Profilage mémoire d'applications OCaml

Çağdaş Bozman

Thèse
pour l'obtention du diplôme de
Docteur de l'École Doctorale de l'École Polytechnique

Spécialité **Informatique**

Profilage mémoire d'applications OCaml

PRÉSENTÉE ET SOUTENUE PAR

ÇAĞDAŞ BOZMAN

LE 16 DÉCEMBRE 2014

Devant le jury composé de

Président	Emmanuel Chailloux	Université Pierre et Marie Curie
Directeurs de thèse	Michel Mauny	ENSTA-ParisTech
	Pierre Chambart	OCamlPro
Rapporteurs	Sylvain Conchon	Laboratoire de Recherche en Informatique
	Manuel Serrano	INRIA Sophia Antipolis
Examinateurs	Damien Doligez	Jane Street
	Alain Frisch	LexiFi
Invité	Fabrice Le Fessant	INRIA Paris-Rocquencourt

Profilage mémoire d'applications OCaml

Résumé

En français

La récupération automatique de la mémoire est une caractéristique commune des langages de programmation. Elle offre certes au programmeur des garanties de fiabilité, mais, en éloignant ce dernier des détails de la gestion de la mémoire, elle rend plus difficile la compréhension et, a fortiori, la maîtrise des allocations de mémoire.

OCaml, langage multi-paradigme statiquement typé développé à l'INRIA. utilise un ramasse-miettes (GC) en charge du recyclage automatique de la mémoire et ses utilisateurs réclament des outils d'analyse du comportement mémoire de leurs programmes.

Dans le cadre de cette thèse, nous avons étudié d'une façon générale le problème de comportement mémoire des applications écrites en OCaml et plus précisément le cas des fuites mémoires, c'est-à-dire des blocs alloués et non recyclés bien qu'ils soient devenus inutiles au calcul.

Cette thèse présente, après un rappel des méthodes connues d'analyse du comportement mémoire des applications, une méthode d'instrumentation de la chaîne de compilation et de la bibliothèque d'exécution des programmes. Cette instrumentation est à la base d'outils que nous avons conçus et mis en oeuvre, et qui permettent de catégoriser, visualiser et localiser les allocations de mémoire.

Les résultats et outils présentés dans cette thèse ont été obtenus dans le cadre d'une collaboration entre l'ENSTA-ParisTech, l'INRIA et la société OCamlPro SAS, et ont été utilisés avec succès dans un contexte industriel.

In english

Automatic memory reclaim is a common feature of programming languages. It offers to the programmers some guarantees about reliability, but, by taking them away from the implementation details of the memory management, it makes it more difficult to understand and control memory allocations.

OCaml is a multi-paradigm statically typed programming language developed at INRIA. It uses a garbage collector (GC) in charge of the automatic reclaim of unused memory and OCaml users need tools for analyzing the memory behavior of their programs.

As part of this thesis, we studied the memory behavior of applications written in OCaml and more specifically the case of memory leaks : still accessible memory blocks (hence not reclaim by the GC) that are no longer needed for the computation.

In this thesis, after describing some known methods for analyzing the memory behavior of applications, we present an instrumentation method of the OCaml toolchain and runtime. This instrumentation then is used by tools that we have designed and implemented, and used to categorize, visualize and locate memory allocations.

Results and tools presented in this thesis were obtained as part of a collaboration between ENSTA ParisTech, INRIA and OCamlPro SAS, and have been successfully used in an industrial context.

Remerciements

Mes premières pensées vont tout naturellement à mes directeurs, Michel et Fabrice, pour avoir su me recadrer, m'aiguiller et surtout avoir pris le temps de m'expliquer les choses patiemment et clairement. Vous êtes les meilleurs.

Je remercie ensuite tous mes collègues d'OCamlPro, Grégoire, Pierre, Benjamin, Mohamed, Thomas, Pierrick, Louis et Michael. Merci pour l'ambiance les gars ! Merci à vous pour avoir su répondre à mes questions, c'est vraiment un réel plaisir de travailler avec des gens comme vous, on ne peut que s'améliorer !

Je remercie aussi tous les stagiaires qui sont passés chez nous et spécialement Ali et Qi, merci pour votre travail sur ocp-memprof, vous avez gagné une licence gratuite du coup :-)

Merci aussi à mes collègues de l'ENSTA, François, Alexandre, Benoit, c'est vraiment cool de travailler avec des chercheurs comme vous !

Merci également à mes collègues de l'IRILL, Vincent, Séverine, Jérôme, Roberto, Gabriel, et tous les autres.

Un grand merci également à mes rapporteurs, Manuel et Sylvain, pour avoir pris le temps et surtout avoir eu le courage de relire après moi. Merci pour vos remarques. Merci également à Alain, Damien et Emmanuel d'avoir accepté de faire partie de mon jury.

Merci à mes collègues de chez Dassault Systèmes, Daniel, Fabien et Etienne, qui m'ont poussé à faire cette thèse quand j'étais en stage de fin d'études.

Merci à mes amis pour m'avoir supporter pendant ces trois ans où j'étais parfois absent et quand j'étais là c'était pour me plaindre que la thèse "c'est trop dur :-(". Merci à vous les amis !

Merci à toi Michael, c'est toi qui m'a le plus supporté, matin, midi et soir, la semaine, le week-end... Depuis la maternelle, le collège, le lycée et la FAC et maintenant au boulot. Arrête de me suivre vieux ! "On descend ?"

Merci Jess, merci de m'avoir écouté, supporté, et merci pour toutes les relectures que tu as du te taper... Mais bon, j'ai lu 3 tomes de 50 shades of grey pour toi, on est quitte ? Merci pour toutes ces séances de visionnage de série, t'es vraiment la meilleure. Merci d'être là...

Merci aussi à ma famille, mes frères, et mes parents, sans qui je n'aurais jamais affronté ce challenge. Vous me supportez depuis 26 ans et merci pour ça ! Sans vous rien de tout ça n'aurait été possible.

Table des matières

Table des figures

"Si l'idée n'est pas a priori absurde, elle est sans espoir."

Albert Einstein

Introduction

Aujourd'hui, lorsqu'on allume son ordinateur, un certain nombre de programmes s'exécutent en même temps. Chacun de ces programmes informatiques utilise une partie de la mémoire vive de l'ordinateur qui les exécute. Il est important d'assurer à la fois la fiabilité des usages que font ces programmes de leur mémoire, en automatisant sa récupération, et de permettre au programmeur de maîtriser son usage.

Ces programmes peuvent être écrits dans différents langages de programmation, chacun gérant sa mémoire de façon manuelle ou automatique. Dans les deux cas, il est important de noter que, par économies de ressources (temps d'exécution, espace mémoire), une application doit allouer de la mémoire lorsque cela lui est nécessaire, et la libérer dès qu'elle est devenue inutile, afin que l'espace correspondant puisse être recyclé.

Pour éviter les gaspillages de ressources, il est important d'essayer de détecter les problèmes mémoires des applications avant leur mise en production, et donc de se doter d'outils qui permettent d'examiner le comportement des programmes en matière de gestion de la mémoire.

Ces outils, dits de *profiling*, peuvent permettre de mieux comprendre le comportement des programmes en donnant des informations sur les blocs alloués, ou en détectant les classes de blocs qui occupent le plus d'espace mémoire. Les informations qu'ils collectent permettent alors l'identification des erreurs de conception ou de programmation, et/ou l'optimisation des performances des applications. Une conséquence classique de ces erreurs de programmation consiste à garder des références sur des blocs mémoire qui sont pourtant devenus inutiles au calcul : des *fuites mémoire*.

Définition 1 (Fuite mémoire)**.** *Dans un environnement où la désallocation et le recyclage de la mémoire sont réalisés automatiquement par un ramasse-miettes, une* fuite mémoire *représente un bloc mémoire qui, à un instant de l'exécution, est vivant, car accessible, du point de vue du ramasse-miettes bien qu'il ne soit plus utile à la suite de l'exécution du programme.*

Dans les langages à gestion manuelle de la mémoire, une fuite mémoire est souvent

due à un oubli de libération explicite (par `free`, en C) d'un bloc alloué (par `malloc`). Dans les langages à gestion automatique de la mémoire, le problème est plus subtil puisqu'il ne peut pas y avoir de libération explicite de la mémoire. Il s'agit alors de comprendre pourquoi tel ou tel bloc mémoire est encore référencé à un point de programme, et comment faire pour que cela ne soit plus le cas si ce bloc est devenu inutile.

Gestion mémoire manuelle

Dans les langages de programmation comme C, la gestion de la mémoire se fait manuellement, avec pour avantage la totale maîtrise de celle-ci. La récupération de la mémoire est faite explicitement par le programmeur. Cette gestion de la mémoire repose donc essentiellement sur le programmeur, et est à l'origine d'erreurs classiques, comme le défaut d'initialisation d'un pointeur, les références multiples à une même zone mémoire, qui posent le problème de s'assurer que l'innocuité de la libération de cette zone, ainsi que les fuites mémoires. Ce dernier cas correspond à la perte du dernier pointeur référençant une zone mémoire, qui ne pourra donc plus être libérée avant la fin de l'exécution. programme.

Pour éviter ce genre de problèmes liés à la gestion de la mémoire, les langages de programmation modernes utilisent des ramasse-miettes qui sont en charge de libérer et recycler la mémoire devenue inaccessible et donc inutile. La récupération automatique de la mémoire offre certes au programmeur des garanties de fiabilité, mais, en éloignant ce dernier des détails de la gestion de la mémoire, elle rend plus difficile la compréhension et, *a fortiori*, la maîtrise des allocations de mémoire.

OCaml, langage de programmation multi-paradigme

OCaml, langage multi-paradigme statiquement typé, utilise un ramasse-miettes en charge du recyclage automatique de la mémoire. Du fait de cette gestion automatique de la mémoire et de l'absence de primitive d'allocation [1], le programmeur n'aura pas besoin de (et ne pourra pas) libérer explicitement de la mémoire.

Le typage statique fort d'OCaml permet à sa bibliothèque d'exécution de ne pas avoir besoin de beaucoup d'informations de type à l'exécution. Par conséquent, la représentation mémoire des valeurs OCaml est très compacte, ce qui résulte en de meilleures performances.

1. Les allocations sont toujours explicites, mais la syntaxe du langage est telle que le programmeur peut ne pas en avoir conscience.

Cependant, lorsqu'il faut profiler une application OCaml, l'absence quasi-complète d'information de type à l'exécution, devient un problème pour retrouver des informations sur les valeurs utilisant la mémoire dans le tas.

Le but de cette thèse est de proposer des outils de profiling pour mieux comprendre le comportement mémoire des applications OCaml. Il est important de noter est que le profiling peut avoir un certain coût. En soi, cela ne pose pas de problème d'avoir un ralentissement pendant un débogage. Cependant, il est très important de ne pas changer le comportement d'une application. Pour un langage à gestion mémoire automatique, il est important que le ramasse-miettes se comporte quasiment de la même manière avant et après profiling pour que les résultats obtenus lors du profiling soient exploitables. En effet, lorsque le profiling occasionne un surcoût mémoire, le ramasse-miettes se lance probablement plus souvent et avec un temps d'exécution différent. L'application ne se comporte donc pas de la même manière dans un tel cas. Nous nous proposons dans cette thèse de profiler des applications OCaml, sans changer leur comportement mémoire.

Motivations et contributions de la thèse

Dans le cadre de cette thèse, nous avons étudié d'une façon générale le problème de comportement mémoire des applications écrites en OCaml et plus précisément le cas des fuites mémoires, c'est-à-dire des blocs alloués et non recyclés bien qu'ils soient devenus inutiles au calcul.

Cette thèse présente, après un rappel des méthodes connues d'analyse du comportement mémoire des applications, une méthode d'instrumentation de la chaîne de compilation et de la bibliothèque d'exécution des programmes. Cette instrumentation est à la base d'outils que nous avons conçus et mis en œuvre, et qui permettent de catégoriser, visualiser et localiser les allocations de mémoire.

Les résultats et outils présentés dans cette thèse ont été obtenus dans le cadre d'une collaboration entre l'ENSTA-ParisTech, l'INRIA et la société OCamlPro SAS, et ont été utilisés avec succès dans un contexte industriel.

Plan de la thèse

Le premier chapitre de cette thèse revient en détail sur les différents types de ramasse-miettes existants, ainsi que les différentes stratégies d'allocation pour les langages à gestion automatique de la mémoire. Le second chapitre donne un aperçu de la gestion mémoire de différents langages de programmation avec des ramasse-miettes. Le troisième

chapitre décrit la gestion mémoire par régions, avec laquelle nous avons tenté d'évaluer la durée de vie des valeurs en mémoire. Le quatrième chapitre fait l'état de l'art des outils de profiling existant pour différents langages de programmation. Le cinquième chapitre décrit l'instrumentation de la chaîne compilation que nous avons réalisée, et l'extraction des données obtenues avec l'introduction des identifiants de location. Le sixième chapitre décrit comment nous traitons ces données introduit dans le cinquième chapitre. Le septième chapitre décrit les différents affichages que nous avons choisis pour mettre en valeur ces informations, ainsi que quelques cas d'usage avec des résultats très encourageants.

“L’amour c’est un dessert : après que le gâteau est mangé, il reste toujours des miettes dans l’assiette.“

Patrick Cauvin

1

Introduction aux ramasse-miettes

Un ramasse-miettes, *garbage collector* en anglais, est un sous-système informatique de gestion automatique de la mémoire. Il permet le recyclage de la partie devenue inaccessible de la mémoire allouée par un programme.

Le premier ramasse-miettes a été inventé et implanté par John MacCarthy, pour le langage Lisp [57]. McCarthy parle alors de *cycle de réclamation* et décrit l’algorithme de ramasse-miettes à balayage (qui ne portait pas ce nom à l’époque).

Le principe de base du ramasse-miettes est assez simple. Il permet de déterminer quels objets en mémoire ne sont plus référencés par le programme exécuté, et dans ce cas, permet de récupérer cette zone mémoire devenue inutile.

Il est difficile de prévoir quand un objet ne sera plus utilisé par un programme, mais il est possible de le savoir durant l’exécution de celui-ci. Lorsque qu’il n’y a plus aucune référence vers un objet, ce dernier est inatteignable, et donc inutile (mort). Il peut alors être collecté par le ramasse-miettes.

Cependant, l’atteignabilité d’un objet n’implique nullement son utilité : un objet accessible peut être devenu inutile.

Dans les langages sans gestion mémoire automatique (c’est-à-dire sans ramasse-miettes) comme C et C++, les problèmes mémoires les plus fréquents sont le déréfé-

rencement des adresses mémoire (pointeurs) déjà libérées (pointeurs « fantômes » ou *dangling pointer*, en anglais), la perte d'un pointeur vers un objet en mémoire que le programme a oublié de libérer, et le fait de garder des pointeurs vers des objets en mémoire qui ne seront plus utilisés dans la suite de l'exécution du programme (fuite mémoire). Un ramasse-miettes règle les deux premiers problèmes de façon automatique. Cependant, les fuites mémoires restent un problème difficile, quelque soit le mode de gestion de la mémoire.

La compréhension du comportement mémoire des applications nécessite au préalable de maîtriser les stratégies d'allocation de mémoire ainsi que les techniques de recyclage automatique par ramasse-miettes. Ce chapitre est une introduction à la gestion automatique de la mémoire présentant les différents ramasse-miettes et la terminologie associée.

1.1 Ramasse-miettes à comptage de références

Souvent utilisés pour leur simplicité [53, 54], ils présentent comme principal inconvénient de ne pas réclamer les cycles et d'être moyennement performants.

1.1.1 Fonctionnement du ramasse-miettes à comptage de références

Avec un ramasse-miettes à comptage de références, chaque bloc mémoire alloué connaît le nombre de fois qu'il est référencé, au moyen d'un compteur porté par le bloc lui-même.

Ce compteur est incrémenté lorsque l'on crée une nouvelle référence sur le bloc, et on le décrémente lorsqu'on supprime l'une de ces références. Lorsque le compteur arrive à zéro, on peut le libérer puisqu'il n'est plus accessible. Avant de libérer complètement l'objet de la mémoire, il faut mettre à jour les compteurs de tous les objets sur lesquels pointait celui-ci, ce qui peut entraîner une cascade de libérations.

1.1.2 Les avantages et inconvénients du comptage de références

Avantage 1 Avec ce système de comptage, on peut très rapidement libérer les blocs dont le compteur est devenu nul : il n'est pas nécessaire d'attendre le prochain cycle de ramassage pour la libération.

Avantage 2 Simple à implémenter, il est souvent utilisé dans les contextes distribués (SSP Chains [77]).

Inconvénient 1 Le problème avec ce genre de ramasse-miettes est le ralentissement de la gestion mémoire, dû à la manipulation de ces compteurs. En effet, pour les programmes manipulant un nombre important de pointeurs, le temps de mise à jour des compteurs peut devenir non-négligeable. Par exemple, si nous prenons le code suivant :

```
1 let x = ref object1 in
2 x  := object2
```

Ici, le pointeur vers `object1` est remplacé par celui de `object2`. Il faut mettre à jour le compteur du premier objet qui doit être lu puis décrémenté. Il faut ensuite tester si le compteur est nul, et si tel est le cas, il faut alors désallouer l'objet. Puis, il faut faire de même avec le nouvel objet `object2`, c'est-à-dire, lire son compteur, l'incrémenter et le mettre à jour.

Ces mécanismes de gestion de compteurs impliquent une augmentation de la taille du code à produire, et du temps d'exécution.

Inconvénient 2 Le comptage de références ne gère pas les valeurs circulaires, ce qui crée par conséquent des fuites mémoire.

Prenons un exemple avec la figure 1.1. Dans la liste `maListe`, on note que le bloc contenant la valeur `toto` a un compteur égal à deux, puis les autres blocs ont des compteurs de un. Or, si le pointeur en rouge est supprimé, le ramasse-miettes ne pourra pas libérer les blocs. En effet, les pointeurs étant cycliques, il y aura toujours une référence sur chacun des objets de la liste (les compteurs des blocs seront toujours égaux à un).

Il est alors nécessaire de compléter l'algorithme de comptage par références en parcourant le graphe mémoire pour libérer ce genre de bloc isolé.

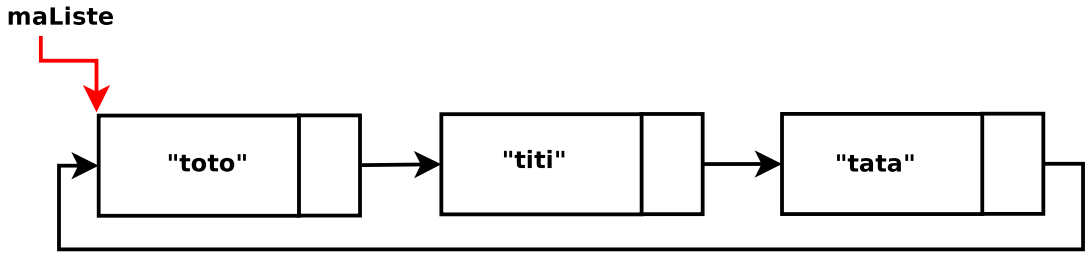

Figure 1.1 – Exemple d'une liste circulaire.

1.2 Ramasse-miettes traversant / traçant

Un ramasse-miettes traversant, ou traçant, est une autre forme de gestion mémoire automatique consistant à considérer la mémoire comme un graphe. Il consiste à déterminer les objets pouvant être désalloués en parcourant les pointeurs depuis des *racines* et en marquant tous les objets atteignables depuis celles-ci, considérant alors tous les autres comme morts (car non atteignables depuis les racines).

Une racine est un pointeur utilisé directement par le programme (mutateur). Les racines habituelles sont dans la pile, les variables statiques, les registres, etc.

Ces types de ramasse-miettes sont les plus communs dans les langages de programmation modernes et il existe plusieurs implantations avec différents algorithmes que nous allons détailler dans la suite.

1.2.1 L'atteignabilité des objets

Le *ramasse-miettes traversant* parcourant le graphe mémoire, il faut comprendre ce que signifie pour un objet d'être vivant ou mort.

Un objet est accessible, ou vivant, s'il est référencé par au moins une variable dans le programme, soit directement, soit à travers des références par d'autres objets vivants/accessibles. Les racines (dont nous avons vu la définition précédemment) représentent un ensemble d'objets considérés comme vivants. La figure 1.2 fournit un exemple d'objets atteignables et non atteignables depuis un ensemble de racines.

1.2.2 Le ramasse-miettes à balayage

Les ramasse-miettes à balayage (*mark and sweep* en anglais) peuvent être modélisés à l'aide d'un coloriage des objets [67, 83, 94] qui indique l'état d'avancement d'un cycle du ramasse-miettes.

Dans la version naïve de l'algorithme, on utilise deux couleurs, les objets en mémoire étant initialement colorés en blanc. En parcourant le graphe depuis les racines, le ramasse-miettes noircit tous les objets atteignables depuis celles-ci. En commençant par les racines, il itère tant qu'il existe des objets noirs avec des fils blancs. Quand aucun objet noir ne pointe plus vers un objet blanc, la phase de marquage est terminée. Cela se fait par un parcours récursif en profondeur du graphe mémoire. Le fait de noircir un objet en noir signifie donc que celui-ci est atteignable depuis une racine. Pour ne pas chercher tous les objets noirs pointant vers des objets blancs, il suffit, après avoir marqué

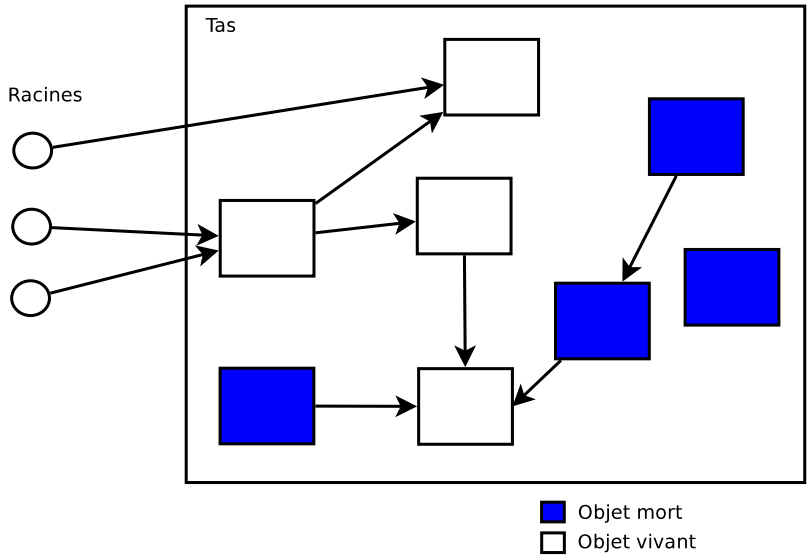

Figure 1.2 – Les objets atteignables et non atteignables depuis un ensemble de racines.

un objet en noir, de parcourir ses fils blancs. L'inconvénient de cette version est que le parcours récursif utilise de la mémoire pendant le parcours des fils, pour se souvenir de l'objet que nous parcourons. Or, lorsque l'on déclenche le ramasse-miettes, c'est qu'il n'y a plus de mémoire.

Dans la version à trois couleurs (blanc, gris et noir), le gris servira au ramasse-miettes pour les noeuds en cours de parcours. À chaque itération,pendant la phase de marquage, le ramasse-miettes va colorier les blocs blancs en gris, s'ils sont toujours vivants. Lorsque tous les fils immédiats d'un nœud gris auront été visités, celui-ci sera alors colorié en noir. Lorsqu'il n'y a plus d'objets gris, le marquage est fini. L'utilisation de la couleur grise permet de rendre le marquage incrémental, c'est-à-dire de le découper en phases plus courtes.

En OCaml, il y a une quatrième couleur, le bleu, servant à identifier un bloc dans la freelist [1].

Pour résumer :

- *Bleu* signifie que le bloc est dans la freelist.
- *Blanc* signifie que le bloc n'est pas encore visité par la phase de *marquage*.
- *Gris* signifie que le bloc a été visité par la phase de marquage, mais certains de ses fils ne l'ont pas encore été.
- *Noir* signifie que le bloc, ainsi que tous ses fils immédiats ont été visités par la phase de marquage.

1. La freelist contient tous les blocs libres prêts à être utilisés lors des futures allocations

Durant la phase de balayage, le ramasse-miettes va colorier les blocs noirs en blanc pour les préparer à la prochaine itération, et les blocs blancs en bleus (les blocs ne sont plus utiles, donc on les ajoute à la freelist).

Avantage 1 Un avantage indéniable des ramasse-miettes traversant, est que la libération de l'espace se fait d'un coup, en récupérant les objets blancs. Cela devient très utile lorsqu'il y a beaucoup d'objets alloués. Le ramasse-miettes ne déplaçant pas les objets, il n'y a pas besoin d'opérations en plus pour mettre à jour les pointeurs.

Avantage 2 Le ramasse-miettes est complet et ne demande que très d'espace supplémentaire par objet (juste les couleurs)

Inconvénients Plusieurs inconvénients subsistent tout de même. Un premier inconvénient est que la mémoire est parcourue en totalité : une fois pour les blocs vivants, pour le marquage et une fois en totalité pour blanchir les blocs vivants et récupérer les morts. De plus, avec ce type de ramasse-miettes, la mémoire peut être rapidement fragmentée, car aucun objet n'est déplacé. En effet, les durées de vie des objets étant différentes, on se retrouve très vite avec des zones vivantes séparées par des zones mémoires libres, de taille trop faibles pour répondre à la demande. Dès lors, il faut un système de compaction pour réunir ces petits espaces libres. Or la compaction a un coût qui peut être parfois non négligeable.

Il faut alors mettre à jour tous les pointeurs de ces objets vers leur nouvelle destination, ce qui peut être plus ou moins complexe avec un ramasse-miettes concurrent par exemple.

1.2.3 Le ramasse-miettes copiant

1.2.3.1 Fonctionnement de base du ramasse-miettes copiant

L'idée principale du ramasse-miettes copiant [11, 41] (*stop and copy* en anglais) est d'utiliser une mémoire secondaire dans le but de copier et compacter la mémoire que l'on va conserver. Le tas est alors divisé en deux parties : la partie utilisée appelée *from-space* et la partie qui va contenir les objets copiés, appelée *to-space*.

L'algorithme est assez simple : les parties intéressantes (objets vivants) de l'espace *from-space* sont copiées dans la zone *to-space*. Les nouvelles adresses de ces objets copiés sont stockées dans leur premier champ dans le but de mettre à jour les objets pointant vers ces objets. Cette étape de réécriture peut créer de nouvelles racines. Tant qu'il y

a alors des racines non explorées, la phase de copie continue. Une fois tous les objets vivants copiés dans la mémoire *to-space*, on échange les statuts des deux zones mémoires : la zone *to-space* devient la *from-space*, et vice-versa, et l'exécution reprend.

Cette technique éliminant la fragmentation lors de la copie, il n'y a pas de freelist à maintenir ou à mettre à jour. Les blocs sont copiés dans la zone *to-space* consécutivement et les nouveaux blocs sont alloués consécutivement dans le *from-space*.

Sur la figure 1.3, on peut observer les déplacements effectués par le ramasse-miettes copiant sur trois valeurs. Dans un premier temps, chaque valeur est copiée dans la mémoire *to-space* avec un nouveau pointeur de l'ancienne valeur vers sa nouvelle destination. À la fin, une fois que toutes les valeurs sont copiées dans la zone *to-space*, l'ensemble des racines pointent vers les nouvelles valeurs et on échange les statuts des deux zones.

1.2.3.2 Avantages et inconvénients

Avantages Un avantage indéniable du ramasse-miettes copiant est sa vitesse et la non-fragmentation de la mémoire. Le temps du ramasse-miettes est proportionnel à la taille totale des objets vivants et la copie remet les objets vivants directement dans l'espace *to-space* les uns à la suite des autres, ce qui laisse une zone libre d'allocation des blocs.

Inconvénients L'inconvénient majeur du ramasse-miettes copiant est l'espace mémoire total qu'il nécessite : le double de la mémoire maximale disponible pour le programme. De plus, le ramasse-miettes déplaçant les objets, il est difficile de rendre ce ramasse-miettes parallèle, concurrent ou incrémental.

1.2.4 Ramasse-Miettes à générations

Le ramasse-miettes à générations [11, 38] peut être combiné aux deux types de ramasse-miettes traversants vus précédemment. Les données d'un programme n'ayant pas toutes la même durée de vie, certaines peuvent être éliminées plus rapidement que d'autres, certaines pouvant même être éliminées très peu de temps après leur création. D'autres, au contraire, vont survivre plus longtemps, voire toute la durée de vie du programme (par exemple les globales). L'idée principale de ce type de ramasse-miettes est alors de traiter différemment ces deux types de données.

On divise la mémoire en deux parties : une contenant les objets fraîchement alloués, que l'on va appeler les objets *jeunes*, et une autre qui va contenir les objets dont la durée de vie a déjà été plus longue, que l'on va appeler les objets *vieux*. En OCaml, la jeune

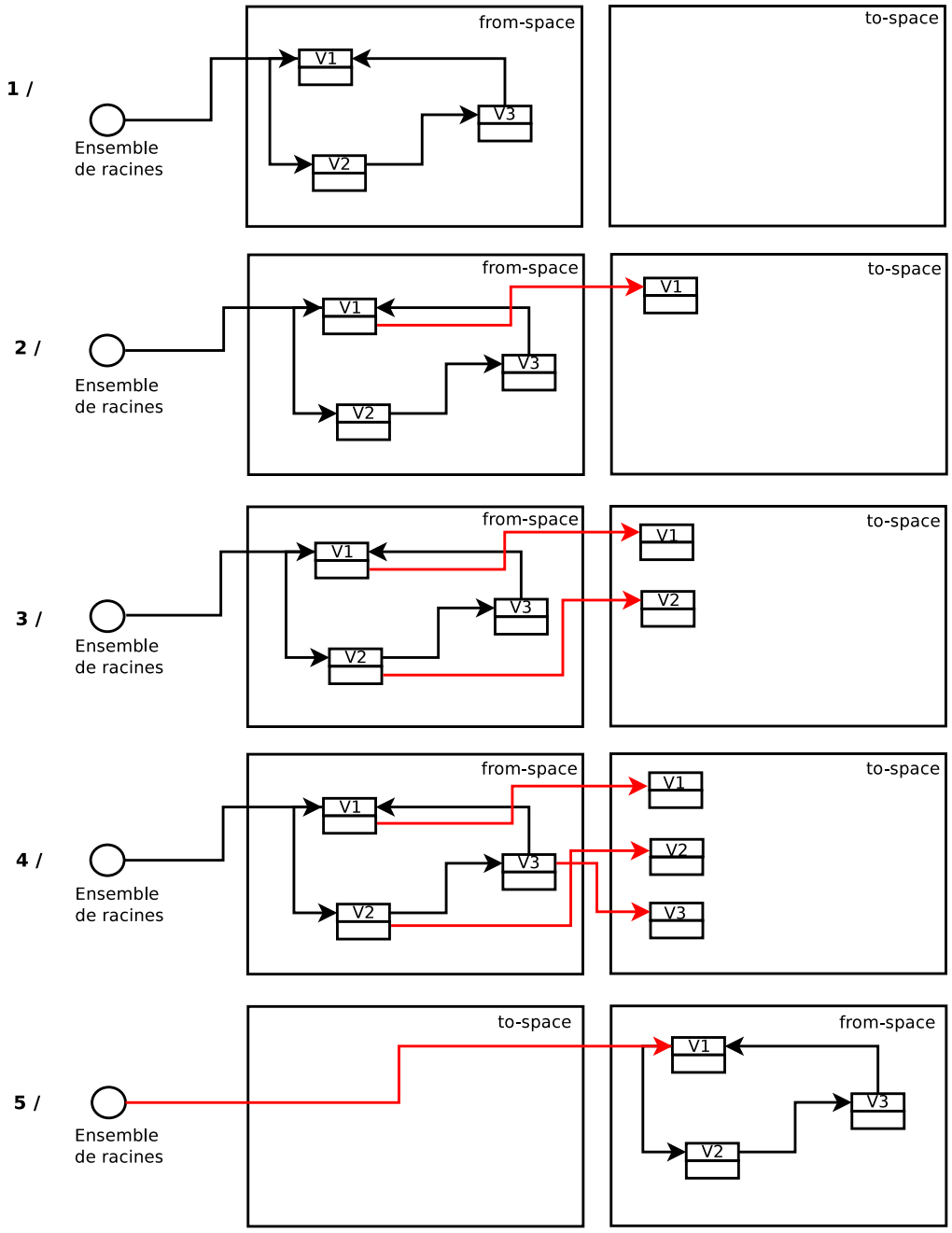

Figure 1.3 – Algorithme *Stop&Copy*

génération va s'appeler le tas mineur et la vieille génération le tas majeur.

