans le cycle de développement. Ce point est essentiel, car notre analyse est destinée à être utilisée interactivement par le programmeur, tout au long de l'écriture du programme. De plus, nous avons vérifié que même sur des programmes déjà écrits, la synthèse de régions produit des résultats satisfaisants dans la majorité des cas. Lorsque l'analyse détecte un risque d'explosion de région, elle indique l'emplacement et le contexte des sites d'allocation, afin de permettre au développeur de prendre une décision. Par contre, si la logique de l'application ne respecte pas du tout notre hypothèse générationnelle, l'analyse indique un nombre de sites très important, ce qui montre que la gestion mémoire en régions ne convient pas pour ce type de programme.

Le gestionnaire mémoire a été implanté dans la machine virtuelle de l'environnement Jits. Grâce à un modèle mémoire *ad hoc*, nous avons implanté les différentes opérations primitives en régions de telle sorte que leur temps d'exécution au pire cas soit constant. Ainsi, le programme peut s'exécuter d'une façon prévisible, puisque la présence d'allocation mémoire dans le programme n'est plus synonyme de temps d'exécution au pire cas irréalistes. D'autre part, les expérimentations nous ont permis de vérifier que le comportement mémoire obtenu était bien conforme aux prévisions de l'analyse statique. Pour la majorité des programmes, la synthèse de régions parvient à désallouer la mémoire aussi bien, ou mieux que le ramasse-miettes. Le comportement mémoire et le comportement temporel du programme sont alors tous les deux prévisibles, ce qui est l'objectif de ce travail. Pour d'autres programmes, dans lesquels les structures de données utilisées ne respectent pas notre hypothèse de départ quant aux durées de vie, on observe à l'exécution l'explosion de région prévue par l'analyse statique.

Nous avons également étudié la possibilité de combiner cette approche avec l'utilisation d'un ramasse-miettes à comptage de références, de façon à réduire les risques d'explosion de région. Les résultats obtenus sont encourageants, mais soulèvent plusieurs questions théoriques qu'il faudrait résoudre pour pouvoir généraliser cette idée.

Ces expérimentations ont ainsi confirmé qu'il est possible de concilier comportement temporel prévisible et allocation de mémoire dynamique dans un programme écrit en Java Standard. Cet aspect est une condition nécessaire pour favoriser l'utilisation de Java Standard dans le monde de l'embarqué et du temps-réel.

Chapitre 9

Conclusion et perspectives

9.1 Bilan

Nous nous sommes intéressés dans ce travail à la problématique de la gestion mémoire temps-réel dans les systèmes Java embarqués. La gestion automatique de la mémoire, à cause de ses implantations courantes, a la réputation d'avoir un comportement temporel non-déterministe, et donc est souvent laissée de côté lors du développement d'application temps-réel. Plusieurs approches de ce problème ont déjà été proposées, comme les ramasse-miettes temps-réel [Sie99, BCR03], ou la programmation RTSJ [BBD+00], mais elles sont trop contraignantes à notre avis pour le programmeur. L'utilisation d'un ramasse-miettes temps-réel nécessite une modélisation très fine du comportement mémoire de l'application, ce qui est difficile à obtenir en pratique. La gestion mémoire dans la RTSJ est entièrement à la charge du programmeur, qui doit se préoccuper de la durée de vie et de la taille occupée par chacun des objets alloués par l'application. Elle est donc très difficile à utiliser en pratique. Notre objectif est de permettre la gestion automatique de la mémoire en Java dans un contexte embarqué et temps-réel, tout en restant le plus proche possible des manières de programmer habituelles.

9.1.1 Résumé des contributions

Nous avons choisi comme plate-forme expérimentale l'environnement Jits [MCG05], dont les objectifs sont également de permettre la programmation Java Standard dans le contexte des systèmes embarqués aux ressources très contraintes. Grâce notamment à un ensemble d'outils de spécialisation du système, Jits permet d'embarquer l'application et la machine virtuelle Java sur des architectures très limitées.

Afin de bénéficier d'un comportement temporel prévisible, nous proposons d'utiliser un gestionnaire mémoire en régions [Tof98]. Nous avons implanté un tel gestionnaire mémoire dans la machine virtuelle de Jits, et décomposé les différentes opérations mémoire (création de région, allocation, destruction de région) en tâches de temps d'exécution prévisibles. La taille des régions n'est pas fixée à l'avance car elles sont implantées sous la forme d'une liste chaînée de pages. Le gestionnaire mémoire peut donc ajouter à la demande de nouvelles pages à une région.

