Unité Mixte de Recherche UJF - CNRS - INPG
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Gestion automatique de la mémoire dynamique pour des programmes Java temps-réel embarqués

Guillaume Salagnac

Master 2 Recherche *Systèmes et Logiciels* Année 2003 - 2004

UNIVERSITE JOSEPH FOURIER GRENOBLE 1 SCIENCES, TECHNOLOGIE, MEDECINE

CNRS

INPG

Résumé Ce rapport s'intéresse à la gestion automatique de la mémoire dynamique pour des programmes Java temps-réel embarqués. Dans la première partie, on détaillera les différentes problématiques ainsi que les solutions apportées à l'heure actuelle. Mais ni la gestion manuelle de la mémoire ni les algorithmes de ramasse-miettes n'apportent de réponse satisfaisante. Nous présenterons ensuite des techniques qui tâchent de garantir *à la fois* performance temporelle, simplicité d'utilisation pour le programmeur, et sécurité. L'objet du stage est l'écriture d'une première implémentation de l'une de ces techniques, de façon à pouvoir expérimenter concrètement cette approche. La seconde partie décrit cette implémentation, avant de commenter les mesures qui ont pu être obtenues, pour pouvoir conclure sur le bien-fondé de cette méthode.

Table des matières

Introduction **4**

1 Etat de l'art **5**

1.1 La mémoire dynamique . . . 5

1.1.1 La gestion de la mémoire dynamique . . . 5

1.1.2 Gestion de la mémoire en régions . . . 6

1.1.3 Le langage C et les régions : RC . . . 7

1.1.4 The Realtime Specification for Java . . . 8

1.2 Automatisation des transformations . . . 11

1.2.1 Analyse d'échappement . . . 11

1.2.2 Instrumentation automatique vers la RTSJ . . . 13

1.2.3 Notre approche . . . 13

2 Contribution **17**

2.1 Réalisations . . . 17

2.1.1 Architecture du prototype . . . 17

2.1.2 Le choix du compilateur : TurboJ ou GCJ . . . 19

2.1.3 Implémentation . . . 21

2.2 Validation de l'approche . . . 24

2.2.1 Le programme `Tile` . . . 25

2.2.2 Expérimentations . . . 27

Conclusion et perspectives **32**

Bibliographie **34**

A Le code de l'API Madeja **36**

A.1 `ScopedMemory.java` . . . 36

A.2 `Region.java` . . . 38

A.3 `natRegion.cc` . . . 38

B Le programme `tile` **42**

B.1 `TileMain.java` . . . 42

B.2 `Htile.java` . . . 44

B.3 `Tokizer.java` . . . 56

Introduction

Le langage, Java, conçu à l'origine pour le développement d'*applets* intégrées dans des pages web, est aujourd'hui largement utilisé dans le développement d'un nombre important d'applications.

Si le langage Java est attractif, la machine virtuelle est elle, en revanche, un frein puissant pour son adoption par certaines communautés, comme celle des systèmes temps-réel. En effet le manque de performance et l'absence de déterminisme de la machine virtuelle Java constituent pour cela un obstacle qui est rédhibitoire.

L'un des atouts de Java est son modèle de programmation orienté objet avec gestion automatique de la mémoire dynamique (GC). Toutefois, le GC n'est pas utilisé dans le développement des logiciels embarqués temps-réel. La principale raison pour cela est que les techniques actuelles de GC ne satisfont pas les exigences temporelles auxquelles ces applications sont confrontées.

Plusieurs approches visent à se passer du GC, par exemple en rendant le programmeur responsable de la gestion de mémoire dynamique, ou en utilisant des techniques d'analyse d'échappement. Cependant, les résultats obtenus à l'heure actuelle restent encore principalement théoriques.

L'objet de ce projet est une implémentation de l'approche poursuivie à Verimag, et présentée dans [GNYZ04] (article paru en cours de stage).

Après un tour d'horizon des articles traitant de ce sujet, l'objectif est l'implémentation d'un prototype, intégré dans l'exécutif Java temps-réel Expresso, ainsi que des expérimentations concrètes, de façon à faire apparaître les forces et les faiblesses de cette approche.

Dans le premier chapitre, nous rappellerons les différents aspects de la problématique, ainsi que les avancées actuelles dans ce domaine. La seconde partie s'attachera à décrire l'implémentation du prototype, puis la réalisation d'expérimentations sur un programme de test.

Chapitre 1

Etat de l'art

La gestion de la mémoire dynamique est depuis toujours un problème important dans l'implémentation des langages de programmation. La problématique est double : faut-il laisser la charge de cette gestion au programmeur ? Cette solution est une source d'erreur importante, et conduit à un processus de développement difficile. Faut-il la gérer automatiquement ? On fait alors face aux performances non prédictibles des algorithmes de GC.

Nous nous attacherons plus particulièrement à deux aspects pertinents pour les programmes Java temps-réel : la gestion de la mémoire en régions, qui permet de garantir le respect de certaines contraintes temporelles mais qui est d'usage complexe pour le programmeur, puis nous verrons comment automatiser son utilisation, en particulier par analyse d'échappement, qui vise à obtenir une gestion automatique de la mémoire dynamique grâce à l'anayse statique du programme.

1.1 La mémoire dynamique

Un programme, lorsqu'il s'exécute, a besoin de mémoire pour stocker les données qu'il manipule. On distingue trois catégories de mémoire : la mémoire globale, de taille fixe, où sont situées les variables globales, la *pile*, qui sert à stocker les *frames* d'appel des fonctions, et le *tas*, dans lequel on alloue la mémoire dynamique.

1.1.1 La gestion de la mémoire dynamique

L'espace du tas n'est pas utilisé de la même manière suivant les langages : C offre pour le manipuler les deux primitives `malloc()` et `free()`, C++ les opérateurs `new` et `delete`. Le programmeur demande au système la mémoire dont il a besoin, et la libère explicitement ensuite.

Le langage Java, quant à lui, permet au programmeur de s'affranchir complètement de la gestion manuelle de la mémoire dynamique, une des sources les plus courantes d'erreur pour un programmeur, et qui devient très problématique sur les gros projets impliquant de nombreux développeurs. En Java, le programmeur ne peut pas demander à désallouer de mémoire. C'est le rôle du *ramasse-miettes* (ou *garbage collector*, GC) de déterminer si une zone peut être réutilisée ou pas.

En général, le ramasse-miettes interrompt périodiquement le programme utilisateur, pour parcourir alors le tas tout entier. Ce faisant, il met de côté les zones de mémoire encore actives, et recycle la mémoire qui n'est plus référençable (*garbage*). pour s'assurer que le tas ne sera pas modifié pendant son exécution.

Il existe de nombreux langages qui font usage d'un ramasse-miettes ([Jon96]), et de nombreux algorithmes ont été proposés pour remplir cette fonction. Le GC de Java, depuis la version 1.2, utilise un algorithme à générations, et est doté d'optimisations spécifiques aux caractéristiques du langage. Malgré ses bonnes performances dans la JVM, un tel ramasse-miettes ne peut être utilisé dans un compilateur *AOT* (Ahead Of Time, c.à.d. qui compile *à l'avance* le programme vers du code objet).

En effet, un GC générationnel *déplace* les objets actifs, puis met à jour tous les pointeurs vers les objets déplacés. Dans du code natif, souvent optimisé par le compilateur, la présence d'arithmétique sur les pointeurs empêche souvent de les manipuler, l'identification même des pointeurs peut devenir problématique.

Le seul GC adapté à un programme *natif* est celui de Boehm-Weiser (détaillé dans [Jon96]), qui opère *conservativement* sur le tas, c'est pourquoi il est utilisé dans tous les compilateurs Java AOT.

Les *gestionnaires de mémoire*, qu'ils soient manuels ou à ramasse-miettes, sont en général des algorithmes sophistiqués, car ils doivent faire face à *deux* problèmes :

- Le tas a naturellement tendance à se *fragmenter* au fur et à mesure des allocations et désallocations, gaspillant ainsi l'espace : en effet, le système peut ne pas parvenir à allouer un grand bloc contigu lorsque l'espace libre est suffisant mais trop morcelé. Les différent gestionnaires de mémoire luttent chacun à leur manière contre la fragmentation, par exemple par *l'algorithme du compagnon*.
- Cette fragmentation peut aussi dégrader considérablement les performances temporelles de l'allocation, car la recherche d'un bloc de mémoire approprié oblige à considérer plusieurs critères contradictoires. Les différentes stratégies (first-fit, best-fit, etc) on chacune leurs inconvénients ([BZM02]).

1.1.2 Gestion de la mémoire en régions

Pour se soustraire au problème de la fragmentation, on peut, plutôt que de confier la gestion du tas à un algorithme classique, l'organiser en *régions* : les objets sont alors alloués «côte à côte», et libérés *en masse* lorsque la région est *quittée*. La gestion de la mémoire en régions ne permet donc pas la libération individuelle d'objets, mais en contre partie elle est bien plus efficace pour l'allocation, qui se fait toujours en temps linéaire. En effet, il n'y a plus besoin de *rechercher* un bloc de mémoire libre, l'allocation est implémentée simplement en incrémentant un pointeur dans la région. Pourtant, elle n'est pas faite en temps constant, car l'allocation d'un nouvel objet necéssite aussi d'initialiser à zéro tous les octets de la de mémoire obtenue.

Certains logiciels notoires (le serveur web Apache, le compilateur GCC) utilisent ainsi leur propre mécanismes de gestion de mémoire en régions, en lieu et place des traditionnels `malloc` et `free`.

A notre connaissance, l'introduction de la gestion de mémoire en régions dans un langage de programmation a été proposée pour la première fois dans [TT97], où les

auteurs présentent une implémentation du ML kit qui utilise des régions au lieu du traditionnel ramasse-miettes.

Leur approche utilise une annotation automatique du programme, qui place chaque variable dans une région différente. C'est le rôle d'un système d'inférence que d'analyser ces annotations et de les *simplifier* (pour regrouper les régions), de façon à ce que l'usage des régions reflète les durées de vie des objets du programme

Les régions sont ainsi organisées en pile à l'exécution, et un *moteur d'exécution* se charge de leur allocation et désallocation.

1.1.3 Le langage C et les régions : RC

Une approche pour étendre le langage C avec une gestion de la mémoire dynamique par régions est proposée dans [GA01]. A l'inverse de [TT97], les régions n'ont pas forcément à être organisées en pile, elles sont à la disposition du programmeur qui décide comment il les utilise.

L'approche est basée sur un dialecte de C, appelé RC, ainsi que sur un compilateur spécifique, appelé `rcc`.

- En RC, les pointeurs comportent, en plus de leur type, une annotation de région. Un pointeur peut être déclaré `traditional` (par exemple `int *traditional x`), ce qui signifie qu'il s'agit d'un pointeur C normal.
  L'intérêt de RC réside cependant surtout dans l'annotation `sameregion`, qui impose à un pointeur de désigner un objet situé dans la même région que lui.
- Ces restrictions, données par le programmeur, sont vérifiées par le compilateur `rcc` : il garantit alors (par analyse statique, ou par l'insertion de tests dynamiques) qu'une variable `traditional` ne pointe jamais vers une région mais toujours dans la mémoire *normale* de C (allouée par `malloc`), qu'une variable `sameregion` pointe toujours vers la même région, etc.

La motivation de ces annotations est assez naturelle : grouper explicitement les données de même durée de vie. En effet, souvent de nombreux pointeurs renvoient sur différentes parties de la *même* structure de données. En RC, toute cette structure sera allouée dans la même région, et désallouée d'un seul coup, garantissant (encore, grâce à une preuve statique ou à des vérifications à l'exécution) que tous les pointeurs contiennent toujours des références valides.

Lorsque la structure du programme le demande, les régions de RC peuvent aussi être organisées en pile, ou en arbre, et alors les sous-régions sont contraintes à avoir une durée de vie plus courte que leur parent. Des pointeurs annotés par l'attribut `parentptr` fourni par RC peuvent alors être utilisés pour indiquer qu'un objet pointe quelque part dans la région parente.

Le programmeur est maître des régions qu'il utilise dans son programme, créées à l'aide d'une série de primitives : `newregion(), deleteregion()`... En allouant des objets dans la même région, il est plus facile pour le programmeur de s'occuper de la gestion explicite de la mémoire. Pourtant, c'est quand même le rôle de l'humain que de déterminer les durées de vie des objets car il faut faire appel «à la main» à `deleteregion()`.

Cette utilisation des régions, implémentées par la bibliothèque `libregions`, avait déjà été présentée dans [GA98], mais le langage, appelé alors C@, n'utilisait pas d'annotations ni d'analyse statique.

La particularité de RC qui le rend réellement différent du langage C est la présence des tests dynamiques : pour s'assurer l'absence de *dangling references*, le compilateur `rcc` insère dans le code binaire généré des mécanismes de comptage des références à chaque affectation d'un pointeur annoté. Le programme est ainsi en mesure de contrôler que tous les pointeurs respectent bien le comportement qui a été spécifié dans le source. De même, à chaque région est associé un compteur qui mémorise le nombre de pointeurs *externes* vers cette région. Lors du `deleteregion()`, si ce compteur est non-nul, plutôt que de laisser le programme s'exécuter avec des *dangling references*, RC l'arrête proprement par un `abort()`.

On peut cependant compiler un programme RC avec un compilateur C classique si l'on met de côté (par préprocesseur) ces annotations, et l'on perd alors cette sécurité à l'exécution. Les auteurs proposent, par exemple, d'utiliser `rcc` pour s'assurer de l'usage correct de la mémoire dynamique pendant la phase de développement, mais pas forcément pour le programme *en production*, de façon à s'épargner les surcoûts à l'exécution : un `abort()` ne vaut en effet guère mieux qu'un `segmentation fault` si personne n'est là pour faire de correction.

Grâce aux vérification dynamiques, on est certain que le programme ne s'exécute pas en utilisant de la mémoire *interdite*, mais on n'est pas pour autant débarrassé de la tâche de gestion de la mémoire dynamique.