Dès qu'un objet est alloué, il passe d'abord par le tas mineur. Le ramasse-miettes passe un cycle sur ces objets jeunes et élimine les objets morts. Si un objet dans cette

génération survit, alors il est promu dans la génération vieille. Un autre cycle de ramasse-miettes examine le tas majeur et élimine à son tour ceux qui sont morts.

L'idée est basée sur l'hypothèse suivante [87] : "la plupart des objets meurent jeunes". Si cette hypothèse se révèle vraie, alors les efforts à fournir doivent se focaliser sur la génération jeune. Les ramasse-miettes à générations exploitent alors cette hypothèse sur les objets jeunes et vont alors souvent ramasser les objets de cette génération, les données de courte durée n'atteignant, pour la plupart, pas la génération supérieure.

De ce fait, des algorithmes différents vont être utilisés dans chacune des générations. En raison de la taille plus petite de la jeune génération, il sera moins gênant d'utiliser un algorithme interrompant l'exécution du programme [59], contrairement au tas majeur, de taille plus importante, pour lequel on préfèrera un algorithme incrémental (voir section suivante).

Les racines du graphe mémoire inclus dans le tas mineur sont les racines classiques, auxquelles on doit ajouter les éventuels pointeurs en provenance des générations plus anciennes, que l'on appelle des pointeurs inverses.

Avantages L'avantage principal de ce type de ramasse-miettes est lorsqu'il y a beaucoup d'allocations d'objets à faible durée de vie. Dans ce type de cas, où le taux de mortalité des objets jeunes est très élevé, une grande partie de la mémoire est récupérée très rapidement. Le travail nécessaire pour chaque bloc décroît avec son âge, et donc croît avec l'ancienneté de sa génération.

De plus, du fait de la petite taille du tas mineur, il est possible d'y effectuer des ramassages très fréquents, tout en impactant faiblement le temps d'exécution.

Inconvénients Un inconvénient de cet algorithme est que la taille du tas mineur doit être choisie avec précaution : une taille trop petite implique des ramassages trop fréquents, promouvant beaucoup d'objets jeunes qui n'auraient peut-être pas survécu si la taille du tas mineur avait été plus grande. Un tas mineur trop petit aura tendance à encombrer le tas majeur avec des objets de courte durée alors qu'il est censé regrouper des objets dont la durée de vie est longue.

De plus, du fait des pointeurs inverses, il faut mettre à jour la liste de ces pointeurs à chaque ramassage. Le surcoût peut être relativement important dans un langage avec affectations, puisque ces dernières sont en mesure d'écrire dans des structures anciennes des pointeurs vers des objets jeunes.

Enfin, l'algorithme du ramasse-miettes à générations est relativement complexe à implanter.

1.2.5 La compaction

Avec les ramasse-miettes traçants comme le ramasse-miettes à balayage, la mémoire est souvent amenée à être fragmentée. Pour pallier ce problème de fragmentation, le ramasse-miettes lance une phase de compactage. Le but de la compaction est de déplacer les objets de manière à ce que la mémoire libre soit contiguë.

Un algorithme à base de tableau est décrit par Haddon et Waite en 1967 [45]. L'algorithme conserve l'ordre relatif des objets vivants en mémoire et a un surcoût constant.

Chaque objet vivant rencontré est déplacé vers la première adresse libre du début du tas. En même temps, l'information de relocation est stockée dans une table annexe. Pour chaque objet vivant, on enregistre dans cette table, le pointeur initial avant la compaction et la différence entre cette adresse et la nouvelle (après la compaction). Cette table est stockée dans un segment mémoire étant dans la zone de compaction, mais dans une zone marquée comme inutilisée. Pour s'assurer du bon déroulement de la compaction, la taille minimale de l'objet dans le tas doit être supérieure ou égale à la taille de cette table.

Au fur et à mesure que la compaction avance, les objets sont copiés dans le début du tas. Si un objet doit être copié dans l'emplacement où est stockée cette table, on la relocalise dans un autre segment mémoire.

Enfin, une fois tous les objets relocalisés, on utilise cette table pour mettre à jour les pointeurs dans les champs appropriés des objets.

Sur la figure 1.4, on voit sur un exemple simple, le déroulement de la compaction étape par étape.

Il existe des algorithmes de compaction en espace constant, comme celui d'OCaml, mais ils sont beaucoup plus complexes à mettre en œuvre.

1.3 Ramasse-miettes Concurrent / Parallèle / Incrémental

Dans l'idéal, il est préférable de ne pas arrêter l'exécution du programme durant un cycle de ramasse-miettes. Pour cela, il existe des ramasse-miettes concurrents, parallèles et incrémentaux.

Le ramasse-miettes *concurrent* [37, 38] va s'exécuter en même temps que le programme, sur un processeur différent de celui ou ceux sur lesquels s'exécute le programme. Cela permet de mettre à profit la puissance de calcul des machines multi-processeurs modernes. De la même manière, les ramasse-miettes *parallèles* s'exécutent en parallèle sur différents processeurs. Le programme est alors arrêté le temps du cycle du ramasse-miettes. Combiné à un ramassage concurrent, il est possible d'éviter l'arrêt du programme pour exécuter parallèlement et simultanément le programme et le

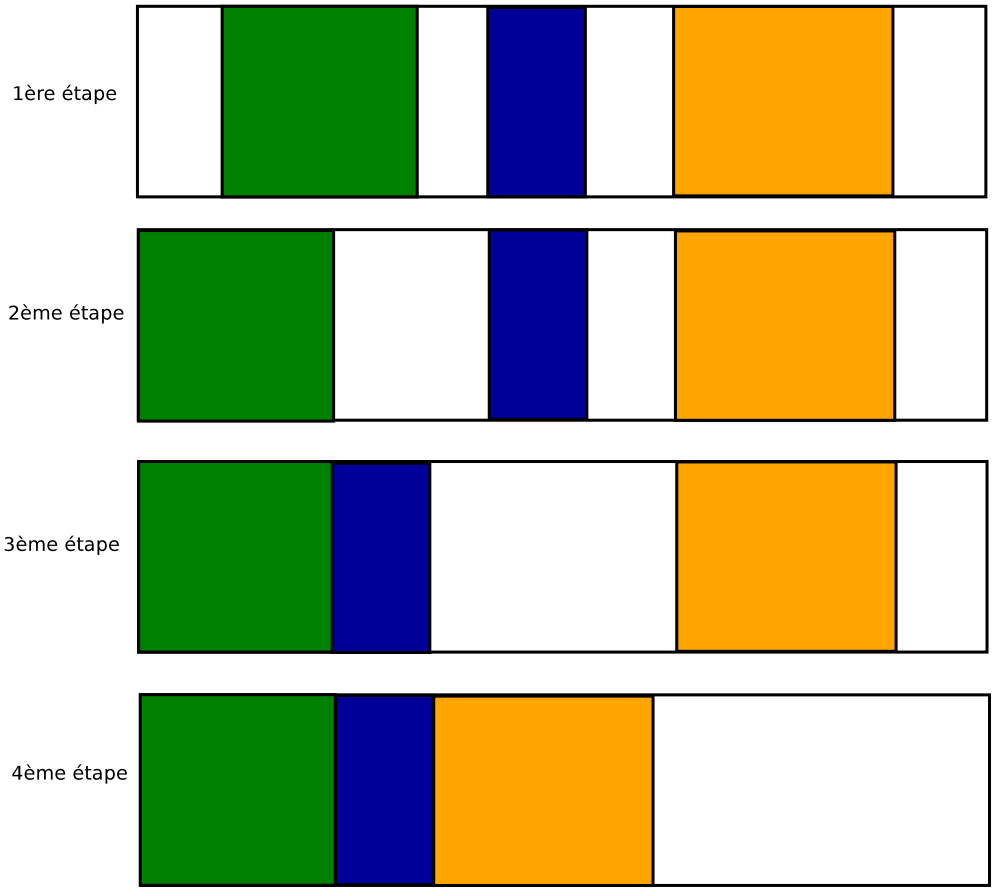

Figure 1.4 – Algorithme de compaction classique. On déplace d'abord le bloc en vert vers le début du segment mémoire. Puis on continue avec le bloc en bleu et enfin avec le bloc en orange. Les nouvelles adresses sont stockées dans une table à part qui permet de mettre à jour les adresses à la fin de la compaction.

ramasse-miettes [36].

Le ramasse-miettes incrémental est un compromis entre les deux ramasse-miettes concurrents et parallèles. Au lieu d'exécuter le programme et le ramasse-miettes en même temps (comme pour le ramasse-miettes parallèle), on alterne les deux sur un seul processeur, ce qui permet de faire des pauses plus courtes, moins susceptibles de gêner l'utilisation si le programme est interactif ou en réseau.

Le programme est stoppé, pour effectuer une «tranche» de ramassage, incrémentalement. Combiné à un ramasse-miettes à générations, par exemple, on procède à une partie du ramassage sur le tas majeur à chaque fois qu'un cycle de ramassage est exécuté sur le tas mineur.

Avantages L'avantage indéniable des ces types de ramasse-miettes est la diminution du temps de latence, voire même sa disparition dans certains cas (ramasse-miettes parallèles et concurrents [36]).

Inconvénients Un des inconvénients, qui est plutôt une difficulté, est la mise en œuvre délicate de ce genre de ramasse-miettes. Dans certains cas, il peut aussi ralentir des programmes séquentiels.

1.4 Ramasse-miettes Conservatif

Comme nous l'avons vu dans la définition, un ramasse-miettes est une forme de gestion mémoire automatique permettant la suppression de la désallocation explicite dans le langage de programmation. Néanmoins, on peut combiner cette gestion automatique avec des langages où la gestion mémoire se fait manuellement, comme en *C*. Cela permet de travailler sur un compilateur donné sans faire de modification sur celui-ci.

Contrairement aux ramasses-miettes que nous avons vus jusque là, le ramasse-miettes conservatif [33, 92] ne sait pas toujours faire la différence entre une valeur scalaire (un entier, par exemple) et un pointeur : il considèrera donc toute adresse plausible (c'est-à-dire pouvant être celle d'un objet alloué dans le tas) comme un pointeur, même si ce n'est pas le cas.

Ce type de ramasse-miettes est adapté aux langages qui n'ont pas été conçus pour être dotés de gestion automatique de mémoire, et qui n'ont donc pas prévu de distinguer les pointeurs des valeurs scalaires dans la représentation de leurs objets en mémoire.

Avantages Ce type de ramasse-miettes étant indépendant de l'implémentation la bibliothèque d'exécution du langage, il peut être compilé par et pour un compilateur quelconque, sans avoir à modifier ce dernier. Il s'agit donc d'un type de ramasse-miettes universel et indépendant.

Inconvénients Le ramasse-miettes conservatif, en considérant à tort certains scalaires comme des pointeurs valides, il est à l'origine de fuites mémoire. De plus, ce type de ramasse-miettes ne peut naturellement pas déplacer les objets. Sans modification du mutateur, il est très difficile d'en faire une version incrémentale, parallèle ou concurrente.

1.5 Stratégie d'allocation et gestion de la freelist

Avec les langages à gestion mémoire automatique, la désallocation se faisant de manière automatique, il faut traiter avec une attention particulière la fragmentation du tas. En effet, la mémoire pouvant être libérée dans un ordre quelconque, l'espace libre peut potentiellement être réparti partout dans le tas. La compaction est une opération qui élimine cette fragmentation, mais avec un coût dans l'exécution du programme. Il est alors important de réduire le nombre de compaction pour ne pas ralentir, trop souvent, une application. C'est pour cela que l'on s'intéresse aux stratégies de d'allocation permettant d'éviter des compactages systématiques. Le choix d'une zone libre à allouer doit s'effectuer rapidement, minimiser la fragmentation et garantir une utilisation globale optimisée.

De plus, selon l'algorithme de ramasse-miettes utilisé, les allocations seront faites de manières différentes. Pour les ramasse-miettes à copie, par exemple, les fonctions d'allocations vont être très efficaces. En effet, la mémoire libre étant une zone contiguë, il suffira de prendre une zone mémoire de la bonne taille, puis copier la donnée.

Par contre, pour les ramasse-miettes à balayage, la mémoire pourrait être fragmentée. En effet, à chaque libération d'une zone, il faudra se rappeler son emplacement et sa taille. On maintiendra ces blocs libres dans une *freelist*. Pour allouer, il faudra alors parcourir cette liste et trouver un bloc dont la taille est supérieure ou égale à la taille demandée.

Nous allons voir dans cette section différentes stratégies d'allocation avec leurs avantages et inconvénients.

1.5.1 Stratégie first-fit

Cette stratégie d'allocation commence par parcourir la freelist depuis le début, et y choisit le premier bloc libre qui correspond à la taille demandée, et la découpe s'il le faut (si le bloc libre est trop gros par exemple).

Cette stratégie permet de limiter la fragmentation du tas, mais avec un temps d'allocation plus lent que la stratégie *next-fit* (section 1.5.2).

Chaque allocation scanne la freelist depuis le début jusqu'à trouver un bloc de taille suffisante pour allouer le bloc au lieu de partir du dernier bloc alloué. Dans ce cas, le ramasse-miettes est parfois obligé de lancer une *compaction* (section 1.2.5), alors même qu'il pourrait y avoir des blocs vides, mais de taille insuffisamment grande pour stocker le bloc.

1.5.2 Stratégie next-fit

Cette stratégie d'allocation est équivalente à la stratégie first-fit, à la différence qu'elle ne re-parcourt pas la freelist depuis le début, mais reprend le parcours depuis le dernier point d'allocation dans cette liste.

Dans cette stratégie, on garde un pointeur vers le bloc le plus récemment utilisé de la freelist. Lors d'une allocation, il suffit alors de parcourir la liste depuis ce pointeur jusqu'à trouver un bloc qui satisfait la demande (bloc de bonne taille). S'il ne trouve pas de bloc de bonne taille, il recommence son parcours depuis le début de cette liste, jusqu'à revenir à son point initial.

Cette stratégie est celle adoptée par défaut par OCaml. Celle-ci étant dans la plupart des cas plus efficace et plus rapide, mais pouvant donner lieu à des fragmentations du tas.

1.5.3 Stratégie best-fit

Cette stratégie d'allocation choisit le plus petit bloc correspondant à la taille demandée dans la freelist.

Sur la figure 1.5, on peut observer l'allocation d'un bloc de taille 5 avec cette stratégie d'allocation.

1.5.4 Stratégie worst-fit

Cette stratégie d'allocation choisit le plus grand bloc correspondant dans la freelist, pour éviter une trop grande fragmentation du tas en plusieurs petits blocs.

Sur la figure 1.5 on peut voir un exemple sur une zone de mémoire donnée où l'on tente d'allouer un bloc de taille cinq. Les différentes stratégies d'allocation y sont représentées.

1.5.5 Stratégie de malloc (*binning*)

L'idée de cette stratégie d'allocation est de séparer les freelist par taille de chunk. Les chunks disponibles sont alors placés dans des bacs/régions (*bins* en anglais). L'idée générale est qu'on garde une liste séparée pour chacune de ces régions de même taille. La recherche d'une région est ainsi plus rapide.

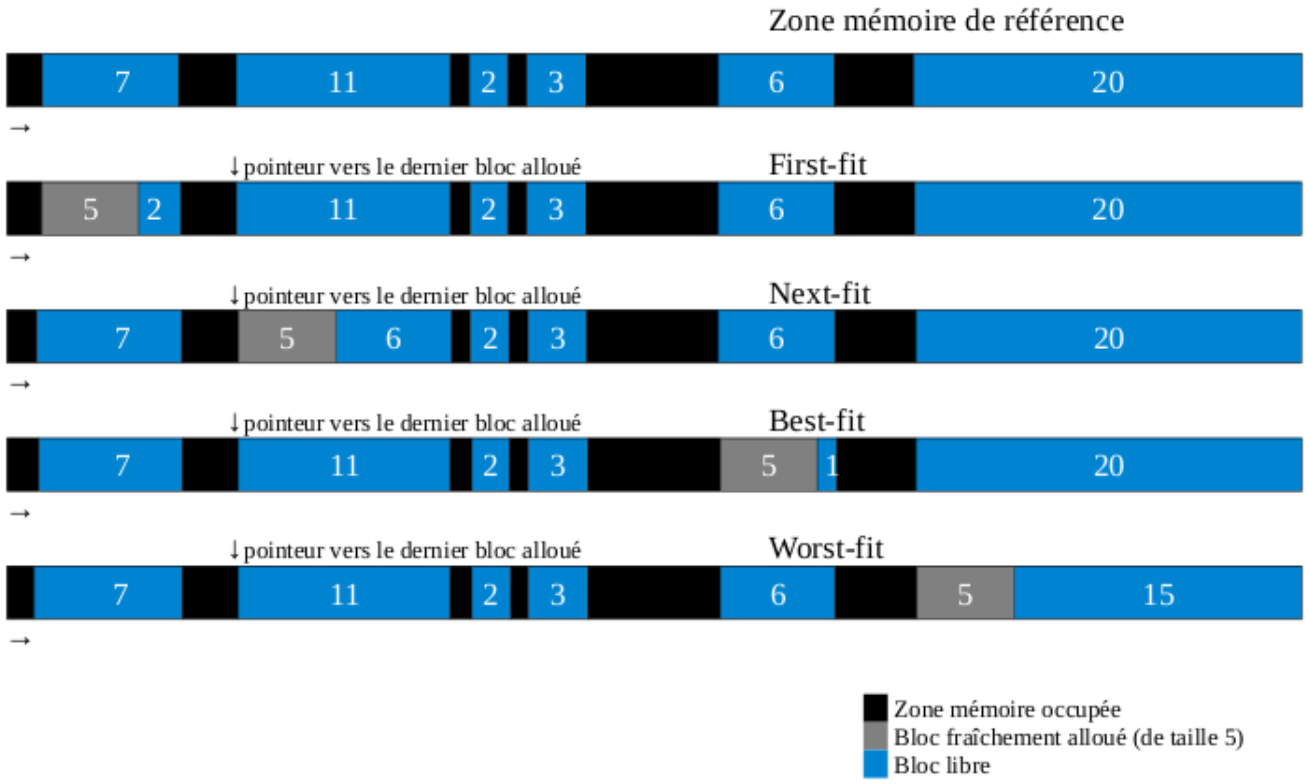

Figure 1.5 – Les différentes stratégies d'allocation

Cette stratégie est utile si le programme alloue et libère beaucoup d'objets de même taille.

1.5.5.1 Stratégie exact-fit

Cette variante partitionne chaque chunk en régions dont chacune correspond à une taille de bloc. L'avantage est qu'il n'y a pas de recherche à faire pour trouver la bonne taille, et l'inconvénient est qu'on se retrouve avec beaucoup de régions.

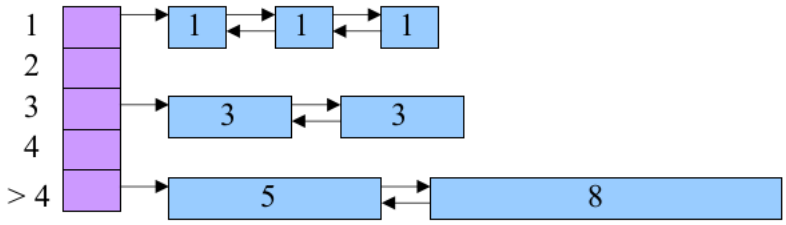

Figure 1.6 – Stratégie d'allocation par malloc (*binning*) : exact-fit

1.5.5.2 Stratégie par intervalles

Cette variante partitionne chaque chunk en régions correspondant à des intervalles de tailles de blocs. L'avantage est qu'il y a moins de régions contrairement à la stratégie *exact-fit* et l'inconvénient est que la recherche peut être un peu plus coûteuse,

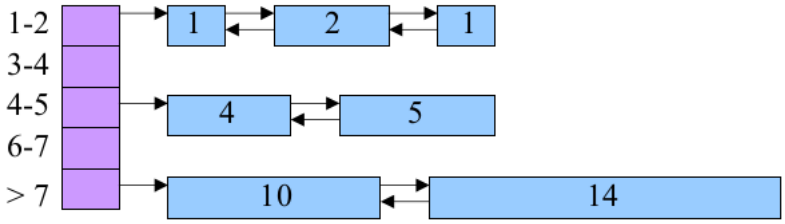

Figure 1.7 – Stratégie d'allocation par malloc (*binning*) : par interval

1.6 Notions et définitions utiles

Les types primitifs

Dans le monde de la programmation, un type primitif est un type de base prédéfini, fourni par le langage. On retrouve souvent le type booléen (valeur vrai ou faux), le type entier, le type réel, les chaînes de caractères, etc.

Les pointeurs faibles

Un pointeur faible est un pointeur dont la zone mémoire référencée est récupérable par le ramasse-miettes à n'importe quel moment durant l'exécution du programme (contrairement à un pointeur fort).

“Il faut avoir une bonne mémoire pour être en mesure de tenir les promesses que l’on fait.”

Friedrich Nietzsche

2 Gestion mémoire

Lorsqu’un programme est exécuté, une zone mémoire est allouée et gérée au fur et à mesure de son exécution. Cette mémoire du programme est généralement disponible sous différentes formes (pile, tas, registre, etc…). Elle va permettre de stocker l’information nécessaire à la bonne exécution du programme.

Le comportement mémoire d’un langage de programmation dépend des styles de programmation du langage considéré et des techniques de représentation des données qu’il utilise. Ce chapitre présente l’organisation mémoire des applications écrites en différents langages de programmation comme Swift, avec un ramasse-miettes à compteur de références, Java et Haskell dont le ramasse-miettes est très proche d’OCaml, le langage auquel nous nous intéressons.

2.1 Les différentes mémoires d’un programme

Un programme en langage machine est une suite d’instructions manipulant des valeurs en mémoire. Cette mémoire est généralement composée de plusieurs éléments, que nous allons détailler dans la suite, selon la stratégie d’allocation. En effet, on peut soit faire une allocation statique, soit une allocation dynamique sur la pile (voir 2.1.1), ou sur le tas (voir 2.1.4).

2.1.1 La pile

La pile est une structure permettant de stocker des données. C'est une structure de conteneur qui obéit à une discipline «dernier entré - premier sorti» (LIFO : Last In First Out) signifiant que les derniers éléments ajoutés à la pile seront les premiers à être récupérés (figure 2.1).

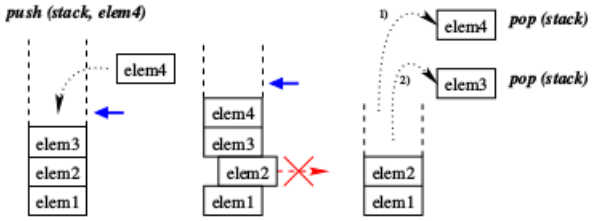

Figure 2.1 – Schéma décrivant le fonctionnement d'une pile.

On alloue, par exemple, dans la pile de la mémoire lors d'un appel de fonction, cette mémoire étant libérée lorsque la fonction termine. Les paramètres des fonctions peuvent être stockés dans la pile et sont libérés à la fin de l'exécution de la fonction. Il existe aussi une structure de tas (voir 2.1.4) qui permet l'allocation de mémoire à tout moment, dont la durée de vie ne dépend pas de l'exécution d'une fonction.

2.1.2 Les registres du processeur

Un registre est une zone de mémoire interne à un processeur. C'est la mémoire la plus rapide d'un ordinateur, est présente en quantité limitée dans le microprocesseur, à cause de son coût de fabrication élevé. La capacité d'un registre est généralement comprise entre 1 et 16 octets. Les valeurs immédiates (comme la liste vide, `unit`, etc. en OCaml) peuvent ainsi être passées directement dans des registres et ne pas apparaître sur la pile (seulement si le nombre de paramètres d'une fonction n'est pas très élevé).

2.1.3 Le segment de données

Le segment de données est une zone mémoire virtuelle allouée statiquement qui permet de stocker les variables globales et les variables statiques. Cette portion de la mémoire a une taille fixe pour chaque programme qui va dépendre des variables initialisées par le programmeur. Il y en a généralement deux : le `.data` pour les variables disponible en lecture et écriture et le `.rdata` pour les variables en lecture seule (par exemple, les constantes).

2.1.4 Le tas

Le tas est l'une des deux sortes de mémoires utilisées pour l'allocation dynamique (l'autre étant la pile d'exécution). Les objets y sont alloués dynamiquement à tout moment de l'exécution du programme. Lors de l'exécution d'un programme, le tas est utilisé pour allouer dynamiquement de l'espace mémoire à durée de vie non bornée, à la demande du programme. L'allocation dans le tas est donc dynamique, par opposition à la mémoire statique, allouée à l'initialisation du programme. Le caractère non borné *a priori* de la mémoire allouée dans le tas diffère aussi de la mémoire allouée dans la pile, dont la durée de vie est limitée à un appel de fonction. C'est précisément cette durée de vie de la mémoire allouée dans le tas qui permet l'avènement des fuites mémoire : des blocs mémoire non libérés peuvent rester alloués très longtemps même si les données qu'ils contiennent sont devenues inutiles.

2.2 Gestion mémoire en Swift et Objective-C

Les langages *Swift* et *Objective-C* utilisent ARC, un ramasse-miettes à compteur de références automatique (voir section 1.1), pour leur gestion mémoire. Nous allons détailler cette gestion automatique et la manière dont Swift gère le problème commun aux ramasse-miettes à compteur de références, c'est-à-dire les valeurs cycliques.

L'avantage de ce ramasse-miettes est qu'il permet l'appel à une fonction de *finalisation* de façon prévisible et immédiate.

Nous allons voir un peu plus en détails le fonctionnement du ramasse-miettes de *Swift*.

2.2.1 Swift, un langage de programmation pour smartphone

Swift [6] est un nouveau langage de programmation multiparadigme (objet, fonctionnel et impératif) (développé par Apple en 2014, pour le développement d'applications sous IOS et OSX. Tout comme C et Objective-C, *Swift* fournit des types de bases d'Objective-C comme les entiers, les flottants, les doubles, les chaînes de caractères, les tableaux, les dictionnaires, etc. Il introduit également des types avancés comme les tuples, les types options, etc.

Swift permet de déclarer des variables mutables, et permet également de définir des variables constantes, plus puissantes que les constantes C. *Swift* favorise ces valeurs constantes (immutables), améliorant la sûreté des programmes, la lisibilité ainsi que les performances. Par défaut, les méthodes déclarées dans des types `struct` ou `enum`

ne permettent pas de modifier la structure. De plus, les paramètres des fonctions sont constants par défaut.

2.2.2 Le ramasse-miettes dans Swift

2.2.2.1 Ramasse-miettes à compteur de références automatique

La gestion mémoire en *Swift* se fait de manière automatique à l'aide d'un ramasse-miettes à compteur de références (*ARC*, Section 1.1). Celui-ci va alors allouer et désallouer les valeurs en mémoire de manière implicite sans intervention explicite du programmeur. Il peut arriver dans certains cas que le ramasse-miettes ait besoin de certaines informations supplémentaires du programmeur pour une meilleure gestion mémoire (les pointeurs circulaires, détaillés dans la suite).

À chaque création d'une instance de classe, le ramasse-miettes alloue un chunk mémoire pour stocker l'information concernant cette classe (type, valeur et propriétés de la classe). Dès que l'instance de la classe n'est plus utile, elle est libérée par le ramasse-miettes (plus aucune référence dessus, le compteur est à zéro).

2.2.2.2 Exemple de code en Swift

Sur la figure 2.2, la classe `Person` a un constructeur initialisant la classe et affiche un message de succès. Elle a également un destructeur, qui affiche un message lors de la suppresion d'une instance de cette classe.

```
1   class Person {
2       let name : String
3       init(name : String) {
4         self.name = name
5         println("\(name) is being initialized")
6       }
7       deinit {
8         println("\(name) is being deinitialized")
9       }
10  }
```

Figure 2.2 – Définition d'une classe `Person` en *Swift* avec son constructeur et son destructeur. Exemple disponible en ligne sur le site du langage.

Sur la figure 2.3, trois variables de type `Person` sont créées (ligne 1, 2 et 3), ainsi qu'une instance de cette classe (ligne 5) :

```
1  var reference1 : Person?
2  var reference2 : Person?
3  var reference3 : Person?
4
5  reference1 = Person(name : "John Appleseed")
6  // affiche "John Appleseed is being initialized"
7
8  reference2 = reference1
9  reference3 = reference1
10
11 reference1 = nil
12 reference2 = nil
13
14 reference3 = nil
15 // affiche "John Appleseed is being deinitialized"
```

Figure 2.3 – Exemple du ramasse-miettes de *Swift* avec une instance de la classe `Person` de la figure 2.2. Exemple disponible en ligne sur le site du langage.

La nouvelle instance crée un pointeur fort vers `reference1` (ligne 5). Ensuite, ligne 8 et 9, deux autres références sont établies. Il y a donc en tout trois pointeurs forts vers cette instance.

Le ramasse-miettes de *Swift* ne désallouera pas l'instance de la classe `Person` tant qu'il y aura une référence vers celle-ci.

Dans cet exemple, on supprime deux des trois références vers la seule instance de la classe `Person`, ligne 11 et 12. À ce moment-là, il ne reste plus qu'un pointeur fort, l'objet est donc toujours vivant. Puis, la dernière référence est mise à `nil`, soit supprimée, à la ligne 14 et c'est seulement à ce moment là que le ramasse-miettes libérera la mémoire (d'où l'affichage juste après cette ligne du message du destructeur).

2.2.2.3 Traitement des valeurs cycliques

Comme nous l'avons vu dans la section 1.1, l'inconvénient de ce type de ramasse-miettes est le fait que les valeurs cycliques ne sont pas gérées (voir figure 1.1). Pour pallier cela, *Swift* fournit deux moyens : les pointeurs faibles et des références appelées *unowned* (sans référants, n'appartenant à personne). Cela ne résout pas le problème, mais donne juste au programmeur, un moyen de s'en sortir.

Si nous prenons l'exemple de deux classes mutuellement récursives, on obtient alors le code suivant :

```
1  class Person {
2    let name : String
3    init(name : String) { self.name = name }
4    var apartment : Apartment?
5    deinit { println("\(name) is being deinitialized") }
6  }
7
8  class Apartment {
9    let number : Int
10   init(number : Int) { self.number = number }
11   weak var tenant : Person?
12   deinit { println("Apartment #\(number) is being deinitialized") }
13 }
```

Figure 2.4 – Définition de deux classes mutuellement récursives en Swift. Exemple disponible en ligne sur le site du langage.

Sur la figure 2.4, à la ligne 11, on voit apparaître le mot clé `weak`. Le pointeur fort sera donc remplacé par un pointeur faible (définit dans la Section 1.6).

```
1
2  var john : Person?
3  var number73 : Apartment?
4
5  john = Person(name : "John Appleseed")
6  number73 = Apartment(number : 73)
7
8  john!.apartment = number73
9  number73!.tenant = john
10
11 john = nil
12 // prints "John Appleseed is being deinitialized"
```

Figure 2.5 – Gestion des valeurs circulaires en Switft à l'aide de pointeurs faibles. Exemple disponible en ligne sur le site du langage.

Concrètement, si nous prenons l'exemple de la figure 2.5, cela signifie que lors de la suppression de la référence forte de la classe `Person` (ligne 11), il n'y aura alors plus de référence forte vers celle-ci. Ainsi, le fait qu'il n'y ait plus de référence forte vers la classe `Person` permettra aux ramasse-miettes de libérer la mémoire des instances des classes `Person` et `Apartment`.