Pour déterminer comment placer les objets en régions, nous présentons une analyse statique appelée analyse d'interférence de pointeurs. L'objectif de cette analyse est de déterminer une sur-approximation des connexions entre les objets de façon à grouper les objets connectés ensemble dans une même région. Cette idée se base sur l'hypothèse générationnelle [HHDH02], selon laquelle il y a souvent une forte corrélation entre les durées de vie des objets et les connexions qui les relient. Cette analyse est conçue pour être utilisée de façon interactive par le programmeur, elle est donc de nature simple, et s'exécute en un temps très court.

En effet, pour certains programmes, la gestion mémoire en régions conduit à un comportement insatisfaisant dans lequel la mémoire n'est pas recyclée à un rythme suffisant, parfois même

aucune mémoire n'est libérée au cours de l'exécution. Pour détecter ce comportement le plus tôt possible dans le cycle de développement, nous proposons une seconde analyse statique appelée analyse d'anticipation des régions, destinée à avertir le programmeur des risques potentiels dans son code, et ainsi lui permettre de modifier son programme.

Pour l'aider à comprendre mieux le comportement mémoire de son programme, nous basons ces deux analyses sur une analyse de pointeurs intra-procédurale appelée analyse de forme locale, qui représente sous la forme d'un graphe les objets manipulés par chaque méthode du programme. Ainsi, le programmeur peut visualiser l'interaction entre les différents objets plus facilement qu'en examinant le texte du programme.

Ce mode de développement interactif s'est révélé utilisable en pratique, car il nécessite peu de changement dans les manières de programmer habituelles en Java. Notre objectif est en effet de s'éloigner le moins possible de la programmation Java Standard, dans une optique d'efficacité de développement.

Nous avons évalué cette approche sur les programmes du banc d'essais JOlden [CM01], ainsi que sur une application plus réaliste, le décodeur MP3 JLayer. Nous avons ainsi pu valider nos hypothèses de départ : pour les programmes qui respectent l'hypothèse générationnelle, la synthèse de régions permet de recycler la mémoire au fur et à mesure de l'exécution, tout en offrant un comportement temporel prévisible.

Dans ces conditions, la présence d'allocation mémoire dans le programme n'est plus rédhibitoire à l'étude des temps d'exécution au pire cas. Notre travail permet donc de lever une des restrictions quant à l'usage de Java dans un contexte embarqué et temps-réel.

9.1.2 Limites de l'approche

Notre proposition pour la gestion mémoire souffre cependant de certaines limites. L'analyse d'interférence de pointeurs, bien que basée sur l'analyse de forme locale qui est elle intra-procédurale, nécessite de disposer de l'ensemble du programme et des bibliothèques pour donner un résultat correct. Cette caractéristique empêche l'extensibilité du logiciel embarqué, et notamment le chargement dynamique de classes. Pour certains systèmes embarqués au cycle de vie ouvert, cette limitation est significative. Afin de permettre le chargement de classes après l'embarquement du logiciel, une solution serait d'utiliser un mécanisme de vérification [GHSR06] sur le système embarqué pour s'assurer que les nouvelles classes chargées ont un comportement compatible avec le reste du système.

Une autre limitation de ce travail est son impact sur la manière de programmer imposée au développeur. Bien que visant à minimiser cet impact, notre analyse n'est pas assez précise pour gérer correctement certaines formes de structures de données. Si le cas des objets auxiliaires connectés à une structure de données principale, comme les itérateurs, pourrait être réglé en combinant notre allocation en régions avec une analyse d'échappement, certains motifs de programmation sont beaucoup plus difficiles à caractériser. Par exemple, les structures de données dans lesquelles on ajoute et on retire sans cesse de nouveaux objets sont très difficiles à gérer en régions.

9.2 Perspectives

À l'issue de ce travail, un certain nombre de problématiques nous semblent mériter d'être approfondies. En premier lieu, l'intégration de notre approche dans le cycle de développement peut être poussée plus loin, par exemple en incorporant les algorithmes d'analyse statique et les retours d'information dans un environnement de développement interactif (IDE). Les diverses représentations du programme (graphes de forme locale, familles, tribus) seraient ainsi plus faciles à visualiser pour le programmeur qui serait alors plus efficace.