De plus, elle est même rendue assez pénible car, à chaque `deleteregion()`, il faut être sûr que plus aucun pointeur ne fait référence à une objet de cette région, même pour les pointeurs qui ne seront plus du tout utilisés dans le programme. L'écriture d'un programme RC est donc assez fastidieuse car il faut s'assurer exhaustivement du bon usage des pointeurs. Dans le cas d'un portage de programme C vers RC, trouver *où* neutraliser ces pointeurs (en les assignant à `NULL`) peut même, de l'aveu des auteurs, devenir un problème non trivial.

1.1.4 The Realtime Specification for Java

En Java, le programmeur n'a pas à se préoccupper de la gestion de la mémoire dynamique, le ramasse-miettes se charge de toutes les libérations de mémoire. Cependant, l'usage d'un GC est souvent incompatible avec la programmation temps-réel. En particulier, l'imprédictabilité des temps d'exécution du GC (*temps de pause*) et des temps d'allocation est un problème pour programmer des tâches à contraintes temps-réel fortes.

C'est pourquoi est apparue une extension de Java centrée sur le temps-réel. [Bol00] propose des modifications à apporter au langage ainsi qu'à la machine virtuelle Java dans plusieurs domaines : *threads* et ordonnancement, gestion de la mémoire, synchronisation, gestion des ressources, ...

La RTSJ étend l'API de la bibliothèque standard de Java (hiérarchie `javax.realtime`) et modifie la sémantique de certains aspects du langage, comme par exemple l'instruction `new`. En particulier, elle définit deux nouveaux types de *threads*, le `RealtimeThread` et le `NoHeapRealtimeThread`.

La gestion de mémoire

La RTSJ définit des *memory scopes*, qui sont des régions de mémoire organisées en pile (durées de vie imbriquées) : un objet alloué dans une région aura une durée de vie égale à celle de cette région. Chaque région est associée à un `Runnable`, et dure autant que dure l'exécution de son thread. Lorsqu'elle sera *quittée*, à la terminaison de celui-ci, tous les objets alloués dans cette région seront désalloués.

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 d'exécution de l'instruction `new` est *linéaire* en la taille de l'objet ; mais rien n'est dit sur la durée d'exécution du constructeur de l'objet, qui est une méthode comme une autre. Connaissant les coûts temporels des allocations, on peut construire un modèle de coût pour prédire les temps d'exécution des programmes qui utilisent des régions `LTMemory`.

Dans une région `VTMemory`, le temps d'exécution de `new` est *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 connaitre par avance la taille totale allouée. Au contraire, pour une `LTMemory`, la taille maximum de la région doit être connue au plus tard à la création de la région.

Les threads

Un `RealtimeThread` diffère d'un thread normal par la présence d'une priorité : l'ordonnanceur RTSJ est préemptif et offre 28 priorité fixes. Le plus faible de ces niveaux de priorité est encore au dessus du plus haut des 10 niveaux de priorité des threads standard de Java. De plus, un `RealtimeThread` est associé avec une `MemoryArea`, dans laquelle il alloue ses objets.

Le `NoHeapRealtimeThread` est un `RealtimeThread` qui n'est pas du tout autorisé à manipuler des données situées dans le tas, ni à y accéder. Les objets alloués par un `NoHeapRealtimeThread` n'ont pas le droit de contenir des références vers des objets du tas.

De cette façon, on est certain qu'il n'interfèrera jamais avec le GC, et donc qu'il ne sera pas interrompu par ce dernier (il a une priorité strictement supérieure à celle du GC).

On peut alors lui imposer des contraintes temps-réel, car s'il alloue tous ses objets dans une région `LTMemory`, tous les coûts d'exécution seront prédictibles.

Restrictions

Du fait des *memory scopes*, il est interdit pour un objet du tas de pointer vers un objet d'une région (car la région pourrait disparaître intempestivement, créant une *dangling reference*).

A l'inverse, un objet d'une région peut pointer vers un objet du tas, c'est pourquoi les `MemoryAreas` associées à ds `RealtimeThreads` sont parcourues par le GC, en quête de références. Mais si une région est associée à un `NoHeapRealtimeThread`,

de telles références sont interdites car le GC ne pourrait pas examiner cette régions sans interrompre le calcul.

La RTSJ impose au compilateur soit de vérifier statiquement ces règles, résumées sur la figure 1.1, soit de placer dans le code des tests dynamiques qui s'assurent qu'elles ne sont pas violées.

Objet	pointer vers le tas	pointer dans une région
Dans le tas	Oui	Interdit
Dans une région (`RealtimeThread`)	Oui	Oui, si région inférieure
Dans une région (`NoHeapRealtimeThread`)	Interdit	Oui, si région inférieure, *et* possédée par un `NoHeapRealtimeThread`

FIG. 1.1 – Les règles d'assignement de 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 visée atteinte, par des appels successifs à `getouterScope()`, on peut y allouer des objets par des primitives comme `MemoryArea.newInstance()`. L'API de gestion de mémoire de la RTSJ est résumée sur la figure 1.2

On peut voir un exemple d'utilisation d'une partie de l'API RTSJ sur la figure 1.5.

```
abstract class MemoryArea {
    void enter( Runnable );
    static void getMemoryArea( Object );
    Object newArray( Class, int );
    Object newInstance( Class ); }

final class HeapMemory extends MemoryArea {
    static HeapMemory instance(); }

final class ScopedMemory extends MemoryArea {
    int getReferenceCount();
    MemoryArea getOuterScope(); }

class VTMemory extends ScopedMemory...
class LTMemory extends ScopedMemory...
```

FIG. 1.2 – L'API de gestion de mémoire de la RTSJ

La RTSJ remet donc à la charge du programmeur une 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 ainsi s'avérer complexe, en particulier du fait du manque de clarté introduit dans le code. Le fait que chaque région est associée à un thread, (et

l'absence d'un `exit()` explicite, équivalent au `deleteregion()` de RC) impose des paradigmes de programmation étranges.

[FHPV04] fait une lecture intéressante de la RTSJ et y relève nombre de difficultés pour le programmeur ainsi que pour l'implémenteur.

Effectivement, la RTSJ n'est pas une implémentation, mais seulement une spécification. Il en existe aujourd'hui plusieurs implémentations, comme par exemple dans le moteur d'exécution d'Expresso, ou l'implémentation de référence de TimeSys, mais elles sont imparfaites.

L'article donne aussi quelques *patrons de conception* pour aider au portage vers la RTSJ de schémas de programmation classiques (producteur-consommateur, etc) en contournant ces problèmes.

1.2 Automatisation des transformations

Les techniques de gestion de la mémoire dynamique en régions ont des avantages certains par rapport aux algorithmes classiques de ramasse-miettes, mais elles compliquent considérablement la tâche au programmeur.

Pour éviter de les utiliser directement, plusieurs approches ont été proposées, par exemple pour instrumenter automatiquement le programme. Il faut alors déterminer à l'avance les durées de vie des objets alloués dynamiquement.

Pour ce faire, [DC02] utilise une technique de profilage du programme, et vise la RTSJ comme plate-forme d'exécution. Au contraire, [Bla99] détermine, par analyse statique, quels objets peuvent être *alloués en pile* dans un compilateur AOT. [GNYZ04] propose d'utiliser *l'analyse d'échappement* de [Bla99] pour déterminer les régions, puis transformer le code.

Ces approches visent à associer des régions aux méthodes, et donc à les créer lors des appels et les détruire lors des retours

1.2.1 Analyse d'échappement

Les techniques d'allocation en régions cherchent à éliminer les problèmes dus à la gestion de la mémoire dynamique (complexité algorithmique, difficulté pour le programmeur) mais elles ne règlent pas le problème de la détermination des durées de vie effectives des objets. L'analyse d'échappement de [Bla99], par une interprétation abstraite du programme, cherche à déterminer à la compilation la durée de vie des objets.

Lorsque la durée de vie d'un objet dépasse sa portée syntaxique (souvent, la fonction dans laquelle il a été alloué), par exemple parce qu'il est utilisé comme valeur de retour de cette fonction, on dit que l'objet s'en *échappe*. A l'inverse, si une variable d'une méthode contient une référence vers un objet et qu'il ne s'échappe pas, on dit que l'objet *est capturé* par cette méthode.

Le programme de la figure 1.3 est assez simple, mais ses objets s'échappent de manière non triviale :

Dans la méthode `m2()`, lorsqu'elle est appelée par `m1()`, l'objet G_2 est référencé par E_2 (l. 20), qui est en réalité un alias de E_0. L'objet G_2 *s'échappe* de `m2()` pour *être capturé* par `m0()`. On ne doit donc pas l'allouer dans la région associée à `m2()`

```
class refObject
{
    Object Ref;
}
void m0()
{
    RefObject E0 = new RefObject();
    m1(E0);
    Object D0 = m2(E0);
}
void m1(RefObject E1)
{
    RefObject A1 = new RefObject();
    Object D1 = m2(E1);
}
Object m2(RefObject E2)
{
    RefObject D2=new RefObject();
    Object F2 = new Object();
    D2.ref = E2;
    Object G2 = new Object();
    E2.ref = G2;
    return D2;
}
```

FIG. 1.3 – Un exemple simple

car sa durée de vie serait trop courte, et `E0.ref` deviendrait une *dangling reference* au retour de `m2()`.

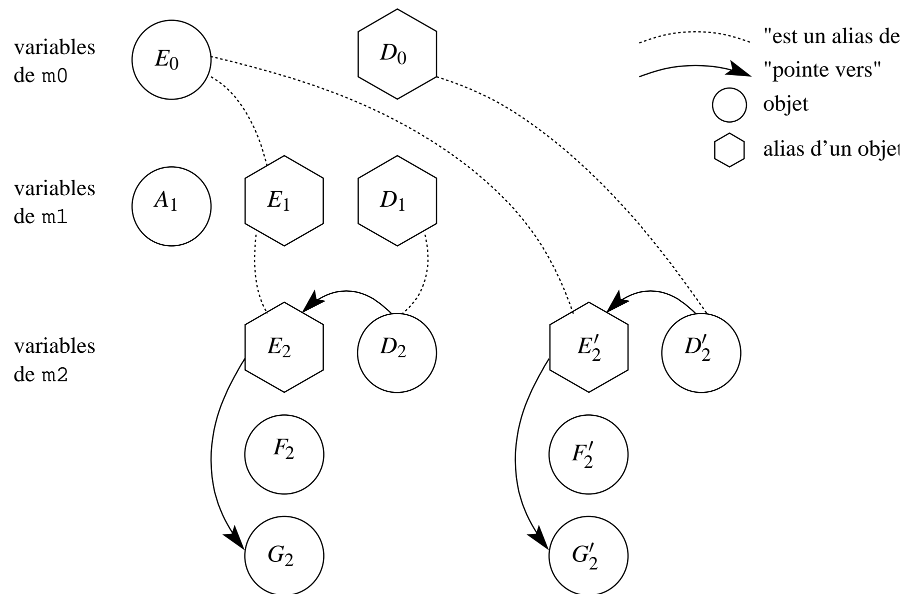

FIG. 1.4 – Les relations entre les objets du programme de la figure 1.3

De même, D_2 est utilisé comme valeur de retour de `m2()`, donc il doit être alloué dans une région inférieure dans la pile. D_2 s'échappe seulement jusqu'à `m1()`, donc il

n'est pas necéssaire (bien que possible) de l'allouer dans la région associée à `m0()`.

Les différents objets et leurs relations sont représentés sur la figure 1.4, qui permet de déterminer facilement où ils s'échappent.

1.2.2 Instrumentation automatique vers la RTSJ

[DC02] propose d'associer un *memory scope* à chaque méthode, et d'allouer les objets, non pas dans la région associée à la méthode du site d'allocation, mais dans la région de la méthode qui capture l'objet.

L'approche est ici de transformer un programme Java naïf en un programme RTSJ par instrumentation du code. Chaque `new` sera remplacé par un `MemoryArea.NewInstance()`.

Pour allouer l'objet dans la bonne région, [DC02] utilise *plusieurs*

```
outerScope = thisScope.getOuterScope()
```

de façon à descendre dans la pile des régions.

Le nombre de niveaux d'imbrication à traverser pour allouer l'objet est déterminé par le profilage. Le programme de la figure 1.3, une fois transformé, est présenté sur la figure 1.5.

Cette approche, si elle a le mérite de la simplicité pour le programmeur, n'est pas exempte de défauts.

En particulier, la mesure des durées de vie n'est pas faite par analyse d'échappement mais par profilage, en faisant tourner le GC très régulièrement pendant une première exécution, puis en instrumentant le programme avec les informations obtenues. Les mesures récoltées ainsi sont optimales, mais sont seulement valables pour une seconde exécution *dans les mêmes conditions* (sur les mêmes données, etc). Il n'y a donc pas de garantie que les désallocations soient sans danger dans le cas général.

L'optimalité des mesures n'est même pas certaine, car les durées de vie sont mesurées par un «dénivelé», compté en nombre de *frames*, sans tenir compte de la chaîne d'appel. Cependant, un objet alloué au même endroit peut s'échapper plus ou moins loin suivant le contexte dans lequel est appelée la méthode.

Par exemple, l'objet G'_2, alloué pendant l'appel à `m2()` par `m0()`, ne s'échappe que d'un niveau, alors que [DC02], qui ne le distingue pas de G_2, l'envoie deux niveaux plus bas.

De plus, le choix a été fait ici d'instrumenter le programme pour lui faire utiliser l'API de la RTSJ. Ce choix est dicutable, car la RTSJ associe une région à un `Runnable`, c'est pourquoi chaque méthode est lancée dans un thread séparé. Le surcoût à l'exécution est certainement considérable vis-à-vis du programme original par appel de méthodes. Les nombreux appels aux fonctions de l'API sont eux aussi synonymes de surcoût : pour allouer un objet n niveaux plus bas dans la pile des régions, il faut n appels à `getOuterScope()`.