Comme les pointeurs faibles, une référence *unowned* ne garde pas une référence forte

sur l'instance sur laquelle elle pointe. La différence majeure avec les pointeurs faibles est le fait qu'une pointeur *unowned* pointe toujours vers une valeur. Par conséquent, il ne peut référencer qu'un type non optionnel.

Comme pour les pointeurs faibles, un mot clé est introduit pour l'utilisation de ces pointeurs. Il suffira d'ajouter le mot clé `unowned` devant une déclaration de propriété ou variable.

```
1  class Customer {
2      let name : String
3      var card : CreditCard?
4      init(name : String) {
5          self.name = name
6      }
7      deinit { println("\(name) is being deinitialized") }
8  }
9
10 class CreditCard {
11     let number : UInt64
12     unowned let customer : Customer
13     init(number : UInt64, customer : Customer) {
14         self.number = number
15         self.customer = customer
16     }
17     deinit { println("Card #\(number) is being deinitialized") }
18 }
```

Figure 2.6 – Exemple de gestion des valeurs circulaires en Switft à l'aide de pointeurs *unowned*. Exemple disponible en ligne sur le site du langage.

Sur la figure 2.6, le programme définit deux classes `Customer` et `CreditCard`. Chacune des deux classes contient une instance de l'autre classe comme propriété de la classe. On peut donc avoir une référence forte cyclique.

La différence avec l'exemple de la figure 2.4 est que le client de type `Customer` peut ne pas avoir de carte de crédit (`CreditCard`), mais une carte de crédit sera toujours associée à un client. Le client aura donc un champ `card` optionnel (représenté avec le *?*), mais le champ `customer` de la classe `CreditCard` ne sera pas optionnel.

Sur la figure 2.7, on définit une instance de la classe `Customer` (`john`, à la ligne 1), puis on l'initialise avec un nom et une carte de crédit (ligne 3 et 4). À ce moment, là l'instance de la classe `Customer` a un pointeur fort sur l'instance de la classe `CreditCard` et l'instance `CreditCard` a une référence *unowned* vers le `Customer`. Il n'y donc plus de

```
1  var john : Customer?
2
3  john = Customer(name : "John␣Appleseed")
4  john!.card = CreditCard(number : 1234_5678_9012_3456, customer : john!)
5
6  john = nil
7  // prints "John Appleseed is being deinitialized"
8  // prints "Card #1234567890123456 is being deinitialized"
```

Figure 2.7 – Gestion des valeurs circulaires en Switft à l'aide de pointeurs *unowned*. Exemple disponible en ligne sur le site du langage.

référence forte vers la classe `Customer` depuis l'instance de `CreditCard`.

À la ligne 6, on supprime cette référence forte de `john` vers l'instance de la classe `Customer` et comme il y a une référence *unowned*, les deux instances vont être désallouées (ligne 7 et 8).

Swift ne permet pas de régler le problème des valeurs cycliques de façon automatique et sûre, mais fournit au programmeur un moyen de contourner ce problème, commun à ce type de ramasse-miettes.

2.3 Gestion mémoire en Java

Java est un langage orienté objet, dont la gestion mémoire se fait à l'aide d'un ramasse-miettes à générations. Du fait de ces contraintes multi-cœurs et objets, nous allons voir dans la suite que sa représentation mémoire est alors très coûteuse, même pour l'allocation de petits objets. Dans la suite, nous détaillons la machine virtuelle d'Oracle.

2.3.1 Java, langage de programmation orienté objet

Java est un langage de programmation orienté objet, créé par James Gosling et Patrick Naugton en 1995. Le trait caractérisant ce langage est sa portabilité sur la plupart des plateformes (UNIX, Windows, Mac OS et GNU/Linux entre autres) avec très peu, voire aucune modification à ajuster au code source, grâce au bytecode.

Java est un langage simple, robuste et multitâche. Sa gestion mémoire est automatique à l'aide d'un ramasse-miettes à génération (voir section 1.2.4). Cette gestion mémoire automatique contribue à la robustesse des programmes. Cependant, sa représentation mémoire est très lourde dû aux contraintes multi-cœurs.

Une de ses grandes forces qui a fait sa popularité est une vaste librairie, permettant de faire des applications assez facilement et rapidement sous forme de console, ou des programmes graphiques à l'aide de librairies graphiques ou sous forme d'applets (pour le web).

Nous allons voir dans la suite la représentation mémoire des objets alloués en mémoire par le ramasse-miettes de Java, ainsi que son ramasse-miettes et l'algorithme qu'il utilise.

2.3.2 Représentation mémoire des objets

Dans cette section, nous allons détailler la représentation mémoire des objets Java [14, 15] sur les architectures 32 bits et 64 bits.

2.3.2.1 Représentation des objets Java en 32 bits et 64 bits

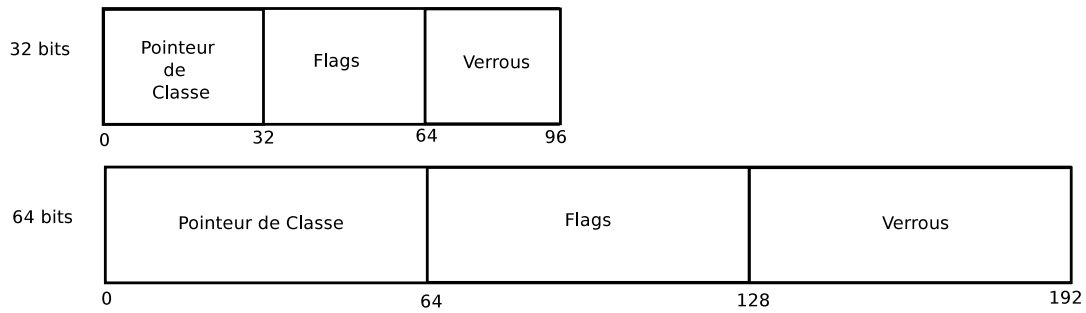

Figure 2.8 – En-tête d'un bloc Java sur les architectures 32 bits et 64 bits.

Sur la figure 2.8, on peut voir une description de l'en-tête d'un objet Java 32 bits et 64 bits. Les champs décrivent :

- le pointeur de classe : un pointeur vers les informations concernant la classe décrivant le type de l'objet. Par exemple, pour le cas d'un tableau d'entiers, le pointeur référencera la classe `int []`
- les flags : un ensemble de flags décrivant l'état de l'objet
- les verrous : ce champ contient les informations de synchronisation de l'objet. En particulier, ce champ nous dit si l'objet est synchronisé ou pas.

Sur la figure 2.9, on peut voir un exemple de la représentation d'un objet `java.lang.Integer` en 32 bits.

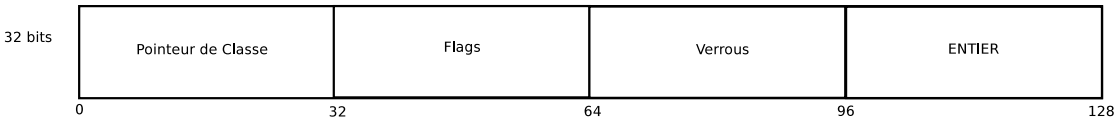

Figure 2.9 – Représentation en mémoire d'un objet `java.lang.Integer` en Java en 32 bits.

2.3.2.2 Représentation des tableaux Java en 32 bits et 64 bits

Sur la figure 2.10, on peut voir la représentation d'un tableau d'entier en Java 32 bits.

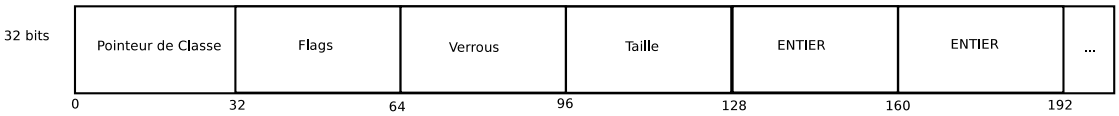

Figure 2.10 – Représentation en mémoire d'un tableau d'entiers en Java en 32 bits.

2.3.2.3 Représentation des objets plus complexes en Java

Une bonne pratique de la programmation orienté Objet est d'utiliser l'encapsulation et la délégation. L'encapsulation permet de fournir une interface pour manipuler les données que contiennent une classe. Mais cette bonne pratique a un surcoût au niveau de la mémoire. En effet, elle augmente de façon non négligeable la taille des blocs alloués pour les données. Si nous prenons la figure 2.11, nous avons l'exemple d'une `java.lang.String` contenant quatre caractères.

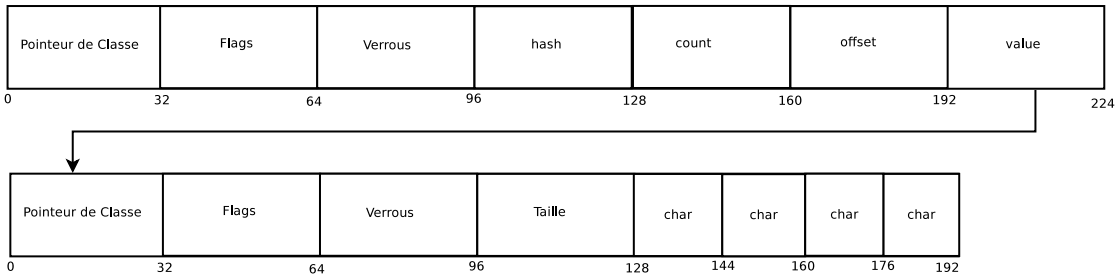

Figure 2.11 – En-tête d'un bloc représentant un objet complexe en Java 32 bits. Cet exemple représente une chaîne de caractères `java.lang.String` de taille 4.

Pour avoir une chaînes de caractères de taille quatre (soit 8 octets), il faut

- 24 octets pour le tableau contenant les caractères avec son en-tête,
- 28 octets pour l'objet encapsulant, permettant de manipuler la chaîne de caractères.

Soit un total de 52 octets pour représenter seulement 8 octets de données.

2.3.3 Le ramasse-miettes à générations de Java

2.3.3.1 Les racines du ramasse-miettes de Java

Comme nous l'avons vu dans le chapitre précédent, les racines sont des emplacements mémoires à partir desquelles sont atteints tous les objets vivants de la mémoire. Tous les objets atteignables depuis ces racines sont alors considérés comme vivants.

En Java, on considère comme racines du ramasse-miettes :

- *les variables locales* gardées vivantes par la pile d'un thread,
- les threads, considérés comme vivants,
- les variables statiques des classes,
- les références JNI (Java Native Interface), les objets natifs en Java. Ces objets sont traités différemment car la machine virtuelle n'a aucune information sur les références de ces derniers. Ils sont donc considérés comme des racines du ramasse-miettes.

En règle générale, une application simple et basique en Java va avoir trois types de racines : les variables locales de la méthode `main`, le thread principal exécutant le `main` et les variables statiques de la classe `main`.

2.3.3.2 Les zones mémoires

En Java, les zones mémoires utilisées se différencient des autres langages, notamment par :

- la pile, une par thread,
- une zone mémoire pour les méthodes.

La pile par thread

Chaque thread possède sa propre pile contenant les variables accessibles seulement par le thread, ou autrement dit les variables locales, les paramètres des fonctions ainsi que les valeurs de retour des méthodes utilisées dans ce thread.

La pile étant limitée, les seules données stockées sur celle-ci seront les valeurs de types primitifs (voir Section 1.6) et les adresses des objets. Les objets créés par `new` seront stockés dans le tas.

Par rapport à une pile *classique*, comme en C, la spécificité ici est que seuls les pointeurs sont stockés sur la pile et non les objets eux-mêmes [19].

Lorsqu'il n'y a plus de place sur la pile, une exception `StackOverflowError` est levée. L'exception `OutOfMemoryError` est levée lorsqu'un thread ne peut allouer l'espace nécessaire pour la pile.

Le tas

Contrairement à la pile, le tas est partagé par tous les threads de la machine virtuelle Java. Toutes les instances des objets créées y seront stockées et partagées par tous les threads. Comme il n'y a qu'un seul tas par instance de machine virtuelle Java, chaque nouvelle application aura son propre tas. Il n'y aura pas possibilité pour deux applications données de partager ou de lire des données du tas de l'autre.

Les tableaux étant représentés comme des objets en Java, ils seront stockés dans le tas malgré leur statut de type primitif.

La zone des méthodes

Comme le tas en Java, cette zone mémoire est partagée par tous les threads. Elle permet de stocker la définition des classes et des interfaces, ainsi que le code des méthodes déclarées. On y trouvera donc le code des constructeurs, des méthodes, des constantes, des variables de classe, etc.

Comme pour la pile, seule des données de type primitif ou des références à des objets peuvent être stockées dans cette zone de mémoire. Étant partagée par tous les threads, il faut alors apporter une attention particulière aux accès des variables statiques.

2.3.3.3 Le ramasse-miettes à générations

Le ramasse-miettes de Java est un ramasse-miettes à 3 générations, la jeune, la vieille et la génération permanente.

Les générations *La jeune génération* est l'endroit où les objets fraîchement alloués sont stockés. Elle est divisée en trois espaces : l'espace *eden*, *from-space* et *to-space*. Lorsque cette génération est pleine, une collection sur ce tas est déclenchée. Un objet fraîchement alloué sera stocké dans l'eden et si l'objet survit à un cycle de ramasse-miettes, il sera déplacé dans un des deux espaces cités précédemment. Si l'objet survit à un deuxième cycle de ramasse-miettes, il est alors copié dans la vieille génération.

L'algorithme utilisé dans ce tas est celui du ramasse-miettes copiant (voir Section 1.2.3). Le ramasse-miettes stoppe l'application jusqu'à ce que le nettoyage soit effectué.

La génération vieille est utilisée pour stocker les objets avec une durée de vie plus longue que les objets de la jeune génération. D'une façon générale, lorsque les objets de la jeune génération atteignent un certain âge (au moins deux cycles de ramasse-miettes), ils sont déplacés dans cette génération.

De la même manière que la jeune génération, la vieille génération utilise aussi un ramasse-miettes copiant (voir section 1.2.3).

La génération permanente permet à la machine virtuelle Java de stocker des données permettant de décrire les classes et les méthodes utilisées pour une application donnée. Elle contient également les classes et méthodes correspondant aux librairies Java.

Les classes peuvent être collectées si la machine virtuelle Java trouve qu'elles ne sont plus utiles et qu'il n'y a plus d'espace mémoire.

LE RAMASSE-MIETTES À GÉNÉRATIONS ET À BALAYAGE Pour résumer, le ramasse-miettes va essentiellement travailler sur deux générations :

- la jeune génération
- la vieille génération

Nous allons maintenant décrire l'algorithme du ramasse-miettes utilisé par la version actuelle distribuée par Oracle.

Sur la figure 2.12, on peut voir l'agencement des générations en Java.

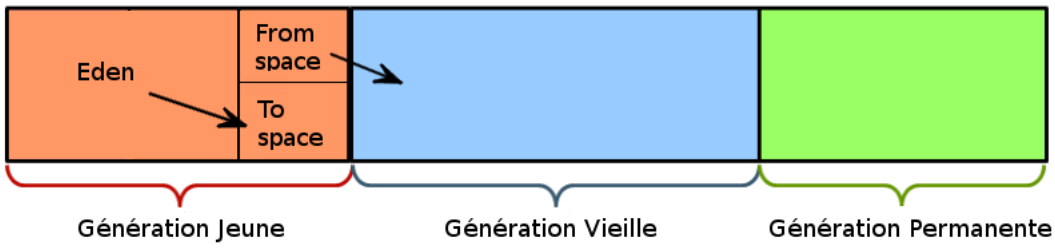

Figure 2.12 – Représentation mémoire des différentes générations en Java.

Tout d'abord les objets sont alloués dans la jeune génération. Après un cycle de ramasse-miettes, les objets alloués dans l'*eden* passe dans la zone *to-space* et les objets

de la zone *from-space* dans la vieille génération. À la fin de cette étape, les deux zones *from-space* et *to-space* sont échangées.

L'allocation d'objets est aussi soumise à des problématiques multithreads puisque chaque thread peut demander l'allocation d'objets. Pour tenir compte de ces contraintes, la machine virtuelle Java réserve à chaque thread une zone de mémoire de l'espace *eden* nommée *Thread-Local Allocation Buffer* (TLAB) dans laquelle les objets du thread sont créés. Une synchronisation est cependant nécessaire si le TLAB est plein et qu'il faut en allouer un supplémentaire au thread. De la même manière, une zone de mémoire, appelée *Promotion-Local Allocation Buffer* (PLAB), est utilisée pour les objets promus vers la *to-space* ou la génération vieille.

L'algorithme de marquage et balayage est celui décrit dans la section 1.2.2.

Du fait de ces contraintes multi-cœurs, la représentation mémoire des objets en Java est très coûteuse, même dans le cas d'allocation de petits objets.

2.4 Gestion mémoire en Haskell/GHC

2.4.1 Haskell, langage de programmation fonctionnel pur

Haskell [48, 4] est un langage de programmation fonctionnel créé en 1990 par un comité de chercheurs en théorie des langages intéressés par les langages fonctionnels et l'évaluation paresseuse. Il est fondé sur le lambda-calcul et la logique combinatoire. Le langage continue d'évoluer, principalement avec le compilateur GHC, constituant ainsi un standard de facto. Comme d'autres langages (Java, OCaml, Lisp, etc.), Haskell/GHC dispose d'un système de gestion mémoire automatique à l'aide d'un ramasse-miettes.

Les fonctionnalités les plus intéressantes de Haskell sont le support des fonctions récursives, l'inférence de types, le filtrage (*pattern matching* en anglais) et l'évaluation paresseuse. Ces fonctionnalités, surtout si on les combine, facilitent l'écriture et l'utilisation de fonctions. Le système de types permet la représentation et la vérification statique de nombreuses contraintes ordinairement vérifiées dynamiquement, à l'exécution. Haskell se distingue également par l'utilisation de monades pour les entrées/sorties, rendues nécessaires par l'une des plus importantes particularités du langage : Haskell est un langage *fonctionnel pur* (par défaut, aucun effet de bord n'est autorisé, comme les entrées/sorties impératives, ou l'affectation d'une variable). Haskell impose ce style de programmation dans tout code qui ne signale pas explicitement par son type qu'il contient des effets de bord.

Dans la suite, nous allons voir en quoi l'évaluation paresseuse fait une différence dans la représentation et la gestion de la mémoire.

2.4.2 Représentation mémoire des valeurs en Haskell/GHC

Les valeurs alloués dans le tas

Haskell étant un langage fonctionnel pur, tous les calculs effectués par l'évaluation des expressions donnent des valeurs. Chaque valeur se voit associer un type. Comme exemple de valeurs, nous avons l'entier `42`, le caractère `'a'`, la fonction `\x -> x * x`, la liste *[1, 2, 3]*, etc.

GHC [56] gère de façon uniforme le tas et la pile. Le tas consiste en deux sortes d'objets, les valeurs, appelées aussi *head normal forms*, et les expressions non évaluées, appelées **thunks**. Une valeur peut contenir un certain nombre de thunks, par exemple pour une liste, le constructeur `Cons` peut avoir sa tête de liste ou sa fin de liste non évaluée. Tous les objets ayant la même représentation, on utilise le terme de *clôture* pour ces deux types d'objets. Sur la figure 2.13, on peut voir la forme d'un objet du tas Haskell. Chaque objet commence par un en-tête contenant un pointeur vers la table d'information. Dans la version par défaut, l'en-tête ne contient que cette information, elle est donc stockée sur un seul mot machine (l'en-tête peut varier par exemple lors de la compilation en mode debug).

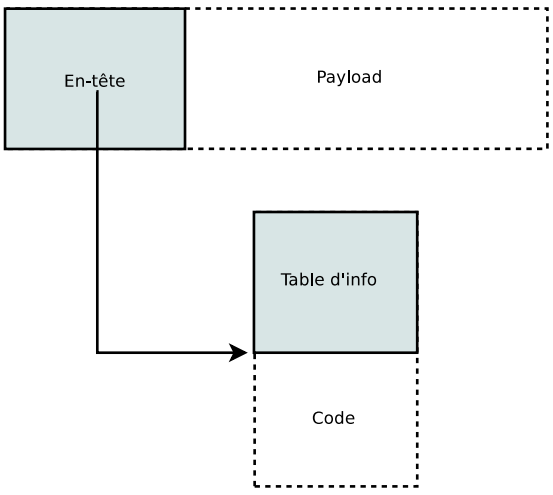

Figure 2.13 – Représentation d'un objet Haskell dans le tas.

Le code est le code qui sera évalué par la clôture. Pour les fonctions, le code appliquera la fonction aux arguments stockées dans les registres ou sur la pile, selon la

convention d'appel. Cette partie suppose que tous les arguments sont bien présents.

La *table d'information* contient toute l'information nécessaire à la runtime à propos des clôtures. Sur la figure 2.14, on peut voir la forme que prend cette table.

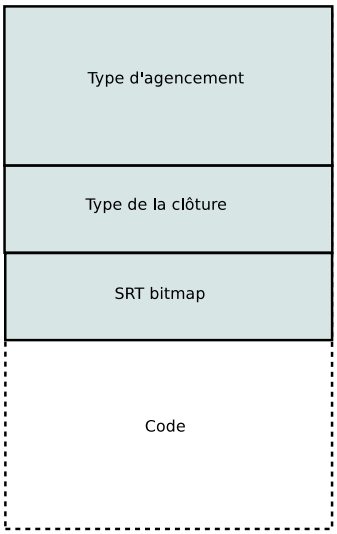

Figure 2.14 – La table d'information nécessaire pour les clôtures en Haskell.

Le champ **agencement** décrit l'agencement du champ **payload** pour le ramasse-miettes. Il a seulement besoin d'avoir deux informations à propos des objets dans le tas : sa taille en mots et l'information indiquant si certains de ces mots sont des pointeurs.

Il existe deux types d'agencements pour le champ `payload`, dont le type dépend du type de la clôture :

- pointers-first : consiste en une suite de pointeurs, suivie de valeurs n'étant pas des pointeurs. La plupart des objets vont utiliser ce type d'agencement, dont les constructeurs, les fonctions ainsi que les thunks (les clôtures non évaluées).
- bitmap : consiste en un mélange de pointeurs et valeurs n'étant pas des pointeurs, décrit par un bitmap. Cet agencement sera utilisé entre autres par les *stack frames* (ou *activation records.*)

Le **type de la clôture** est une constante décrivant le genre de la clôture (fonction, thunk, constructeur, etc.). Le ramasse-miettes va d'abord scanner ce champ pour déterminer comment interpréter le type d'agencement.

Le champ **SRT bitmap** (pour *static reference table*) est utilisé par le ramasse-miettes pour les valeurs définies à top-level.

Les **flags** des clôtures permettant d'avoir des informations rapides sur des propriétés

d'une clôture comme par exemple si elle est ou non statique.

Certains types d'objets vont ajouter plus de champs à la fin de cette table, comme les fonctions, adresses de retour et les thunks.

La particularité d'Haskell sont les *thunks* qui sont des expressions non évaluées contenant du code et des pointeurs si nécessaire. Comme tous les objets il aura la même représentation que nous avons décrit précédemment. Lorsqu'un thunk est forcé, le calcul est alors effectué. Il sera ensuite mis à jour et écrasé avec la valeur calculée. Un pointeur de code lui sera assigné pour que les fois suivantes, la valeur calculée soit retournée à chaque fois sans réévaluation du thunk.

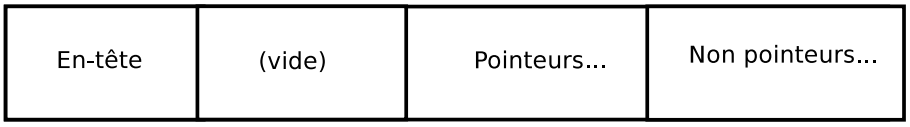

Figure 2.15 – Représentation mémoire d'un thunk en Haskell/GHC.

La partie vide est réservée pour le code de mise à jour du thunk pour écraser la cible d'une indirection sans pour autant perdre l'information des variables libres. L'avantage de cela est qu'une mise à jour du thunk peut être faite sans synchronisation (la pureté garantissant que deux écritures concurrentes écrivent le même résultat).

La pile

La pile consiste en une séquence de frames, chaque frame étant représentée exactement par la *même forme qu'un objet du tas*. Ainsi, les valeurs sur la pile et le tas peuvent être traitées de façon uniforme.

Le ramasse-miettes à générations

Le modèle de calcul de Haskell est très différent de celui de langues impératifs classiques. L'immuabilité de la plupart des données nous oblige à produire un grand nombre de données temporaires, mais il contribue également à nettoyer ces valeurs assez rapidement.

Pour les structures de données mutables (très peu utilisées), il peut arriver qu'une valeur dans le tas majeur pointe vers le tas mineur. Le ramasse-miettes garde alors ces objets dans un ensemble qui sera considéré comme une racine du programme. En particulier, les *thunks* ne sont mutables qu'une seule fois. Une fois le calcul des thunks

effectués, ils deviennent alors immuables jusque la fin du programme. Ainsi, les thunks sont immédiatement éliminés de cet ensemble.

Cela simplifie grandement la collecte des valeurs mortes par le ramasse-miettes (voir Section 1). Par défaut, *GHC* utilise un ramasse-miettes à générations (voir section 1.2.4). Les nouvelles données sont allouées dans la jeune génération. Une fois que cet espace est plein, une collection mineure est lancée - il scanne cet espace mémoire et libère les valeurs inutilisées (mortes). Les valeurs vivantes sont copiées dans la zone mémoire principale. Moins il y a de valeurs qui survivent, moins il y a de travail de copie à faire.

Si nous prenons l'exemple d'un algorithme récursif remplissant rapidement la mémoire de données avec ces valeurs temporaires d'induction, seules les variables de la dernière étape de récursion seront copiées dans la zone mémoire principale. Il y a donc un comportement contre-intuitif, puisque plus la durée de vie des valeurs est courte, plus le programme ira vite.

Pour conclure, la représentation des données en mémoire d'Haskell/GHC est uniforme. La particularité de ce langage est l'évaluation paresseuse : les thunks qui sont des expressions non évaluées. L'immuabilité de la plupart des données oblige à produire un grand nombre de données temporaires, mais contribue également à les nettoyer assez rapidement.

2.5 Gestion mémoire en OCaml

2.5.1 OCaml, langage de programmation fonctionnel, impératif et orienté objet

OCaml est un langage multiparadigme, fonctionnel, impératif, orienté objet, avec un typage fort et une évaluation stricte. Cela implique la possibilité d'utiliser un format compact des données (l'absence de tests de types dynamiques permet de se passer d'information de type durant l'exécution), une mutation possible des objets pendant l'exécution et une durée de vie faible pour la plupart des objets.

Dans cette section, nous allons expliquer en détails la façon dont est gérée la mémoire en OCaml, la représentation de celle-ci, ainsi qu'une explication sur le ramasse-miettes avec les algorithmes utilisés pour allouer et désallouer les blocs devenus inaccessibles durant l'exécution du programme. OCaml étant le langage auquel nous nous intéressons, nous détaillerons plus certains aspects du langage ainsi que sa représentation et sa gestion mémoire.

2.5.2 Représentation mémoire des données

2.5.3 Une unité d'allocation de mémoire (*chunk*)

Un *chunk*[1] est une région mémoire allouée via la fonction C `malloc`. Les chunks mémoire sont des éléments de mémoire virtuelle, alignés sur des pages mémoires : leurs tailles sont des multiples de la taille des pages.

2.5.3.1 Représentation des valeurs

Dans une application OCaml, une valeur représente soit une valeur *entière immédiate* (ou équivalent), soit un *pointeur* vers un autre bloc mémoire.

Une *valeur entière immédiate* peut représenter soit un entier (constante), un caractère ou un constructeur constant (sans paramètre) (par exemple, les booléens `true` et `false`, la liste vide `[]`, la valeur `()` de type `unit`).

L'organisation de la mémoire en OCaml est telle qu'il est toujours possible de distinguer un pointeur d'une valeur immédiate : l'espace d'adressage et celui des entiers sont tous deux amputés d'un bit, dont la valeur sert précisément à distinguer entiers et pointeurs :

- les entiers sont donc représentés sur 31/63 bits,
- et les blocs, avec un tag supérieur à la valeur *No_scan_tag* pour les flottants, les chaînes de caractères, etc.

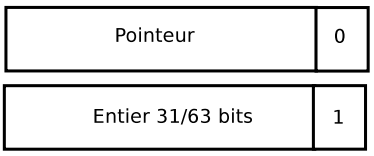

Figure 2.16 – Représentation mémoire d'un entier et d'un pointeur OCaml

Le ramasse-miettes d'OCaml peut donc être précis et n'a pas besoin d'être conservatif.

Les *valeurs boxées* sont des indirections permettant d'avoir des *wrappers* autour de types primitifs. Pour faire simple, ce sont des pointeurs sur des valeurs dans le tas. Elles sont allouées dans le tas. Par exemple, les flottants sont souvent alloués dans le tas (il existe certains cas où le compilateur sait optimiser ces valeurs pour ne pas les boxer). La principale raison est que le ramasse-miettes a besoin d'identifier les valeurs à l'aide du champ `tag` (voir 2.5.3.2) pour pouvoir les nettoyer lorsqu'elles ne seront plus utilisées.

1. Unité d'allocation (de mémoire)

Enfin, il faut faire la différence entre les pointeurs classiques, pointant vers une valeur dans le tas OCaml, et les pointeurs C. La runtime d'OCaml garde une trace des pointeurs sur une valeur du tas, il suffit alors de tester si un pointeur est à l'intérieur d'un *chunk* (voir section 2.5.3) ou pas. Si le pointeur pointe en dehors de ces chunks, alors il est traité comme un pointeur C, qui ne sont pas concernés par le ramasse-miettes d'OCaml.

2.5.3.2 Représentation des blocs

Nous avons vu que les pointeurs OCaml dans le tas pointaient vers des valeurs boxées. Sur la figure 2.17, on peut voir la composition d'un bloc en mémoire qui consiste en un en-tête (header) suivi d'une suite de *mots* (word). Un mot OCaml est codé sur 4/8 octets (selon l'architecture 32/64 bit).

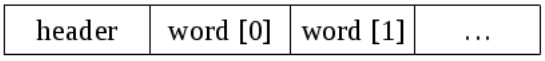

Figure 2.17 – Bloc mémoire OCaml.

Lorsque nous regardons l'en-tête de plus près (figure 2.18), on remarque que celui-ci est également codé sur 4/8 octets et contient trois champs :

- *la taille* (22/54 bits) : contient le nombre de mots du bloc.
- *la couleur* (2 bits) : utilisée par le ramasse-miettes (mark&sweep).
- *le tag* (8 bits) : encode l'information de type minimale requise pour des opérations classiques comme la discrimination entre les variants avec arguments, la comparaison polymorphe, *etc.* et est également utilisé par le ramasse-miettes. Si le tag est inférieur à `No_scan_tag`, alors il sera scanné par celui-ci.

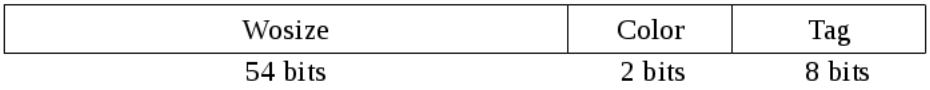

Figure 2.18 – En-tête d'un bloc OCaml.

2.5.4 Le ramasse-miettes d'OCaml

Le ramasse-miettes d'OCaml est un GC à deux générations qui combine deux techniques différentes selon qu'il agit sur le tas mineur ou sur le tas majeur :

- tas mineur : petit et de taille fixe (dans la version 4.01.0), la plupart des blocs alloués passent par ce tas avant d'être déplacés dans le tas majeur. Souvent, la

durée de vie des blocs alloués dans ce tas est courte. Il utilise l'algorithme de *Stop&Copy*

- tas majeur : de taille variable, il permet de stocker les blocs de durée de vie plus longue. Il utilise l'algorithme de *Mark&Sweep*

Le ramasse-miettes va donc agir différemment selon le tas qu'il va examiner. Nous verrons dans la section suivante les différents algorithmes utilisés sur les deux tas d'OCaml.