Un autre point intéressant est l'étude des modèles de programmation compatibles avec

l'hypothèse générationnelle sur laquelle s'appuie notre approche. En effet, si ce travail vise à rester au plus près des manières de programmer habituelles en Java, on a vu que certains programmes se révélaient incompatibles avec la gestion mémoire en régions. Dégager certains patrons de conception appropriés aux régions engendrerait une plus grande efficacité de programmation.

Pour maîtriser les programmes provoquant une explosion de région, ainsi que pour mieux comprendre les quantités de mémoire allouées par la phase d'initialisation, il faudrait pouvoir évaluer quantitativement la mémoire allouée par le programme. Plusieurs travaux s'intéressent justement à des problématiques de ce genre, en se basant sur des analyses statiques sophistiquées [MDLC05, BGY06]. Leurs résultats pourraient être combinés avec notre approche par exemple pour *laisser s'agrandir* une région de façon contrôlée.

Il paraît toutefois important de perfectionner la politique d'allocation en régions elle-même pour permettre une plus grande liberté de programmation. Des pistes dans cette direction ont déjà été évoquées dans la section 7.5. On pourrait notamment combiner notre approche avec une analyse d'échappement, de façon à allouer sur la pile d'exécution certains objets à la durée de vie très courte, et réduire la taille des régions. Plusieurs programmes de notre banc d'essais allouent en effet pour leurs calculs des objets auxiliaires de courte durée de vie qui se retrouvent placés dans la même région qu'une structure de données durable. L'espace qu'ils occupent n'est pas recyclé car la mémoire d'une région n'est libérée que lors de sa destruction. Cette idée de combiner l'allocation en régions avec l'allocation sur la pile nécessiterait cependant de revoir notre politique d'allocation car elle dérogerait à l'invariant de groupement des objets connectés.

Une autre approche a également attiré notre attention, celle des ramasses-miettes à comptage de références temps-réel. En effet, le modèle mémoire offert par ces techniques présente également un comportement temporel prévisible. Il semble donc pertinent d'avoir recours à un ramasse-miettes de ce type pour gérer les objets que la synthèse de régions ne parvient pas à désallouer. Nous avons entamé au laboratoire un travail dans ce sens au cours de la thèse [Ber07], et avons obtenu des résultats encourageants.

Cette technique est prometteuse, car elle combine les avantages de plusieurs approches du problème de gestion mémoire. Cependant, elle soulève également plusieurs questions. L'invariant de groupement des structures de données n'est plus respecté, car certains objets situés dans des régions pointent sur des objets situés dans la zone gérée par le ramasse-miettes, et réciproquement. La stratégie de destruction des régions doit donc être mise à jour pour s'assurer de ne pas détruire d'objet encore vivant. De plus, un ramasse-miettes à comptage de références ne peut pas détecter la mort de structures comportant un cycle de pointeurs. Pour garantir l'absence de fuite de mémoire, il faut donc un moyen de vérifier que les objets confiés au ramasse-miettes ne forment jamais de cycle isolé.

Nous avons fait jusqu'ici cette vérification en examinant manuellement le code, mais cette tâche pourrait probablement être automatisée par une analyse statique basée sur les graphes de forme locale. En effet, comme nous l'avons vu, ils forment une représentation du programme propice aux analyses de pointeurs. Il serait par exemple intéressant de les utiliser pour exprimer d'autres analyses statiques, comme une analyse d'échappement, ou des analyses plus sophistiquées comme l'analyse de pureté de Sălcianu et Rinard [SR05].

Enfin, nous ne nous sommes pas intéressés dans ce travail au problème de la gestion mémoire pour des applications concurrentes. La synthèse de régions que nous proposons pourrait également traiter des programmes constitués de plusieurs tâches parallèles, mais le comportement mémoire obtenu ne serait probablement pas très satisfaisant. Il serait donc intéressant d'étudier comment étendre les notions de durées de vie et d'interférence de pointeurs au contexte de la programmation concurrente.