1.2.3 Notre approche

[GNYZ04] propose une méthode qui ne souffre pas de l'inconvénient majeur de celle présentée ci-dessus : la non-sûreté de l'analyse par profilage.

```
   class ScopeReturnValue {
      Object obj;
      void set(Object o) { obj = o; }
      void get() { return obj; }
   }
   void m0()
   {
      RefObject E0 = new RefObject();
      ScopeReturnValue rv=new ScopeReturnValue();
10    new LTMemory(1024,1024).enter(
         new Runnable() {
            public void run() {
               m1(E0);
               rv.set(m2(E0));
            }
      });
      Object D0 = rv.get();
   }
   void m1(RefObject E1)
20 {
      ScopeReturnValue rv=new ScopeReturnValue();
      new LTMemory(1024,1024).enter(
         new Runnable() {
            public void run() {
               RefObject A1 = new RefObject();
               rv.set(m2(E1));
            }
      });
      Object D1 = rv.get();
30 }
   Object m2(RefObject E2)
   {
      RealtimeThread thisThread =
         RealtimeThread.getRealtimeThread();
      ScopedMemory thisScope =
         (ScopedMemory)thisThread.getMemoryArea();
      MemoryArea outerScope = thisScope.getOuterScope();
      MemoryArea outerOuterScope =
         ((ScopedMemory) outerScope).getOuterScope();
40
      RefObject D2 = (RefObject)
         outerScope.newInstance(RefObject.class);

      Object F2 = new Object();

      D2.ref = E2;

      Object G2 =
         outerOuterScope.newInstance(Object.class);
50    E2.ref = G2;

      return D2;
   }
```

FIG. 1.5 – Programme de la figure 1.3, instrumenté avec la méthode de [DC02]

L'idée est de faire statiquement une analyse d'échappement et une analyse de pointeurs (*points-to analysis*), de façon à déterminter *conservativement* où s'échappent les objets, et d'instrumenter le programme grâce aux informations obtenues.

L'instrumentation est faite vers une API spécifique, plus adaptée que l'API RTSJ, et surtout qui ne necéssite pas de créer un thread par méthode.

Tout en conservant ses avantages (pas de GC donc pas de coût non borné, pas de tests dynamiques), cette méthode n'a pas les inconvénients de celle de [DC02] :

– Les objets sont alloués dans une région dont on est *sûr* qu'elle sera active assez longtemps
– Les méthodes ne sont pas encapsulées chacune dans un thread, donc il n'y a pas de surcoût de ce côté là.

De plus, comme le mécanisme d'enregistrement (voir ci-après) est dynamique, les allocations se font différemment si le contexte d'appel est différent. Ceci évite d'allouer sytématiquement «très bas» dans la pile des régions un objet lorsqu'il s'échappe à des distances variables (comme l'objet G_2 et G'_2 dans l'exemple).

Présentation de l'API Madeja

Le choix a été fait ici d'un mode de programmation différent de celui de [DC02] : les objets capturés sont *enregistrés* à l'avance par la méthode qui les capture. Le nombre de niveaux à traverser dans la pile des régions n'est pas fixe, et un même objet (ou plutôt, différents objets alloués au même site d'allocation) peut s'échapper plus ou moins loin suivant le contexte dans lequel est appelée la méthode.

Les différentes fonctions de l'API Madeja sont donc :

– `ScopedMemory.enter(Region)` et `ScopedMemory.exit()` permettent, au début et à la fin de chaque méthode, d'ajouter puis de retirer une région du sommet de la pile des régions.
– `ScopedMemory.DetermineAllocationSite(String[])` est utilisée pour enregistrer les demande de capture. Elle maintient une structure de données appelée le *registre de captures*.
– `ScopedMemory.newInstance(String,Class)` remplace les `new` du programme original. Le paramètre `String` est le nom du site d'allocation considéré. Cette méthode recherche dans le registre de capture une indication de région où allouer l'objet. Si elle ne trouve pas, c'est qu'aucune méthode n'a capturé l'objet (il ne s'échappe pas), et il est alors alloué dans la région du sommet de la pile.
– `ScopedMemory.newAInstance(String,Class,int)` permet, de façon similaire, d'allouer un tableau.

Le code est instrumenté automatiquement pour utiliser cette API, comme présenté sur la figure 1.6.

Cette transformation du code est assez mécanique : à chaque site d'allocation, le `new` est remplacé par un `newInstance()` (ou `newAInstance()` pour les tableaux), et à chaque site d'appel on insère un `DetermineAllocationSite()`. Le seul problème non trivial est de correctement calculer les captures, pour initialiser les tableaux correspondants.

Un exemple plus complexe de cette transformation est présenté en détail dans [GNYZ04].

```
String [ ] m0_2 ={"m2_3"};
String [ ] m0_3 ={"m2_1","m2_3"};
String [ ] m1_2 ={"m2_1"};
void m0()
{
   ScopedMemory.enter(new Region("m0"));
   RefObject E0=(RefObject)
      ScopedMemory.newInstance("m0_1",RefObject.class);
   ScopedMemory.determineAllocationSite(m0_2);
10 m1(E0);
   ScopedMemory.determineAllocationSite(m0_3);
   Object D0=m2(E0);
   ScopedMemory.exit();
}
void m1(RefObject E1)
{
   ScopedMemory.enter(new Region("m1"));
   RefObject A1=(RefObject)
      ScopedMemory.newInstance("m1_1",RefObject.class);
20 ScopedMemory.determineAllocationSite(m1_2);
   Object D1=m2(E1);
   ScopedMemory.exit();
}
Object m2(RefObject E2)
{
   ScopedMemory.enter(new Region("m2"));
   Region myRegion=ScopedMemory.current();
   RefObject D2=(RefObject)
     ScopedMemory.newInstance("m2_1",RefObject.class);
30 Object F2=(Object)
     ScopedMemory.newInstance("m2_2",Object.class);
   D2.ref=E2;
   Object G2=(Object)
     ScopedMemory.newInstance("m2_3",Object.class);
   E2.ref=G2;
   ScopedMemory.exit();
   return D2;
}
```

FIG. 1.6 – Programme de la figure 1.3, instrumenté avec l'API Madeja

Comme dans [DC02], chaque méthode est associée à une région, et les objets sont alloués indirectement, par le biais de `newInstance()`, mais ici il n'y a pas de création de threads, et l'objet G_2' s'échappe correctement (i.e. vers la région de `m0()`) les deux fois. En effet, les sites d'appels `m0_2` et `m0_3` déclarent tous les deux qu'ils capturent `G2`.

Chapitre 2

Contribution

2.1 Réalisations

Mon travail a consisté à implémenter l'API Madeja d'enregistrement et de gestion des régions, de façon à ce que des programmes, déjà correctement analysés et instrumentés, s'exécutent en allouant leurs objets dans des régions en utilisant la bibliothèque de régions de RC.

2.1.1 Architecture du prototype

Principe

Le programme Java est instrumenté pour utiliser l'API Madeja et n'alloue plus lui-même ses objets par `new`, mais le contrôle *réel* de l'allocation des objets ne peut se faire au niveau Java.

Dans une JVM, ce serait le code de bas niveau de la machine virtuelle qui interpréterait le *bytecode* `new`. Dans un compilateur AOT, l'instruction `new` est compilée en un appel de fonction C qui se charge d'assurer l'allocation.

De façon à allouer nous-mêmes l'objet, il faut utiliser une *interface native*. Une interface native est une API permettant d'interfacer du code Java avec du code objet natif de la plate-forme cible. On perd les avantages dûs à la portabilité de Java, mais on gagne en performance, et on peut faire des opérations de bien plus bas niveau.

Un programme simple qui l'interface JNI, définie par le JDK de Sun, est donné sur la figure 2.1

Une bonne partie du *Classpath* (la bibliothèque standard du langage Java) est ainsi implémentée en natif, en particulier toutes les entrées-sorties.

De même, pour faire appel aux fonctions d'allocation de la bibliothèque de RC, une interface native nous sera necéssaire, comme montré sur la figure 2.2.

L'API Madeja pourra cependant en grande partie être implémentée en Java, car l'instrumentation est faite au niveau source, et les mécanismes sont assez simples, d'un point de vue algorithmique : Il suffit de gérer la pile des régions et la table d'enregistrement.

```java
public class HelloWorld
{
  public native void displayHelloWorld();

  public static void main(String[ ] args)
  {
    new HelloWorld().displayHelloWorld();
  }
}
```

HelloWorld.java

```c
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
 Java_HelloWorld_displayHelloWorld
  (JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
10    return;
}
```

HelloWorldImp.c

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
10 #endif
/*
 * Class :      HelloWorld
 * Method :     displayHelloWorld
 * Signature :  ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
  (JNIEnv *, jobject);

#ifdef __cplusplus
20 }
#endif
#endif
```

HelloWorld.h

FIG. 2.1 – Un Hello World écrit en JNI

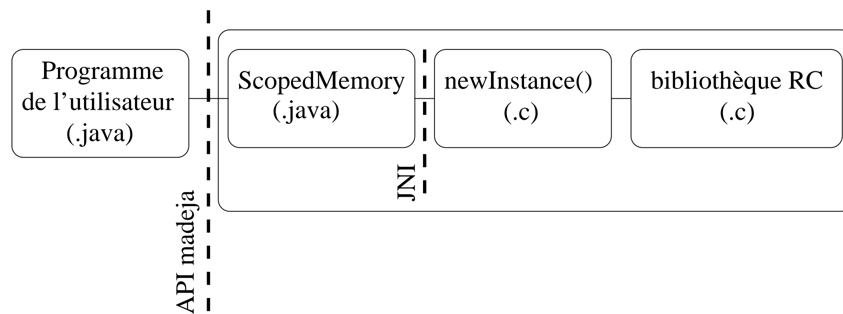

FIG. 2.2 – Architecture du prototype

2.1.2 Le choix du compilateur : TurboJ ou GCJ

TurboJ

Nous avions initialement prévu d'impléménter l'API dans le compilateur AOT TurboJ de Silicomp. TurboJ traduit le *bytecode* Java en un langage machine appelé RiscJ, représenté en XML, puis utilise une feuille de style pour transformer le RiscJ en C.

On a alors un ensemble de fichiers C correspondant aux diverses classes utilisées dans le programme, qui sont compilés tous ensemble pour produire un exécutable.

La chaîne de compilation de TurboJ (simplifiée) est présentée sur la figure 2.3. Les rectangles représentent les fichiers, les losanges les programmes, les flèches simples sont les opérations de compilation, les flèches à tête pleine veulent dire *est constitué de*.

Nous avions prévu d'intégrer la bibliothèque de RC dans le moteur d'exécution Expresso, qui implémente déjà une partie de la RTSJ, grâce à l'interface native XNI de TurboJ.

Un inconvénient majeur des programmes utilisant une interface native traditionnelle est qu'ils ne peuvent faire l'objet de traitement automatiques : en effet, bien qu'exécutés hors de l'environnement Java, ils *peuvent* manipuler l'état de la JVM, créer des objets, etc. Chaque compilateur Java utilise pour cela des API différentes côté natif, car le programme doit utiliser des fonctions internes de bas niveau dans la JVM, qui ne sont pas standardisées. Un programme qui analyserait seulement le bytecode ne pourrait que considérer les appels aux méthodes natives comme des «boîtes noires».

TurboJ transforme le bytecode Java en XML pour faciliter son traitement et son analyse, et propose une interface native qui utilise elle aussi le XML : En XNI (eXtensible Native Interface), toutes les interactions avec la JVM sont exprimées non grâce à une API, mais grâce à des fragments de XML.

Ainsi, il devient possible d'analyser automatiquement le code natif, et ce sans avoir à connaitre l'API utilisée pour accéder aux mécanismes internes de la JVM. C'est un programme appelé Xni2c qui est chargé de traduire en code (par une feuille de style) ces fragments de XML.

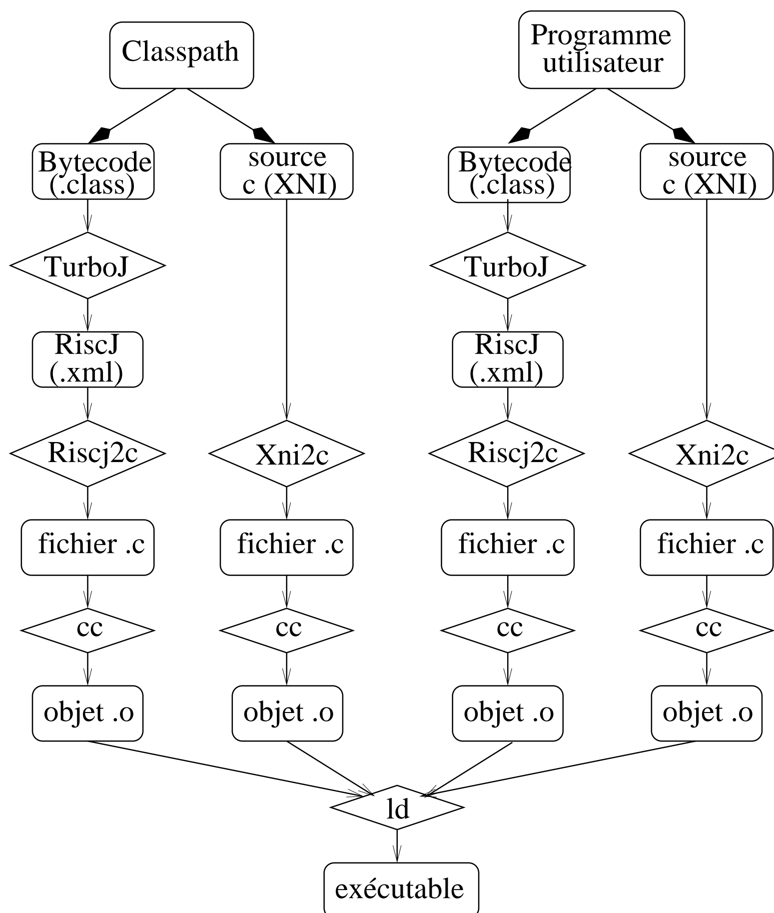

FIG. 2.3 – La chaîne de compilation de TurboJ

Le problème auquel on s'est heurté à ce moment-là essentiellement technique, mais capital : la version de TurboJ dont on disposait à ce moment-là ne supportait pas la réflexivité, et il n'était pas possible du tout d'instancier des objets par nom de classe comme nécessite l'API MADEJA.

Après une longue et fastidieuse période de tâtonnements dans le code source (non documenté) du compilateur et du Classpath, j'ai compris que je cherchais en vain : Le code XNI de `Class.newInstance()`, que je comptais reprendre en grande partie (seule changeait la manière d'allouer la zone mémoire pour y loger l'objet) était tout simplement absent, et produisait une exception dès que l'on écrivait une expression comme `RefObject.class`...

GCJ

On a donc décidé ensuite d'écrire le prototype à l'aide du compilateur Java AOT GCJ (composant de GCC), qui lui, supporte la réflexivité.

GCJ compile directement le Java en code objet, de façon assez similaire à `g++`, le compilateur C++ de GCC, de même que Java et C++ sont assez similaires. L'interface native de GCJ se nomme CNI, et consiste à programmer les parties natives en C++ (moyennant quelques restrictions).

La chaîne de compilation de GCJ est présentée sur la figure 2.4.

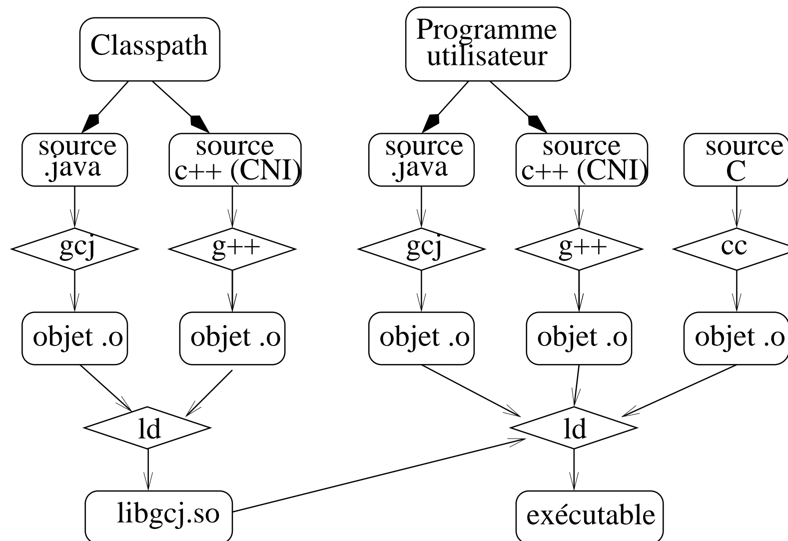

FIG. 2.4 – La chaîne de compilation de GCJ

L'API a été réprise à l'identique puisqu'elle est écrite en Java, seule a changé la partie qui instancie les objets, écrite maintenant en CNI, et inspirée du code (CNI) de `Class.newInstance()`.

2.1.3 Implémentation

L'implémentation de l'API Madeja suit rigoureusement le schéma de la figure 2.2 :

ScopedMemory

C'est la seule interface avec le programme utilisateur. Elle maintient deux variables privées :