2.5.4.1 Le tas mineur

Une particularité de la gestion mémoire en OCaml est que le tas n'est pas alloué définitivement au début du programme, mais il évolue au cours du temps (augmentation ou diminution d'une taille donnée).

Initialisation La première étape est donc d'initialiser le tas avec une certaine taille et de placer les pointeurs au bon endroit. Ces pointeurs vont nous permettre de naviguer dans le tas mineur.

On remarquera qu'il y a essentiellement quatre pointeurs sur ce tas :

- `caml_young_start` : ce pointeur marque le début du tas mineur. Il est aligné sur une page mémoire et ne commence pas forcément au début de la zone allouée pour le tas mineur. Aucun objet ne pourra être alloué avant cette limite.
- `caml_young_end` : ce pointeur marque la fin du tas mineur. Aucun objet ne pourra être alloué au delà de cette limite
- `caml_young_ptr` : ce pointeur va permettre de suivre l'allocation des blocs dans le tas. En effet, à chaque allocation dans le tas mineur, ce pointeur va descendre de la taille du bloc alloué, ajouter un header avec la taille de ce bloc alloué. On peut voir ce comportement sur la figure 2.19
- `caml_young_limit` : qui marque la limite du tas mineur.

On peut se demander l'utilité du pointeur `caml_young_limit` qui semble être toujours être égal à `caml_young_start`. La runtime d'OCaml, déclenche une collection sur le tas mineur en mettant le `caml_young_limit` à `caml_young_end`. Grâce à cette astuce, les prochaines allocations n'auront plus assez de place et déclencheront une collection mineure. Le compilateur, pour des raisons internes, a besoin de déclencher des collections mineures en avance, comme pour la gestion des signaux.

Algorithme copiant, *Stop&Copy* L'algorithme copiant est décrit dans la section 1.2.3.

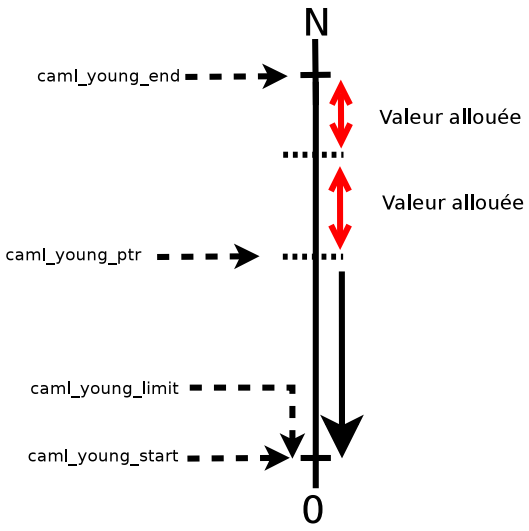

Figure 2.19 – Description des pointeurs sur le tas mineur d'OCaml.

La différence majeure est qu'OCaml étant un langage fonctionnel impur, il faut garder dans une table, les références du tas majeur vers le tas mineur (ceci n'étant pas possible dans un langage fonctionnel pur et strict). Ces pointeurs sont communément appelés les pointeurs inverses. À la figure 2.20, on peut voir exemple de code de ce cas d'usage.

```
1 let ancien = ref [ 1 ];;
2     val ancien  : int list ref = {contents=[1]}
3
4 (* on manipule ancien ici *)
5
6 let jeune = [2; 5; 8] in ancien  := jeune;;
7     -  :unit = ()
```

Figure 2.20 – Exemple de code où il y a une référence d'une valeur du tas majeur vers une valeur du tas mineur.

Remarque Cette table de références grossit peu et est vidée après une collection mineure, puisque les valeurs vivantes du tas mineur sont copiées au même endroit que les objets pointant vers ces valeurs (c'est-à-dire le tas majeur). Chaque collection mineure considère ces pointeurs comme des racines.

Partage Lorsque l'on tente de copier un objet qui a déjà été copié, il suffit alors de réutiliser la nouvelle adresse de celle-ci stockée dans son premier champ.

Vieillissement On appelle le «vieillissement» le fait de copier un objet du tas mineur vers le tas majeur. Pour faire simple, la phase de «vieillissement» des objets est divisée en quatre parties :

- lorsque le tag des objets est inférieur à `Infix_tag` (249) : on alloue d'abord un bloc de la taille de l'objet, on met à jour tous les pointeurs vers ce nouveau bloc, puis on met à jour la valeur.
  - si l'objet est plus grand qu'un seul mot machine alors : on ajoute le bloc à la "to do" liste
  - sinon on avance le pointeur
- lorsque le tag des objets est supérieur ou égal à `No_scan_tag` (251) : on alloue un nouveau bloc de la taille de l'objet, change son adresse en mettant à jour son premier champ, et enfin marque l'objet comme étant déplacé.
- lorsque le tag vaut la valeur `Infix_tag` (249) : on calcule l'offset de l'objet, puis on déplace son pointeur au pointeur courant, moins cet offset.
- lorsque le tag vaut `Forward_tag`, `Lazy_tag` ou `Double_tag` : on copie comme un bloc classique

2.5.4.2 Le tas majeur

Algorithme traversant, *Mark&Sweep* L'idée du Mark&Sweep est de parcourir le graphe depuis les racines en marquant les sommets comme accessibles, puis de balayer le tas en désallouant les blocs qui ne sont pas marqués (voir section 1.2.2 pour plus de détails).

Le ramasse-miettes d'OCaml utilise quatre couleurs pour colorier les cellules du tas. Chaque couleur a une signification bien particulière dans une des deux étapes.

Phase de marquage Le principe de base de cette phase est de reconnaître à un instant donné l'état de l'ensemble des objets vivants. Pour cela, on va utiliser deux couleurs :

- on marque en **gris** tous les objets que l'on est en train de parcourir, mais dont tous les fils n'ont pas encore été tous parcourus. On ajoute cet objet dans une pile `gray_vals` pour accélérer cette phase.
- on marque en **noir** tous les objets que l'on a parcouru et dont tous les fils ont également été parcourus.

À la fin de cette phase, tous les objets doivent être soit blancs, soit noirs.

Tas Pur On dit que le tas est "pur", si tous les objets marqués en gris qui sont avant le pointeur `markhp` sont également dans la pile `gray_vals`. Si jamais il y a dépassement de la pile `gray_vals`, alors tous les objets gris ne seront pas sur cette pile ; on dit alors que le tas est "impur" et une deuxième phase de marquage est alors nécessaire pour scanner le tas entièrement.

On peut débuter la deuxième phase de l'algorithme (le balayage) une fois que tous les objets gris sont coloriés en noir (une fois que la pile `gray_vals` est vide).

Phase de balayage Le principe de cette phase est de libérer les objets morts marqués dans la phase précédente. Durant cette phase, on utilisera les couleurs bleu et blanc. Au début de cette phase, il n'existe plus aucun objet gris dans le tas.
Dans cette phase, nous allons utiliser essentiellement deux couleurs :

- on va noter en **blanc** tous les objets noirs pour les préparer au prochain scan du tas majeur
- on va noter en bleu tous les objets blancs (donc non visités) et les rajouter à notre `freelist`. Les objets déjà marqués en bleu n'ont pas de traitement particulier car ce sont déjà des cellules vides prêtes à contenir de nouvelles valeurs.

Après le balayage, il n'y a alors plus aucun objet gris ou noir.
Lorsque l'on alloue depuis la `freelist`, les objets bleus deviennent blancs si cela se produit durant l'étape de marquage et que la nouvelle adresse a déjà été balayée. Si jamais, cela arrive durant la phase de marquage, on marque en gris/noir l'objet que l'on vient d'allouer.

2.5.4.3 La compaction

La compaction procède en plusieurs étapes :

- $1^{ère}$ passe : encodage des en-têtes non infixes
- 2^{nde} passe : inversion des pointeurs pour former une liste
- $3^{ème}$ passe : choix des nouvelles positions et mise à jour des racines
- $4^{ème}$ passe : déplacement des objets
- $5^{ème}$ passe : désallocation du tas

Encodage des en-têtes pour la compaction Pendant la compaction, les en-têtes sont encodés sous une forme différente (`wosize|tag|color` au lieu de `wosize|color|tag`), pendant la $1^{ère}$ phase. Après cette phase, les seuls en-têtes non encodés sont les en-têtes en position infixe, qui seront encodés dans la 2^{nde} phase s'ils sont atteignables.

$1^{ère}$ PASSE : ENCODAGE DES EN-TÊTES NON INFIXES On parcourt le tas, chunk par chunk, et pour chaque bloc, on transforme son en-tête de la représentation habituelle en la représentation propre au GC (i.e. `wosize|tag|color` au lieu de `wosize|color|tag`, avec 3 comme valeur par défaut pour la couleur). On peut voir cette représentation sur la figure 2.21.

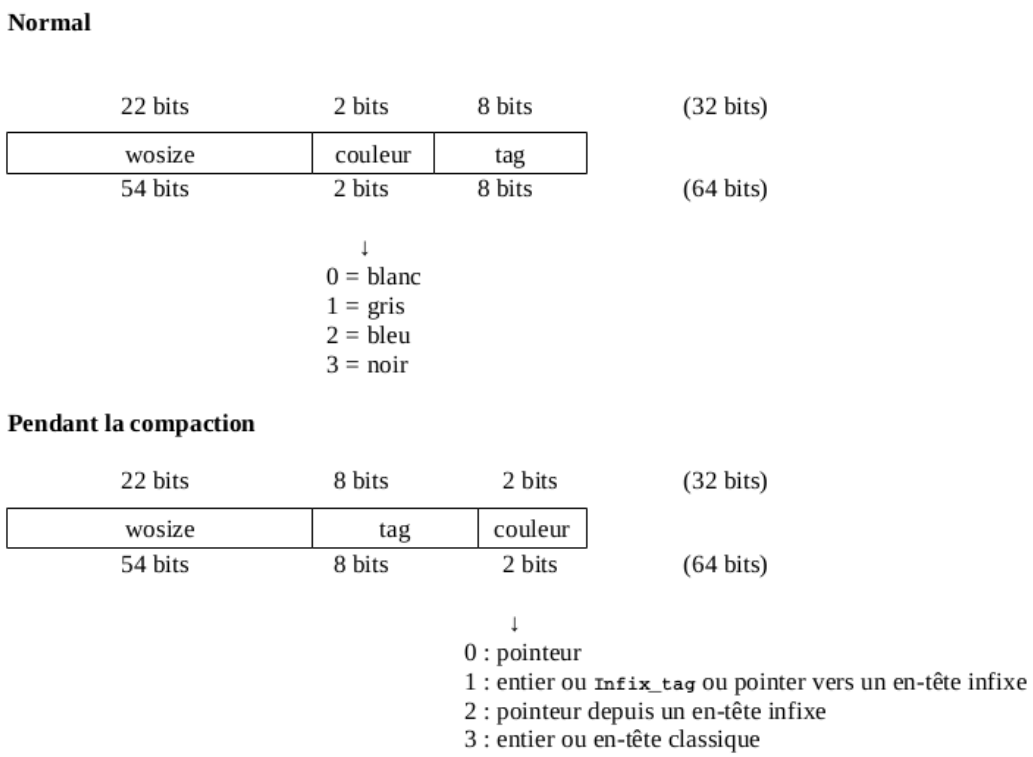

Figure 2.21 – Format des en-têtes pendant la compaction.

Les en-têtes infixes ne sont pas modifiés, car impossibles à détecter facilement.

Les en-têtes des blocs libres (couleur bleue) sont remplacés par des en-têtes de blocs `String_tag` pour indiquer qu'il ne faut pas les scanner (ils seront de toute façon détectés plus tard comme non atteignables et donc supprimés). On garde le même tag sinon.

2^{nde} PASSE : INVERSION DES POINTEURS POUR FORMER UNE LISTE On inverse tous les pointeurs : l'objectif est que l'en-tête de chaque bloc pointe sur une liste dont les éléments sont les pointeurs vers ce bloc (soit directement vers l'en-tête, soit vers des en-têtes infixes).

Pour cela, on commence par appeler la fonction `invert_pointer_at` sur toutes les racines car la librairie de threads a besoin de structures de données du tas pour trou-

ver ses racines, puis sur chaque pointeur de bloc de chaque chunk (*i.e.* en dessous de `No_scan_tag`).

Comme sur la figure 2.22, quand on suit un pointeur dans le tas, celui-ci pointe soit sur un en-tête de bloc, soit sur un en-tête infixe. Un en-tête de bloc est soit présent (avec une couleur 3), soit pointe vers la liste des racines de ce bloc (liste qui se termine par l'en-tête avec la couleur 3). Un en-tête infixe est soit présent (avec la couleur 1, puisque son tag 249 n'a pas été changé), soit pointe vers la liste des pointeurs vers ce pointeur infixe (avec la couleur 2).

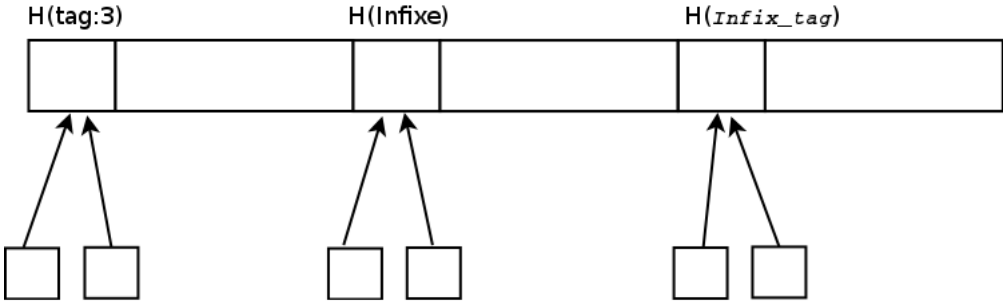

Figure 2.22 – Compaction des blocs initiaux.

Cas d'un bloc avec uniquement un en-tête non-infixe À la fin, l'en-tête pointe vers une liste faite des racines qui pointaient originalement vers ce bloc. Le dernier élément de la liste contient l'en-tête original encodé pour la compaction (figure 2.23).

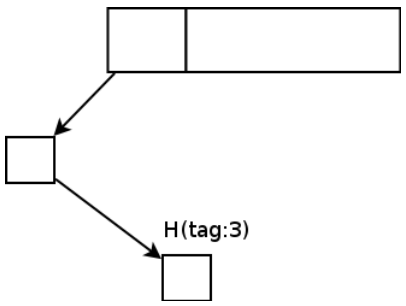

Figure 2.23 – Compaction des blocs non infixes.

Cas d'un bloc avec un en-tête infixe Sur la figure 2.24 Les racines pointant vers l'en-tête non-infixe forment une liste partant de l'en-tête non-infixe. Cette liste ne se termine pas par l'en-tête original, mais par un en-tête portant le tag `Infix_tag`, et

indiquant comme taille la position de l'en-tête infixe.

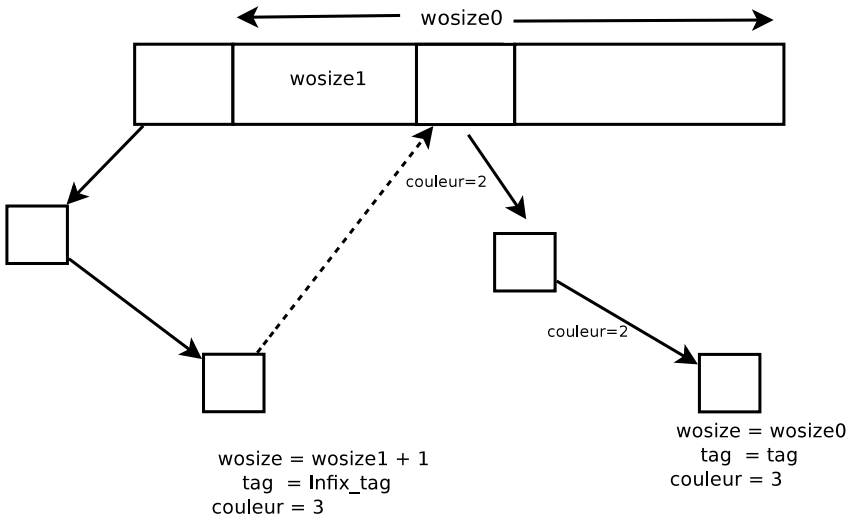

Figure 2.24 – Compaction des blocs avec un en-tête infixe.

Cas d'un bloc avec plusieurs en-têtes infixes Sur la figure 2.25, on peut voir l'encodage d'un bloc avec plusieurs en-têtes infixes.

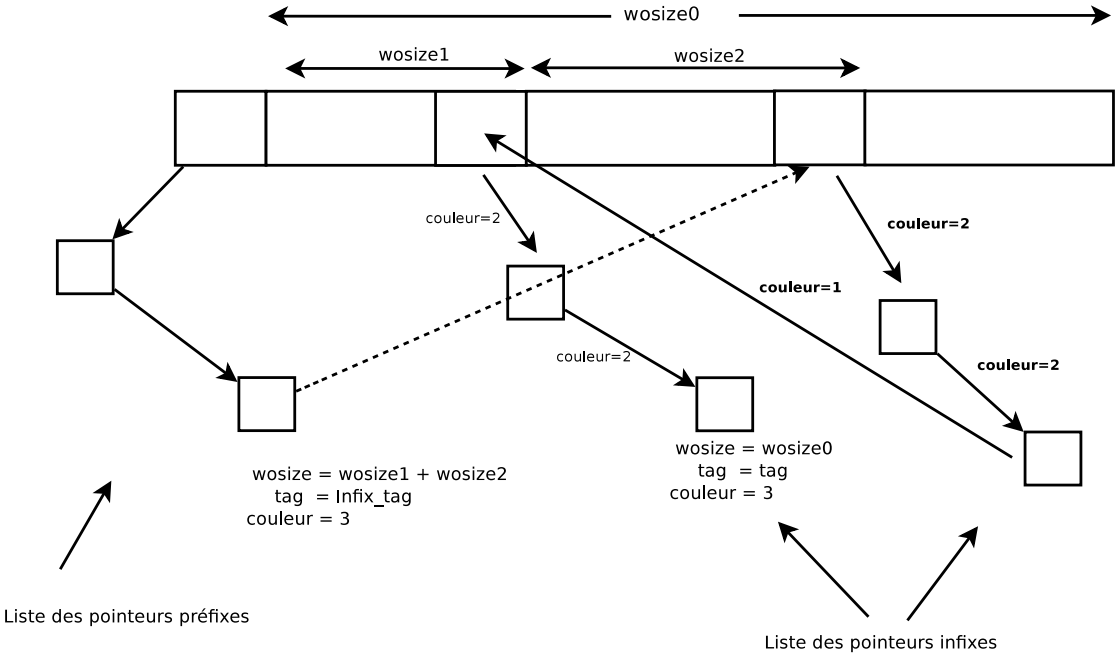

Figure 2.25 – Compaction des blocs avec plusieurs en-têtes infixes.

$3^{ème}$ PASSE : MISE À JOUR DES RACINES Cette passe fait une réallocation virtuelle, puis un inversement des pointeurs et finit par un décodage des en-têtes.

On parcourt le tas, et pour chaque bloc vivant (i.e. dont la liste des racines n'est pas vide), on choisit une nouvelle adresse, et on met à jour toutes les racines de la liste. On ne copie pas encore l'objet à sa position finale, et on ne sauvegarde pas la nouvelle adresse, celle-ci sera recalculée pendant la $4^{ème}$ phase.

$4^{ème}$ PASSE : DÉPLACEMENT DES OBJETS C'est durant cette passe qu'il y a une vraie réallocation et des déplacements d'objets. L'algorithme utilisé est le même que celui de la $3^{ème}$ passe.

$5^{ème}$ PASSE : MISE À JOUR DES RACINES On met à jour les racines et on réduit le tas si cela est nécessaire et on finit par reconstruire la nouvelle freelist.

Pour conclure, par contraste avec Java, par exemple, le typage statique fort d'OCaml permet à sa bibliothèque d'exécution de ne pas nécessiter d'informations de type à l'exécution. Par conséquent, la représentation mémoire des valeurs OCaml est très compacte, ce qui résulte en de meilleures performances. En revanche, lorsqu'il faut profiler une application OCaml, l'absence quasi-complète d'information de type à l'exécution, devient un problème pour retrouver des informations sur les valeurs utilisant la mémoire dans le tas. C'est l'un des principaux obstacles que nous avons essayé de surmonter dans cette thèse.

“La certitude est une région profonde où la pensée ne se maintient que par l’action.“

Richard Nixon

3

La gestion mémoire par régions

3.1 Définitions et rappels de la gestion mémoire par régions

Dans le monde des langages de programmation, il existe différents types de gestion de mémoire :

- automatique et dynamique, à l’aide de ramasse-miettes (*Java, OCaml*, etc.) vu de façon détaillée précédemment.
- manuelle, avec des manipulations explicites de la mémoire avec `malloc/free` par exemple (*C*)
- automatique et statique, à l’aide des régions, en groupant les objets de même durée de vie dans des régions pour qu’ils soient désalloués d’un seul bloc pour plus d’efficacité

La gestion mémoire par région est un compromis entre la gestion mémoire manuelle et automatique. Chaque objet alloué est assigné à une région.

Une région représente une zone en mémoire contenant alors un ensemble d’objets alloués pouvant être désallouée efficacement en une seule opération. Comme l’allocation par pile, les régions facilitent l’allocation et la désallocation de la mémoire à moindre coût, la différence étant une meilleure souplesse permettant aux objets de rester plus longtemps que la pile d’activation. Les régions sont allouées dans une plage contiguë

d'adresses mémoires.

Ce type de gestion mémoire n'est pas concurrent des ramasse-miettes, mais complémentaire, en faisant bénéficier la gestion mémoire d'informations en provenance des programmes, souvent calculées par analyse statique, au moment de la compilation. Pour les objets gérés par les régions, il n'y a alors plus d'allocation ou de désallocation dans un tas, mais :

création de région, une zone contenant un ensemble d'objets destinés à être désalloués en même temps, et qui peut être
- ou bien un vrai bloc, une zone de mémoire contiguë,
- ou alors un ensemble de zones qui, ensemble, forment une même région ;

allocation dans une région : alloue les objets côte à côte dans la région,

destruction de région : libération d'une région entière.

La sémantique statique et dynamique des langages de haut niveau comme OCaml ou Java les rend aptes à être soumis à des analyses statiques diverses. Pour ces langages, il est donc naturel de se poser la question de savoir si des analyses statiques pouvaient être conçues pour détecter les fuites mémoire. L'analyse de régions, initialement proposée par Talpin et Tofte [85, 86], cherche justement à approximer statiquement la durée de vie des objets manipulés par les programmes.

Dans le ce chapitre, nous présentons l'utilisation des régions en Java, C et SML pour la gestion automatique de la mémoire. Ensuite, nous présentons notre étude qui consiste en la mise en œuvre d'une analyse de régions pour OCaml, par l'introduction de cette analyse sur l'un des langages intermédiaires utilisés par le compilateur, et l'implantation d'un *plugin* pour *Typerex* qui fournit une interface graphique à la visualisation des résultats.

3.2 Implantation des régions en C avec *Cyclone*

Cyclone [44] est un langage de programmation, dérivé du *C*, corrigeant les défauts inhérents à ce dernier notamment concernant l'arithmétique des pointeurs très libérale de C entraînant, par exemple, des *buffer overflow*. La plupart de ces problèmes liés aux pointeurs, sont dus à des erreurs de programmation.

Ainsi, le but premier de Cyclone est de garder les avantages du langage C, c'est-à-dire une gestion mémoire explicite et une programmation bas niveau, sans pour autant sacrifier le typage.

Cyclone insère des annotations de régions inférées, mais pour supporter la compilation séparée, le programmeur aura à en ajouter manuellement. Le surcoût remarqué, au niveau de la taille du code, est d'environ 8%, dont seulement 6% représentent des annotations de régions et 2% des changements dans le code source pour l'adaptation aux régions (déclaration de variables, *etc.*).

Le système est :

- sûr : les programmes compilés avec Cyclone ne peuvent plus déréférencer des *dangling pointers* (référence qui ne pointe pas vers un objet valide en mémoire).
- statiquement vérifié, les erreurs de ce type sont détectées durant la compilation.
- facile d'accès, les annotations explicites sont minimisées
- transparent, le programmeur choisit la région ainsi que la durée de vie de l'objet qu'il alloue
- uniforme, toutes les sortes de mémoires sont traitées de manière uniforme, que ce soit la pile, le tas ou les régions.
- modulaire, le système supporte la compilation séparée, en effet les analyses sont intra-procédurales.

Sur la figure 3.1, on peut voir un exemple d'un programme permettant de calculer la factorielle. À gauche le code de la factorielle en C sans annotations de régions et à droite le même code compilé avec Cyclone, avec les annotations de régions.

Lors de la déclaration de la fonction `fact`, la région ρ en argument est générique et abstraite. Le premier argument `result` est un pointeur dans la région ρ.

Une fois exécuté, le programme renvoie 720. Dans la fonction `fact`, ρ est instancié avec la région ρ_H. Dans l'appel récursif, ρ est instancié avec ρ_L représentant la région pour la pile. Le premier appel à la fonction `fact` modifiera `g` et ensuite, chaque appel récursif modifiera la valeur de `x` dans la structure de pile de l'appelant.

3.3 Implantation des régions en Java avec *RTSJ*

Dans les travaux de Cherem et Regina [24], l'analyse de régions consiste à prendre un programme Java en entrée et à produire un programme, sémantiquement équivalent, avec des annotations explicites de région (création, destruction de régions et allocation d'objets).

Les avantages sont divers, comme par exemple une amélioration de la localisation des données permettant de réduire la durée de vie des objets et ainsi limiter leurs occupations

Sans les annotations de régions :

```
1  void fact(int* result, int n) {
2    int x = 1;
3    if(n > 1) fact(&x, n-1);
4    *result = x * n;
5  }
6
7  int g = 0;
8
9  int main() {
10   fact(&g, 6);
11   return g;
12 }
```

Avec les annotations de régions :

```
1  void fact<ρ>(int*ρ result, int n)
       {
2    int x = 1;
3    if(n > 1) fact<ρ_L>(&x, n-1);
4    *result = x * n;
5  }
6
7  int g = 0;
8
9  int main() {
10   fact<H>(&g, 6);
11   return g;
12 }
```

Figure 3.1 – Exemple d'un programme Cyclone, permettant de calculer la factorielle, avant et après les annotations de régions.

en mémoire. Il peut également y avoir une amélioration des performances, car on a une désallocation de régions entières, au lieu d'avoir une désallocation d'objets un par un. Enfin, grâce à des analyses statiques permettant d'annoter le programme source, ce système fournit une gestion de mémoire automatique.

Une implantation concrète a été réalisée en Java avec RTSJ [71] (**R**eal **T**ime **S**pecification for **J**ava), un langage temps réel utilisant la gestion de mémoire en régions.

3.3.1 Spécificité de l'approche

Par comparaison avec l'approche initiale de Tofte et Talpin, deux différences importantes apparaissent :

1. la portée des régions n'est pas lexicale.
2. il peut y avoir des pointeurs invalides (*dangling pointers*), aucune vérification n'est faite lors de la suppression d'une région sur la vivacité de ses pointeurs. Ceci est dû au fait que les analyses statiques calculent les endroits exacts où ces pointeurs ne seront plus jamais utilisés, ceux-ci ne pourront donc pas causer d'erreurs à l'exécution.

3.3.2 L'algorithme utilisé

L'algorithme se décompose globalement en plusieurs phases :

Analyse *points-to :* *(Quels objets ce pointeur peut-il référencer ?)*
Dans cette phase, le compilateur effectue une analyse sensible au contexte, mais pas au flot (*flow-insensitive* et *context-sensitive*), pour partitionner la mémoire en régions. Pour chaque méthode, il construit un graphe de pointeurs de régions capturant les effets de l'exécution de la méthode et de toutes les méthodes qu'il invoque transitivement.

Graphe de pointeurs de régions pour chaque méthode.
Les nœuds représentent les régions et les arêtes sont les relations entre les pointeurs.
Ensuite, une analyse intra-procédurale est effectuée sur ce graphe. Puis, un algorithme standard d'analyse de pointeurs non sensible au flot avec unification de contraintes est utilisé [82].

Analyse de vivacité : le compilateur effectue une analyse sensible au flot et calcule les régions vivantes à chaque point du programme.
L'analyse garde à la fois une trace des variables et des régions vivantes, et utilise le graphe précédemment calculé pour déterminer l'atteignabilité des régions par le biais des variables vivantes.
Une région est vivante si :

1. la région est atteignable par au moins une variable vivante à cet endroit du programme
   ET
2. l'exécution du programme peut faire un accès (lecture, écriture ou allocation d'objets) dans cette région par la suite.

La difficulté pour Java est au niveau des méthodes virtuelles, pouvant invoquer dynamiquement différentes méthodes durant l'exécution du programme, avec des threads, car il est difficile d'identifier la durée de vie des objets partagés. Les exceptions compliquent également le contrôle du flot.

3.3.3 Exemple

Sur la figure 3.2, on observe un programme Java avant et après les annotations de régions. L'objet `Complex tmp` est alloué dans la région 11 pour la multiplication et sera supprimé seulement à la fin de l'itération, après avoir été utilisé pour la somme. Celle-ci sera faite dans la région 9. On remarque que la région 9 n'est pas supprimée à la fin de l'itération. étant utilisée à chaque tour, car elle contient l'objet `sum` qui est l'accumulateur.

Avant l'inférence de région :

```
[...]
Iterator it = rev.iterator();
while (!it.empty()) {
  Complex coeff =
          (Complex)it.next();
  Complex tmp = sum.mul(x);
  sum = tmp.plus(coeff);
}
[...]

public Complex plus(Complex c) {
   double r = this.re + c.re;
   double i = this.im * c.im;
   return new Complex(r, c);
}
```

Après l'inférence de région :

```
[...]
Iterator it = rev.iterator<r10>();
while (!it.empty()) {
  Complex coeff = (Complex)it.next();
  create r11;
  Complex tmp = sum.mul<r11>(x);
  remove r9;
  create r9;
  sum = tmp.plus<r9>(coeff);
  remove r11;
}
remove r10;
[...]

public Complex plus<r13>(Complex c) {
   double r = this.re + c.re;
   double i = this.im * c.im;
   return new Complex(r, c) in r13;
}
```

Figure 3.2 – Exemple d'inférence de régions sur un programme en Java compilé avec RTSJ

3.4 Implantation des régions en ML

Pour ML, nous avons étudié l'inférence de régions introduite par Tofte et Talpin [85, 86]. Cette gestion mémoire reste un bon compromis entre la gestion manuelle et automatique de la mémoire. En effet, les régions étant inférées par le système, le compilateur ajoute dans le code les instructions d'allocation et de désallocation des blocs mémoire du programme. Dans ce système, chaque valeur est affectée à une région, qui n'est qu'une collection de blocs alloués, et qui sera récupérée entièrement quand l'exécution aura dépassé la portée de la région. Par rapport à Cyclone, l'ordre supérieur rend l'inférence beaucoup plus complexe.

Totfte et Birkedal [84, 86] expliquent que dans certains cas, il est même possible de prouver l'absence de fuites mémoire.

Prenons l'exemple d'une fonction, dont l'inférence de région renvoie le type suivant :

$$\forall \overrightarrow{\rho}\,\overrightarrow{\alpha}\,\overrightarrow{\epsilon}.\mu_1 \xrightarrow{\{put(\rho)\}} \mu_2$$

Ici, $\overrightarrow{\rho}$ représente l'ensemble des régions initiales, ρ représente une variable de région et $\overrightarrow{\epsilon}$ l'ensemble des effets initiaux. Un effet peut être un `put(`ρ`)` ou un `get(`ρ`)`.

Dans l'exemple, si dans les effets, nous n'avons que l'effet atomique `put(`ρ`)` et si ρ n'apparaît pas dans l'ensemble des $\overrightarrow{\rho}$ initial, alors il y a un risque de fuite. Chaque appel à la fonction risque d'ajouter des objets dans une région différente de celle de la fonction. De plus, si ρ ne fait pas partie du résultat μ_2, alors les objets ajoutés ne sont pas retournés, une raison de plus pour s'alerter. En d'autres termes, la fonction fait un effet de bord allouant des objets dans une autre zone mémoire.