Liste de publications

Nous présentons ci-dessous la liste des publications attenantes à cette thèse. Une version électronique de ces publications est disponible sur le site Internet du laboratoire à l'adresse `http://www-verimag.imag.fr/~salagnac/`

Conférences et ateliers internationaux avec actes et comité de lecture

- Guillaume Salagnac, Christophe Rippert, Sergio Yovine. Semi-Automatic Region-Based Memory Management for Real-Time Java Embedded Systems ; In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, Daegu (Corée du Sud), 2007

- Guillaume Salagnac, Chaker Nakhli, Christophe Rippert, Sergio Yovine. Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems ; In *Proceedings of the 1st Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'06)*, workshop associé à la conférence ECOOP'06, Nantes (France), 2006

- Guillaume Salagnac, Sergio Yovine, Diego Garbervetsky. Fast Escape Analysis for Region-Based Memory Management ; In *Proceedings of the 1st Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL'05)*, workshop associé à la conférence VMCAI'05, Paris (France), 2005

Posters

- Guillaume Salagnac. Automatic Region-Based Memory Management for Real-time Embedded Systems ; In *ACM Conference on Languages, Compilers, and Tools for Embedded Systems LCTES'06, Poster Session*, Ottawa (Canada), 2006

Publications universitaires

- Guillaume Salagnac. Gestion automatique de la mémoire dynamique pour des programmes Java temps-réel embarqués ; Rapport de D.E.A. de l'Université Joseph Fourier, 2004.

Bibliographie

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan J. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the 6th Static Analysis Symposium (SAS'99)*. Springer, 1999. [p. 70]

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI'88)*, pages 11–20. ACM Press, 1988. [p. 46]

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization (PLDI'70)*, pages 1–19. ACM Press, 1970. [p. 59]

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1994. DIKU Research Report 94/19. [p. 69]

[ATCL+98] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 280–290. ACM Press, 1998. [p. 22]

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 296–306. ACM Press, 1988. [p. 67]

[Bak91] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. In *Proceedings of the OOPSLA'91 Workshop on Garbage Collection in Object-Oriented Systems*, pages 66–70. ACM Press, 1991. Position paper. [p. 46]

[Bar89] Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical note, DEC Western Research Laboratory, Palo Alto, CA, 1989. [p. 48]

[BBD+00] Gregory Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The real-time specification for Java*. Addison-Wesley, 2000. [p. 10, 29, 51, 141]

[BBD+04] Rudy Belliardi, Ben Brosgol, Peter Dibble, David Holmes, and Andy Wellings. *The Real-Time Specification For Java, Version 1.0.1*. Java Community Process, 2004. [p. 30]

[BBD+06] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, Mark Turnbull, Rudy Belliardi, David Holmes, and Andy Wellings. *The Real-Time Specification For Java, Version 1.0.2*. Java Community Process, 2006. [p. 30]

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the SIGPLAN 2003 Conference on Programming Languages Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, 2003. [p. 68]

[BCJ+02] Andrew Black, Magnus Carlsson, Mark Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. Technical report, Oregon Graduate Institute School of Science & Engineering, 2002. [p. 20]

[BCR03] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 285–298. ACM Press, 2003. [p. 47, 48, 121, 141]

[BCR04] David F. Bacon, Perry Cheng, and V.T. Rajan. A unified theory of garbage collection. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04)*. ACM Press, 2004. [p. 39]

[BCS02] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP Workshop on Component-Oriented Programming (WCOP'02)*, 2002. [p. 18]

[Ber07] Nicolas Berthier. Gestion hybride de la mémoire dynamique dans les systèmes Java temps-réel. Rapport de magistère 2, Université Joseph Fourier, Grenoble, France, 2007. [p. 115, 139, 143]

[BGY06] Victor A. Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006. [p. 54, 143]

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 35–46. ACM Press, 1999. [p. 70]

[Bla03] Bruno Blanchet. Escape analysis for Java^{TM}: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003. [p. 70]

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262. ACM Press, 1984. [p. 46]

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 324–34. ACM Press, 1996. [p. 69]

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, William S. Beebee, and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. ACM Press, 2003. [p. 53]

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988. [p. 48, 50]

[BZM02] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'02)*. ACM Press, 2002. [p. 38, 50, 54, 71, 79, 109]

[CC77] Patrick Cousot and Radhia Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977. [p. 68]

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282. ACM Press, 1979. [p. 68]

[CCF+02] Vincent Colin de Verdière, Sébastien Cros, Christian Fabre, Romain Guider, and Sergio Yovine. Speedup prediction for selective compilation of embedded Java programs. In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT'02)*, pages 227–239. Springer, 2002. [p. 22]

[CCQR04] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *Proceedings of the SIGPLAN 2004 Conference on Programming Languages Design and Implementation (PLDI'04)*, pages 243–254. ACM Press, 2004. [p. 10, 50, 70, 76, 79, 109, 118, 123]

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 25–35. ACM Press, 1989. [p. 67]

[CGS+03] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, 2003. [p. 70]