```
// la pile des regions
static private Stack regionStack;

// le registre des captures
static private HashMap escapeMap;
```

```
public static void enter(Region r)
public static void exit()
```

Ces deux méthodes permettent d'empiler et de dépiler une région sur `regionStack`. Elles sont insérées automatiquement par l'instrumentation en début et en fin de chaque méthode.

```
public static void determineAllocationSite(String [ ]capturedSites)
```

Un appel à cette méthode est inséré à chaque site d'appel pour enregistrer les captures. C'est elle qui renseigne le registre `escapeMap`.

```
public static Object newInstance(String site,Class klass)
    throws InstantiationException,
          IllegalAccessException

public static Object newAInstance(String site,Class klass,int count)
    throws InstantiationException,
          IllegalAccessException
```

Ces deux méthodes sont insérées à l'instrumentation pour remplacer respectivement les instructions `new` et `new[]`. Elles consultent `escapeMap` pour savoir dans quelle région allouer l'objet. Pour se faire, elles font appel au méthodes natives de la classe `Region`.

A titre d'exemple, le code de `newInstance()` est donné par la figure 2.5. La méthode `r.allocObject()` est détaillée ci-après : c'est une méthode native de la classe Région.

Le code comple de l'API Madeja est donnée en annexe A.

```
public static Object newInstance(String site,Class klass)
    throws InstantiationException,
          IllegalAccessException
{
    Region r=(Region)escapeMap.get(site);

    // si personne n'a demandé a capturer l'objet on l'alloue dans
    // la région de la méthode courante.
    if( r == null)
10      r = current();

    return r.allocObject(klass);
}
```

FIG. 2.5 – Le code de `ScopedMemory.newInstance()`

Region

C'est dans cette classe que se fait le lien entre l'API Java et la bibliothèque RC. Elle ne sert de *wrapper* pour encapsuler les appels aux fonctions C.

Sa seule variable est le pointeur `r` vers la région RC correspondante.

- `private native void createRCRegion()` est appelée par le constructeur, lui-même appelé par `ScopedMemory.enter()`.
- `native void deleteRCRegion()` est appelée par `ScopedMemory.exit()`, car Java ne possède pas de notion de destructeur. Elle permet de libérer explicitement la mémoire à chaque sortie de région.
- `native Object allocObject(Class klass)` et `native Object allocObjectArray(Class klass, int count)` sont les pendants natifs des méthodes de `ScopedMemory`, qui les appelle une fois la région déterminée. Ce sont elles qui allouent la zone de mémoire qui va abriter l'objet, puis qui font les manipulations necéssaires dans la JVM (chargement de la classe, etc)

Le code (CNI) de `Region.allocObject()` est donné par la figure 2.6. Successivement, on le voit *charger la classe* (en particulier, exécuter les *initialisation statiques*), puis allouer la mémoire nécessaire grâce à la fonction `ralloc()` de la bibliothèque RC, et enfin appeler le constructeur par défaut.

```
 ::java::lang::Object *
madeja::Region::allocObject (::java::lang::Class *klass)
{
   _Jv_InitClass (klass);

   _Jv_Method *meth = _Jv_GetMethodLocal (klass, gcj::init_name,
                                          gcj::void_signature);
   if (! meth)
      throw new java::lang::NoSuchMethodException;

   jobject o=(jobject)ralloc(r,klass->size());
   if (!o) _Jv_ThrowNoMemory();

   *((_Jv_VTable **) o) = klass->vtable;
   ((void (*) (jobject)) meth->ncode) (o);
   return o;

}
```

FIG. 2.6 – Le code de `Region.AllocObject()`

Ce sont ces quelques fonctions qui ont été les plus difficiles à implémenter, car elles necéssitent une connaissance approfondie du fonctionnement interne de GCJ et de la `libgcj`, qui ne sont pas documentés.

Dans GCJ, l'ensemble du classpath est implémenté sous la forme d'une bibliothèque partagée (la `libgcj`), et l'instanciation par réflexion est opérationnelle. Mais pour comprendre comment elle fonctionnait et pouvoir reproduire ce comportement

dans ScopedMemory, il m'a fallu parcourir en profondeur le code, modifier plusieurs fois ses sources (pour voir quels fragments étaient exécutés).

La libgcj est faite pour être compilée une fois pour toutes, à chaque installation, comme la libc, or chaque modification (ajout d'une trace, etc) necéssitait de recompiler outre GCJ, tout le classpath, ce qui prenait plus de dix minutes !

Par exemple, dans le code ci-dessus, il y a un accès à class->vtable, qui est un champ privé de la classe java : :lang : :Class. Je n'ai pas trouvé de manière plus satisfaisante de résoudre le problème que de redéfinir entièrement le fichier d'entête Class.h. Pour ne pas avoir à modifier l'original, et rester compatible avec la distribution GCJ standard, j'ai créé un fichier MadejaClass.h qui, lui, permet l'accès à ce champ, et qui empêche l'inclusion du fichier d'en-tête de la distribution.

Ainsi, le code peut fonctionner sans avoir à ré-installer GCJ.

La bibliothèque de RC

Les méthodes natives de la classe Region font appel aux fonctions présentées à la section 1.1.3 : newregion(), deleteregion(). L'allocation à proprement parler se fait grâce à la primitive RC ralloc(region,size_t).

Cette bibliothèque est elle-même écrite en RC, et nécessite donc pour être compilée une série d'options de préprocesseur pour neutraliser les annotations de pointeurs.

De plus, j'ai dû modifier légèrement la primitive ralloc() car originellement sa signature est ralloc(region,type_t), ce qui permet de l'appeler avec des types en paramètre.

La primitive ralloc() était implémentée par la construction suivante :

```
#define ralloc(r, type) typed_ralloc((r), sizeof(type), rctypeof(type))
```

La forme rctypeof est spécifique à RC et ne pouvait fonctionner sans le compilateur rcc, mais je n'ai pas eu grand chose à modifier pour supprimmer son usage dans typed_ralloc() : de toutes façons, on n'a besoin ici que des régions, et pas des vérifications de types de RC.

2.2 Validation de l'approche

Pour vérifier le bon fonctionnement de mon code ainsi que pour mesurer son efficacité, j'ai fait plusieurs expérimentations avec des programmes différents : tout d'abord avec le programme donné dans [GNYZ04] (RegisterExample, assez similaire au programme donné en 1.3). Ce programme utilise plusieurs régions, et les objets s'échappent de manière non triviale, mais il utilise très peu de mémoire, et n'est certainement pas réaliste d'un point de vue programmation.

J'ai alors recherché des programmes de *benchmark* en Java dont les sources soient disponibles, pour pouvoir les instrumenter. J'ai donc examiné la suite de programmes SPEC JVM98, les programmes du Java Grande Forum Benchmark Suite ou de CaffeineMark, mais ils sont presque tous fournis sous forme *bytecode*. Les rares programmes *Java* que j'ai trouvé étaient inexploitables, surtout du fait de leur taille trop importante.

J'ai donc examiné les programmes présentés dans [GA01], qui étaient fournis à titre d'example dans la distribution de RC. Il s'agit de programmes C *portés* en RC pour profiter de l'allocation en régions. Mon but étant de les porter en Java pour leur faire utiliser l'API `ScopedMemory`, j'en ai éliminé certains (Apache, `rcc` etc), pour me concentrer sur `tile` et `cfrac`. Cependant, je n'ai pu exploiter que `tile`.

2.2.1 Le programme `Tile`

Ce programme manipule du texte. Son but est de regrouper les différents paragraphes d'un document, chaque *groupe* traitant d'un même sujet. Pour cela, il cherche quels mots-clés sont redondants entre les paragraphes en faisant des statistiques sur les occurrences des mots.

J'ai donc porté ce programme vers Java, pour faire des mesures de son usage mémoire.

Structure du programme original

J'ai tout d'abord cherché à comprendre la structure du programme : il fait deux passes sur le texte, la première pour répertorier tous les mots et leurs positions relatives, la seconde pour calculer des grandeurs significatives (distances entre les mots...). Du point de vue de la mémoire, la majorité des allocations se fait pendant la première passe, puisqu'il faut allouer les structures de données stockant les mots. La phase de calcul génère moins d'allocations.

Le programme RC est une adaptation du programme C original, modifiée pour utiliser l'API de RC (`ralloc, newregion, deleteregion`) à la place des habituels `malloc` et `free`. Cependant, son utilisation des régions n'est pas conforme à notre approche : les régions ne sont pas associées particulièrement aux méthodes, elles sont créées et détruites selon les besoins du programmeur, sans structure de pile...

L'arbre d'appel du programme RC, tel que je l'ai relevé pour y appliquer l'analyse d'échappement est donnée par la figure 2.7. Toutes les créations et destructions de régions sont indiquées.

Portage en Java

Le portage en Java a été assez naturel, méthode par méthode, en remplaçant les `struct` par des classes, les `char*` par des `String`, etc. Dans la version Java, j'ai donc adapté l'usage des régions pour les lier aux méthodes et que leur durée de vie soit imbriquée correctement, et j'ai donc fait une grossière analyse d'échappement pour assigner les bons objets aux bonnes régions. Ceci a été possible car le programme original (en C) utilise principalement des variables globales pour ses structures de données. Il suffit donc de chercher les occurrences des *noms* des variables pour déterminer dans quelles parties du programme quelles structures sont actives.

L'arbre d'appel est presque le même que l'original, seule change la manière d'utiliser les régions. Le code complet de la version Java de `Tile` est donné en annexe B