Concrètement, si nous prenons un exemple, le code suivant :

```
let x = (2, 3) in
  (fun y -> (fst x, y)) 5
```

sera transformé en :

```
letregion ρ4, ρ5 in
letregion ρ6 in
let x = (2 at ρ2, 3 at ρ6) at ρ4 in
  (λy.(fst x, y) at ρ1) at ρ5
  5 at ρ3
```

3.5 Inférence de régions en OCaml pour calculer la durée de vie des valeurs

Pour vérifier cette hypothèse, nous avons modifié le générateur de code du compilateur OCaml pour inférer les régions et ainsi annoter les valeurs avec celles-ci. Ces annotations nous ont alors permis de comprendre l'interaction des valeurs entre elles (*i.e.* stockées dans la même région). Cela nous a permis d'avoir une simplification de l'analyse de la durée de vie des valeurs, basée sur les régions, chaque valeur étant affectée à une région. Cela peut permettre de tracer de façon simple les valeurs non désirées dans une région.

Nous avons écrit un prototype sous forme de plugin pour *Typerex* [63]. Sur la figure 3.3, nous pouvons voir les résultats dans l'éditeur de code `emacs` pour la visualisation de ces régions en surlignant les valeurs d'une même région de la même couleur. Dans certains cas, cela facilite la détection des valeurs non désirées dans certaines régions [21].

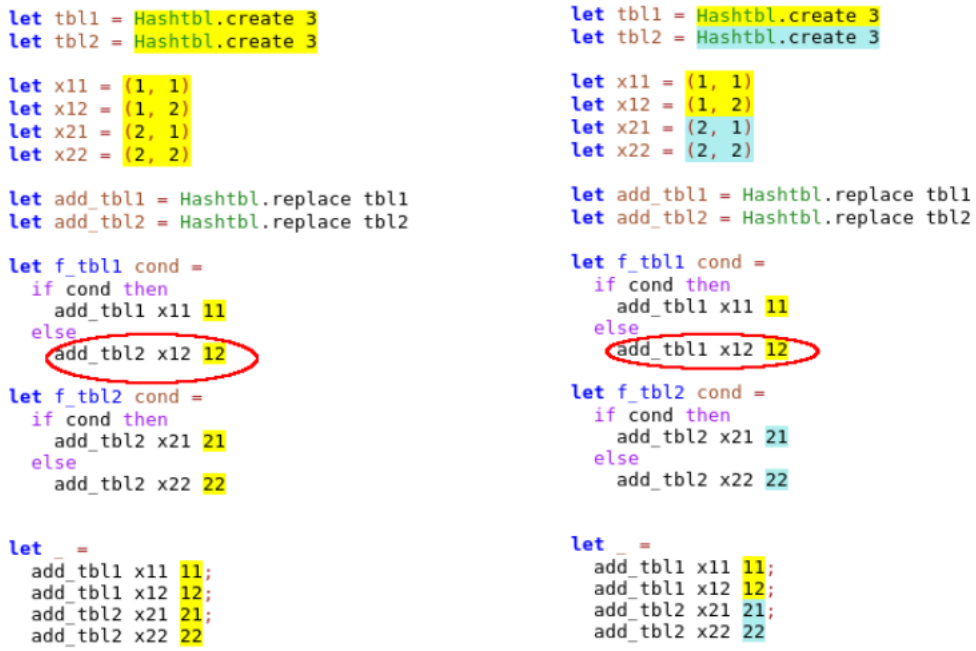

Figure 3.3 – L'inférence de régions peut permettre de détecter des bugs. Ici, notre inférence de régions sur OCaml, affichée sous forme de plugin dans Typerex, montre une erreur à gauche, détectée car les deux tables `tbl1` et `tbl2` se retrouvent dans la même région.

Il est apparu que les résultats de ce type d'analyse étaient mitigés. En effet, dans certains cas, l'analyse classait une grande majorité les valeurs dans la même région, n'apportant donc pas ou peu d'information statique pertinente sur le comportement dynamique du programme. Nous avons donc dû nous résoudre à abandonner la piste des analyses statiques à base de régions pour explorer la collecte dynamique d'information.

"Le débogage est deux fois plus dur que d'écrire le code initial. Par conséquent, si vous écrivez le code aussi intelligemment que possible, vous êtes, par définition, trop peu intelligent pour le déboguer."

Brian W. Kernighan

4

Introduction aux outils de profiling

Les langages à gestion automatique de la mémoire sont de plus en plus communs : JAVA, OCaml, Haskell, C#, etc., Cette gestion mémoire offre certes au programmeur des garanties de fiabilité, mais, en éloignant ce dernier des détails de la gestion de la mémoire, elle rend plus difficile la compréhension et, *a fortiori*, la maîtrise des allocations de mémoire.

Les utilisateurs de ces langages à ramasse-miettes ont progressivement découvert de nouveaux types de problèmes liés à cette gestion automatique de la mémoire. Les fuites mémoires, *i.e.* la non libération de la mémoire n'étant plus utile à l'exécution du programme, font partie de ce genre de problèmes très compliqués à identifier et localiser, et donc à corriger. En effet, les blocs mémoire restant atteignables du point de vue du ramasse-miettes, celui-ci est incapable de libérer la mémoire. Ce type de problèmes se traduit souvent par le fait que le programme continue de demander de la mémoire : au mieux, les performances de l'application seront dégradées et dans le pire cas, l'application n'a plus assez de mémoire pour continuer à s'exécuter, atteint une limite et se termine par une erreur (manque de mémoire). Sur des applications dont l'exécution est longue (serveurs, par exemple), ces erreurs peuvent arriver après des jours ou des semaines d'exécution. C'est une des raisons pour lesquelles il est très difficile de les corriger.

Aujourd'hui il n'existe pas d'outil pour détecter de façon *sûre* et automatique ce type

de problème mémoire. Néanmoins, on peut tenter d'aider les développeurs en faisant quelques analyses qui peuvent parfois être très coûteuses, à cause de grandes quantités de données à traiter (cela peut atteindre plusieurs gigaoctets alloués et désalloués en très peu de temps [55]).

La recherche de techniques de collecte dynamique d'informations, appelée *profiling*, nous a alors amené à étudier les différents outils de profiling existants que nous présentons dans ce chapitre. Selon le langage de programmation et la représentation mémoire qu'il utilise, certaines informations peuvent être obtenues très facilement. En Java, par exemple, les informations de classes sont encodées dans les en-têtes des objets, ce qui facilite grandement la compréhension des résultats obtenus par ces outils, et le lien avec le code source. On comprend alors que le manque d'informations dynamiques de types en OCaml devient un réel problème pour le profiling, puisque des valeurs avec la même forme peuvent être de types très différents et correspondre à des valeurs n'ayant rien en commun.

Dans le monde Java, il existe des outils permettant de comprendre les types, les instances et l'utilisation de la mémoire, mais ne donnant aucune information concrète au programmeur sur la localisation de la fuite mémoire (JProfiler [40] et JProbe [79] par exemple).

On retrouve également des logiciels basés sur des heuristiques tentant de détecter certaines fuites mémoire, comme **LeakBot** [58] et **Cork** [50] dont l'heuristique est basée sur l'augmentation de la taille mémoire. Le problème de ces deux outils est la présence de faux-positifs : le fait que la mémoire augmente ne signifie pas forcément qu'il y a une fuite mémoire.

Enfin, il y a également des outils comme **Sleigh** [20] dont l'heuristique est basée sur l'état d'utilisation (caducité) des objets (temps écoulé depuis la dernière utilisation). Encore une fois, il y a des faux-positifs : par exemple, une *frame* dans Java Swing peut ne jamais être utilisée, mais ne peut être considérée comme une fuite.

4.1 Outils de profiling pour la performance / optimisation de la mémoire

Certains outils de profiling permettent de mieux comprendre le comportement mémoire d'une application donnée et aide à les améliorer, que ce soit pour optimiser la vitesse d'exécution ou l'espace mémoire.

Chis *et al.* [25] proposent comme approche de définir un certain nombre de modèles servant à montrer des motifs d'inefficacité de la gestion mémoire. Un des buts est d'évaluer la rentabilité du travail à fournir pour optimiser le code. Sur la figure 4.1, on peut noter les *instantanés* du tas sur lesquels on applique un certain nombre d'analyses pour trouver des motifs dont on peut voir quelques-uns sur la figure 4.2. Ces motifs sont implantés comme des greffons pour un algorithme de parcours de graphe.

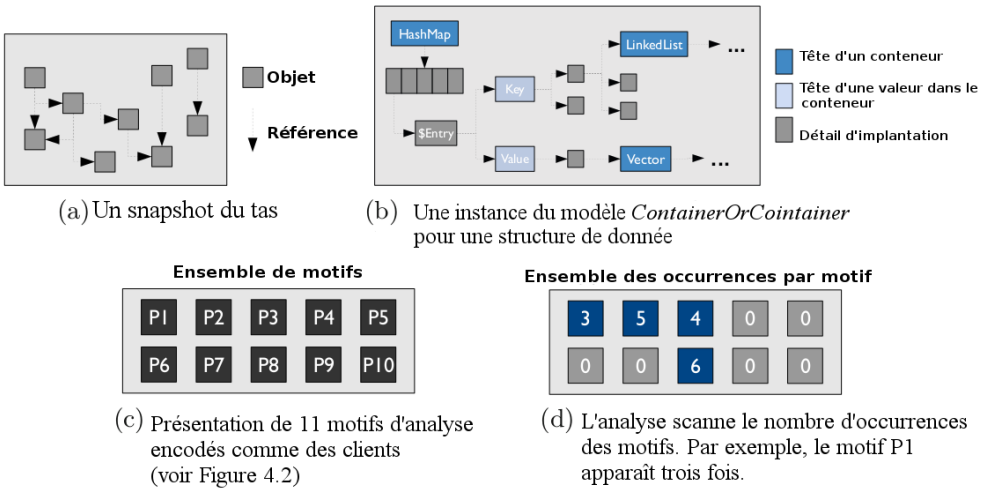

Figure 4.1 – Vérification par motifs (figure traduit de Chis et al. [25]).

Mesurer la taille de la mémoire seule ou de chaque structure de données ne suffit pas ; le programmeur a aussi besoin d'évaluer rapidement si une inspection du code plus approfondie sera un investissement rentable en temps.

Cette approche est plus dans une logique d'optimisation que de recherche de fuites mémoire. Le point intéressant est de définir des motifs qui reviennent souvent, pour un langage donné, pour détecter des inefficacités dans la mémoire.

Dans notre approche, que nous détaillerons dans la suite, la même méthode pourrait être appliquée aux instantanés de la mémoire pour OCaml. Ces mêmes motifs pourraient être implantés comme des greffons qu'on pourrait utiliser sur un algorithme de parcours de graphe de la mémoire OCaml.

Memory Pattern	Identifier
Empty collections	P1
Fixed-size collections	P2
Small collections	P3
Sparsely populated collections	P4
Small primitive arrays	P5
Boxed scalar collections	P6
Wrapped collections	P7
Highly delegated structures	P8
Nested collections	P9
Sparse references	P10
Primitive array wrappers	P11

Figure 4.2 – Les motifs prédéfinis pour Java.

4.2 Profiler l'occupation mémoire

Certains outils de profiling permettent de visualiser et de comprendre l'occupation mémoire d'une application. Ces outils permettent de mieux comprendre les éléments en mémoire et font des hypothèses sur l'occupation en mémoire de ces valeurs. Parfois ils peuvent donner des informations intéressantes concernant des fuites mémoire.

4.2.1 *LeakBot*, profiler l'augmentation mémoire

LeakBot [58] est un outil de profiling pour le langage Java, tentant de détecter les fuites mémoire de façon automatique. Partant du constat que malgré la gestion automatique de la mémoire en Java, les fuites mémoire restent un problème important, donnant suite à des crashes fréquents de serveurs en production. La complexité du code, avec l'utilisation de différentes librairies, ainsi que sa taille, rendent très difficile l'utilisation des outils existants, demandent beaucoup d'expertise de la part des développeurs pour tenter de détecter la source du problème et les surcoûts de ces outils les rendent parfois inutilisables en production.

Cet outil intègre des nouvelles techniques afin d'augmenter les chances de trouver ces fuites mémoire de façon automatique.

L'idée de base est de changer le niveau d'analyse en ne se concentrant plus sur chaque objet, mais sur des zones dans des structures qui fuient potentiellement. D'abord il classe les structures de données par leur probabilité de contenir une fuite mémoire.

Cette étape élimine une grande quantité de structures à analyser. Ensuite, il identifie les zones suspectes dans les structures de données et décrit leur évolution attendue et enfin il compare l'évolution attendue avec la vraie évolution durant l'exécution du programme. Après cette dernière étape, on reclasse les structures par rapport aux résultats obtenus.

L'avantage est qu'on se concentre seulement sur les structures qui fuient potentiellement et on les reclasse, au fur et à mesure, avec les résultats obtenus grâce à l'exécution du programme. Toutes ces étapes sont faites de façon automatique sans intervention du programmeur.

Pour classer et trouver les structures de données qui fuient potentiellement, *LeakBot* introduit la notion de *leak root*, différente des racines du programme, représentant un nœud dans le graphe, étant un parent d'une zone critique identifiée. Pour cela, il se base sur des propriétés structurelles et temporelles. Le but est de diminuer le nombre de ces racines pour mettre en valeur seulement les zones vraiment critiques.

Conclusion

L'augmentation de la mémoire n'étant pas forcément synonyme de fuite mémoire, il peut y avoir beaucoup de faux-positifs et également de faux-négatifs, c'est-à-dire des fuites mémoires qui ne seront pas détectées par l'outil. De plus, le fait de ne pouvoir annoter ni identifier certaines structures manuellement, peut également être un point négatif.

4.2.2 *Cork*, profiler l'augmentation mémoire à moindre coût

Cork [50] est un outil de profiling pour le langage Java, avec un faible surcoût pour tenter de détecter les fuites mémoires. Pour cela, après chaque collection majeure sur la vieille génération (voir 2.3.3.3), l'outil construit un rapport avec tous les points du graphe mémoire où le volume en taille a augmenté. Par exemple, entre deux collections, Cork va identifier les structures de données qui auront augmenté en taille mémoire et les mettra en valeur dans le rapport émis.

Cork ajoute des *hooks* sur le ramasse-miettes permettant de traquer facilement les objets alloués en mémoire.

Une fois les structures de données critiques repérées par Cork, leur site d'allocation est ajouté au rapport pour que le debug soit plus simple et plus efficace pour l'utilisateur.

Le surcoût de l'application en temps d'exécution est minimal, entre 1,7% et 2.4% en moyenne et pouvant aller jusque 12.9% pour les gros tas [50]. Il est important de ne pas modifier le comportement de l'application pour rester le plus proche d'une exécution réelle. Cork pourra alors être utilisé sur une application en production.

Ce surcoût correspond à la durée de pause du ramasse-miettes. Il est généralement dû au calcul du graphe mettant en avant les structures de données ayant augmenté.

Conclusion

Le manque de précision et le fait de n'utiliser qu'un seul facteur (l'augmentation mémoire) ne suffit pas pour mettre en avant une fuite mémoire. En effet, l'augmentation mémoire en général ou d'une structure de données en particulier ne signifie pas forcément qu'il y a une fuite mémoire.

De plus, Cork ne faisant pas une analyse par objet, mais examinant tout le graphe mémoire (tout le tas), il ne vérifie pas si les structures de données qui augmentent sont entièrement utilisées par le programme ou pas. Dans le cas où une structure est vraiment utile et augmente de façon attendue, Cork ne fera pas la différence avec une structure augmentant, mais qui ne sera jamais utilisée.

4.3 Outils de profiling sur la durée de vie des objets

Certains outils de profiling se focalisent sur la durée de vie des objets en mémoire. En faisant cette hypothèse, ces outils tentent de trouver les objets en mémoire les moins utilisés ou de détecter ceux qui ne le sont plus depuis une certaine durée.

4.3.1 *Sleigh*, détecter les fuites mémoire

Sleigh [20] est un outil de profiling tentant de détecter des fuites mémoire, en se basant sur la date de dernière utilisation d'un objet (*staleness* en anglais). Si un objet est inutilisé pendant une longue durée, il sera signalé lors du rapport émis.

De plus, les objets en mémoire sont encodés d'une façon astucieuse de manière à n'utiliser qu'un bit dans l'en-tête pour stocker l'information sur le site d'allocation, ainsi que son site de dernière utilisation. Une fois les objets suspicieux en mémoire détectés, il est alors plus aisé de retrouver dans le code source l'endroit exact où ils ont été alloués et l'endroit exact où les objets ont été utilisés. Bond *et al.* détaillent dans [20] cette approche statistique pour encoder les sites d'allocation dans l'en-tête des objets.

Sleigh utilise des emplacements non-utilisés dans l'en-tête de chaque bloc mémoire, il n'y a donc aucun surcoût au niveau de l'espace utilisé. Cependant, il y a un surcoût de 29% à l'exécution dû à l'instrumentation ajoutée par l'outil.

Le code est instrumenté pour chaque allocation, et les en-têtes sont lus pour identifier les objets non utilisés depuis un certain temps. 4 bits seront introduits dans le header de sorte à ajouter :

- 1 bit pour le site d'allocation.
- 1 bit pour le site de dernière utilisation
- 2 bits pour le temps de dernière utilisation.

Sleigh commence par ajouter dans cet espace le site d'allocation ainsi que le champ de dernière utilisation. Si un objet n'est jamais utilisé, alors ces deux champs auront toujours la même valeur.

Les 2 bits pour le temps de dernière utilisation sont mis à 0 à chaque nouvelle allocation (initialisation de l'objet) et à chaque fois que l'objet est utilisé. Ces 2 bits sont incrémentés de `k` à `k + 1` si le GC courant est divisible par $\texttt{b}^k$, où b représente la base du compteur logarithmique (la valeur 4 est utilisé par défaut). Ce compteur utilise une échelle logarithmique, ce qui permet de sauver de l'espace mémoire sans perdre beaucoup d'informations. La saturation pour ce compteur est atteinte à 3, à cause du codage sur deux bits.

Conclusion

Le surcoût en temps d'exécution de 29% en moyenne est un inconvénient non négligeable sur des applications tournant en production.

De plus, *Sleigh* ne supporte par le déplacement d'objets. Il fonctionne avec un ramasse-miettes à balayage sans déplacements d'objets.

Il peut également y avoir quelques imprécisions sur les objets reportés. Le fait de n'utiliser que le temps de dernière utilisation fait que les objets plus gros en mémoire peuvent passer inaperçus entre tous les objets reportés.

Dans le monde Java, il peut y avoir quelques faux-positifs notamment lors de l'utilisation de *frames* dans Java Swing, qui sont alloués une fois en début de programme et peuvent ne pas être utilisés pendant une durée très longue.

Enfin, l'outil utilise l'information de classe à runtime de Java et n'est donc pas facilement applicable à d'autres langages.

4.3.2 Profiler les conteneurs

On a également des approches avec un niveau d'abstraction basé sur les conteneurs [97] et ses opérations. Les auteurs introduisent deux concepts qui vont permettre de décider de l'importance des fuites d'un conteneur :

- quantité de la mémoire utilisée
- la caducité (*staleness*)

L'idée de cette nouvelle approche est de tracer seulement les conteneurs responsables de la plupart des fuites mémoire en Java, au lieu de tracer tous les objets. Tous les conteneurs sont alors suspectés de fuites mémoires. Cette approche peut être complétée par des techniques de suppression automatique de fuites liées aux tableaux [76].

L'idée de base est concentrée sur le fait qu'un conteneur a trois opérations courantes : l'ajout, la suppression et la lecture d'un élément de ce conteneur. À la fin de l'exécution d'un programme, pour une structure donnée, si le nombre d'ajouts est équivalent au nombre de suppressions, alors celle-ci n'est pas considérée comme fuyante.

Cette idée est complétée par un calcul sur la fiabilité et l'importance de la fuite mémoire, basée sur la quantité consommée (MC) et la caducité (SC).

Le MC est la quantité de mémoire que le conteneur consomme durant sa vie. La valeur relative à un temps τ du MC (qui est dans l'intervalle [0, 1]), est le ratio entre la somme de toute la mémoire consommée par tous les objets atteignables par le contenur et le total de mémoire consommé par le programme à τ.

Le SC est la distance entre le moment où un élément est enlevé du conteneur et le moment le plus récent où cet élément est récupéré/ajouté de/à celui-ci.

Les auteurs définissent alors un indice de fiabilité sur la fuite (LC : Leaking Confidence) :

$$\texttt{LC} = \texttt{SC} \times \texttt{MC}^{1-SC} \text{ où } \texttt{LC} \in \texttt{[0, 1]}$$

On peut alors déduire les propriétés suivantes :

- pour `MC=0 et SC` $\in$ `[0, 1]` alors `LC = 0` : si le MC d'un conteneur est assez petit, son LC sera proche de 0 (la caducité ici n'importe pas). Cette propriété permet de filtrer les petits objets comme les chaînes de caractère (`String`) ;
- pour `SC=0 et MC` $\in$ `[0, 1]` alors `LC = 0` : si chaque élément d'un conteneur est retiré dès lors qu'il n'a plus d'utilité, son LC est de 0 quelque soit sa taille ;
- pour `SC=1 et MC` $\in$ `[0, 1]` alors `LC = 1` : si aucun élément d'un conteneur n'est jamais retiré, son LC est de 1 quelque soit la taille du conteneur ;
- `MC=1 et SC` $\in$ `[0, 1]` alors `LC = SC` : si le MC d'un conteneur est très élevé, son LC est décidé par la valeur de son SC.

Conclusion

Pour utiliser cet outil, il est nécessaire d'instrumenter manuellement le code des conteneurs. En effet, une fois passée cette étape, l'outil fera une analyse hors ligne (post-mortem), après que le programme se soit arrêté normalement ou pas.

Un inconvénient de cette approche est qu'elle ne capture qu'une partie des fuites mémoires : celles basées sur les conteneurs. Pour finir, une recherche plus approfondie est nécessaire sur les conteneurs *non mutables*, ainsi que sur les tableaux de pointeurs faibles (*weak arrays*).

Dans le cas d'OCaml, il serait possible de faire la même chose pour certaines structures mutablesIl faudrait pour cela complètement annoter la librairie standard et fournir des nouveaux modules pour ce genre de valeurs.

4.4 Assertions sur le ramasse-miettes

Il existe plusieurs travaux effectués pour Java, visant à introduire des assertions sur le ramasse-miettes. Aftandilian *et al.* [8] est un premier essai d'assertions sur le tas qui propose un ensemble d'assertions à vérifier pendant l'exécution par le ramasse-miettes qui a des informations sur la durée de vie des objets et leurs connexions. Le but est de proposer une interface système pour vérifier des erreurs comme des invariants sur des structures de données et diagnostiquer des fuites mémoires.

Pour cela les assertions sont classées en trois catégories, selon qu'elles concernent :

- la durée de vie
- la connectivité/structure
- le volume d'allocation

Cette approche ne sera pas très efficace avec un ramasse-miettes incrémental : puisque la collection majeure est faite de façon incrémentale, certaines assertions mettront un peu plus de temps à être vérifiées.

Au niveau des performances, le surcoût est très faible. De plus, il n'y aura aucun faux-positif, hormis les erreurs liées au programmeur.

Le gros désavantage est que ce système est dépendant des compétences du programmeur pour trouver où placer les assertions.

Plusieurs outils basés sur cette idée d'assertions existent dans le monde Java. DEAL [68] (**DE**clarative **A**ssertion **L**anguage), est un langage d'assertions qui contient un certain nombre de types d'assertion. La nouveauté ici est que DEAL assure la traversée **unique** du tas pour vérifier une assertion. QVM [12] fournit un riche ensemble de vérifications sur le tas appelées des *heap probes*, vérifiées à l'aide du ramasse-miettes aux endroits où

apparaissent les assertions. PHALANX [89] quant à lui, est une extension de QVM avec comme avantage, l'ajout des vérifications sur l'atteignabilité et la dominance de façon parallèle en utilisant les différents cœurs du système.

Dans la suite, nous allons détailler certains de ces outils.

4.4.1 Assertion sur le tas avec *DEAL*

Contrairement à l'approche d'Aftandilian et al.[8], DEAL formalise un langage générique pour les assertions évaluées efficacement durant le GC. De plus, chaque assertion assure un seul parcours du tas, contrairement à Aftandilian et al. qui peut traverser plusieurs fois le tas pour évaluer une assertion. DEAL est intégré à Java comme une extension de la machine virtuelle.

Le but de DEAL est de profiter de la traversée du GC, pour récupérer un maximum d'informations (pas seulement sur la vivacité).

De plus, il n'y a pas de calculs avec effets de bords, pas de modification du flux de contrôle du programme ou de calcul, que le programme utiliserait plus tard.

Les primitives de DEAL Pour `F` une formule de la logique du premier ordre :

- **assertion(F)** : au plus une relation d'accessibilité peut se produire dans F. De plus, la même relation peut se produire plusieurs fois. Cette primitive vérifie la condition dans l'assertion en ne traversant qu'une seule fois les objets vivants.
- **assertDisjoint(F)** : relations d'accessibilité multiples, mais si deux de ces relations se produisent, alors en plus de vérifier `F`, il faut vérifier que les domaines des deux relations sont disjoints. S'il existe un nœud qui satisfait les deux relations, alors une erreur est détectée. (Si les domaines sont disjoints, F peut être évaluée en une seule passe durant le GC, sinon ceci est détecté durant l'unique passe.)
- **unsafeAssert(F)** : on peut avoir plusieurs relations `R` sans avoir besoin de la disjonction. Dans ce mode, DEAL se comportera comme si les domaines des relations `R` étaient tous disjoints sans faire aucune vérification. Si les domaines ne sont pas disjoints, alors aucune garantie ne peut être donnée sur la détection de ce problème, ni sur la vérification correcte de `F`. De plus, DEAL ne vérifiera pas correctement la formule si la valeur de vérité de `F` dépend d'objets qui satisfont simultanément plusieurs relations `R`. La raison est que DEAL ne traverse de tels objets qu'une fois pendant le traitement d'une des relations `R` et sans savoir si l'objet satisfait d'autres relations `R`. Cependant, F peut exprimer différentes exigences pour un objet selon le `R` qu'il satisfait.

Ces trois primitives tentent de vérifier si la formule est vraie, mais ils ont des exigences et des garanties différentes.

Exemples

- On peut spécifier qu'une structure de donnée doit être cyclique :
  `assertion ("∀x ∈ Node: R`$_C$`(x) ⇒ x.next ≠ null");`
  "Pour tout nœud `x`, si `x` est atteignable depuis une structure de données `C`, alors le successeur de `x` n'est jamais nul."
- On peut spécifier des propriétés sur l'atteignabilité des objets qui satisfont une certaine condition :
  `assertion ("∀x ∈ Node: (x.next = null) ⇒ R`$_{leaves}$`(x)");`
  "Pour tout nœud `x`, si le successeur de `x` est nul, alors `x` est atteignable depuis la structure de données `Leaves`."
- On peut exprimer des propriétés de dominance comme des propriétés de non-atteignabilité si le dominateur est exclu :
  `assertion ("∀x ∈ Node: ¬R/`$_C$`(x) ⇒ x.data > 0");`
  "Pour tout nœud `x`, s'il n'y a aucun chemin depuis le graphe des objets qui permet d'atteindre l'objet `x` sans passer par `C`, alors `x.data` doit être supérieur à 0."
- On peut également combiner les propriétés de dominance : "Aucun chemin depuis le graphe des objets ne doit être capable d'atteindre l'objet `Person` sans passer par la structure de données `males` ou l'objet `females`" :

```
assertion("∀x ∈ Person: ¬R/males,females(x)");
assertDisjoint("∀x ∈ Person: Rmales(x) ∨ Rfemales(x)");

assertDisjoint ("∀x ∈ Person :(Rmales(x) ∨ Rfemales(x))
                              ∧ ¬R/males,females(x)");
```

Modèle d'évaluation

L'assertion devrait être évaluée quand elle est rencontrée au lieu de la retarder jusqu'au prochain GC, car le tas peut changer durant cette attente. Se pose alors le problème avec le fait d'évaluer directement l'assertion : si chaque assertion cause un GC, alors on se retrouve avec un surcoût non négligeable.

DEAL décide s'il faut évaluer une assertion en regardant le temps entre l'assertion et le dernier GC. Si le temps passé est suffisamment long, il évalue l'assertion et déclenche

un GC. Sinon l'assertion est ignorée. L'utilisateur peut par contre choisir la fréquence d'évaluation des assertions.

Conclusion

L'avantage de DEAL par rapport à Aftandilian et al.[8] est qu'il y a une formalisation du langage d'assertions.

Ensuite, le but avec DEAL est de ne faire qu'une seule traversée du tas pour évaluer une assertion.

Un problème non négligeable est qu'avec cette façon d'évaluer, on peut lancer un GC à chaque fois qu'on rencontre une assertion, donc un surcoût très important.
Si on veut éviter ce problème, on peut réduire la fréquence de parcours du tas, par contre on se retrouve alors avec des assertions tout simplement ignorées.

4.4.2 Framework d'assertion sur le tas avec *LeakChaser*

Dans la même idée, LeakChaser [95] est un *framework* d'assertions sur le tas qui autorise le programmeur à affirmer des propriétés de vivacité. Les outils existants, contiennent beaucoup de faux positifs et manquent d'informations sur la sémantique, pour aider le programmeur à déterminer la source du problème.

Utilisant une syntaxe très légère pour annoter les programmes, les rapports générés seront plus pertinents. L'outil peut également fournir des diagnostics liés à la sémantique (quels types d'objets ? quelles transactions ? etc.). Il existe trois niveaux d'abstractions pour aider le programmeur quelque soit son niveau de familiarité avec le code.

Détails de l'approche

L'approche est basée sur deux observations :

1. On veut pouvoir spécifier des invariants implicites sur la durée de vie des objets :

   Ex : "Objet α doit mourir en même temps que l'objet β"
   "L'ancienne instance d'un objet doit être déréférencée avant que la nouvelle ne soit créée "

2. Dans certaines régions de la mémoire, des objets sont fortement liés à la survie de la région entière :

   Ex : Web ou database transaction en J2EE.
   (1) création de l'objet de transaction
   (2) création de tous les objets liés à cette transaction

(3) suppression de ces objets

(4) Suppression de l'objet de transaction

Lorsque ces invariants ne sont pas respectés, on a des fuites mémoires.

Une *transaction* est la borne d'une zone à vérifier. Les objets associés à ces transactions sont divisés en trois catégories :

- identificateur d'objet de la transaction
- objets locaux [1] de la transaction
- objets partagés [2] entre les transactions

Les différents niveaux d'abstraction :

- Niveau H : (plus haut niveau) tente d'inférer les objets locaux et partagés. Le programmeur doit uniquement spécifier la borne de la transaction et l'identificateur de l'objet de transaction. Une fois les objets partagés inférés, il vérifie que ces objets soient bien utilisés dans d'autres transactions. Une erreur est reportée si ces objets ne sont pas utilisés durant un certain temps.
- Niveau M : Le programmeur spécifie la borne et l'identificateur de la transaction, mais également les objets partagés. Une erreur est reportée si des objets locaux d'une transaction sont visibles depuis une autre instance (transaction).
- Niveau L : Ce niveau est principalement un framework d'assertions qui autorise le programmeur à spécifier des invariants sur la durée de vie pour détecter les fuites mémoires. Le framework contient des assertions binaires qui expriment les relations entre durée de vie des objets (e.g. `assertDB`)

Exemple d'utilisation :

```
void run Compare(ISelection s){
    transaction(s, CHECK /*INFER*/) {
        fInput = new ResourceCompareInput(s);
        assertDB(fInput, s);
        share {
            visitedFiles.add(new String(s.left));
            visitedFiles.add(new String(s.right));
        }
```

1. Ces objets doivent être déréférencés avant l'identificateur de la transaction.
2. Ces objets peuvent être référencés après le déréférencement de l'identificateur.

```
            openCompareEditoOnPage(fInput, fWorkBenchPage);
            fInput = null; //don't reuse this input
        }
}
```

Conclusion

Un des grands avantages de LeakChaser, est de pouvoir écrire des transactions vues comme des greffons de ces applications. Par exemple, pour un problème de base de données, il suffira d'écrire un greffon exécutant des requêtes. Cela évite d'entrer dans les détails d'implémentation de l'application.