[CGVSR05] Alexandre Courbot, Gilles Grimaud, Jean-Jacques Vandewalle, and David Simplot-Ryl. Application-driven customization of an embedded Java virtual machine. In *Proceedings of the 2nd Symposium on Ubiquitous Intelligence and Smart Worlds (UISW'05)*, pages 81–90. Springer, 2005. [p. 20, 24]

[CHB+01] David E. Culler, Jason L. Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT'01)*, pages 114–130. Springer, 2001. [p. 18]

[CHM+98] Charles Consel, Luke Hornof, Renaud Marlet, Gilles Muller, Scott Thibault, Eugen-Nicolae Volanschi, Julia L. Lawall, and Jacques Noyé. Tempo: specializing systems applications and beyond. *ACM Computing Survey*, 30(3), 1998. [p. 20]

[CLM04] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming*, 52(1-3):341–370, 2004. [p. 20]

[CM01] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 10th Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, pages 280–291. IEEE Computer Society, 2001. [p. 123, 142]

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960. [p. 40]

[Cou06] Alexandre Courbot. *Spécialisation tardive de systèmes Java embarqués pour petits objets portables et sécurisés.* Thèse de doctorat, Université de Lille 1, France, 2006. [p. 25, 118]

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 178–188. ACM Press, 1987. [p. 21]

[CPRS03] Antoine Colin, Isabelle Puaut, Christine Rochange, and Pascal Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Technique et Science Informatiques*, 22(5):651–677, 2003. [p. 27]

[CR95] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, pages 29–38. ACM Press, 1995. [p. 123]

[CR04] Sigmund Cherem and Radu Rugina. Region analysis and transformation for Java programs. In *Proceedings of the 4th International Symposium on Memory Management(ISMM'04)*, pages 85–96. ACM Press, 2004. [p. 10, 50, 54, 70, 76, 79, 88, 109, 117, 123, 136]

[CR06] Sigmund Cherem and Radu Rugina. Compile-time deallocation of individual objects. In *Proceedings of the 5th International Symposium on Memory Management (ISMM'06)*, pages 138–149. ACM Press, 2006. [p. 70]

[CV65] Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 185–196, 1965. [p. 17, 34]

[CWZ90] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation (PLDI'90)*, pages 296–310. ACM Press, 1990. [p. 70, 88]

[Dau05] Jean-Marie Dautelle. Validating Java for safety-critical applications. In *Proceeedings of the AIAA Space 2005 Conference*. American Institute of Aeronautics and Astronautics, 2005. [p. 30, 53]

[DB98] Brian Dobbing and Alan Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 ACM SIGAda international conference on Ada (SIGAda '98)*, pages 1–6. ACM Press, 1998. [p. 36]

[DC02] Morgan Deters and Ron Cytron. Automated discovery of scoped memory regions for real-time Java. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*, pages 25–35. ACM Press, 2002. [p. 53]

[Det04] David Detlefs. A hard look at hard real-time garbage collection. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 23–32. IEEE Computer Society, 2004. Invited paper. [p. 48]

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77–101. Springer, 1995. [p. 69]

[DLM+78] Edsgar W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, 1978. [p. 42]

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL'84)*, pages 297–302. ACM Press, 1984. [p. 22]

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003. [p. 21]

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM Press, 1995. [p. 18]

[Fas01] Jean-Philippe Fassino. *Think : vers une architecture de systèmes flexibles*. Thèse de doctorat, École Nationale Supérieure des Télécommunications, Paris, France, 2001. [p. 18]

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for Kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 38–51. ACM Press, 1997. [p. 18]

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX'02)*, pages 73–86. USENIX Association, 2002. [p. 18]

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969. [p. 43]

[GA98] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 313–323. ACM Press, 1998. [p. 49]

[GA01] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*. ACM Press, 2001. [p. 49]

[GHSR06] Gilles Grimaud, Yann Hodique, and Isabelle Simplot-Ryl. Can small and open embedded systems benefit from escape analysis ? In *Proceedings of the 1st ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, 2006. [p. 142]

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The *nesC* language: A holistic approach to networked embedded systems. In *Proceedings of the SIGPLAN 2003 Conference on Programming Languages Design and Implementation (PLDI'03)*, pages 1–11. ACM Press, 2003. [p. 18, 20]

[GM07] Gilles Grimaud and Kevin Marquet. A complexity study of garbage collection for small devices. In *2007 International Conference on Sensor Technologies and Applications (SENSORCOMM 2007)*, pages 127–133. IEEE Computer Society, 2007. [p. 120]