```
  hmain.c:52:region_main() :
  |
66+----htile.c:310:tile() :
  |     |
  | 325+---- region_hashtable = newregion()
  |     |
  | 326+----htile.c:828:mksentarrays()
  |     |     |
  |     | 831+-- newh = newregion()
  |     |     |
  |     | 850+-- if (region_arrays) deleteregion_ptr(&region_arrays);
  |     | 851`-- region_arrays = newh;
  |     |
  | 335+----htile.c:242:init_stopword_table()
  |     |     |
  |     | 245`-- region_stop = newregion()
  |     |
  | 338+----htile.c:338: while( str = yylex() )
  |     |     |
  |     | 366`--htile.c:828:mksentarrays()
  |     |        |
  |     |    831+-- newh = newregion()
  |     |        |
  |     |    850+-- if (region_arrays) deleteregion_ptr(&region_arrays);
  |     |    851`-- region_arrays = newh;
  |     |
  | 377+----htile.c:390:process_input()
  |     |     |
  |     | 392+--htile.c:577:compute_all_sim()
  |     |     |
  |     | 393+--htile.c:492:smooth()
  |     |     |
  |     | 394`--htile.c:673:tile_locs()
  |     |        |
  |     |    690+-- local = newregion()
  |     |        |
  |     |    775+-- region_tiledoc = newregion();
  |     |        |
  |     |    803`-- deleteregion(local)
  |     |
  | 382+----htile.c:277:ai_free()
  |     |     |
  |     |    `-- deleteregion_ptr(&region_hashtable)
  |     |
  | 383`----htile.c:867:freesentarrays()
  |           |
  |          `-- deleteregion_ptr(&region_arrays)
  |
75+----hmain.c:133:showtext()
  |    hmain.c:170:showoffsets()
  |
77 `---htile.c:859:freetiledoc()
        |
    861`-- deleteregion_ptr(&region_tiledoc)
```

FIG. 2.7 – Structure du programme `Tile`

2.2.2 Expérimentations

Pour évaluer l'efficacité de l'allocation en régions, j'ai fait des mesures de l'utilisation mémoire du programme.

Efficacité de l'allocation en régions

Sur la figure 2.8, on a représenté, en trait pointillé, l'espace occupé par les régions, et en trait plein l'espace occupé par le tas du programme.

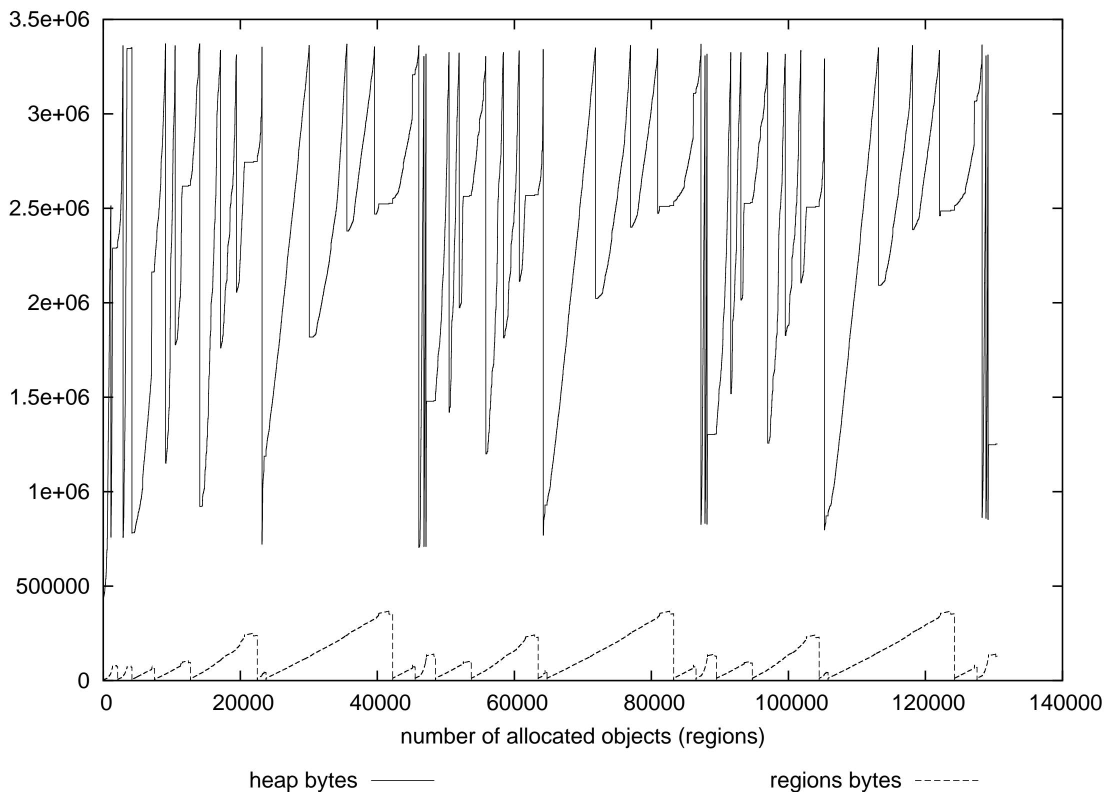

FIG. 2.8 – Espace utilisé par les régions et par le tas

La courbe pleine reflète la présence d'une grande quantité de mémoire *hors* des régions, car il m'a été impossible de tenir le compte de *toutes* les allocations faites par le programme. En effet il existe de nombreuses allocations *automatiques* en Java, impossibles à détecter au niveau source (comme par exemple chaque concaténation de chaînes, qui donne lieu à un `new StringBuffer`).

Pourtant, une quantité considérable de mémoire a été allouée *dans* les régions.

Cette mémoire est ici bornée avec le temps, car le programme ne comportait pas de région *immortelle*. En effet, certains schémas de programmation peuvent mener à des régions jamais désallouées, comme le souligne [FHPV04]. Dans une programmation en régions, lorsqu'une structure de données comme une table de hachage ou une liste, est amenée à pointer sur de nombreux objets qui changent avec le temps, *tous* ces objets doivent vivre *au moins aussi longtemps* que la structure. C'est ce qui arrive ici

avec certains tableaux de mots, qui deviennent inactifs au cours de la première passe pour être remplacés par des tableaux plus grands.

Le programme RC contourne ce problème en détruisant la région puis en en recréant une nouvelle au milieu de la boucle (dans la fonction `mksentarrays()`). Les durées de vie des régions ne sont alors plus imbriquées. Dans la version Java, toute cette mémoire est allouée dans la même région, et ne peut être désallouée qu'à la fin de la seconde passe.

Ce genre de problème relève plus de l'analyse d'échappement, qui pourrait déterminer que la durée de vie d'un objet n'excède pas une certaine boucle, que de l'API de régions elle-même, c'est pourquoi je n'ai pas cherché à le régler.

Comparaison avec le ramasse-miettes

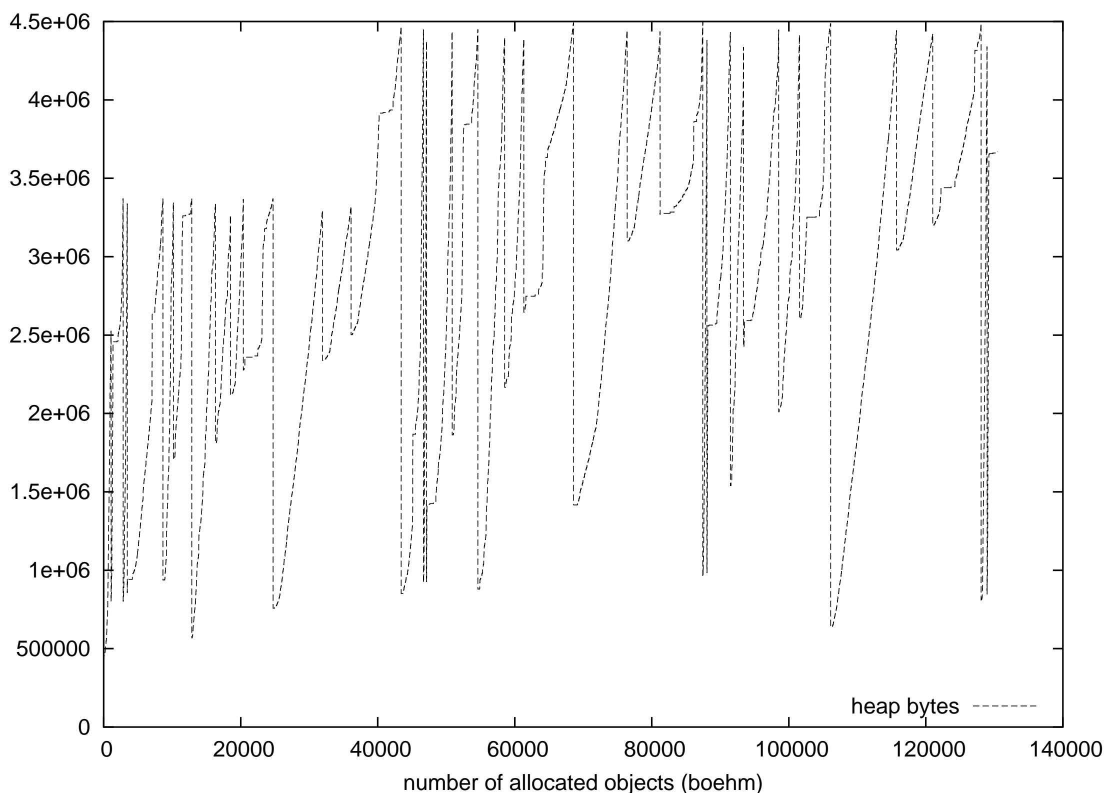

FIG. 2.9 – Le programme est ici exécuté sans gestion de la mémoire par régions. Toutes les allocations se font par l'intermédiaire du GC.

On peut aussi comparer la courbe de la figure 2.8 et celle obtenue avec un GC. Celui de GCJ est de type Boehm-Weiser : il arrête le programme périodiquement pour examiner le tas et collecter la mémoire inutilisée, mais sans *modifier* le contenu de la mémoire active.

La courbe présentée sur la figure 2.9 montre la taille totale du tas de la JVM.

Les traits verticaux décroissants reflètent les collectes, pendant lesquelles le ramasse-miettes parcourt le tas pour détecter la mémoire inutilisée.

Il est donc intéressant de tracer sur un même graphique les deux consommations mémoire. La figure 2.10 montre, en trait pointillé, la taille du tas, pour une exécution avec ramasse-miettes (la même que sur la figure 2.9). En trait plein est représentée l'addition de la taille du tas et de la place occuppée par les régions, pour une exécution *avec* régions.

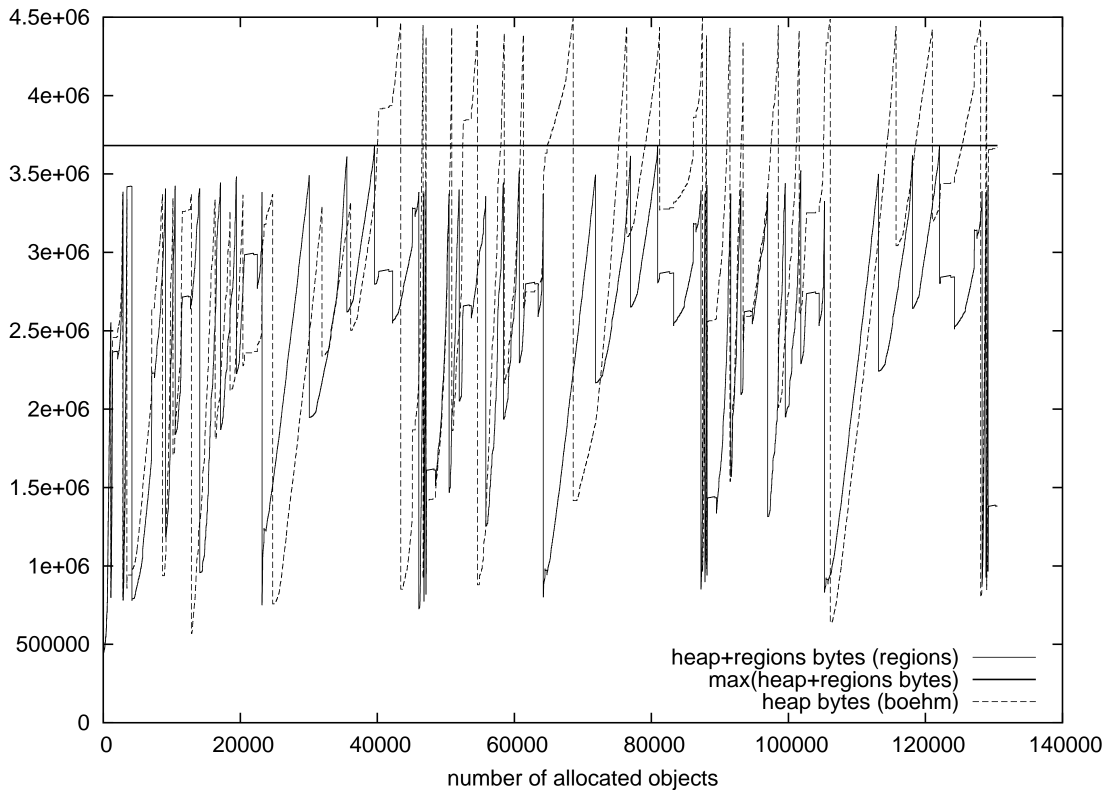

FIG. 2.10 – Comparaison des consommations mémoire. Les deux gestionnaires de mémoire sont représentés.

Les deux courbes sont assez similaires, ce qui signifie que la grande majortié de la mémoire est allouée *hors* des sites d'allocations identifiés. Seule une analyse d'échappement plus fine (en particulier faite au niveau bytecode, par exemple) permettrait d'allouer ces données en régions.

Néanmoins on peut remarquer que les pics d'utilisation de mémoire sont moins hauts *avec* les régions (un trait horizontal est tracé pour refléter le plus haut pic d'occupation du programme avec régions)

C'est grâce à la désallocation précoce de la mémoire permise par l'analyse d'échappement que cette mémoire est économisée. Le ramasse-miettes détecte bien la mémoire inutilisée, mais plus tard, et entre-temps la JVM est obligée d'augmenter la taille du tas.

La figure 2.11 montre en détail le comportement des deux gestionnaires de mémoire.

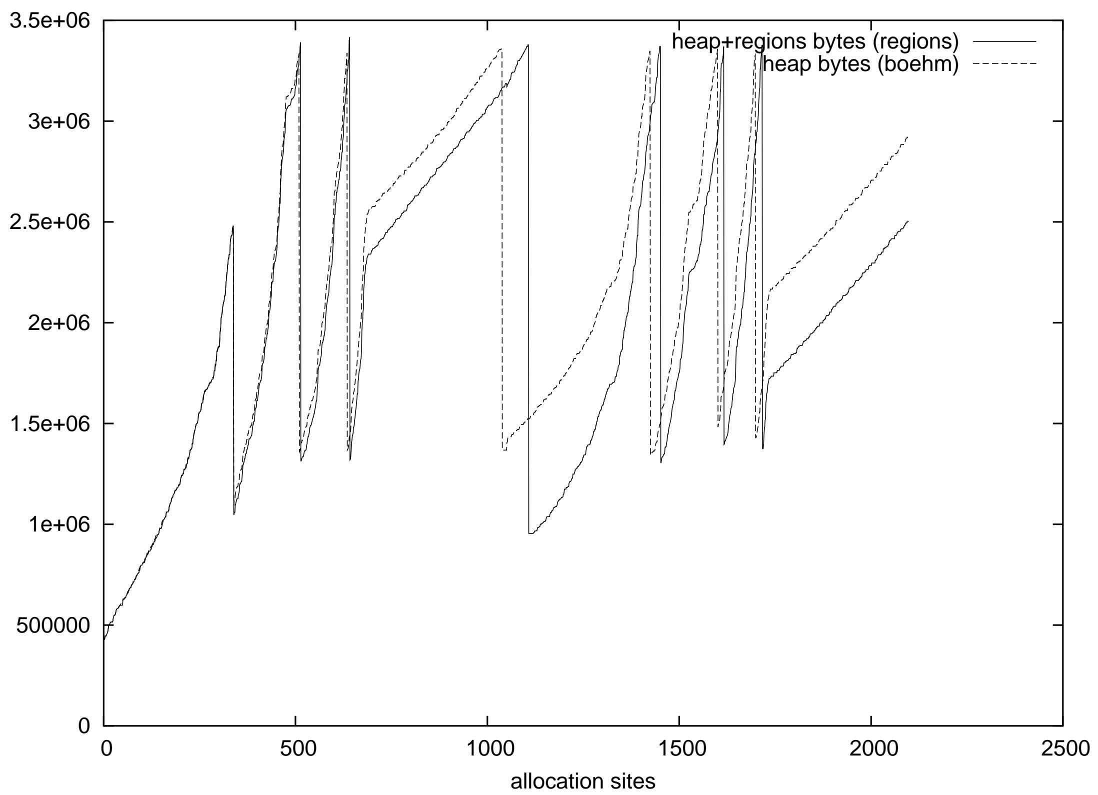

FIG. 2.11 – Comparaison des consommations mémoire (détail) : Les deux gestionnaires de mémoire sont représentés.

Pendant le permière partie de la trace (jusqu'à 650 objets alloués environ), les deux courbes sont similaires.

C'est alors que la version sans régions du programme commence à utiliser plus de mémoire, car certaines régions on déjà été quittées (donc leur mémoire libérée) alors que le GC n'a pas remarqué la mort de ces objets.

Pendant toute cette période, le programme avec régions s'exécute dans un espace plus restreint que la version GC. Cette partie de la courbe est intéressante, car, dans une optique de *multithreading*, un *autre* thread pourrait exploiter la mémoire gagnée pendant cette période.

Les allocations se font à peu près au même rythme dans les deux programmes, donc le seuil qui déclenche la collecte est atteinte *plus tôt* par le programme sans régions, c'est pourquoi la courbe pointillée subit sa décroissance en premier.

Le programme en régions n'a une occuppation mémoire supérieure à la version GC que pendant un court moment, car notre programme subit lui aussi une collecte du tas, suivie d'une libération de la mémoire des régions.

Dans une système multithread, cette utilisation explicite des régions permettrait de faciliter l'ordonnancement du programme : lors d'un `exit()`, si un thread plus prioritaire était suspendu en attente de mémoire, l'ordonnanceur peut alors le lancer.

Tout au long de l'exécution, l'utilisation des régions, même sur une faible proportion de la mémoire puisqu'elles sont utilisées seulement pour les sites d'allocation du programme utilisateur, permet ainsi de réduire sensiblement la consommation mémoire moyenne.

Conclusion et perspectives

Ce travail était essentiellement tourné vers l'implémentation, cependant il a nécéssité un certain travail de compréhension théorique préalable. Les théories qui soustendent l'allocation de mémoire par régions ne sont pas encore bien établies, et faire le lien entre les différents articles lus était parfois difficile.

Une fois bien indentifié le travail à faire, l'étude des outils à utiliser (RC, GCJ et la `libgcj`, TurboJ) et de leur fonctionnement interne a été très longue, surtout à cause du manque de documentation disponible. J'ai cependant acquis beacoup de connaissances sur les compilateurs et sur le langage Java (bytecode, structure du Classpath, sandboxing, fonctionnement des JVM, etc).

Même si l'implémentation n'a pas pu être réalisée dans le compilateur TurboJ, j'ai pu écrire pour gcj un prototype rudimentaire (environ 1000 lignes en tout) mais suffisant pour faire des premières mesures et vérifier la pertinence de l'approche. Le portage de programmes de tests a aussi constitué une tâche notable, car le programme `tile` représente environ 1300 lignes.

Grâce aux mesures qui comparent l'occupation mémoire du programme *avec* régions et celle avec le GC seul, on a pu constater que la gestion de la mémoire dynamique par régions permet de réduire sensiblement les besoins en mémoire.

Pourtant, dans ce travail, seule une faible proportion des sites d'allocation sont considérés. Une piste intéressante, à poursuivre dans le cadre d'une thèse, serait de raffiner l'analyse, pour pouvoir contrôler l'allocation de *tous* les objets.

En particulier, cette analyse devra se faire au niveau *bytecode*, car certaines allocations sont invisibles au niveau source (par exemple le `new StringBuffer` lors d'une concaténation de chaînes). Pour cela, il sera indispensable d'effectuer cette analyse dans TurboJ, car GCJ passe directement du source Java au code objet.

L'instrumentation du code n'est même pas forcément nécessaire : dans le compilateur TurboJ, le programme C est produit par la combinaison d'un fichier XML et d'une feuille de style. En modifiant le fragment de code émis pour l'instruction `new`, pour l'entrée dans une méthode, etc, on peut intégrer la gestion des régions sans avoir à modifier le source Java, ni le bytecode. En faisant aussi l'analyse d'échappement pendant la compilation, on peut, tout en rendant le processus complètement transparent pour le programmeur, aboutir à une chaîne de compilation bien plus simple et naturelle.

De plus, il faudra aussi analyser la bibliothèque standard, car de nombreuses allocations y sont faites. L'analyse du programme utilisateur seul ne permet pas de rendre compte fidèlement du comportement mémoire du programme.

Les perspectives pour approfondir ce travail sont nombreuses : outre l'analyse à déplacer à plus bas niveau, on peut envisager plusieurs points intéressants :

Actuellement, l'analyse d'échappement construit l'arbre d'appel complet du programme, ce qui suppose qu'on peut le déterminer entièrement à l'avance. Dans le cas général, ce ne serait certainement pas aussi simple, et trouver des représentations appropriées de l'arbre d'appel permettra de traiter des programmes plus complexes, donc plus réalistes. En particulier, sachant que le problème général est indécidable, comment cette analyse statique se comporte-t-elle en présence de récursivité ?

Les programmes réels utilisent souvent plusieurs threads, et la RTSJ est centrée autour du multithreading. Comment appréhender ce problème du point de vue de l'analyse statique ?

Les régions sont actuellement associées aux méthodes. Pourtant, on a remarqué pendant ce stage que cette stratégie n'était pas forcément la meilleure dans tous les cas : par exemple, si une boucle utilise des structures de données qui ne restent pas actives d'une itération sur l'autre, il serait intéressant de placer l'entrée et la sortie de région (`enter()` et `exit()`) en début de boucle et en fin de boucle. Différentes stratégies de *placement* des régions sont ainsi envisageables, qu'il serait instructif d'étudier.

La gestion de la mémoire dynamique en régions permet, de plus, d'améliorer les performances temporelles du programme, et en particulier rend son temps d'exécution plus facilement prédictible.

L'interprétation abstraite permet aussi de *prévoir* dans une certaine mesure la consommation mémoire du programme, pour dimmensionner correctement les régions à l'avance. Il serait intéressant d'utiliser notre prototype pour valider expérimentalement les mesures paramétriques obtenues par la méthode de [BGY03]

Bibliographie

[BGY03] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. On synthesizing parametric specifications of dynamic memory utilization. Technical report, VERIMAG, 2003.

[Bla99] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 20–34, Denver, CO, October 1999. ACM Press.

[Bol00] RTSJ Gregory Bollella. *The real-time specification for Java*. Java series. Addison-Wesley, Reading, MA, USA, 2000.

[BZM02] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, November 2002. ACM Press.

[DC02] Morgan Deters and Ron Cytron. Automated discovery of scoped memory regions for real-time Java. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 25–35, Berlin, June 2002. ACM Press.

[FHPV04] Jason M. Fox, David Holmes, Filip Pizlo, and Jan Vitek. Real-time java scoped memory; design patterns and semantics, 2004. http ://www.ovmj.org/documents.html.

[GA98] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 313–323, Montreal, June 1998. ACM Press.

[GA01] David Gay and Alex Aiken. Language support for regions. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[GNYZ04] Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in java. In *Proceedings of "Runtime Verification (RV'04)", ETAPS'04*, 2004.

[Jon96] Richard E. Jones. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997.

Annexe A

Le code de l'API Madeja

A.1 ScopedMemory.java

C'est cette classe qui contient toutes les méthodes publiques de l'API. son fonctionnement et son utilisation on été expliqués à la section 2.1.3