4.5 Outils de profiling en Scheme

Dans cette section, nous allons présenter deux outils de profiling, pour l'analyse de la mémoire en Scheme. Ces deux outils sont inclus dans l'environnement de développement BEE [74] pour le langage de programmation Scheme [81].

4.5.1 `Kprof`, profiler la fréquences des allocations

`Kprof` [75] est un profileur des points d'allocations d'un programme Scheme. Pour chaque fonction, il reporte le nombre et le genre d'allocations effectuées. L'idée derrière cet outil est similaire à celui de `gprof` [43], qui est un outil de profiling de code calculant des statistiques sur le programme. Concrètement, il calcule la durée passée dans chaque fonction, le nombre de fois qu'une fonction est exécutée, etc.

En plus de cela, `Kprof` donne des informations sur les opérations du ramasse-miettes. Il affiche également des informations comme l'évolution du tas durant une exécution donnée, la taille du tas et le nombre d'objets vivants. Un léger surcoût est à noter durant l'exécution, mais aucun surcoût mémoire n'est nécessaire.

Dans la même idée, Haskell (section 4.6) ainsi qu'OCaml (voir section 4.7 fournissent ce genre d'outils très utiles lors des profilings, pour mieux comprendre les points critiques d'allocations.

4.5.2 `Kbdb`, un outil pour inspecter le tas

`Kbdb` [75] est un outil d'inspection du tas. Inclus dans l'environnement de développement BEE, le programme est exécuté de manière interactive et peut être stoppé à n'importe quel moment pour examiner les variables, la pile ainsi que le tas. Le tas peut

être affiché de manière à ce que chaque pixel représente une cellule du tas. Chaque élément du tas peut être inspecté pour afficher des informations de type, de point d'allocation dans le code source, ainsi qu'une approximation de son "âge" (durée de vie dans le tas). De plus, `Kbdb` peut afficher les chemins vers des cellules depuis les racines. Cet outil est très utile pour comprendre pourquoi une valeur est considérée comme vivante par le ramasse-miettes.

`Kbdb` ajoute un surcoût mémoire non négligeable aux applications profilées. Serrano *et al.*[75] explique que, sur certains exemples, le nombre d'allocations peut augmenter d'un facteur entre 3,6 et 4,9 pour des exemples en mode debug. Cela est dû au surcoût mémoire ajouté à chaque valeur en mémoire. en effet, dans cette version de la runtime Scheme, chaque bloc se voit augmenter de 4 mots mémoires : 1 pour le pointeur arrière, 1 pour les informations d'allocation, 1 pour l'âge et un pour être utilisé en interne pour supporter le profiling C.

Dans la section 5, nous expliquerons en détails comment nous avons introduit ces informations utiles dans les blocs mémoires en OCaml, sans ajouter de surcoût mémoire, très important lors des profilings.

4.6 Outils de profiling en Haskell/GHC

Haskell est un langage fonctionnel de haut niveau. Il existe différentes méthodes de profiling comme par exemple avoir des statistiques sur la runtime, le profiling du temps d'exécution, de l'espace mémoire, etc.

Une grande partie du contenu de ce chapitre vient du chapitre 5 de la documentation en ligne d'Haskell [3] ainsi que du chapitre 25 du livre Real World Haskell [64].

4.6.1 Statistiques sur la runtime

Pour obtenir ces informations sur la runtime, il faut utiliser différentes options du compilateur, notamment `+RTS` et `-RTS` utilisées à la runtime. L'application en elle-même ne sera donc pas affectée par ces options.

Dans la suite, le programme profilé sera celui de la figure 4.3. Le programme calcule de façon naïve la somme des éléments d'une grande liste.

Avec cela, la runtime pourra par exemple recueillir les informations sur la consommation mémoire ainsi que sur les performances du ramasse-miettes. Par exemple, sur la figure 4.4, en activant les informations de la runtime avec l'option `+RTS -sstderr`, on peut observer cette trace pour le programme de la figure 4.3.

```haskell
import System.Environment
import Text.Printf

main = do
[d] <- map read `fmap` getArgs
printf "%f\n" (mean [1..d])

mean  :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)
```

Figure 4.3 – Programme Haskell calculant la moyenne de la somme des éléments d'une liste.

```
$ ./A 1e7 +RTS -sstderr
  5000000.5
  1,689,133,824   bytes allocated in the heap
  697,882,192     bytes copied during GC (scavenged)
  465,051,008     bytes copied during GC (not scavenged)
  382,705,664     bytes maximum residency (10 sample(s))

  3222 collections in generation 0 (  0.91s)
  10 collections   in generation 1 ( 18.69s)

  742 Mb total memory in use

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    0.63s  (  0.71s elapsed)
  GC    time   19.60s  ( 20.73s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time   20.23s  ( 21.44s elapsed)

  % GC time      96.9%  (96.7% elapsed)

  Alloc rate    2,681,318,018 bytes per MUT second

  Productivity   3.1% of total user, 2.9% of total elapsed
```

Figure 4.4 – Statistiques sur la runtime Haskell obtenues en utilisant l'option `+RTS -sstderr`.

L'option `-sstderr` permet d'afficher les résultats sur la sortie d'erreur. On observe le temps utilisé par le ramasse-miettes ainsi que le maximum de la taille de la mémoire vivante.

Il s'avère que pour calculer la moyenne de la somme d'une liste de 10 millions d'élé-

ments, le programme atteint une mémoire maximum de 742Mo dans le tas, et 96.9% du temps est utilisé par le ramasse-miettes. Au total, seulement 3.1% est vraiment utilisé par le programme pour faire le calcul.

4.6.2 Profiling du temps

GHC permet, entres autres, le profiling du temps d'exécution. Cette démarche se déroule en trois temps :

- compilation du programme avec l'option `-prof` du compilateur
- lancement de l'application avec les options de profiling activées pour obtenir des informations de débogage
- analyse de ces informations

En plus d'ajouter l'option `-prof` au compilateur, il y a la possibilité d'annoter le code source pour profiler certaines fonctions ou expressions particulières. GHC générera alors du code pour calculer le coup d'évaluation de ces expressions à chacune de ces locations.

À la fin, un fichier `.prof` sera généré et contiendra les informations de débogage, comme sur la figure 4.5.

Ce fichier donne un aperçu du comportement de la runtime. Dans la première partie, on observe le nom du programme ainsi que les options utilisées pour l'exécution. La ligne *total time* est le temps utilisé par la runtime du point de vue du système, et la ligne *total alloc* est le nombre correspondant à toutes les allocations en octet pendant l'exécution du programme.

La deuxième section du fichier correspond à la partie sur le profiling en temps d'exécution et en espace pour chacune des fonctions qui en sont responsables.

Enfin, la dernière section est la partie correspondant aux expressions annotées manuellement par le programmeur. La colonne *individual* correspond au coût pour l'expression annotée et la colonne *inherited* correspond aux fils de ces expressions.

Les observations qu'on peut faire après lecture de ces informations sont que la majorité du temps d'exécution est passé dans deux CAFs (*Constant Applicative Form*), le premier étant le calcul de la somme, et l'autre étant sur la partie sur les flottants. Combiné à l'observation faite sur le temps passé par le ramasse-miettes (96.9%), on peut déduire que la cause du problème de cette application est l'allocation des nœuds de la liste, contenant des valeurs flottantes.

Pour conclure, dans des gros programmes, pour identifier les points critiques posant des problèmes de performances, l'utilisation de ces informations de débogage devient un

```
 Time and Allocation Profiling Report  (Final)

          A +RTS -p -K100M -RTS 1e6

       total time  =        0.28 secs   (14 ticks at 20 ms)
       total alloc = 224,041,656 bytes  (excludes profiling
           overheads)

COST CENTRE  MODULE               %time %alloc

CAF sum      Main                  78.6   25.0
CAF          GHC.Float             21.4   75.0

                                          individual     inherited
COST CENTRE MODULE         no.    entries  %time %alloc   %time %
    alloc

MAIN        MAIN            1          0   0.0    0.0   100.0  100.0
 main       Main          166          2   0.0    0.0     0.0    0.0
 mean       Main          168          1   0.0    0.0     0.0    0.0
CAF sum     Main          160          1  78.6   25.0    78.6   25.0
CAF ilvl    Main          158          1   0.0    0.0     0.0    0.0
 main       Main          167          0   0.0    0.0     0.0    0.0
CAF         Numeric       136          1   0.0    0.0     0.0    0.0
CAF         Text.Read.Lex 135          9   0.0    0.0     0.0    0.0
CAF         GHC.Read      130          1   0.0    0.0     0.0    0.0
CAF         GHC.Float     129          1  21.4   75.0    21.4   75.0
CAF         GHC.Handle    110          4   0.0    0.0     0.0    0.0
```

Figure 4.5 – Exemple d'un fichier de profiling en Haskell.

atout non négligeable. Une fois que ces points critiques sont identifiés, on peut lancer des outils de profiling permettant d'avoir plus d'informations.

4.6.3 Profiling de l'espace mémoire

En plus du profiling du temps d'exécution et des statistiques sur la runtime sur les allocations, GHC propose de générer un graphe de l'utilisation mémoire durant l'exécution d'une application. Cet outil permet d'identifier quelques fuites mémoires potentielles, les objets retenus en mémoire inutilement, ce qui stresse le ramasse-miettes inutilement au fur et à mesure de l'exécution de l'application.

Pour construire un graphe mémoire, l'idée est la même que pour profiler le temps

d'exécution. Il faut activer l'option `-prof` du compilateur, et rajouter les options `-auto-all` et `-caf-all`. La possibilité d'avoir plusieurs façons de trier est également proposée, par module, par constructeur, par type de données, etc. Un fichier `.hp` est ainsi généré puis traité par l'outil `hp2ps` qui produit un fichier visuel du graphe mémoire dans le format PostScript.

Pour extraire cette information, l'application doit être exécutée avec l'option `-hc`. Sur la figure 4.6, un exemple de fichier `.hp` est représenté.

```
JOB         "A␣1e6␣+RTS␣-hc␣-p␣-K100M"
SAMPLE_UNIT "seconds"
VALUE_UNIT  "bytes"
BEGIN_SAMPLE 0.00
END_SAMPLE   0.00
BEGIN_SAMPLE 0.24
  (167) main/CAF ìvl   48
  (136) Numeric.CAF    112
  (166) main           8384
  (110) GHC.Handle.CAF 8480
  (160) CAF sum        10562000
  (129) GHC.Float.CAF  10562080
END_SAMPLE 0.24
```

Figure 4.6 – Exemple d'un fichier `.hp` généré par GHC.

Des dumps du tas sont pris à des intervalles réguliers durant l'exécution. Cet échantillonnage peut être modifié avec l'option `-iN` où N est le nombre de seconde (*e.g.* 0,01) entre les intervalles. Plus il y aura de dumps, plus les informations seront précises, mais aussi plus le programme sera lent. Sur la figure 4.7, on peut voir le résultat obtenu après avoir utilisé *hp2ps* pour générer le graphe visuel pour l'exemple de la figure 4.3.

Une partie importante du travail est faite dans l'allocation des nœuds des listes contenant des flottants. Les listes d'Haskell étant paresseuses, la liste entière est construite au fil du temps d'exécution.

Cependant, la liste n'est pas libérée car elle est traversée, ce qui conduit à une augmentation de l'utilisation de la mémoire utilisée. Enfin, durant un peu plus de la moitié du temps d'exécution du programme, l'application se termine en additionnant la liste, et commence le calcul de la longueur. Si nous regardons le fragment d'origine pour la moyenne, nous pouvons voir exactement pourquoi la mémoire est retenue.

Au début, nous calculons la somme de notre liste, ce qui déclenche l'allocation des nœuds de la liste, mais ceux-ci ne seront libérés qu'une fois que la liste entière sera créée,

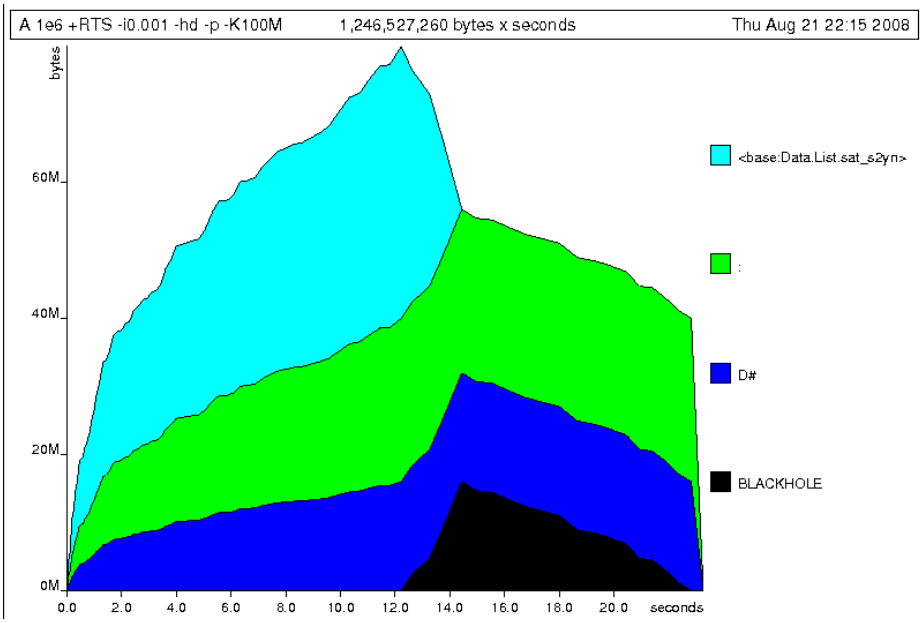

Figure 4.7 – Graphe mémoire obtenu à l'aide de l'outil *hp2ps* à partir d'un fichier `.hp` généré par GHC.

car ils sont nécessaires pour le calcul de la taille de la liste. Dès le moment où le calcul de la somme est terminé, le ramasse-miettes pourra libérer ces nœuds.

4.7 Outils en OCaml

4.7.1 Profiling d'une application avec `ocamlcp` et `ocamlprof`

Les outils `ocamlcp` et `ocamlprof` permettent de profiler des applications bytecode OCaml. Ils annotent le code avec des commentaires sur le nombre de fois qu'est exécuté une expression. Sur la figure 4.8, un exemple de teste utilisant le module `Graphics` d'OCaml. L'exemple est pris du site de la communauté d'OCaml [3].

Les commentaires, avec un nombre, sont ajoutés par `ocamlprof` et indiquent combien de fois chacune des parties du code est appelée.

Pour obtenir cette trace, il faut d'abord compiler avec `ocamlcp` qui va compiler et instrumenter le code ajoutant des enregistrements pour compter le nombre de fois que les fonctions sont appelées, le nombre de fois qu'une des branches d'une conditionnelle sont prises, etc.

```
$ ocamlcp -p a graphics.cma graphtest.ml
```

3. https ://ocaml.org/learn/tutorials/performance_and_profiling.html

```
1   let () =
2     Random.self_init ();
3     Graphics.open_graph " 640x480"
4
5   let rec iterate r x_init i =
6     (* 12820000 *) if i == 1 then (* 25640 *) x_init
7     else
8       (* 12794360 *) let x = iterate r x_init (i-1) in
9       r *. x *. (1.0 -. x);;
10
11  let () =
12    for x = 0 to 640 do
13      (* 641 *) let r = 4.0 *. (float_of_int x) /. 640.0 in
14      for i = 0 to 39 do
15        (* 25640 *) let x_init = Random.float 1.0 in
16        let x_final = iterate r x_init 500 in
17        let y = int_of_float (x_final *. 480.) in
18        Graphics.plot x y
19      done
20    done
```

Figure 4.8 – Exemple tiré du site de la communauté OCaml utilisant le module `Graphics`. Cette exemple est annoté de commentaires ajoutés par `ocamlprof`.

Ensuite, il suffit d'exécuter le programme qui va générer un fichier `ocamlprof.dump`, Celui-ci sera lu par `ocamlprof` pour l'annotation du code source (voir figure 4.8).

```
$ ./a.out
$ ocamlprof graphtest.ml
```

Sous linux, pour profiler et obtenir ce genre de trace pour des programmes natifs, il existe l'outil `grof` qui va générer un fichier `gmon.out`. Celui-ci sera interprété par `gprof`. Sur la figure 4.9, un programme permettant de calculer les 3000 premiers nombres premiers.

Après interprétation du fichier `gmon.out`, sur la figure 4.10, une partie de la trace de notre programme précédent est obtenue.

La majeure partie du temps est consommée par le ramasse-miettes. Le programme étant écrit à des fins de tests, les résultats ne sont pas surprenants. Pour améliorer cela, l'utilisation de boucles agissant sur un tableau aurait été plus efficace.

```
1   let rec filter p = parser
2     [< 'n; s >] ->
3        if p n then [< 'n; filter p s >] else [< filter p s >]
4
5   let naturals =
6     let rec gen n = [< 'n; gen (succ n) >] in gen 2
7
8   let primes =
9     let rec sieve = parser
10      [< 'n; s >] ->
11       [< 'n; sieve (filter (fun m -> m mod n <> 0) s) >] in
12    sieve naturals
13
14  let () =
15    for i = 1 to 3000 do
16      ignore (Stream.next primes)
17    done
```

Figure 4.9 – Programme OCaml calculant les 3000 premiers nombres premiers.

4.7.2 Profiling d'une application avec `perf`

`perf` est un outil disponible sous Linux permettant d'avoir des informations sur les performances d'un programme. Il fournit un framework d'analyse permettant d'avoir des informations au niveau du hardware (CPU/PMU), ainsi que des traces plus précises dans le source.

Sur un article de blog du site d'OCamlPro [4], nous expliquons un exemple d'utilisation de `perf` sur le compilateur natif. Il suffit d'appeler `perf` avec les options `record` et `-g` pour commencer l'enregistrement des données de debug.

```
$ perf record -g ./ocamlopt.opt -c -I utils -I parsing -I typing
    typing/*.ml
```

`perf` va générer un fichier `perf.data` contenant tous les événements reçus pendant l'exécution de la commande `ocamlopt.opt`. Le fichier contient entre autres, le compteur de performance sur le processeur amd64 et les backtraces des événements.

Pour afficher les résultats, il suffit alors d'appeler `perf` avec les options `report` et `-g`.

```
$ perf report -g
```

Le résultat de cette commande sur notre exemple est visible sur la figure 4.11.

4. http ://www.ocamlpro.com/blog/2012/08/08/profile-native-code.html

```
  [...]
Flat profile :

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
10.86      0.57     0.57     2109     0.00     0.00  sweep_slice
9.71       1.08     0.51     1113     0.00     0.00  mark_slice
7.24       1.46     0.38  4569034     0.00     0.00  Sieve__code_begin
6.86       1.82     0.36  9171515     0.00     0.00
    Stream__set_data_140
6.57       2.17     0.34 12741964     0.00     0.00  fl_merge_block
6.29       2.50     0.33  4575034     0.00     0.00  Stream__peek_154
5.81       2.80     0.30 12561656     0.00     0.00  alloc_shr
5.71       3.10     0.30     3222     0.00     0.00  oldify_mopup
4.57       3.34     0.24 12561656     0.00     0.00  allocate_block
4.57       3.58     0.24  9171515     0.00     0.00  modify
4.38       3.81     0.23  8387342     0.00     0.00  oldify_one
3.81       4.01     0.20 12561658     0.00     0.00  fl_allocate
3.81       4.21     0.20  4569034     0.00     0.00  Sieve__filter_56
3.62       4.40     0.19     6444     0.00     0.00  empty_minor_heap
3.24       4.57     0.17     3222     0.00     0.00
    oldify_local_roots
2.29       4.69     0.12  4599482     0.00     0.00  Stream__slazy_221
2.10       4.80     0.11  4597215     0.00     0.00  darken
1.90       4.90     0.10  4596481     0.00     0.00  Stream__fun_345
1.52       4.98     0.08  4575034     0.00     0.00  Stream__icons_207
1.52       5.06     0.08  4575034     0.00     0.00  Stream__junk_165
1.14       5.12     0.06     1112     0.00     0.00  do_local_roots
  [...]
```

Figure 4.10 – Trace obtenue avec l'outil `gprof` sur l'exemple de la figure 4.9.

En regardant de plus près le résultat obtenu, l'essentiel du temps est passé dans la primitive `compare_val`. En naviguant dans les résultat obtenus, comme sur la figure 4.12, on remarque que cette primitive est appelée depuis `Pervasives.compare`, elle-même appelée depuis le module `Set` dans le fichier `asmcomp/interp.ml`.

En regardant de plus près, dans le module `IntPairSet`, la fonction de comparaison polymorphe peut être remplacée par une fonction de comparaison monomorphe.

Dans notre exemple, le code suivant :

```
Events : 3K cycles
+   9.81%  ocamlopt.opt  ocamlopt.opt       [.] compare_val
+   8.85%  ocamlopt.opt  ocamlopt.opt       [.] mark_slice
+   7.75%  ocamlopt.opt  ocamlopt.opt       [.]
    caml_page_table_lookup
+   7.40%            as  as                 [.] 0x5812
+   5.60%  ocamlopt.op[kernel.kallsyms]     [k] 0xffffffff8103d0ca
+   3.91%  ocamlopt.opt  ocamlopt.opt       [.] sweep_slice
+   3.18%  ocamlopt.opt  ocamlopt.opt       [.] caml_oldify_one
+   3.14%  ocamlopt.opt  ocamlopt.opt       [.] caml_fl_allocate
+   2.84%            as  [kernel.kallsyms]  [k] 0xffffffff81317467
+   1.99%  ocamlopt.opt  ocamlopt.opt       [.] caml_c_call
+   1.99%  ocamlopt.opt  ocamlopt.opt       [.] caml_compare
+   1.75%  ocamlopt.opt  ocamlopt.opt       [.] camlSet__mem_1148
+   1.62%  ocamlopt.opt  ocamlopt.opt       [.] caml_oldify_mopup
+   1.58%  ocamlopt.opt  ocamlopt.opt       [.] camlSet__bal_1053
+   1.46%  ocamlopt.opt  ocamlopt.opt       [.] camlSet__add_1073
+   1.37%  ocamlopt.opt  libc-2.15.so       [.] 0x15cbd0
+   1.37%  ocamlopt.opt  ocamlopt.opt       [.]
    camlInterf__compare_1009
+   1.33%  ocamlopt.opt  ocamlopt.opt       [.] caml_apply2
+   1.09%  ocamlopt.opt  ocamlopt.opt       [.] caml_modify
+   1.07%            sh  [kernel.kallsyms]  [k] 0xffffffffa07e16fd
+   1.07%            as  libc-2.15.so       [.] 0x97a61
+   0.94%  ocamlopt.opt  ocamlopt.opt       [.] caml_alloc_shr
```

Figure 4.11 – Résultat de la commande `perf` sur un exemple avec le compilateur natif d'OCaml (1).

```
1  module IntPairSet =
2    Set.Make(struct type t = int * int let compare =
         compare end)
```

sera transformé en :

```
1  module IntPairSet =
2    Set.Make(struct type t = int * int
3                    let compare (a1,b1) (a2,b2) =
4                      if a1 = a2 then b1 - b2 else a1 -
                           a2
5              end)
```

Sur la figure 4.13, une comparaison des deux exécutables, avant et après optimisation de la fonction de comparaison.

```
Events : 3K cycles
-   9.81%  ocamlopt.opt  ocamlopt.opt          [.] compare_val
   - compare_val
      - 71.68% camlSet__mem_1148
         + 98.01% camlInterf__add_interf_1121
         + 1.99% camlInterf__add_pref_1158
      - 21.48% camlSet__add_1073
         - camlSet__add_1073
            + 93.41% camlSet__add_1073
            + 6.59% camlInterf__add_interf_1121
      + 1.44% camlReloadgen__fun_1386
      + 1.43% camlClosure__close_approx_var_1373
      + 1.43% camlSwitch__opt_count_1239
      + 1.34% camlTbl__add_1050
      + 1.20% camlEnv__find_1408
+   8.85%  ocamlopt.opt  ocamlopt.opt          [.] mark_slice
-   7.75%  ocamlopt.opt  ocamlopt.opt          [.]
    caml_page_table_lookup
   - caml_page_table_lookup
      + 50.03% camlBtype__set_commu_1704
      + 49.97% camlCtype__expand_head_1923
+   7.40%            as  as                    [.] 0x5812
+   5.60%  ocamlopt.opt  [kernel.kallsyms]     [k] 0
    xffffffff8103d0ca
+   3.91%  ocamlopt.opt  ocamlopt.opt          [.] sweep_slice
```

Figure 4.12 – Résultat de la commande `perf` sur un exemple avec le compilateur natif d'OCaml (2).

```
$ time ./ocamlopt.old -c -I utils -I parsing -I typing typing/*.ml
   ./ocamlopt.old  7.38s user 0.56s system 97% cpu 8.106 total

$ time ./ocamlopt.new -c -I utils -I parsing -I typing typing/*.ml
   ./ocamlopt.new  6.16s user 0.50s system 97% cpu 6.827 total
```

Figure 4.13 – Comparaison des performances de l'exécutable `ocamlopt` avant et après optimisation gtâce à `perf`.

On remarque une augmentation de la vitesse d'exécution après ce changement. L'idée est que cette étape d'itération peut être faite plusieurs fois afin de trouver les points critiques d'une application donnée.

4.7.3 Profiling d'une application avec `ocamlviz`

OCamlViz [28] est un outil d'instrumentation d'applications OCaml, permettant de visualiser facilement en temps réel l'évolution de certains indicateurs du programme. L'application doit être liée statiquement avec une bibliothèque OCaml qui lance un thread de communication en arrière plan. L'application OCamlViz se connecte alors à ce thread, pour recevoir en temps quasi réel les mises à jour des valeurs. Par défaut, seules les statistiques du GC sont transmises, mais l'utilisateur peut modifier son application pour signaler les modifications de certaines variables. Outre la valeur de la variable, la bibliothèque permet aussi de calculer la quantité de mémoire retenue par la valeur, ou pour des conteneurs (Hashtbl, etc.), le nombre de valeurs stockées dans ces conteneurs.

Comparé au travail présenté ici, OCamlViz ne fournit que des informations très globales sur l'utilisation mémoire, des informations plus fines demandant un lourd travail d'instrumentation du code par l'utilisateur, et ralentirait considérablement les performances de l'application observée. OCamlViz s'adresse donc plus aux utilisateurs voulant observer l'évolution de paramètres autres de leur application, voir monitorer à distance des compteurs dans une application serveur, sans avoir à construire une interface graphique spécifique.

4.7.4 Échantillonneur d'allocations (*Poor man's profiler*)

L'idée générale de l'algorithme

"Poor man have no time to learn complicated new tools. Poor man need simulation results."[5]

L'idée derrière ce profileur est de simplifier le profiling d'une application au maximum. Le but principal est de pouvoir extraire des informations sans forcément connaître les détails du programme, mais de pouvoir comprendre pourquoi l'application est lente.

Avec les debuggers classiques, pour profiler les programmes lents, la façon standard d'obtenir des traces intéressantes est la suivante :

1. Lancer le programme avec le debugger
2. L'arrêter après un certain temps puis regarder le code exécuté
3. Répéter l'étape 2 autant que nécessaire

5. poormansprofiler.org

Ensuite selon les outils, une représentation graphique peut être obtenue pour pouvoir naviguer entre le code source et les données obtenues avec la trace. Il faut alors comprendre le fonctionnement de l'outil, lire la documentation, refaire ces étapes de profiling jusqu'à obtenir l'information pertinente nous expliquant une ou les causes de la lenteur du programme.

L'idée du *profileur du pauvre* est de simplifier ce processus en enlevant les parties superflues, comme le fait d'avoir une trace toutes les microsecondes ou l'apprentissage d'un nouvel outil de profiling.

La philosophie du profileur est d'utiliser les outils classiques disponibles en les poussant le plus loin possible.

`alloc-prof` une variante en OCaml

`alloc-prof` [21] permet au programmeur de trouver les points critiques d'allocation dans leur programme. Inspiré d'un échange sur la mailing-list d'OCaml présentant le profileur du pauvre, il consiste en une modification de la runtime d'OCaml qui sauvegarde dans un journal la pile du programme à chaque fois qu'une certaine quantité de mémoire est allouée. Les traces peuvent être affichées sur une page web écrit avec `js_of_ocaml` [90] pour l'affichage et la navigation dans le graphe. Sur la figure 4.14, on peut voir l'exemple de ce graphe généré par `alloc-prof` pour la commande `ocamlopt`. On peut alors naviguer dans ce graphe en cliquant sur les points critiques d'allocations.

Le constat que nous faisons est qu'en OCaml, très peu d'outils nous permettent aujourd'hui de faire du profiling mémoire comme en Java, Haskell ou Scheme. Contrairement à Java dont les blocs mémoires contiennent beaucoup d'informations utile pour le profiling, en OCaml, la runtime n'offre très peu d'informations de ce genre.

Nous avons vu qu'analyser l'exécution d'une application avec un profileur permet de mieux comprendre son comportement. Les résultats peuvent permettre de faire de l'optimisation de la mémoire ou de la performance et dans certains cas de détecter des problèmes liés à la mémoire.

Nous nous sommes fixés comme objectif de ne proposer pour OCaml que des techniques de profiling qui ne modifiaient pas le comportement mémoire des applications, afin de ne pas être contraints de collecter des informations qui pourraient ne plus être pertinentes, ni les performances, afin de pouvoir être utilisées en production.

Dans la suite, nous introduisons les *identifiants de points d'allocation* qui vont nous

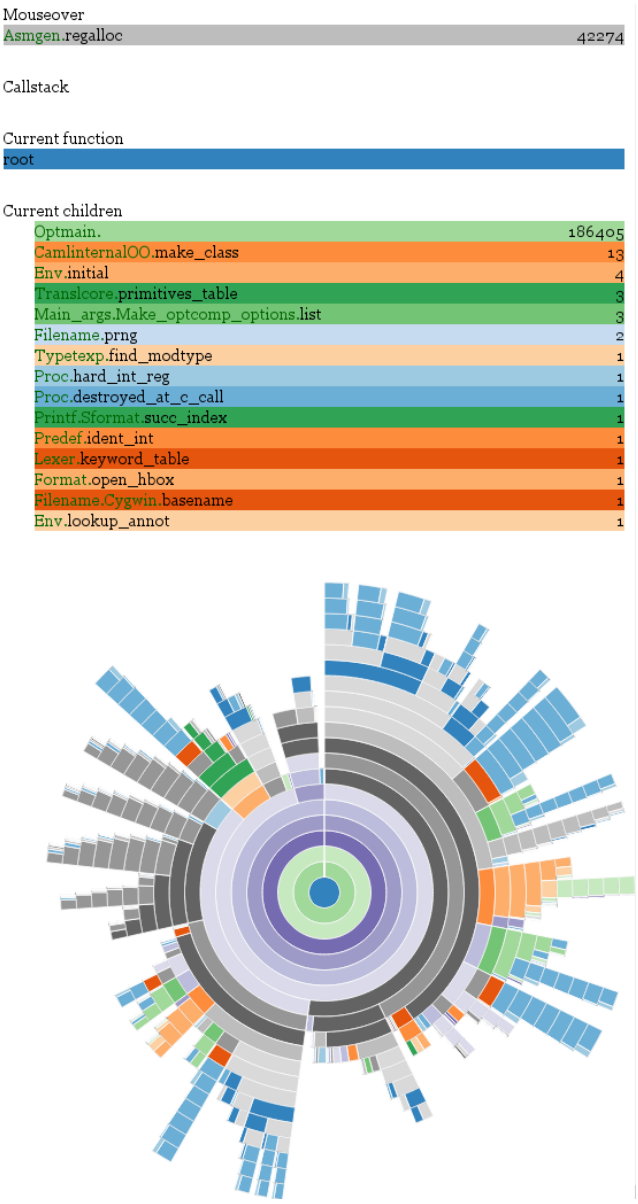

Figure 4.14 – Exemple d'affichage des points d'allocations chauds pour `ocamlopt` avec l'outil `alloc-prof`.

permettre, sans aucun surcoût mémoire, de retracer les points d'allocation de chaque bloc en mémoire. Comme nous l'avons vu précédemment, cette absence de surcoût lors du profiling est très importante, car le comportement mémoire doit rester le plus proche possible de l'application réelle, sinon les résultats peuvent devenir inexploitables, car faussés par ce surcoût.