[GMF06] Samuel Z. Guyer, Kathryn McKinley, and Daniel Frampton. Free-Me: A static analysis for automatic individual object reclamation. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, pages 364–375. ACM Press, 2006. [p. 70]

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 282–293. ACM Press, 2002. [p. 50]

[GNYZ04] Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. In *Proceedings of the 4th ETAPS Workshop on Runtime Verification (RV'04)*, 2004. [p. 50, 86]

[Gri00] Gilles Grimaud. *Camille : un système d'exploitation ouvert pour carte à microprocesseur*. Thèse de doctorat, Université de Lille 1, France, 2000. [p. 19]

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the 9th International Conference on Compiler Construction (CC'2000)*. Springer, 2000. [p. 70]

[GSR03] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, 2003. [p. 94]

[Ham00] Kevin Hammond. Hume: a concurrent language with bounded time and space behaviour. In *Proceedings of the 7th IEEE International Conference on Electronic Control Systems (ICECS 2000)*, pages 407–411. IEEE Computer Society, 2000. [p. 20]

[Hat04] Les Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information & Software Technology*, 46(7):465–472, 2004. [p. 36]

[HB05] Matthew Hertz and Emery Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05)*. ACM Press, 2005. [p. 48]

[HHDH02] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*, pages 36–49. ACM Press, 2002. [p. 10, 57, 76, 141]

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet ? In *Proceedings of the 2001 ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. ACM Press, 2001. Invited paper. [p. 69]

[HR04] Grégoire Hamon and John M. Rushby. An operational semantics for stateflow. In *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE'04)*, pages 229–243. Springer, 2004. [p. 21]

[HS02] Patricia M. Hill and Fausto Spoto. A foundation of escape analysis. In *Proceedings of the 9th Conference on Algebraic Methodology and Software Technology (AMAST'02)*, pages 380–395. Springer, 2002. [p. 70]

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 93–104. ACM Press, 2000. [p. 18]

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981. [p. 70]

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 66–74. ACM Press, 1982. [p. 70, 88]

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX Association, 2002. [p. 50]

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996. [p. 35, 39, 40, 48]

[Jon97] Mike Jones. What really happened on mars rover pathfinder. *ACM Forum on Risks to the Public in Computers and Related Systems*, 19(49), 1997. [p. 28]

[JR06] Richard Jones and Chris Ryder. Garbage collection should be lifetime aware. In *Proceedings of the first ECOOP Workshop in Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, 2006. [p. 46]

[KWK02] Jagun Kwon, Andy J. Wellings, and Steve King. Ravenscar-Java: a high integrity profile for real-time Java. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 681–714. ACM Press, 2002. [p. 30, 135]

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI'88)*, pages 21–34. ACM Press, 1988. [p. 70]

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC'2003)*, pages 153–169. Springer, 2003. [p. 69, 93, 117]

[LL73] Chung L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. [p. 27]

[LPB98] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In *Proceedings of the First International Symposium on Memory Management (ISMM'98)*, pages 118–129. ACM Press, 1998. [p. 46]

[Mas03] Anthony J. Massa. *Embedded software development with eCos*. Prentice-Hall, 2003. [p. 19]

[McB63] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, 1963. [p. 41]

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960. [p. 39]

[MCG05] Kevin Marquet, Alexandre Courbot, and Gilles Grimaud. Ahead of time deployment in ROM of a Java-OS. In *Proceedings of the 2nd International Conference on Embedded Software and Systems (ICESS'05)*, pages 63–70. Springer, 2005. [p. 20, 24, 119, 141]

[MDLC05] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *Proceedings of the 2005 ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 193–202. ACM Press, 2005. [p. 48, 54, 143]

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 66–77. ACM Press, 1995. [p. 38]

[MG07] Kevin Marquet and Gilles Grimaud. A DSL approach for object memory management of small devices. In *Proceedings of the 5th Symposium on Principles and Practice of Programming in Java (PPPJ'07)*, pages 155–164. ACM Press, 2007. [p. 120]

[Min02] Antoine Miné. A few graph-based relational numerical abstract domains. In *Proceedings of the 9th Static Analysis Symposium (SAS'02)*, pages 117–132. Springer, 2002. [p. 68]

[MIS98] Motor Industry Software Reliability Association – MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, 1998. [p. 36]

[MM01] Fabrice Mérillon and Gilles Muller. Dealing with hardware in embedded software: A general framework based on the Devil language. In *Proceedings of the 2001 ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'01)*, pages 121–127. ACM Press, 2001. [p. 20]