```
package madeja;

import java.util.*;
import java.io.*;

public final class ScopedMemory
{
    private ScopedMemory() {}
10
    // la pile des regions
    static private Stack regionStack;

    // le registre des captures
    static private HashMap escapeMap;

    static
    {
        //System.out.println("ScopedMemory.<clinit>");
20      regionStack=new Stack();
        escapeMap=new HashMap();
    }

    public static Object newAInstance(String site,
                                       Class aklass,
                                       int count)
        throws InstantiationException,
               IllegalAccessException
    {
30      boolean top=false;

        Region r=(Region)escapeMap.get(site);

        // si personne n'a demandé a capturer l'objet on l'alloue dans
        // la région de la méthode courante.
```

```
            if( r == null)
            {
                r = current();
                top=true;
40          }

            Class klass=aklass.getComponentType();


            return  (Object)r.allocObjectArray(klass,count);
        }

        public static Object newInstance(String site,Class klass)
            throws InstantiationException,
50                 IllegalAccessException
        {
            boolean top=false;

            Region r=(Region)escapeMap.get(site);

            // si personne n'a demandé a capturer l'objet on l'alloue dans
            // la région de la méthode courante.
            if( r == null)
            {
60              r = current();
                top = true;
            }

            return r.allocObject(klass);
        }

        public static void determineAllocationSite(String [ ]capturedSites)
        {
            for(int i=0; i<capturedSites.length; i++)
70          {
                escapeMap.put(capturedSites[i],current());
            }
        }

        public static void enter(Region r)
        {
            regionStack.push(r);
        }

80      public static Region current()
        {
            return (Region)regionStack.peek();
        }

        public static void exit()
        {
            Region r=(Region)regionStack.pop();
            r.deleteRCRegion();
        }
90  }
```

A.2 `Region.java`

La classe `Region` fait le lien entre `ScopedMemory` Java et la bibliothèque RC. Elle sert de *wrapper* pour encapsuler les appels aux fonctions C : en effet, la plupart de ses méthodes sont natives.

```
package madeja;

import gnu.gcj.*;

public class Region
{
    static
    {
        initRCRegions();
    }

    String name;

    gnu.gcj.RawData r;

    public Region(String name)
    {
        this.name=name;

        createRCRegion();
    }

    // createRCRegion est appelée par le constructeur,
    private native void createRCRegion();

    // mais deleteRCRegion est a appeler par ScopedMemory.exit()
    native void deleteRCRegion();

    // celle-la est appelée par le <clinit> de la classe Region
    private static native void initRCRegions();

    native Object allocObject(Class klass)
        throws InstantiationException,
               IllegalAccessException;

    native Object allocObjectArray(Class klass,
                                   int count)
        throws InstantiationException,
               IllegalAccessException;

}
```

A.3 `natRegion.cc`

C'est dans ce fichier que sont implémentées les méthodes natives de `Region`. Les méthodes `allocObjectArray()` et `allocObject()` effectuent, en plus des allocations mémoire proprement dites, les opérations necéssaires dans la JVM.