"L'important c'est de savoir ce qu'il faut observer."

Edgar Allan Poe

5 Extraction de données

Grâce à son système de typage fort, statique et inféré, OCaml n'a pas besoin d'informations de types pendant l'exécution d'un programme. C'est grâce à cela que la représentation mémoire vue dans la Section 2.5 est très compacte, contrairement aux langages avec un système de types moins fort, et par conséquent les programmes ont de meilleures performances pendant leur exécution.

Malheureusement, cet avantage devient rapidement un inconvénient lorsque l'on tente de faire du profiling mémoire, à cause de l'absence d'information de types durant l'exécution d'un programme. Il devient alors très compliqué d'essayer de comprendre l'occupation mémoire et la nature des valeurs allouées.

Afin de pouvoir relier les informations d'allocation recueillies dynamiquement par un profileur et les points de programme à l'origine de ces allocations, nous avons ajouté un identifiant de site d'allocation sur chacun des blocs en mémoire, pour pouvoir les retracer par la suite.

Pour cela, après avoir décoré l'arbre de syntaxe abstraite du programme avec un identifiant unique pour chaque site d'allocation, nous avons propagé ces identifiants le long de toute la chaîne de compilation, jusqu'aux instructions d'allocation elles-mêmes, afin de les insérer dans les en-têtes des blocs mémoires alloués. La représentation de ces identifiants est très compacte, et, au prix d'une légère restriction de la taille maximale

des blocs alloués, l'insertion des identifiants a un coût mémoire nul.

Durant l'analyse de la mémoire dynamique d'un programme, ces identifiants nous permettent de retrouver facilement depuis un bloc mémoire, son point d'allocation dans le programme source. De plus, la génération de ces identifiants se faisant sur l'arbre de syntaxe abstraite typé, il est également possible de leurs associer le type statique de la valeur correspondante.

La génération d'identifiants nécessite un travail conséquent sur le compilateur, pour tracer ces points d'allocation et de propager les identifiants. Une modification supplémentaire est également nécessaire dans la bibliothèque d'exécution d'OCaml afin de pouvoir sauvegarder l'image de la mémoire d'une application durant son exécution. En effet, une fois les identifiants générés et sauvegardés dans les en-têtes des blocs, nous avons mis en œuvre plusieurs façons de dumper le tas (les instantanés) sur disque, afin de pouvoir ensuite relire la mémoire et ainsi récupérer les nouveaux blocs mémoires avec ces identifiants de site d'allocation. Ce chapitre décrit notre technique d'extraction de ces données.

5.1 Ajout des identifiants de site d'allocation

Le but est d'ajouter à chaque bloc du tas une information utile pour le profiling (position de l'allocation dans le code source, son type, etc.), sans augmenter l'occupation mémoire des blocs.

L'idée de notre outil est de décorer l'arbre de syntaxe abstraite avec un identifiant unique pour chaque emplacement qui puisse provoquer une allocation, puis de propager ces identifiants durant la compilation jusqu'aux instructions d'allocation afin de les conserver explicitement dans chaque bloc mémoire. Sur les architectures 64 bits, un certain nombre de bits des en-têtes de blocs sont utilisables, si on réduit la taille maximale des blocs alloués, et c'est dans cet espace que nous stockons les identifiants. Durant l'analyse d'un tas mémoire, ils nous permettent de « remonter » d'un bloc mémoire à son point d'allocation dans le programme source. De plus, en effectuant la génération des identifiants sur l'arbre de syntaxe abstraite typé, on peut aussi associer à chaque identifiant le type *statique* de la valeur correspondante. Cela nécessite également de modifier la machine virtuelle.

5.1.1 Générations des identifiants

5.1.1.1 Qu'est-ce qu'un identifiant ?

Un identifiant (ou locid) est un entier dont la valeur maximale peut atteindre $2^{22} - 1$ (soit un peu plus de 4×10^6 possibilités).

Ces identifiants vont être assignés à chaque point d'allocation d'un programme OCaml. Ainsi, seules les $2^{22} - 1$ premières allocations seront tracées. Une erreur est générée à l'exécution si le nombre d'identifiants dépasse les 2^{22} possibilités (cela pourrait être vérifié lors de l'édition de liens du programme, mais l'erreur resterait néanmoins possible avec des chargements dynamiques).

5.1.1.2 Calcul des identifiants de location

Nous allons voir en détails dans la section 5.1.2 comment ne pas ajouter de surcoût au niveau de l'occupation mémoire. Dans cette partie, nous allons voir le calcul et la génération de ces identifiants durant la compilation.

Lors de la génération du code d'une unité de compilation, nous numérotons séquentiellement chacun de ses points d'allocation. Le $i^{\text{ème}}$ point d'allocation de cette unité se verra associer, au moment du chargement du module, l'identifiant $n + i$, où n est l'identifiant du dernier point d'allocation de la dernière unité précédemment chargée. Ainsi, les identifiants d'un exécutable sont les numéros d'ordre absolus des points d'allocation des programmes source qui le composent. Cette création d'identifiants utilise au mieux l'espace d'identifiants, et nous permet de retrouver sans ambiguïté chacun des points d'allocation du programme.

Dans le compilateur (bytecode et natif)

Dans une première version, nous avons généré les identifiants de façon aléatoire, de manière à associer une location à chacun. Pour cela, nous avons propagé ces identifiants dans les différentes phases de compilation jusqu'à la génération du code.

Ce calcul était un simple hash entre la location et le nom du module. Un inconvénient avec cette méthode était la forte probabilité de collision entre deux identifiants, qui plus est, très difficile à détecter.

Une autre manière est d'utiliser la taille, un nombre aléatoire et le tag d'un bloc comme son identifiant unique. Plusieurs blocs pourraient alors avoir la même composante aléatoire, mais comme il serait identifié par ce triplet, les risques de collisions seraient

diminués. De plus, cette méthode pourrait s'appliquer à des machines 32-bits, avec des nombres aléatoires calculés sur un intervalle plus petit.

La méthode que nous adoptons est différente et compatible uniquement avec les machines 64-bits. Les identifiants sont générés sur l'arbre de syntaxe abstraite typé et sont propagés durant les étapes de compilation, que ce soit en bytecode ou en code natif, pour être inclus dans les en-têtes lors de leur génération.

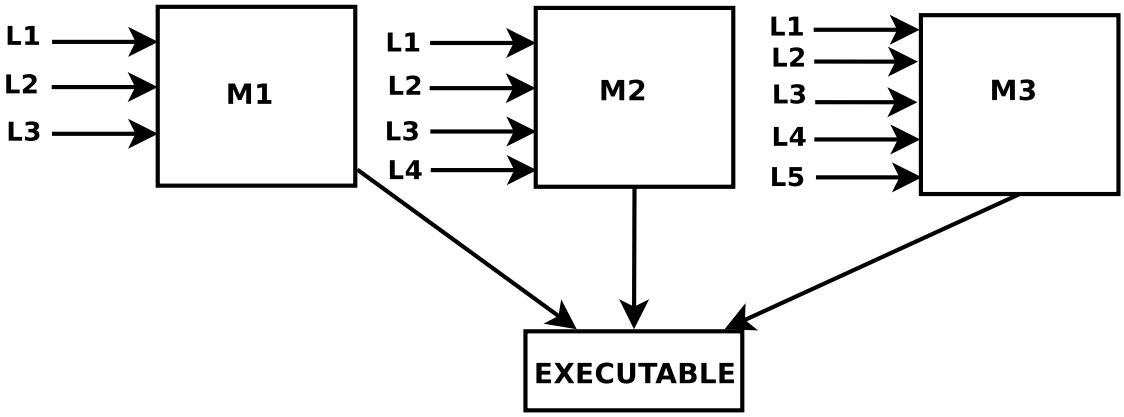

Figure 5.1 – Génération des identifiants de sites d'allocation.

Sur la figure 5.1, on peut voir les identifiants de site d'allocation par module. Dans le module `M1` il y a trois locations, dans le module `M2` quatre et dans le module `M3` cinq.

Le calcul de ces identifiants se fait dans un premier temps par module avec une variable globale appelée `{MODULE_NAME}_locid_base` qui va servir de compteur pour les modules. La mise à jour de cette globale se fait à l'exécution après avoir chargé tous les modules, pour connaître l'ordre dans lequel ils sont chargés.

Pour `M1` :	Pour `M2` :	Pour `M3` :
$L_1 = 0 +$ `M1_locid_base`	$L_1 = 0 +$ `M2_locid_base`	$L_1 = 0 +$ `M3_locid_base`
$L_2 = 1 +$ `M1_locid_base`	$L_2 = 1 +$ `M2_locid_base`	$L_2 = 1 +$ `M3_locid_base`
$L_3 = 2 +$ `M1_locid_base`	$L_3 = 2 +$ `M2_locid_base`	$L_3 = 2 +$ `M3_locid_base`
	$L_4 = 3 +$ `M2_locid_base`	$L_4 = 3 +$ `M3_locid_base`
		$L_5 = 4 +$ `M3_locid_base`

Figure 5.2 – Calcul des identifiants de site d'allocation, après avoir chargé les modules.

Sur la figure 5.2, la partie calculée statiquement par le compilateur est l'ordre dans lequel nous rencontrons un point d'allocation dans le module, les offsets. On leur ajoute la valeur de la variable globale qui sera initialisée plus tard, après le chargement des

modules. Ainsi, la locid est l'addition entre l'offset et la valeur de l'identifiant.

Par exemple, pour le module `M1`, L_1 vaut `0 + M1_locid_base`, L_2 vaut `1 + M1_locid_base` et L_3 vaut `2 + M1_locid_base`.

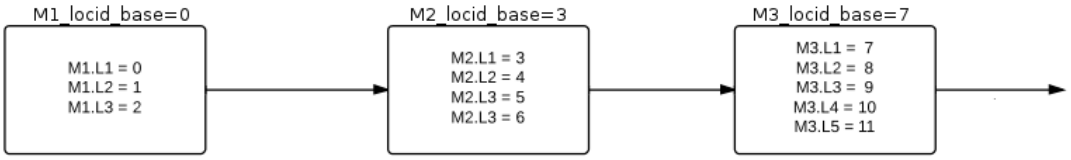

Figure 5.3 – Calcul des identifiants de location après le chargement des modules.

Sur la figure 5.3, on peut voir la partie allant être calculée après le chargement des modules. En effet, à cet instant nous sommes capables de mettre à jour les variables globales `{MODULE_NAME}_locid_base` de telle sorte qu'à la fin de l'initialisation de chaque module, elles prennent le nombre d'identifiants générés dans chacun des modules. Une fois cette globale initialisée pour un module donné, grâce à l'offset calculé statiquement, nous allons pouvoir générer un identifiant unique par point d'allocation. De plus, le calcul dans le module suivant ne collisionnera pas avec les autres valeurs et continuera d'augmenter.

En reprenant la figure 5.3, à l'étape 1, `M1_locid_base` vaut 0, le calcul des identifiants de ce module sont les premiers. À la fin, nous avons trois allocations dans ce module, donc `M2_locid_base` va augmenter de trois. Ainsi, à l'étape 2, les valeurs des identifiants vont être additionnées avec `M2_locid_base` qui vaut 3. Enfin, à la dernière étape, nous avons ajouté à `M3_locid_base` les quatre allocations du module `M2`, donc les identifiants du module `M3` vont être additionnés avec la valeur 7.

Pour résumer, la partie calcul des identifiants par module se fait statiquement durant la compilation. Ensuite, nous générons le code d'addition entre cet offset et l'identifiant de base du module qui sera, elle, mise à jour après le chargement des modules.

Sur la figure 5.4, un exemple de code de test pour montrer le code produit en assembleur par le compilateur natif. Un enregistrement à trois champs est créé par la fonction create.

Sur la figure 5.5, on peut voir la sortie du code intermédiaire cmm du compilateur natif, sans les modifications apportées sur les en-têtes. Sur la figure 5.6, la sortie du code intermédiaire, ainsi que le code assembleur du code qui va modifier l'en-tête et rajouter l'addition entre l'identifiant de base du module et la locid calculée par module. On peut

```
1  type toto = I of int | F of float | TT of toto * toto
2  type titi = {f  :float; t1  :toto; t2  :toto}
3  let create () =
4   {
5     f = 1.1 +. 45.;
6     t1 = TT (I (42), F 42.);
7     t2 = TT (I (43), F 43.);
8   }
```

Figure 5.4 – Exemple de code créant un enregistrement à trois champs, dans le but de montrer le code intermédiaire avec l'ajout de l'identifiant du site d'allocation dans les en-têtes des blocs.

voir le chargement de l'identifiant de base, puis l'addition dans l'en-tête.

```
1  (function camlTest2__create_1016 (param/1020 : addr)
2   (alloc 3072
3     (alloc 1277 (+f 1.1 45.))
4     "camlTest2__3" "camlTest2__4"))
5
6      qui va générer le code assembleur suivant :
7
8        [...]
9    leaq      16(%rbx), %rax
10   movq      $3072, -8(%rax)
11   movq      %rbx, (%rax)
```

Figure 5.5 – Extrait du code intermédiaire, ainsi que code assembleur, du programme de la figure 5.4, avec le compilateur OCaml 4.01.0 non patché.

De cette manière, nous traçons les allocations faites de façon implicite par le compilateur et nous pouvons associer à chaque point d'allocation un de ces identifiants. Nous stockons en mémoire une table nous permettant de retrouver un site d'allocation depuis un identifiant. Nos expérimentations ont montré que l'addition (voir le code assembleur de la figure 5.6) supplémentaire nécessaire pour placer les identifiants d'allocation dans l'en-tête des blocs n'avait pas d'influence perceptible sur la vitesse d'exécution des programmes.

Modification de la machine virtuelle

Il se peut que certaines allocations soient faites depuis la machine virtuelle OCaml. Certains appels de fonction OCaml sont des appels externes (fonctions déclarées par

```
1  (function camlTest2__create_1017 (param/1022 : addr)
2   (alloc (+ 12884902912 (load "camlTest2__locid_base"))
3     (alloc (+ 4294967549 (load "camlTest2__locid_base")) (+f 1.1 45.)
          )
4     "camlTest2__3" "camlTest2__4"))
5
6     qui va générer le code assembleur suivant :
7
8        [...]
9    leaq        16(%rbx), %rax
10   movq        camlTest2__locid_base@GOTPCREL(%rip), %rdi
11   movq        (%rdi), %rdi
12   movabsq     $12884902912, %rsi
13   addq        %rdi, %rsi
14   movq        %rsi, -8(%rax)
15   movq        %rbx, (%rax)
```

Figure 5.6 – Extrait du code intermédiaire, ainsi que code assembleur, du programme de la figure 5.4, avec le compilateur OCaml 4.01.0 patché qui insère les identifiants d'allocation dans les en-têtes des blocs.

`external`) comme par exemple la fonction de création d'un tableau dans le module `Array`. Il faut donc pouvoir profiler également ces allocations.

Pour cela, dans une première version, nous avons d'abord tracé toutes les fonctions allouant dans le tas, créant des en-têtes.

Une fois ces fonctions identifiées, nous avons généré des identifiants statiques par point d'allocation dans le code `C`. De cette façon, les en-têtes pourront être modifiés avec ces identifiants et nous pourrons faire la différence entre du code allouant depuis le code `OCaml` avec des identifiants générés comme expliqué précédemment, et du code allouant dans la runtime.

Si nous prenons l'exemple de la création d'un tableau, le code dans le module `Array` (fichier `array.ml`) est le suivant :

```
external make : int -> 'a -> 'a array = "caml_make_vect"
```

En regardant de plus près le code `C` de la runtime sur la figure 5.8, on s'aperçoit qu'il y a quatre points d'allocation. Ligne 7, 14, 19 et 23 on remarque que les fonctions `caml_alloc`, `caml_alloc_small` et `caml_alloc_shr` prennent un nouvel argument supplémentaire correspondant à l'identifiant. Grâce à cela, nous allons pouvoir insérer cet identifiant dans les en-têtes des blocs (voir 5.1.2). En effet, ces fonctions, ainsi que toutes

celles qui allouent, feront appel à une macro de création d'en-tête (figure 5.7) définie dans `gc.h`. Cette macro prend également un argument supplémentaire correspondant à l'identifiant de location.

```
#define Make_header(wosize, tag, color, prof)   \
    (((header_t) (((header_t) (wosize) << 32)   \
                  + ((profiling_t) (prof) << 10) \
                  + (color)                      \
                  + (tag_t) (tag))))             \
```

Figure 5.7 – La macro C permettant la création d'en-têtes dans la runtime OCaml.

```
if (Is_block(init)
        && Is_in_value_area(init)
        && Tag_val(init) == Double_tag) {
  d = Double_val(init);
  wsize = size * Double_wosize;
  if (wsize > Max_wosize) caml_invalid_argument("Array.make");
  res = caml_alloc(wsize, Double_array_tag, PROF_ARRAY_MK_VECT_1);
  for (i = 0; i < size; i++) {
    Store_double_field(res, i, d);
  }
} else {
  if (size > Max_wosize) caml_invalid_argument("Array.make");
  if (size < Max_young_wosize) {
    res = caml_alloc_small(size, 0, PROF_ARRAY_MK_VECT_2);
    for (i = 0; i < size; i++) Field(res, i) = init;
  } else if (Is_block(init) && Is_young(init)) {
    caml_young_limit = caml_young_start; /* SPACE */
    caml_minor_collection();
    res = caml_alloc_shr(size, 0, PROF_ARRAY_MK_VECT_3);
    for (i = 0; i < size; i++) Field(res, i) = init;
    res = caml_check_urgent_gc (res);
  } else {
    res = caml_alloc_shr(size, 0, PROF_ARRAY_MK_VECT_4);
    for (i = 0; i < size; i++) caml_initialize(&Field(res, i), init);
    res = caml_check_urgent_gc (res);
  }
}
```

Figure 5.8 – Extrait du code du fichier `array.c` de la runtime d'OCaml 4.01.0

Ces identifiants sont également calculés statiquement dans la partie runtime, pour les

primitives C qui font des allocations. De ce fait, nous sommes capables de façon précise, de donner la location dans le code source des allocations faites directement depuis la runtime OCaml.

5.1.2 Enregistrement des identifiants

Maintenant que nous sommes capables de générer un identifiant, de lui associer des informations telles que sa location dans le programme source et son type, nous allons voir les modifications que nous avons apportées à la représentation mémoire, pour étiqueter chaque bloc sans augmenter la taille du tas, ni des blocs en question.

5.1.2.1 Modification des en-têtes

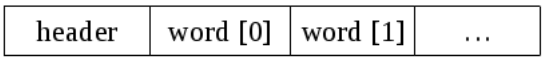

Figure 5.9 – Rappel : bloc mémoire OCaml

Sur la figure 5.9, nous faisons un petit rappel sur la forme des blocs mémoire. Section 2.5.3.2 nous avons expliqué en détails la représentation d'un bloc mémoire OCaml. La partie qui nous intéresse étant l'en-tête encodé sur `64-bits`, nous allons utiliser une partie sous-utilisée de celui-ci.

Pour cela, comme représenté sur la figure 5.10, nous réduisons le champ `Wosize` contenant la taille en mot (8 octets) d'un bloc. Sa taille passe alors de `54-bits` à `32-bits`.

⊕ Avec cette modification, les avantages sont qu'il n'y a :

- aucun impact visible durant l'exécution d'un programme,
- pas de surcoût en mémoire, car nous insérons ces identifiants dans l'en-tête des blocs,
- pas d'impact sur le ramasse-miettes. En effet, n'ayant pas modifié l'occupation mémoire, ni l'implantation de celui-ci, le ramasse-miettes se comportera de la même manière qu'une version non modifiée.

⊖ Les inconvénients de cette approche sont que pour le moment, pour éviter une trop grosse contrainte sur la taille des blocs, ces modifications ne sont applicables que sur les plateformes 64 bits. En effet, le fait de réduire la taille des blocs ne serait pas acceptable sur les machines 32 bits. De plus, les identifiants à ce jour sont limités à 2^{22}, soit environ quatre millions d'allocation. Enfin, la taille maximale d'un bloc sera de `32Go`, ce qui reste

tout de même raisonnable.

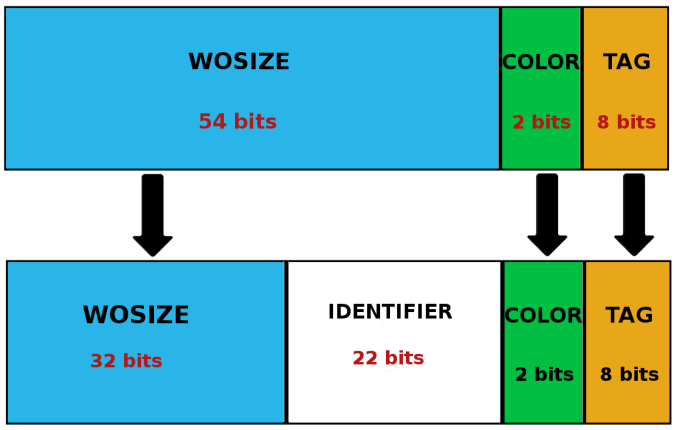

Figure 5.10 – En-tête d'un bloc OCaml après ajout des identifiants de location.

5.1.2.2 Enregistrement dans le tas

Les identifiants sont donc propagés durant les étapes de compilation, que ce soit en bytecode ou en code natif, pour être inclus dans les en-têtes lors de leurs générations. Sur la figure 5.11, nous avons comparé un code allouant des listes pour comprendre l'impact durant la génération de ces identifiants. En effet, comme il y a cette addition avec l'identifiant de base, on peut s'attendre à un léger ralentissement lors de l'exécution.

Sur la figure 5.12, on ne remarque pas d'impact entre le compilateur version 4.01.0 et notre compilateur modifié. En effet, le code étant écrit pour stresser l'allocateur, on remarque que le temps d'exécution est comparable (environ 1,6s pour les deux versions). De plus, le comportement du ramasse-miettes reste le même. Le nombre de collections mineures et majeures, ainsi que le nombre de compactions restent inchangés. Comme nous ne rajoutons aucun surcoût mémoire, le nombre de mots alloués dans le tas reste identique (visible avec les valeurs *minor_collections*, *major_collections* et *compaction* qui restent identiques dans les deux version).

5.1.3 Compilation

Nous avons vu dans les sections précédentes le principe de la génération des identifiants de site d'allocation et leur insertion dans les en-têtes des blocs mémoires. Nous allons maintenant voir dans cette section, de façon détaillée, la chaîne de compilation complète jusqu'à la génération de code et le traitement de la liaison dynamique.

```
let rec alloc i =
  if i <= 0 then []
  else i  : i  : i  : i  : i  : i  : i  : :
       i  : i  : i  : i  : i  : i  : i  : :
       i  : i  : i  : i  : i  : i  : alloc (i-1)

let () =
  for i = 0 to 999999 do
    ignore(alloc 50)
  done

let () =
  Gc.print_stat stdout
```

Figure 5.11 – Programme de test, faisant beaucoup d'allocations, nous permettant de tester le surcoût lors du calcul des identifiants.

	Standard OCaml 4.01.0	Patched OCaml 4.01.0
minor_words	3000045896	3000047456
promoted_words	16822395	16822755
major_words	16822404	16822764
minor_collections	11444	11444
major_collections	8692	8692
heap_words	126976	126976
heap_chunks	1	1
top_heap_words	126976	126976
live_words	2124	2904
live_blocks	704	964
free_words	124852	124072
free_blocks	1	1
largest_free	124852	124072
fragments	0	0
compactions	4345	4345
Elapsed time	1.609999278s	1.598891083s

Figure 5.12 – Résultats obtenus avec la commande `perf`, résumés sous forme de tableau pour comparer les performances du programme de la figure 5.11 avec le compilateur standard OCaml version 4.01.0 et notre compilateur modifié.

5.1.3.1 Rappels

Le compilateur OCaml accepte un fichier source en entrée (fichier `.ml` pour les modules et `.mli` pour leurs signatures). Chacun de ces fichiers `.ml` représente une unité de compilation, et un module du même nom. Au moment de la génération de l'exécutable ou d'une librairie (fichier `.cma`), chacun de ces modules, dont les dépendances seront calculées préalablement, seront donc compilés et liés.

Sur la figure 5.13, on peut voir un aperçu des étapes de compilation du compilateur OCaml que ce soit dans la branche bytecode ou native.

On remarque donc les phases communes aux deux branches :

- Code source
- Analyse syntaxique : génération de l'AST [1] non typé
- Typage : inférence et vérification, génération d'un AST typé
- Traduction en code intermédiaire : forme simple proche du lambda-calcul où les types ont complètement disparu. Dans cette forme, les constructions de haut niveau disparaissent (les modules, les objets) et sont remplacées par des valeurs simples comme les pointeurs de fonction. Dans cette phase, le filtrage par motif est analysé et compilé.
  Cette phase est très importante, car c'est celle qui supprime l'information de type et spécialise le code source avec le modèle de représentation mémoire que nous avons vu à la section 2.5.

Après ces phases communes aux deux compilateurs OCaml (`ocamlc` et `ocamlopt`), la génération et l'optimisation du code intermédiaire n'engendrent pas de message d'erreur ou d'avertissement (dans la version 4.01.0). Ces phases de manipulation de structures intermédiaires permettent de factoriser le développement des différents compilateurs d'OCaml.

Compilateur Bytecode `ocamlc` et l'interpréteur `ocamlrun`

L'avantage d'utiliser ce compilateur bytecode est la portabilité du code. Le passage de la forme lambda vers le bytecode est assez simple. Un exécutable généré par ce compilateur a le désavantage d'être plus lent que s'il était généré par le compilateur natif.

Les étapes de compilation spécifiques d'`ocamlc` sont les suivantes :

- Génération du bytecode (instruction)

1. Arbre de syntaxe abstraite (*Abstract Syntax Tree*))

- Génération de l'exécutable et liaison avec l'interpréteur `ocamlrun`

Compilateur natif (`ocamlopt`)

Le compilateur natif `ocamlopt` compile des fichiers sources OCaml en code objet natif et les lie pour produire un exécutable.

Ce compilateur n'est disponible que sur certaines plateformes ; le code produit est plus rapide que celui produit par `ocamlc`, ceci ayant comme coût l'augmentation du temps de compilation, ainsi que la taille de l'exécutable.

Les étapes de compilation spécifiques d'`ocamlopt` sont les suivantes :

- CMM (génération de code), optimisation (conversion des fermetures, inlining inter-modules, etc.).
- Génération de code assembleur et génération de l'exécutable

Le code produit par `ocamlc` et `ocamlopt` s'exécutent de façon identique.

Comme nous l'avons dit précédemment, l'idée est de décorer l'arbre de syntaxe abstraite typé avec ces identifiants uniques pour chaque site d'allocation, jusqu'aux instructions d'allocation. Ces identifiants sont alors propagés durant les différentes phases de compilation, que nous venons de voir, pour être inclus dans les en-têtes des blocs.

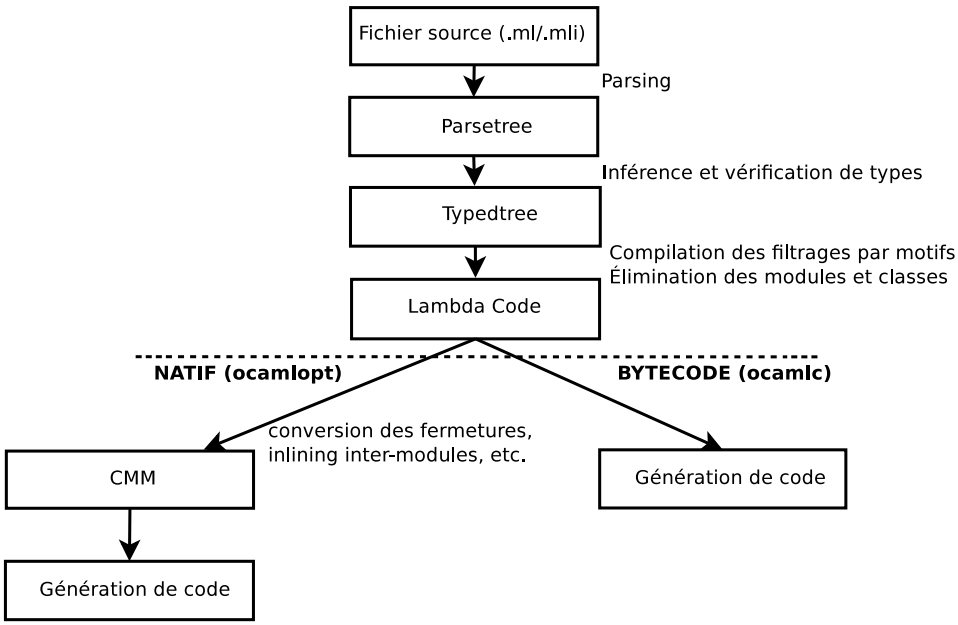

Figure 5.13 – Étapes de compilation du compilateur OCaml.

5.1.3.2 Chargement dynamique (`Dynlink`)

Le chargement dynamique permet de modules pendant l'exécution d'un programme peut être utile, si sous certaines conditions, le programmeur veut charger un module plutôt qu'un autre, ou dans un contexte plus général, cela permet de modifier le comportement d'un programme, pendant son exécution.

Ce qui nous intéresse ici, ce sont les points d'allocation dans ces modules chargés dynamiquement et les identifiants qui y sont associés. En effet, il est important de bien traiter ces identifiants pour ne pas qu'ils collisionnent avec ceux déjà initialisés.

Pour cela, les offsets des points d'allocation étant calculés statiquement lors de la compilation, lors du chargement du module, il suffira de faire comme expliqué précédemment et d'additionner avec la globale de base initialisée correctement.

5.1.3.3 Génération de la table des identifiants

La génération de la table des identifiants se fait dès le début de l'exécution de notre application. Celle-ci va générer un fichier de la forme `memprof.PID.process_info` organisé en sections (figure 5.14) qui contiendra essentiellement deux types d'informations :

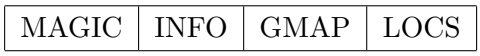

Figure 5.14 – Aperçu de l'organisation du fichier `process_info` en sections.

- **MAGIC**, une valeur permettant d'identifier une version de ce fichier. Cette valeur permet de suivre l'évolution du format de ce fichier durant le développement.
- Section **INFO** de type `raw_process_info` (figure 5.15). Cette section contient des informations génériques sur l'application comme son pid, la commande exécutée, etc.

```
1  type raw_process_info = {
2    rpi_pid : int ;
3    rpi_exe_name : string * string array ;
4    rpi_runparam : string option ;
5    rpi_value_size : int ;
6    rpi_native : bool ;
7  }
```

Figure 5.15 – Interface du type `raw_process_info` contenant les informations générique de l'application profilée.

- Section **GMAP** : contient le contenu exact de la globale C `caml_globals_map`. C'est une valeur sérialisée de type :
  — en bytecode :

```
type map = Ident.t Symtable.numtable
```

  — en natif :

```
type item = {
  name :string;
  interf_crc :Digest.t;
  implem_crc :Digest.t;
  syms :string list
}

type map = item list * string list
```

- Section **LOCS** de type `table_desc list`, contenant l'information sur le tableau des locations. Après lecture de cette liste de description, elle est transformée en un tableau de type `location_table` dont les indices sont des locids (figure 5.16).

```
type table_desc =
| Indirect of string * string * int (* fname,digest,size *)
| Direct of cmg_infos

type location_table = {
  lt_env :Env.t;
  lt_types :(Types.type_expr, string, string) block_info array;
  lt_inlined_locations :(string, string) block_location list
      array
}
```

Figure 5.16 – Interface du type `table_desc` contenant l'information sur le tableau des locations.