[MMBC97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, pages 1–20. USENIX Association, 1997. [p. 22]

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005. [p. 66]

[NKH04] Erik M. Nystrom, Hong-Seok Kim, and Wen-Mei Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th Static Analysis Symposium (SAS'04)*, pages 165–180. Springer, 2004. [p. 69]

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Christopher L. Hankin. *Principles of Program Analysis*. Springer, 1999. [p. 57, 62, 68]

[PFHV04] Filip Pizlo, Jason M. Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 101–110. IEEE Computer Society, 2004. [p. 30, 53, 54]

[Pop06] Sebastian Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. PhD thesis, École des Mines de Paris, France, 2006. [p. 67]

[QH02] Feng Qian and Laurie Hendren. An adaptive, region-based allocator for Java. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*, pages 127–138. ACM Press, 2002. [p. 50]

[RBF+89] Richard Rashid, Robert Baron, Ro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A foundation for open systems. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, pages 109–113, 1989. [p. 18]

[RD04] Christophe Rippert and Damien Deville. On-the-fly metadata stripping for embedded Java operating systems. In *Proceedings of the 6th IFIP International Conference on Smart Card Research and Advanced Applications (CARDIS'04)*, pages 17–32. Kluwer, 2004. [p. 24, 119]

[RF02] Tobias Ritzau and Peter Fritzson. Decreasing memory overhead in hard real-time garbage collection. In *Proceedings of the 2nd International Conference on Embedded Software (EMSOFT'02)*. Springer, 2002. [p. 41, 47, 121]

[Rit03] Tobias Ritzau. *Memory Efficient Hard Real-Time Garbage Collection*. PhD thesis, Linköping University, Sweden, 2003. [p. 47, 48]

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218. ACM Press, 2000. [p. 70]

[Sal06] Alexandru Salcianu. *Pointer Analysis for Java Programs ; Novel Techniques and Applications*. PhD thesis, Massachusetts Institute of Technology, 2006. [p. 88, 94]

[Shi88] Olin Shivers. Control flow analysis in Scheme. In *Proceedings of the the SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI'88)*, pages 164–174. ACM Press, 1988. [p. 66]

[SHM+06] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62(2):122–144, 2006. [p. 50]

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00)*, pages 264–280. ACM Press, 2000. [p. 69]

[Sie98] Fridtjof Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In *Proceedings of the First International Symposium on Memory Management (ISMM'98)*, pages 130–137. ACM Press, 1998. [p. 48]

[Sie99] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society, 1999. [p. 47, 48, 141]

[Sie00] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 3rd Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2000)*. ACM Press, 2000. [p. 47, 121]

[SMB04] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04)*. ACM Press, 2004. [p. 46]

[SP81] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981. [p. 66]

[SR01] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, pages 12–23. ACM Press, 2001. [p. 11, 70, 86, 94, 98]

[SR05] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, pages 199–215. Springer, 2005. [p. 94, 143]

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. [p. 28]

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. [p. 69]

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41. ACM Press, 1996. [p. 69]

[Str92] Jan Stransky. A lattice for abstract interpretation of dynamic (LISP-like) structures. *Information and Computation*, 101(1):70–102, 1992. [p. 70]

[Tan87] Andrew Tanenbaum. A UNIX clone with source code for operating systems courses. *ACM Operating Systems Review*, 21, 1987. [p. 18]

[TBEH04] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004. [p. 54]

[Tof98] Mads Tofte. A brief introduction to Regions. In *Proceedings of the First International Symposium on Memory Management (ISMM'98)*, pages 186–195. ACM Press, 1998. [p. 9, 48, 141]

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 188–201. ACM Press, 1994. [p. 49, 70]

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. [p. 49, 70]

[Uma06] Navindra Umanee. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, Montreal, Canada, 2006. [p. 95]

[UML06] Richard Urunuela, Gilles Muller, and Julia L. Lawall. Energy adaptation for multimedia information kiosks. In *Proceedings of the 6th International Conference on Embedded Software (EMSOFT'06)*, pages 223–232. ACM Press, 2006. [p. 26]

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *Proceedings of the 1984 ACM Software Engineering Symposium on Practical Software Development Environments (SDE'84)*, pages 157–167, 1984. [p. 44]

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, page 13. IBM, 1999. [p. 83, 117]

[VRH98] Raja Vallée-Rai and Laurie Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, 1998. [p. 83]