```
#include "madejaClass.h"

// GCJ qu'il faut les sources et/ou les objets...
//#include <jvm.h>            // src/libjava/include/jvm.h
#include "myjvm.h"

// la libregions de RC
#include "regions.h"

10 #include <stdio.h>
#include <stdlib.h>

#include <assert.h>

#include <Region.h>
#include <gcj/cni.h>
#include <java/io/PrintStream.h>
#include <java/lang/InstantiationException.h>
#include <java/lang/NoSuchMethodException.h>
20 #include <java/lang/System.h>
#include <java/lang/UnsupportedOperationException.h>
#include <java/lang/IllegalArgumentException.h>
#include <java/lang/NegativeArraySizeException.h>

// src/libjava/prims.cc :74
// Largest representable size_t.
#define SIZE_T_MAX ((size_t) (~ (size_t) 0))

void
30 madeja : :Region : :createRCRegion()
{
    r=newregion();
}

void
madeja : :Region : :deleteRCRegion()
{
    deleteregion(r);
}
40
void
madeja : :Region : :initRCRegions()
{
    region_init();
}

 : :java : :lang : :Object *
madeja : :Region : :allocObjectArray( : :java : :lang : :Class *klass,
                                    jint count)
50 {
    // Nota bene : on remarque que, a la différence de
    // allocObject, la classe elle-même (klass) n'est pas
    // initialisée (<clinit>) ni que les constructeurs
    // appelés pour les éléments du tableau : effectivement,
    // le tableau alloué est plein de 'null' a ce moment la
    // (sémantique idem java)
```

```
       // src/libjava/java/lang/reflect/natArray.java :31
        if (klass->isPrimitive())
60      {
            if (klass == JvPrimClass (void))
               throw new java : :lang : :IllegalArgumentException ();

            // src/libjava/prims.cc :519 :_Jv_NewPrimArray
            int elsize = klass->size();

            if (count < 0)
                throw new java : :lang : :NegativeArraySizeException;

70          jobject dummy = NULL;
            size_t size = (size_t)
                _Madeja_GetArrayElementFromElementType (dummy, klass);

            // Check for overflow.
            if ((size_t) count > (SIZE_T_MAX - size) / elsize)
                _Jv_ThrowNoMemory();

            // dest/include/java/lang/Class.h :334 :_Jv_GetArrayClass
            if ( !klass->arrayclass)
80              _Jv_NewArrayClass (klass, NULL);
            jclass aklass = klass->arrayclass;
            // fin

            // src/libhava/nogc.cc :29 :_Jv_AllocObj
            __JArray *arr = (__JArray*) ralloc(r,size + elsize * count);
            if ( !arr) _Jv_ThrowNoMemory();
            *((_Jv_VTable **) arr) = aklass->vtable;
            // fin
            jsize *lp = const_cast<jsize *> (&arr->length);
90          *lp = count;

            return arr;
        }
        else
        {
            // src/libjava/prims.cc :486 :_Jv_NewObjectArray

            if (count < 0)
                throw new java : :lang : :NegativeArraySizeException;
100
            // Ensure that elements pointer is properly aligned.
            jobjectArray obj = NULL;
            size_t size = (size_t) elements (obj);
            size += count * sizeof (jobject);

            // dest/include/java/lang/Class.h :334 :_Jv_GetArrayClass
            if ( !klass->arrayclass)
                _Jv_NewArrayClass (klass, NULL);
            jclass aklass = klass->arrayclass;
110         // fin

            // src/libjava/nogc.cc :49 :_Jv_AllocArray
            obj=(jobjectArray)ralloc(r,size);
            if ( !obj) _Jv_ThrowNoMemory();
```

```
        *((_Jv_VTable **) obj) = aklass->vtable;
        // fin

        // Cast away const.
        jsize *lp = const_cast<jsize *> (&obj->length);
120     *lp = count;

        return obj;
    }
}

 ::java::lang::Object *
madeja::Region::allocObject (::java::lang::Class *klass)
{
    // src/libjava/java/lang/natClass.cc:700:newInstance
130
    _Jv_InitClass (klass);

    _Jv_Method *meth = _Jv_GetMethodLocal (klass, gcj::init_name,
                                           gcj::void_signature);
    if (! meth)
        throw new java::lang::NoSuchMethodException;

    // src/libjava/prims.cc:429:_Jv_AllocObject
    jobject o=(jobject)ralloc(r,klass->size());
140 if (!o) _Jv_ThrowNoMemory();
    *((_Jv_VTable **) o) = klass->vtable;

    // nota bene : vtable est un champ privé de Class. il a
    // donc fallu insérer friend class madeja::Region; dans
    // madejaClass.h:316, qui #define __JAVA_LANG_CLASS_H__
    // De cette manière, on n'inclut pas le
    // dest/include/java/lang/Class.h

    // retour dans natClass.cc:newInstance
150
    // appeler le constructeur par défaut.
    ((void (*) (jobject)) meth->ncode) (o);
    return o;
}
```

Annexe B

Le programme tile

B.1 TileMain.java

Ici se trouve la fonction main du programme tile. Elle ne fait rien de plus qu'une boucle sur ses arguments pour traiter successivement tous les fichiers d'entrée. Chaque fichier est passé à la fonction Htile.tile(), puis les résultats sont affichés.

```
import java.io.*;
import java.util.*;

import madeja.*;

class TileMain
{
    static TileDoc tdp;

10  static boolean f_showoffsets=false;
    static boolean verbose      =false;
    static int indented         =0; /* process indented text as a paragraph */
    static boolean not_para_boundaries=false;

    static int bound =2;
    static int numiter = 1;
    static int this_k = 4;
    static int word_sep_num = 20;

20  public static void main(String args[ ])
    {
        int i=getopts(args);

        for(;i<args.length;i++)
        {
            ScopedMemory.enter(new Region("main"));
            String main_captures[ ]={"tile_locs_6","tile_locs_7","tile_locs_8"};
            ScopedMemory.determineAllocationSite(main_captures);

30          if(verbose)
                System.err.println("processing "+args[i]);

            if( (tdp = HTile.tile(args[i])) == null )
            {
```

```
                System.err.println("Erreur! "+args[i]);
                System.exit(1);
            }

            if( f_showoffsets)
40              showOffsets(args[i],tdp);
            else
                showText(args[i],tdp);

            ScopedMemory.exit();
        }

    }

    // returns the index or the first non-option arg
50  static int getopts(String[ ] args)
    {
        int i;
        for(i=0;i<args.length;i++)
        {
            if(args[i].equals("-o"))
                f_showoffsets=true;
            else if (args[i].equals("-i"))
            {
                System.err.println("non implémenté!");
60              System.exit(1);
                indented++;
            }
            else if (args[i].equals("-b"))
            {
                i++;
                bound=Integer.parseInt(args[i]);
            }
            else if (args[i].equals("-n"))
            {
70              i++;
                numiter=Integer.parseInt(args[i]);
            }
            else if (args[i].equals("-k"))
            {
                i++;
                this_k=Integer.parseInt(args[i]);
            }
            else if (args[i].equals("-w"))
            {
80              i++;
                word_sep_num=Integer.parseInt(args[i]);
            }
            else if (args[i].equals("-p"))
                not_para_boundaries=true;
            else if (args[i].equals("-v"))
                verbose=true;
            else
                break;
        }
90      return i;
    }
```

```
        static void showOffsets(String fname,
                                TileDoc tdp)
        {
            System.out.println("Not implemented!\n");
        }


100     static void showText(String fname,
                        TileDoc tdp)
        {
            try{
                RandomAccessFile in=new RandomAccessFile(fname,"r");

                int tx, c;
                long i;
                Tile tp[] = tdp.tilearray;

110             for (tx = 0; tx < tdp.numtiles; tx++)
                {
                    if (verbose)
                        System.out.println
                             ("<TILE "+tx+" - FILE: "+fname+" START: "
                              +tp[tx].startoff+" END: "+tp[tx].endoff+">");
                    in.seek(tp[tx].startoff);
                    for(i=tp[tx].startoff;i<tp[tx].endoff;i++)
                    {
                        c=in.read();
120                     if(c==-1)
                            break;
                        else
                            System.out.print((char)c);
                    }

                    if (verbose)
                        System.out.println
                             ("</TILE "+tx+" - FILE: "+fname+" START: "
                             +tp[tx].startoff+" END: "+tp[tx].endoff+">");
130
                }
            }
            catch(Throwable t)
            {
                t.printStackTrace();
                System.exit(1);
            }
        }
    }
```

B.2 Htile.java

Dans ce fichier se trouvent les méthodes principales du programme. Pour la plupart, je n'ai pas bien compris ce qu'elles faisaient. Seul m'importait leur comportement mémoire. J'ai instrumenté le code pour lui faire utiliser l'API Madeja.

```
import java.util.*;

import madeja.*;

// cf htile.c:119
class AInfo
{
    int current_count;
    int para_count;
    int last_para;
    int wloc[];
}

class Tile
{
    int startoff;
    int endoff;
}

class TileDoc
{
    long numtiles;
    Tile  tilearray[];
}



class Sortf implements Comparable
{
    double val;
    int    sgap;

    public int compareTo(Object o)
        throws ClassCastException
    {
        Sortf s=(Sortf)o;
        if(val > s.val) return -1;
        if(val < s.val) return +1;
        return 0;
    }

}

class TileLocEnv
{
    long    bounds[];
    double  sss [];
    int     count;
}

class HTile
{
    static long  startoftext,endoftext;
    static int          wordcount;
    static int          max_sent;
    static int          max_word;
    static int          max_para;
```

```
        static int    sentpara[];
60      static int    paralocs[];
        static double w1[];
        static double w2[];
        static int    sentarray[];

        static int cur_sentsize=200; // htile.c:104
        static int GROWSENT= 200;     // htile.c:321

        static TileDoc tile(String filename)
        {
70          int outwordcount = 0;

            String str;

            TileDoc td;

            int snum = 1;
            int para = 1;
            boolean seen_para_break = false;

80          Tokizer tk=null;
            try
            {
                startoftext = 0;
                max_sent=0;
                max_para=0;


                ScopedMemory.enter(new Region("region_tile"));
                String tile_captures[]={"process_token_1","process_token_2",
90                                      "ai_find_1","realloc_harray_1"};
                ScopedMemory.determineAllocationSite(tile_captures);

                // init_stopword_table
                try{
                    sw_hm=(HashMap)HashMap.class.newInstance();
                }
                catch(Throwable t)
                { t.printStackTrace(); System.exit(1); }

100             for(int i=0; i<common_words.length; i++)
                    stopword_add(common_words[i]);
                // fin init_stopword_table

                mksentarrays(0,cur_sentsize);

                // fopen et compagnie
                tk=new Tokizer(filename);

                // init_ai
110             try{
                    ai_hm=(HashMap)HashMap.class.newInstance();
                }
                catch(Throwable t)
                { t.printStackTrace(); System.exit(1); }
```

```
            // fin init_ai

            while( (str= tk.lex()) != null  )
            {
                if(TileMain.verbose)
120                 System.out.println("token: "+str+
                                    "    type: "+tk.token_type);

                if(tk.token_type == tk.TEXT_TOKEN )//1
                {
                    process_token(str,snum,para);
                    outwordcount++;
                    seen_para_break=false;
                    if((outwordcount % TileMain.word_sep_num) == 0)
                        snum++;
130
                }
                else if ((tk.token_type == tk.BLANK_TOKEN)//3
                         && (sentpara[snum] == 0) && (sentpara[snum-1]==0))
                {
                    if( !seen_para_break)
                    {
                        sentpara[snum]=para;
                        paralocs[snum]=tk.bytes;
                        para++;
140                 }
                    seen_para_break=true;
                }
                if(snum >= cur_sentsize )
                {
                    mksentarrays(cur_sentsize, cur_sentsize + GROWSENT);
                    cur_sentsize += GROWSENT;
                }
            }
        } catch(Throwable t)
150     {t.printStackTrace();System.exit(1);}

        endoftext=tk.bytes;
        max_sent= snum;
        max_para=para;

        /*   Determine the tile locations.          */
        td = process_input();

        // cleanup:
160     // == stopword_free,        freesentarrays, ai_free
        ScopedMemory.exit(); // deletes region_tile

        return td;

    }

    /* htile.c:611
     *
     * Record the fact we saw a term.        We use a tcl hash table for no
170  * better reason than it was handy at the time.           We keep track of the
     * sentence number and paragraph number it occured in.
```

```
 */
static void process_token(String str,int sent,int para)
{
    String term=str.toLowerCase();
    if(!stopword_find(term))
    {
        // align_hashkey???
        String buf=term;
180
        boolean nouveau[]=new boolean[1];
        nouveau[0]=false;
        AInfo infoPtr = ai_find(buf,nouveau);

        if(nouveau[0])
        {
            try{// capturé par region_hashtable
                infoPtr.wloc=(int[])ScopedMemory.newAInstance("process_token_1",
                                                              int[].class,10);
190         } catch(Throwable t)
            { t.printStackTrace(); System.exit(1); }
            infoPtr.current_count=0;
            infoPtr.last_para= -1;
            infoPtr.para_count=0;
        }

        if(infoPtr.current_count >= infoPtr.wloc.length)
        {
            int [] new_wloc=null;
200         try{// capturé par region_hashtable
                new_wloc=(int[])ScopedMemory.newAInstance("process_token_2",
                                                          int[].class,
                                                          infoPtr.wloc.length*2);
            } catch(Throwable t)
            { t.printStackTrace(); System.exit(1); }

            for(int i = 0; i < infoPtr.current_count; i++)
                new_wloc[i] = infoPtr.wloc[i];

210         /* rest of the fields are already zero from initialization. */
            infoPtr.wloc=new_wloc;
        }

        infoPtr.wloc[infoPtr.current_count] = sent;

        if( para > infoPtr.last_para)
        {
            infoPtr.last_para = para;
            infoPtr.para_count++;
220     }
        infoPtr.current_count++;

        if(TileMain.verbose)
        {
            System.out.println("* wrd: "+buf
                               +" current: "+infoPtr.current_count
                               +" paracnt: "+infoPtr.para_count
                               +" lastpapr: "+infoPtr.last_para
```

```
                                    +" wlocsize : "+infoPtr.wloc.length);
230             }
            }
        }

        /*****************************************************
         *
         *                   Process-input et ses amies
         *
         *****************************************************/

240     static TileDoc process_input()
        {
            ScopedMemory.enter(new Region("region_process_input"));

            String process_input_captures[ ]={""};
            ScopedMemory.determineAllocationSite(process_input_captures);

            try{
                // compute_all_sim, smooth
                w1=(double[ ])ScopedMemory.newAInstance("process_input_1",
250                                             double[ ].class,cur_sentsize);
                // smooth, tile_locs_compute
                w2=(double[ ])ScopedMemory.newAInstance("process_input_2",
                                                double[ ].class,cur_sentsize);
                // seulement (!) compute_all_sim
                sentarray=(int[ ])ScopedMemory.newAInstance("process_input_3",
                                                    int[ ].class,cur_sentsize);
            }
            catch(Throwable t)
            { t.printStackTrace(); System.exit(1); }
260
            compute_all_sim();
            smooth();
            ScopedMemory.exit();
            TileDoc td=tile_locs();
            return td;
        }

        static void compute_sim(int ss,int k)
        {
270         double          num, w1s, w2s, s1, s2, temp;
            int j;
            int ss2;
            AInfo ip;

            ss2 = ss + k;
            num = w1s = w2s = 0.0;
            Collection col=ai_hm.values();
            Iterator it=col.iterator();

280         while( it.hasNext() )
            {
                ip=(AInfo)it.next();
                s1 = s2 = 0.0;
                wloc_to_sentarray(ip.wloc, ip.current_count);
                for (j = ss; j < (ss + k); j++) {
```

```
                s1 += sentarray[j];
            }
            for (j = ss2; j < (ss2 + k); j++) {
                s2 += sentarray[j];
290         }
            num += s1 * s2;
            w1s += s1 * s1;
            w2s += s2 * s2;
        }
        temp = Math.sqrt(w1s * w2s);
        w1[ss+k−1] = (temp!=0)? (num / temp) : 0.0;
    }

    static void compute_all_sim()
300 {
        int i;
        /* use a smaller amount of context at the beginning and end */
        for (i = 1; i < TileMain.this_k; i++) {
            compute_sim(i, 3);
        }
        /*??? - doesn't this want to start at "this_k"? */
        for (i = 1; i <= (max_sent − (TileMain.this_k * 2) + 2); i++) {
            compute_sim(i, TileMain.this_k);
        }
310     for (i = (max_sent − TileMain.this_k); i < (max_sent − 4); i++) {
            compute_sim(i, 3);
        }
    }

    /* htile.c :595
     *
     * Convert the sentence location array (wloc) to an array indexible
     * by sentence number.
     */
320 static void wloc_to_sentarray(int wloc[ ], int current_count)
    {
        int i;
        for(i=0;i<cur_sentsize;i++)
            sentarray[i]=0;
        for (i = 0; i < current_count; i++)
            sentarray[wloc[i]]++;
    }

    static String format_3lf(double d)
330 {
        String s=Double.toString(Math.round(d*1000)/1000.);
        while(s.length()<5) s+="0";
        return s;
    }

    static void smooth()
    {
        int i, j, k;
        double       nw;
340     int count;

        if (TileMain.verbose)
```

```
        {
            System.out.print("\n\n");
            for (j = 1; j < max_sent; j++)
                System.out.println("**** "+j+" "+format_3lf(w1[j]));
        }
        for (j = 1; j < TileMain.this_k; j++)
            w2[j] = w1[j];
350
        for (j = (max_sent − TileMain.this_k); j < max_sent; j++)
            w2[j] = w1[j];
        for (k = 0; k < TileMain.numiter; k++)
        {
            for (j = TileMain.this_k; j < (max_sent − TileMain.this_k); j++)
            {
                nw = 0.0;
                count = 0;
                for (i = (−TileMain.bound + 1); i < TileMain.bound; i++)
360             {
                    nw += w1[j+i];
                    count++;
                }
                w2[j] = nw / (double)count;
            }
            for (j = 1; j < max_sent; j++)
                w1[j] = w2[j];
        }
    }
370
    /* htile.c :671
     *
     * Build the list of tile offset locations to be returned.
     */
    static TileDoc tile_locs()
    {
        int       i, j;
        long prev;
        int       count = 0;
380     double          avg, sd, limit;
        TileDoc         tdp=null;   /* tiledoc return structure */
        Tile            tp[ ]=null; /* pointer to tilearray in TILEDOC */
        long            bounds[ ]=null;
        double          sss [ ]=null;

        //    ('local' dans htile.c)
        ScopedMemory.enter(new Region("region_tile_locs"));
        String tile_locs_captures[ ]={"tile_locs_compute_1","tile_locs_compute_2"};
        ScopedMemory.determineAllocationSite(tile_locs_captures);
390
        TileLocEnv tle=new TileLocEnv();
        tile_locs_compute(tle);
        count =tle.count;
        sss   =tle.sss;
        bounds=tle.bounds;


        avg = 0.0;
        sd = 0.0;
```

```
400         for (i = 0; i < max_para; i++)
                avg += sss[i];
            avg = avg / (double)((count!=0)? count : 1);

            for (i = 0; i < max_para; i++)
                if (sss[i] > 0.0)
                    sd += ((sss[i] − avg) * (sss[i] − avg));
            if ((count − 1) <= 0)
                sd = 0;
            else
410             sd = Math.sqrt(sd / (double)(count − 1));
            limit = avg − (sd / 2.0);

            /* record tiles */
            /*
             * Create the TILEDOC structure to return the tile information
             * in.    We create a maximum sized tilearray and then shrink it
             * after filling it in.
             */

420         try{
                // capturé par main()
                tdp=(TileDoc)ScopedMemory.newInstance("tile_locs_6",TileDoc.class);

                // capturé par main()
                tdp.tilearray = tp =(Tile[ ])ScopedMemory.newAInstance("tile_locs_7",
                                                                         Tile[ ].class,
                                                                         max_para);
                // capturé par main()
                for(i=0;i<tp.length;i++)
430                 tp[i]=(Tile)ScopedMemory.newInstance
                                ("tile_locs_8",Tile.class);
            }
            catch(Throwable t)
            { t.printStackTrace(); System.exit(1); }

            j=0;
            prev = startoftext;
            for (i = 0; i < max_para; i++)
                if ((sss[i] > limit) && (bounds[i] > prev))
440             {
                    tp[j].startoff = (int)prev;
                    tp[j].endoff = (int)bounds[i];
                    j++;
                    prev = bounds[i] + 1;
                }
            tp[j].startoff = (int)prev;
            tp[j].endoff = (int)endoftext;
            j++;

450         tdp.numtiles=j;

            // on fait pas la partie “shrink array” etc...

            // deleteregion local htile.c :803
            ScopedMemory.exit();
            return tdp;
```

```
        }

        // une sous-région pour faire des calculs sur 'used' et 'scores'
460     // dont la durée de vie ne court pas sur toute la méthode tile_locs
        static void tile_locs_compute(TileLocEnv tle)
        {
            ScopedMemory.enter(new Region("region_tile_locs_compute"));

            int i,j;
            Sortf    scores[]=null;
            int      used[]=null;
            long             bounds[];
            double           sss [];
470         int      para, sentgap;
            long     parabyte;
            int count;
            count=tle.count;
            sss   =tle.sss;
            bounds=tle.bounds;

            try{
                bounds=(long[])ScopedMemory.newAInstance
                          ("tile_locs_compute_1",      long[].class,max_para);
480             sss   =(double[])ScopedMemory.newAInstance
                          ("tile_locs_compute_2",double[].class,max_para);
                scores=(Sortf[]) ScopedMemory.newAInstance
                          ("tile_locs_compute_3", Sortf[].class,max_sent);
                used  =(int[])  ScopedMemory.newAInstance
                          ("tile_locs_compute_4",       int[].class,max_sent);
                for(i=0;i<scores.length;i++)
                    scores[i]=(Sortf)ScopedMemory.newInstance
                                  ("tile_locs_compute_5",Sortf.class);
            }
490         catch(Throwable t)
            { t.printStackTrace(); System.exit(1); }

            determine_scores(w2,scores);
            Arrays.sort(scores);

            for (i = 0; i < max_sent; i++) {
                para = 0;
                sentgap = scores[i].sgap;
                parabyte = paralocs[sentgap];
500             for (j = 0; j < 6; j++) {
                    if (sentpara[sentgap+j] > 0) {
                        para = sentpara[sentgap+j];
                        parabyte = paralocs[sentgap+j];
                        break;
                    } else if ((sentgap-j>0)&&(sentpara[sentgap-j] > 0)) {
                        para = sentpara[sentgap-j];
                        parabyte = paralocs[sentgap-j];
                        break;
                    }
510             }
                if (scores[i].val > 0.0) {
                    if ((used[para]!= 1) &&
                        ((parabyte > startoftext) && (parabyte < endoftext))) {
```

```
                    sss[para] = scores[i].val;
                    bounds[para] = parabyte;
                    count++;
                }
                used[para] = 1;
            } else {
520             break;
            }
        }

        ScopedMemory.exit();

        tle.bounds=bounds;
        tle.sss=sss;
        tle.count=count;
    }
530
    static double determine_scores(double w[ ], Sortf scores[ ])
    {
        int i, j, sentgap;
        double          value, score;
        double          max = 0.0;

        for (i = 0; i < max_sent; i++) {
            scores[i].val = 0.0;
            scores[i].sgap = 0;
540     }

        for (sentgap = 1; sentgap < max_sent; sentgap++) {
            value = w[sentgap];
            if (w[sentgap] > max)
                max = w[sentgap];

            score = 0.0;
            j = sentgap − 1;
            while (j > 1) {
550             if (w[j] < value)
                    break;
                value = w[j];
                j−−;
            }
            score = value − w[sentgap];
            j = sentgap + 1;
            value = w[sentgap];
            while (j < max_sent) {
                if (w[j] < value)
560                 break;
                value = w[j];
                j++;
            }
            if ((score > 0.0) && ((value − w[sentgap]) > 0.0)) {
                score += value − w[sentgap];
                scores[sentgap].val = score;
                scores[sentgap].sgap = sentgap;
            }
        }
570     return(max);
```

```
    }

    /*******************************************************
     *
     *                  Ré-allocation des tableaux
     *
     *******************************************************/

    static int[ ] realloc_harray(int[ ] a,int old,int nnew)
580 {
        int b[ ]=null;
        try{ b=(int[ ])ScopedMemory.newAInstance("realloc_harray_1",
                                          int[ ].class,nnew*2);
        }
        catch(Throwable t)
        { t.printStackTrace(); System.exit(1); }
        for(int i=0;i<old;i++)
            b[i]=a[i];

590     return b;
    }

    static void mksentarrays(int old,int nnew)
    {
        // le while de 'tile', tile_locs_compute
        sentpara =realloc_harray(sentpara ,old,nnew);

        // le while de 'tile', tile_locs_compute
        paralocs =realloc_harray(paralocs ,old,nnew);
600 }


    /*******************************************************
     *
     *                      AI
     *
     *******************************************************/

    static HashMap ai_hm;
610
    // static void init_ai() est maintenant inlinée au début de 'tile()'

    static AInfo ai_find(String str,boolean created[ ])
    {
        AInfo p=(AInfo)ai_hm.get(str);
        if(p==null)       {
            try{ p=(AInfo)ScopedMemory.newInstance("ai_find_1",
                                              AInfo.class);
            }
620         catch(Throwable t)
            { t.printStackTrace(); System.exit(1); }
            created[0]=true;

            ai_hm.put(str,p);
        }
        return p;
    }
```

```
    /*****************************************************
630  *
     *                        StopWord
     *
     *****************************************************/
    static HashMap sw_hm=null;

    // static void init_stopword_table() est maintenant inlinée au début de 'tile()'
    // static void stopword_free() est maintenant inlinée a la fin de 'tile()'

    static boolean stopword_find(String str)
640 {
        Object o=sw_hm.get(str);
        return (o!=null);
    }

    static void stopword_add(String str)
    {
        sw_hm.put(str,str);
    }

650 static String common_words[ ]={
        "a","about","above","accordingly","across","after","afterwards",
        "again","against","all","allows","almost","alone","along",

        "...",

        "Whereby","Wherein","Whereupon","Wherever","Whether","Which",
        "While","Whither","Who","Whoever","Whole","Whom","Whose","Why",
        "Will","With","Within","Without","Would","Wouldn't","X","Y",
        "Yes","Yet","You","Your","You're","Yours","Yourself","Yourselves",
660     "Z","Zero"
    };
}
```

B.3 Tokizer.java

L'analyseur lexical du programme en C avait été écrit avec Lex. Comme il était néanmoins assez simple, je l'ai ré-implémenté directement en Java, sans passer par un outil spécialisé.

```
import java.io.*;

class Tokizer
{

    PushbackReader in;
    int token_type;

    int bytes=0;
10  int lastbytes=0;

    boolean beginning_of_line;
```

```
        final int TEXT_TOKEN=1;      /* a word */
        final int NULL_TOKEN=2;      /* characters not part of a word (ignore) */
        final int BLANK_TOKEN=3;     /* a blank line */
        final int INDENT_TOKEN=4;    /* whitespace preceding text */

        Tokizer(String filename)
20      {
            try
            {
                in=new PushbackReader(new FileReader(filename),10);
            }
            catch(Throwable t)
            {
                t.printStackTrace();
                System.exit(1);
            }
30
            beginning_of_line=true;

        }




        String lex()
            throws IOException
40      {
            int i=in.read();

            if (i== -1)
                return null;

            char c=(char)i;
            Character ch=new Character(c);

            if(Character.isLetter(c))
50          {
                beginning_of_line=false;

                String token="";

                while(Character.isLetter(c))
                {
                    token+=ch.toString();


60                  i=in.read();
                    c=(char)i;
                    ch=new Character(c);

                    if( c=='-' || c=='\'' )
                    {
                        int j=in.read();
                        char d=(char)j;
                        if(Character.isLetter(d))
                        {
70                          token+=ch.toString();
```

```
                    c=d;
                    ch=new Character(d);
                }
                else
                {
                    in.unread(d);
                }

            }
80
        }

        in.unread(c);

        bytes+=lastbytes;
        lastbytes=token.length();
        token_type=TEXT_TOKEN;//1

        return token;
90  }
    else if(beginning_of_line && Character.isWhitespace(c) && c!='\n')
    {
        beginning_of_line=false;

        String token="";
        while(Character.isWhitespace(c))
        {
            token+=ch.toString();

100         i=in.read();
            c=(char)i;
            ch=new Character(c);
        }

        in.unread(c);

        bytes+=lastbytes;
        lastbytes=token.length();
        token_type=INDENT_TOKEN; //4
110
        return token;
    }
    else if(ch.toString().equals("\n"))
    {
        if(beginning_of_line)
        {
            // ceci est une ligne blanche

            bytes+=lastbytes;
120         lastbytes=1;
            token_type=BLANK_TOKEN;//3
            return ch.toString();
        }
        else
        { // ceci est une fin de ligne occupée
            beginning_of_line=true;
```

```
                    bytes+=lastbytes;
                    lastbytes=1;
130                 token_type=NULL_TOKEN;//2
                    return ch.toString();
                }
            }

            bytes+=lastbytes;
            lastbytes=1;

            beginning_of_line=false;
            token_type=NULL_TOKEN;//2
140         return ch.toString();

        }

    }
```