[WDDF98] Michael Weiss, François De Ferriere, Bertrand Delsart, and Christian Fabre. Turbo-J, a Java bytecode-to-native compiler. In *Proceedings of the 1998 ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, pages 119–130. ACM Press, 1998. [p. 22]

[Wei63] Joseph Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, 1963. [p. 43, 47]

[Wel04] Andrew J. Wellings. *Concurrent and real-time programming in Java*. John Wiley and Sons, 2004. [p. 30, 52]

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 International Workshop on Memory Management (IWMM'92)*, pages 1–42. Springer, 1992. [p. 39, 42]

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the 1995 International Workshop on Memory Management (IWMM'95)*, pages 1–116. Springer, 1995. [p. 38]

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 187–206. ACM Press, 1999. [p. 94]

[ZNV04] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251. IEEE Computer Society, 2004. [p. 53]

Résumé

La problématique abordée dans ce travail est celle de la gestion mémoire automatique pour des programmes Java temps-réel embarqués. Dans des langages comme le C ou le C++, la mémoire est typiquement gérée explicitement par le programmeur, ce qui est la source de nombreuses erreurs d'exécution causées par des manipulations hasardeuses. Le coût de correction de telles erreurs est très important car ces erreurs sont rarement reproductibles et donc difficiles à appréhender. En Java la gestion mémoire est entièrement automatique, ce qui facilite considérablement le développement. Cependant, les techniques classiques de recyclage de la mémoire, typiquement basées sur l'utilisation d'un ramasse-miettes, sont souvent considérées comme inapplicables dans le contexte des applications temps-réel embarquées, car il est très difficile de prédire leur temps de réponse. Cette incompatibilité est un frein important à l'adoption de langages de haut niveau comme Java dans ce domaine.

Pour résoudre le problème de la prévisibilité du temps d'exécution des opérations mémoire, nous proposons une approche fondée sur l'utilisation d'un modèle mémoire en régions. Cette technique, en groupant physiquement les objets de durées de vie similaires dans des zones gérées d'un seul bloc, offre en effet un comportement temporel prévisible. Afin de décider du placement des objets dans les différentes régions, nous proposons un algorithme d'analyse statique qui calcule une approximation des relations de connexion entre les objets. Chaque structure de données est ainsi placée dans une région distincte. L'analyse renvoie également au programmeur des informations sur le comportement mémoire du programme, de façon à le guider vers un style de programmation propice à la gestion mémoire en régions, tout en pesant le moins possible sur le développement.

Nous avons implanté un gestionnaire mémoire automatique en régions dans la machine virtuelle JITS destinée aux systèmes embarqués à faibles ressources. Les résultats expérimentaux ont montré que notre approche permet dans la plupart des cas de recycler la mémoire de façon satisfaisante, tout en présentant un comportement temporel prévisible. Le cas échéant, l'analyse statique indique au développeur quels sont les points problématiques dans le code, afin de l'aider à améliorer son programme.

Mots-clés: Gestion mémoire, Langage Java, Systèmes embarqués, Systèmes temps-réel, Analyse statique.

Abstract

In this thesis, we address the problem of dynamic memory management in real-time embedded Java systems. When programming in C or C++, all memory management is done explicitly by the programmer, inducing numerous execution faults because of hazardous use of memory operations. This greatly increases software development costs, because such errors are very hard to debug. The Java language tackles this problem by offering automatic memory management, thanks to the use of a garbage collector. However, garbage collection is often deemed to be unsuited to a real-time embedded context, because of its unpredictable execution times. This problem hinders the spread of modern languages like Java in the world of real-time embedded systems.

To settle the problem of execution times, we propose to use a region-based memory manager. The idea is to group objects of similar lifetimes into memory regions, which are deallocated as a whole. This paradigm offers predictable execution times for all memory operations. We propose a static analysis algorithm that predicts connections between objects, so that every data structure is grouped into one region. The analysis also produces results describing the memory behaviour of the program, helping the developer to write his code in a style suitable for region-based memory management.

We implemented an automatic region allocator in the virtual machine of the JITS project, dedicated to bringing full Java support to resource-constrained embedded devices. Experiments show that for most programming patterns, our system behaves as efficiently as a garbage collector, while retaining predictable execution times. Our analysis tool is furthermore able to provide useful feedback to the programmer to pinpoint problematic constructs.

Keywords: Memory Management, Java, Embedded Systems, Real-Time Systems, Static Analysis.