Avec ce système de sections, nous avons donc la possibilité d'en ajouter d'autres au fichier `process_info`, et ainsi, une compatibilité peut être assurée dans la lecture de ces fichiers.

Une fonction de lecture `read_process_info` dans le module `Heapdump_reader` permet de relire ce fichier et de renvoyer un type `process_info` qui a les propriétés de la figure 5.17, avec toutes les informations communes à tous les dumps.

```
type process_info = {
  pi_pid : int;            (* PID du processus *)
  pi_exe_name : string;    (* Nom de l'exécutable *)
  pi_argv : string array; (* Argument(s) de la commande exécutée
      *)
  pi_runparam : string option;
  pi_native : bool;
  pi_value_size : int;
  pi_dump_files : string list; (* Liste des fichiers de dumps *)
  pi_globals : Globals.map;    (* Les globales *)
}
```

Figure 5.17 – Interface du type `process_info` contenant l'information sur le tableau des locations.

5.2 Génération des instantanés

Maintenant que nous avons introduit les identifiants dans les en-têtes des blocs mémoire, nous avons besoin de les relire durant l'exécution, ou à la fin du programme.

Dans cette section, nous présentons une première technique basée sur la sauvegarde des *instantanés* (ou *snapshots*) qui représentent des sauvegardes de la mémoire OCaml dans des fichiers.

Concrètement, un instantané est un fichier contenant le graphe mémoire des blocs OCaml contenus dans le tas avec leurs identifiants. L'image est celle obtenue après une collection majeure complète. Le tas majeur d'OCaml est composé de chunks (voir section 2.5.3) qui contient un ensemble de blocs mémoire. Pour obtenir ces graphes, nous parcourons linéairement tous les chunks mémoires. Dans la suite, nous expliquons en détails le contenu de ces fichiers et comment les obtenir.

5.2.1 Déclenchement de la génération

Les instantanés sont principalement constitués d'une représentation compressée du tas mémoire. Pour que la génération de ces fichiers soit la moins coûteuse possible, le tas est parcouru linéairement de la même manière que la phase de nettoyage du ramasse-miette ; ce choix permet en prime de conserver aussi dans l'instantané, et à moindre coût, l'ensemble des blocs non-alloués et des listes d'allocations.

Nous avons mis en œuvre trois façons d'obtenir les images de la mémoire, selon le besoin. Dans un premier temps, une façon automatique de générer ces images, est fournie au travers d'une commande, que nous nommerons `ocp-memprof` dans la suite, et de la variable d'environnement `OCAMLRUNPARAM`. Pour obtenir une image, il suffit alors

d'exécuter son programme par :

```
ocp-memprof --exec ./executable.exe ARGS
```

qui se chargera de positionner le paramètre `m` de la variable d'environnement `OCAMLRUNPARAM`. Ce paramètre forcera le programme à produire un instantané après chaque collection majeure. Cette production régulière d'instantanés est très utile pour suivre l'évolution de l'utilisation mémoire d'une application donnée durant tout au long de son exécution. Cette méthode présente cependant l'inconvénient de produire rapidement des dizaines, voire des centaines de fichiers et, si l'occupation du mémoire du processus est conséquente, il est possible de rapidement saturer le disque dur.

Une deuxième méthode est de générer des fichiers à la demande. Pour cela, nous pouvons envoyer un signal `SIGHUP` au processus qui s'empressera de générer un instantané. Cela peut être très utile pour des applications de type serveur dont la durée de vie est infinie [2].).

Enfin, la possibilité d'annoter le code et de générer des instantanés à des endroits critiques du code est possible. Pour cela, il suffit d'utiliser la fonction disponible de notre bibliothèque (figure 5.18).

```
1  external dump : string -> unit = "caml_ml_dumpheap"
```

Figure 5.18 – Fonction de dump du tas

La chaîne en entrée permet de nommer un instantané et ainsi de profiler très précisément les fichiers que nous voulons. Par défaut, les instantanés générés après chaque collection majeur prendront la chaîne "after_major_collection" et "signal" pour ceux générés à l'aide d'un signal.

Ces techniques peuvent être paramétrées par plusieurs filtres (liste non-exhaustive) :

- un intervalle ou un ensemble de numéro de dumps : cela peut être très utile lorsque l'on veut profiler seulement une partie des dumps ou des dumps en particulier ;
- un minimum ou un maximum pour profiler à partir d'un dump ou jusqu'à un dump en particulier ;
- une taille minimum pour les dumps : seuls les dumps plus grands ou plus petits que cette taille seraient pris en compte ;
- la possibilité de profiler les dumps par la façon dont ils ont été générés : par un signal, manuellement dans le code source ou automatiquement après chaque collection majeur

2. Par exemple, MLDonkey ou des serveurs web

5.2.2 Contenu des instantanés

Ces fichiers sont principalement constitués d'une représentation compressée du tas mémoire. Pour que la génération des fichiers d'instantané soit la moins coûteuse possible, le tas est parcouru linéairement de la même manière que la phase de nettoyage du ramasse-miettes ; ce choix permet en prime de conserver aussi dans l'instantané, et à moindre coût, l'ensemble des blocs non-alloués et des listes d'allocations.

Les instantanés peuvent être reconstruits en mémoire (type `heap`, figure 5.21). Ils peuvent être ensuite explorés récursivement depuis les racines (`type roots`) à l'aide de fonctions non-décrites ici, mais similaires à celles du module `Obj` de la bibliothèque standard d'OCaml (type `value`) ; ou bien encore parcourus linéairement.

Ces fichiers contiennent aussi quelques informations annexes, comme les informations de statistiques du ramasse-miettes (le type `Gc.stats` usuel), la date de création de l'instantané et une table de correspondance entre noms de symboles et racines globales.

Le fichier est constitué d'un en-tête, d'un nombre variable de sections et se termine par un index des sections. Sur la figure 5.19, on peut voir la description de l'en-tête d'un instantané. Cet en-tête contient les informations suivantes :

`MAGIC` (12 octets) : *"Caml2014H001"*

`VALUE_SIZE` (1 octet) : 32 ou 64, nombre de bits dans un mot mémoire

`LOCID_SIZE` (1 octet) : inférieur ou égale à 22, nombre de bits pour représenter le plus grand locid.

`NATIVE` (1 octet) : 0 pour le bytecode, 1 pour le natif

`ZERO` (1 octet) : 1 octet zéro

Figure 5.19 – Aperçu de l'en-tête d'un instantané.

Actuellement, les sections sont comme décrit sur la figure 5.20 :

Figure 5.20 – Aperçu de l'organisation d'un instantané organisé en sections.

On trouve alors :

- l'en-tête ;
- la table des chunks ;
- la table des chaînes de caractères ;

- le contenu des chunks ;
- la liste des racines ;
- les informations sur l'instantané
- quelques statistiques utiles, dont `Gc.stat` ;
- l'index des sections.

Contenu des chunks Les chunks sont dans l'ordre de la table des chunks.

Chaque chunk est constitué de blocs et de «*freeblocks*», dans l'ordre d'apparition dans la mémoire. Ainsi, l'adresse mémoire d'un bloc peut être calculée en ajoutant à l'adresse du début de son chunk les tailles des blocs et freeblocks qui le précèdent.

Le premier octet de la représentation d'un bloc commence par 0 ; celui d'un freeblock commence par 1.

Les freeblocks sont encodés sur 1 ou 5/9 octets. Si le premier octet est 0xFF, le mot mémoire suivant contient la taille du freeblock. L'octet de poids faible est en premier. Sinon, la taille, inférieure à 127, est encodée dans les 7 bits restants du premier octet.

La représentation d'un bloc est contituée d'un en-tête (de 4, 5 ou 8 octets), suivi, pour les blocs dont le tag est inférieur à la valeur de `No_scan_tag`, de la représentation en séquence de ses champs. Les champs sont représentés par un nombre variable d'octets, compris entre 1 et 9.

Les trois représentations possibles pour les en-têtes de blocs sont les suivantes, où les octets de poids fort sont écrits en premier.

010 . size (25-locid_size) . locid(locid_size) . tag (4bits)

011 . size (29-locid_size) . locid(locid_size) . tag (8bits)

00 . size (32bits) . locid(22bits) . tag (8bits)

Les champs des blocs sont soit des constantes sur 1 octet, soit la différence entre l'adresse du bloc cible et celle du bloc courant. Cette différence est encodée avec un nombre variable d'octets, compris entre 1 et 9. En complément à 2 avec l'octet de poids fort en premier :

- 0 . offset (7bits) = 1 octet
- 10 . offset (14bits) = 2 octets
- 110 . offset (21bits) = 3 octets
- 1110 . offset (28bits) = 4 octets
- 1111_0 . constant (3bits) = 1 octet
  - 7 : Pointeur de code (application partielle, bytecode)
  - 6 : Pointeur de code (bytecode)

  - 5 : Pointeur en dehors du tas d'OCaml
  - 4 : Pointeur vers une valeur allouée statiquement
  - 0-3 : Entier (resp. 1-4 champs entiers successifs)
- 1111_10 . offset (34bits) = 5 octets
- 1111_110 . offset (41bits) = 6 octets
- 1111_1110 . offset (48bits) = 7 octets
- 1111_1111 . offset (1 mot mémoire) = 5/9 octets

Table des chunks

`BLOCK_COUNT` (1 mot mémoire) : nombre de blocs alloués dans le tas

`CHUNKS_COUNT` (1 mot mémoire) : nombre de chunks constituant le tas

Suivi de 2 mots par chunk :

`CHUNKS_ADDR` (1 mot mémoire) : adresse de début du chunk

`CHUNKS_SIZE` (1 mot mémoire) : taille du chunk en octets

Racines globales Bien sûr, nous sauvegardons les racines dans chaque instantané.

La section se termine par un mini-index de 6 mots-mémoire, comprenant respectivement le nombre de racines :

`hp_globals :` les racines globales

`hp_dyn_globals :` les racines dynamiques qui n'existent pas en bytecode.

`hp_stack :` la pile

`hp_C_globals :` les racines globales en C

`hp_finalised_values :` valeurs à finaliser

`hp_hook :` autres racines, en général, les piles des autres threads

En bytecode, la liste des racines commence par un mot contenant l'adresse du bloc `caml_global_data`. En code natif, elle commence la description en séquence des blocs constituant les racines «statiques». Chaque bloc est constitué de :

`BLOCK_SIZE` (1 mot mémoire) : la taille du bloc

`BLOCK_FIELDS` (`BLOCK_SIZE` * 1 mot mémoire) : le contenu brut du champ (valeur immédiate ou adresse mémoire)

Section *INFO* Si nous revenons de façon plus détaillée sur le `type info` champ par champ, nous avons :

`hp_pid` : l'identifiant du processus ;

`hp_dump_number` : dans le cas de la production de plusieurs instantanés lors d'une même exécution, ce champ permet de retrouver l'ordre dans lequel les instantanés ont été produits ;

`hp_kind` : indique la nature de l'évènement ayant déclenché la génération de cet instantané : fin d'un GC majeur, signal ou appel explicite à la fonction `dump` (cf. section 5.2.1) ;

`hp_gc_stat` : informations de statistiques du ramasse-miettes au moment de la génération ;

`hp_dump_start_time / hp_dump_end_time` : date de début et de fin de géneration, exprimée en *temps processus*.

Index des sections L'index des sections se situe à la fin des fichiers. Le dernier mot du fichier est le nombre de section (en commençant par l'octet de poids faible).

Précédant ce dernier mot, on trouve par section :

`SECTION_NAME` (4 octets) : nom de la section

`SECTION_OFFSET` (1 mot mémoire) : position dans le fichier (*little-endian*)

```
type value
type heap

type kind = Major_gc | Signal | User

type info = {
  hp_pid : int;
  hp_dump_number : int;
  hp_kind : kind;
  hp_gc_stat : Gc.stat;
  hp_dump_start_time : float;
  hp_dump_end_time : float;
}

type roots = {
  hp_globals : (string * value array) array;
  hp_dyn_globals : value array;
  hp_stack : value array;
  hp_C_globals : value array;
  hp_finalised_values : value array;
  hp_hook : value array;
}

type dump = {
  hp_filename : string;
  hp_info : info;
  hp_roots : roots;
  hp_heap : heap;
  hp_globals_map : Heapdump_globals.map;
}
```

Figure 5.21 – Format des instantanés.

"Un programme qui manipule un grand nombre de données le fait d'un petit nombre de manières."

Alan Jay Perlis

6

Traitement des données

Dans le chapitre précédent, nous avons introduit la notion d'identifiant nous permettant de marquer chaque bloc mémoire par son site d'allocation dans le code source ainsi que diverses informations comme son type.

Ce chapitre présente les techniques d'exploitation des informations recueillies. L'exploitation des instantanés nécessite en effet la conception d'outils de relecture des graphes mémoire, de navigation dans ces graphes, de reconstruction et d'examen d'informations comme le type, le site d'allocation, mais aussi le module et la fonction contenant ce site d'allocation. Une première représentation de ces résultats a été un graphe temporel qui montrait le volume de chacun des ensembles de valeurs allouées par un même site, dans le tas, au fil des GCs. Le graphe obtenu permet, dans certains cas, de détecter les sources de problèmes mémoire. Cet affichage n'exploitant pas le graphe mémoire, il était difficile de comprendre l'atteignabilité d'une valeur en mémoire. Nous avons donc complété cet affichage par une représentation du graphe mémoire qui permet d'observer la taille retenue par chaque nœud en partant des racines. Il est alors possible de naviguer dans le graphe et d'avoir cette information pour chacun des nœuds.

Dans ce chapitre, nous allons détailler comment nous exploitons ces identifiants pour mieux comprendre le comportement mémoire et tenter de détecter certains problèmes comme la surconsommation ou les fuites de mémoire. Nous détaillerons l'implantation de

notre outil `ocp-memprof` nous permettant de lier chaque bloc à son type et d'afficher ces valeurs au fil du temps d'exécution. Cet outil nous permet de visualiser la consommation mémoire des valeurs en mémoire, agrégées de différentes manières de sorte à visualiser le problème sous différentes formes.

Ensuite, nous détaillerons certaines approches concernant des outils en cours de prototypage pour faire du profiling continu, c'est-à-dire durant l'exécution du programme. Ce profiling continu nous permet de comprendre l'évolution des valeurs en temps réel, comme par exemple les valeurs promues du tas mineur vers le tas majeur ou les valeurs allouées directement dans le tas majeur.

Enfin, nous expliquerons une approche basée sur la durée de vie des valeurs légèrement différente de celles vues avec *Sleigh* et l'approche des conteneurs de la section 4.3. En effet, nous avons vu que se baser sur la seule idée de la durée de vie nous mène à des faux-positifs. En regardant de plus près chaque bloc sur les écritures et les lectures de ces blocs, on peut mieux comprendre ce qui se passe en mémoire ainsi que dans le programme.

6.1 Navigation post-mortem dans le tas

Pour comprendre le comportement mémoire d'une application OCaml, la première chose à faire est de pouvoir naviguer dans celle-ci. Pour cela, nous avons opté, dans un premier temps, pour la navigation post-mortem dans le tas. Grâce aux instantanés générés d'une des manières vues dans la section 5.2, nous allons pouvoir naviguer dans la mémoire pour récupérer diverses informations comme le graphe avec les pointeurs, les globales, la quantité de blocs en mémoire à un moment donné, la taille de ces blocs, et également les identifiants de sites d'allocation que nous avons introduits dans la section 5.1.

6.1.1 Représentation d'un tas (instantané)

Sur la figure 5.21, nous avons vu le format des instantanés. Chaque fichier de dump contient son nom, les informations comme son pid, des statistiques sur le ramasse-miettes, comment il a été généré (après les collections majeures, par un signal ou avec la fonction de dump). Chacun des dumps garde également la liste des racines du programme.

Pour faciliter le chargement de plusieurs fichiers de dumps, sur la figure 6.1, nous avons deux fonctions de lecture : pour un dump avec `read_dump`, ou pour plusieurs dumps avec `read_session`. Des fonctions d'itérations sont aussi disponibles pour pouvoir naviguer dans les dumps.

```
1          (* Extract from Heapdump_reader.mli *)
2
3   val read_dump : string -> dump
4
5   type session = {
6     session_pi : process_info;
7     session_dumps : dump list;
8   }
9
10  val read_session : string -> session
11  val iter_session : (process_info -> dump -> 'a) -> session ->
        unit
12  val map_session : (process_info -> dump -> 'a) -> session -> 'a
13  list
```

Figure 6.1 – Fonctions de lecture et d'itérations sur les dumps.

6.1.2 Les valeurs dans le tas

Après avoir relu tous les tas, il est possible de naviguer dans la mémoire, bloc par bloc et ainsi de récupérer diverses informations. Une valeur est soit une constante soit un pointeur vers un bloc. Sur la figure 6.2, avec une première interface de bas niveau (à la `Obj`), il est possible pour une valeur de récupérer son `tag`, indiquant la nature du bloc. Les tags contiennent une information de type minimale pour distinguer notamment les blocs contenant une chaîne de caractères, une valeur fonctionnelle, un objet, etc.

La fonction `size` permet de récupérer la taille du bloc en mots mémoire.

Il est également possible de récupérer le contenu des blocs en utilisant les fonctions `field` et `fields`. Cela peut permettre d'inspecter chaque champ des blocs pour récupérer des informations plus détaillées.

Enfin il est possible de récupérer les identifiants de sites d'allocation, ou *locid*, stockés dans les en-têtes des blocs. Cette information nous permettra, comme expliqué précédemment, de relier chaque bloc à un site d'allocation dans le code source du programmeur et de retrouver le type de chaque bloc par exemple. Une *locid* pouvant se retrouver dans l'en-tête de plusieurs blocs, s'il y a plusieurs allocations dans le même site d'allocation, un identifiant de bloc nous permet également de tracer chacun d'eux de façon unique, indépendamment de son locid.

Après avoir relu les dumps et navigué dans la mémoire, nous allons pouvoir travailler sur chacun des blocs mémoire pour les décorer avec des informations pertinentes, comme leur type, le site d'allocation, etc.

```
1          (* Extract from Heapdump_reader.mli *)
2  (* Constant or (symbolic) pointer to a block allocated in heap.
       *)
3   type value = int
4
5   (* Blocks *)
6   val tag : heap -> value -> int
7   val size : heap -> value -> int
8   val field : heap -> value -> int -> value
9   val fields : heap -> value -> value array
10  val locid : heap -> value -> locid
11  val blockid : heap -> value -> blockid
```

Figure 6.2 – Informations que l'on peut récupérer sur les blocs pour un tas donnée (interface à la Obj).

6.1.3 Informations sur les blocs par site d'allocation

Dans cette approche avec les *locids*, il devient alors possible d'avoir des informations très précises sur chaque bloc. Depuis une locid, il va alors être possible de récupérer des informations sur son site d'allocation, son type, mais aussi le module dans lequel la valeur aura été allouée ainsi que la fonction dans laquelle elle aura été allouée.

Pour cela, sur la figure 6.3, la fonction `static_types` est la fonction nous permettant d'associer ces informations à chacun des blocs du tas (type `heap`) à l'aide de la table des sites d'allocation (figure 5.16). Ce sont les types statiques associés aux locids par le compilateur. La fonction `reconstructed_types` est la fonction qui raffine les types statiques en faisant de la propagation de types à travers le graphe mémoire.

```
1   val static_types :
2       location_table -> heap -> block_info_id ArrayMap.t
3  val reconstructed_types :
4       location_table -> heap -> block_info_id ArrayMap.t
```

Figure 6.3 – Fonctions `static_types` et `reconstructed_types` permettant d'associer les informations de types et de site d'allocation à chaque bloc du tas.

Avec un contexte donné (figure 6.4) et un identifiant de bloc `block_info_id`, la récupération des informations concernant un bloc est alors simple. À l'aide de la fonction `get_block_info`, on peut alors récupérer le `block_info` (figure 6.5) qui représente ces blocs décorés.

Le type `block_info` contient l'information sur le site d'allocation dans le source

```
type Types.type_expr context = {
  ctx_types : Types.type_expr array;
  ctx_block_infos : (type_id, module_id, function_id) block_info
      array;
  ctx_modules : string array;
  ctx_functions : string array;
}

val get_context : unit -> Types.type_expr context

val get_block_info : Types.type_expr context -> block_info_id ->
                     (Types.type_expr, string, string) block_info
```

Figure 6.4 – Interface du type `context` nous permettant de récupérer une valeur de type `block_info`.

(`block_location`), le type du bloc (`block_type`) et la nature du bloc (`block_kind`, qui peut être une valeur, un module, un objet, etc.). Le type `block_location` en plus de l'information sur le site d'allocation et de la locid du bloc, aura le module et la fonction qui contient ce site d'allocation.

```
    (* Extract from memprof_types.mli *)

type (Types.type_expr, string, string) block_info =
  { block_loc : (string, string) block_location;
    block_type : Types.type_expr block_type;
    block_kind : block_kind;
  }

and Types.type_expr block_type

and block_kind

and (string, string) block_location =
  { block_locid : Heapdump_reader.locid;
    block_location : Location.t;
    block_module : string;
    block_function : string;
  }
```

Figure 6.5 – Interface du type `block_info` nous permettant de récupérer diverses informations sur les blocs en mémoire.

Toutes ces informations disponibles dans plusieurs dumps, vont nous permettre de

trier les valeurs dans le tas de différentes manières (taille des blocs, nombre d'occurrences d'un type de bloc, etc.) et d'afficher sous forme de graphe l'évolution de ces valeurs au fil des dumps. En effet, l'affichage de ce graphe est alors trivial. Il suffit de parcourir chacun des dumps et de récupérer ces informations sur les blocs.

6.1.3.1 Agrégation des valeurs

Une fois ces informations récupérées, d'après nos expérimentations, il faut pouvoir regrouper ces valeurs selon différents critères, comme le tri par taille en mémoire pour mettre en valeur les blocs prenant le plus de place.

Tri par taille `ocp-memprof` propose deux critères de tri nous donnant deux informations différentes. En effet, pour un type donné, il peut y avoir peu de blocs en mémoire, mais avec une taille importante, et au contraire il peut y avoir un nombre élevé de bloc ne prenant pas beaucoup de place en mémoire.

Le fait d'afficher par taille en mémoire nous donnera une information très utile pour comprendre l'occupation mémoire des valeurs. En effet, lors d'un profiling, on pourra apporter une attention particulière à ces valeurs prenant une place importante en mémoire. Ce sont ces valeurs qui seront ciblées dans un premier temps.

L'affichage par occurrence des blocs, nous apportera l'information du nombre d'allocations par site. Lors d'un profiling, il peut être intéressant de comprendre le nombre d'allocation faite pour un site donné.

Le tri des valeurs par leur *taille en mémoire* (en mots) est le critère choisi par défaut par `ocp-memprof`. Il permet d'afficher les valeurs triées selon la taille en mémoire de chaque bloc. Par défaut, les valeurs étant agrégées avec leur type, il suffira alors de regrouper chacun de ces types, d'additionner leur taille et d'afficher le total.

Le tri des valeurs par le nombre d'occurrences des blocs est un deuxième critère. Il permet de regrouper les valeurs en comptant le nombre de blocs vivants pour un type donné.

Sur la figure 6.6, le type `agregate` nous permet de récupérer simplement ces deux informations stockées dans les champs `sizes` et `occurrences`.

Par type, module, fonction ou site d'allocation
En plus des deux critères de taille, il y a la possibilité de combiner ce processus à d'autres types d'agrégation, comme l'agrégation par type, module, fonction ou son site d'allocation dans le code source.

```
type agregate =
  { key : string ;
    sizes : int array ;
    occurrences : int array ;
  }
```

Figure 6.6 – Interface du type `agregate`, nous permettant d'agréger les valeurs par leur taille en mémoire ou leur nombre d'occurrence.

L'agrégation par type est celle par défaut dans `ocp-memprof`. Elle permet, pour une méthode de tri des tailles, d'agréger les valeurs par leur type.

L'agrégation par module permet de regrouper les sites d'allocation d'un module. Toutes les allocations faites dans un module donné seront regroupées et triées selon la méthode de tri choisie.

De la même manière, il y a la possibilité de regrouper les valeurs par la fonction qui les alloue. Cela permet de descendre d'un niveau par rapport au module et de donner des informations concernant le nombre d'allocations faites par une fonction donnée.

Enfin, l'agrégation la plus précise est celle par locid. Elle permet de regrouper les valeurs par leur site d'allocation dans le code source. Pour un type donné, chaque site d'allocation sera séparé. Cela devient très utile lorsque l'on tente de comprendre de façon très précise les allocations faites pour chaque valeur.

6.1.3.2 Reconstruction de types

En générant les identifiants d'allocation directement sur l'arbre de syntaxe typé, on peut retrouver lors de l'analyse d'un tas mémoire à la fois l'emplacement dans le code source de cette valeur et son type. En pratique, cette information de type est souvent partielle : elle ne correspond qu'au type *statique* de la valeur. Par exemple, dans le cas d'allocation ayant lieu dans une fonction polymorphe telle que `List.map` l'information de type associée au point d'allocation sera α `list`. L'information sur le paramètre de type n'est pas connue statiquement.

La même difficulté apparaît avec les foncteurs : l'information de type associée à un point d'allocation situé à l'intérieur du corps d'un foncteur ne contient pas d'information sur les paramètres du foncteur. Cela peut s'avérer gênant par exemple avec les foncteurs instanciant des «conteneurs», tel que `Map.Make` : si dans l'analyse d'un tas mémoire une grande partie du tas est constituée de blocs alloués dans `Map.Make`, il n'est pas possible de distinguer simplement si ces blocs proviennent d'une même instanciation du foncteur ou si elles proviennent de plusieurs instancations distinctes (`IntMap` ou `StringMap`).

Pour pallier cette limitation, notre outil permet d'effectuer, optionnellement et sur un instantané donné, une passe de propagation des types. En effet, en grande majorité, les blocs alloués par du code polymorphe sont soit accessibles depuis un bloc alloué dans du code monomorphe, soit contiennent eux-même des blocs alloués dans du code monomorphe. En propageant ces informations de proche en proche (unification), il est possible de reconstruire une grande partie des informations manquantes.

Pour effectuer cette propagation, nous avons modifié le compilateur OCaml afin qu'il exporte pour chaque module l'environnement « concret » des définitions des définitions de types internes.

Ce travail peut être complété en tentant de retrouver des informations de types manquants, en partant des racines du programme [47].

6.2 Profiling continu

Une autre approche pour exploiter les locids consistent à profiler une application pendant son exécution. Avec cette approche, il est alors possible d'obtenir d'autres informations qui ne sont pas disponibles dans les instantanés. En plus d'obtenir les informations sur les blocs dans le tas majeur, il va alors être possible de tracer les valeurs du tas mineur, celles promues du tas mineur vers le tas majeur, celles allouées directement dans le tas majeur, etc.

Grâce à cela, on aura un historique complet des allocations et désallocation.

6.2.1 Valeurs dans le tas mineur et majeur

L'idée du profiling continu est de garder en mémoire un certains nombres de tableaux indexés par les locids. On pourra alors avoir deux premiers tableaux des valeurs allouées dans le tas mineur et celles allouées dans le tas majeur. Il sera alors possible de suivre l'évolution en temps réel des valeurs disponibles dans chacun des tas.

6.2.2 Les valeurs dans le tas majeur

Grâce aux deux tableaux précédents, on va pouvoir afficher l'information des valeurs qui auront été promues du tas mineur vers le tas majeur. La différence entre les valeurs dans le majeur et celles promues indique les valeurs directement allouées dans ce dernier.

Cette information devient alors utile pour suivre l'évolution des valeurs qui survivent dans le tas majeur.

Avec le profiling continu, un surcoût durant l'exécution est à noter. Le travail étant en cours de prototypage, nous n'avons pas encore de mesure. Cependant, les dumps sont plus rapides : en effet, il suffit alors de dumper les différents tableaux et d'afficher leurs contenus. Cela suffit pour reconstruire l'histoire des allocations et désallocations des valeurs de notre application.

6.3 Parcours du graphe mémoire

Dans la section 5.2, nous avons vu en détail le contenu des instantanés, plus précisément, nous avons vu qu'ils contenaient le graphe mémoire.

Le tas étant parcouru linéairement de la même manière que la phase de nettoyage du ramasse-miettes, cela nous permet de conserver aussi dans l'instantané, et à moindre coût, l'ensemble des blocs vivants et la freelist.

6.3.1 Analyse des blocs libres et des blocs vivants

En analysant les instantanés, on peut ainsi faire des analyses sur le ratio entre les blocs libres et les blocs vivants. Dans certains cas, lorsque le tas est fragmenté, l'allocateur peut passer un temps non négligeable à la recherche de blocs libres pour allouer une valeur.

Nous nous sommes concentrés sur l'analyse de la distribution des blocs par leur taille, dans le tas (blocs vivants) et dans la freelist (blocs libres/morts) pour mieux comprendre les options que nous pourrions utiliser pour le ramasse-miettes. En mettant en avant la fragmentation du tas et le temps passé par l'allocateur, certaines réflexions sont faites sur le format de la freelist qui pourrait être organisée différemment. Nous expliquerons cela en détail dans la suite.

`ocp-memprof` permet de visualiser cette distribution des blocs libres/vivants. Sur la figure 6.7, on peut voir un extrait coupé, de la distribution des blocs pour le programme `why3` sur un exemple critique, dans le cadre du projet ANR *BWare*.

Le tableau est trié par la taille des blocs en mémoire. Dans la première colonne, il y a la taille des blocs, dans la deuxième colonne le nombre de blocs libres présents dans la freelist pour une taille donnée et dans la dernière colonne le nombre de blocs vivants dans le tas pour une taille donnée. Sur la figure 6.7, il est intéressant de noter que la plupart des allocations sont de tailles inférieures ou égales à 6 : elles représentent 96,19% des allocations.

Cette information peut être utilisée pour simuler le coût de l'allocation pour une application, en modifiant le comportement de l'allocateur.

Block Size	Free Blocks		Live Blocks		Cumul Live
1	50662	27.89%	114002	22.33%	22.33%
2	50446	27.78%	159266	31.20%	53.53%
3	18212	10.03%	24628	4.82%	58.35%
4	32543	17.92%	77676	15.22%	73.57%
5	14667	8.08%	82879	16.24%	89.81%
6	4196	2.31%	32557	6.38%	96.16%
7	1255	0.69%	5820	1.14%	97.33%
8	1263	0.70%	5602	1.10%	98.43%
9	30	0.02%	286	0.06%	98.49%
12	4	0.00%	109	0.02%	98.51%
13	38	0.02%	2660	0.52%	99.03%
14	38	0.02%	2668	0.52%	99.55%
20	0	0.00%	32	0.01%	99.56%
21	0	0.00%	2	0.00%	99.56%
22	0	0.00%	79	0.02%	99.58%
23	0	0.00%	82	0.02%	99.60%
35	0	0.00%	27	0.01%	99.61%
[...]					
58	0	0.00%	30	0.01%	99.62%
[...]					
63 & more	1	0.00%	373	0.07%	100%
	181621		510484		692105

Figure 6.7 – Tableau représentant la distribution de blocs libres et vivants sur un fichier de dump données.

Heuristique pour l'allocateur

Sur la figure 6.8, un exemple de code n'allouant que des blocs de taille 3 et 5. Deux tableaux sont créés pour stocker ces valeurs. Un de ces tableaux est maintenu vivant jusque la fin du programme à l'aide d'une globale.

Lorsque nous analysons le graphe mémoire de ce programme (figure 6.9), à la fin de l'exécution, la grande majorité des objets alloués sont de type `Alloc.t` (environ 76Mo sur les 91Mo, soit 82% de la mémoire).

Lorsque nous regardons le temps mis par le programme pour s'exécuter :

```
$time ./alloc.asm
        1.97s user 0.05s system 99% cpu 2.027 total
```

```
type t = T3 of int * int * int | T5 of int * int * int * int * int

let size_glob = 200
let minor_heap_size = 43690

let glob = Array.make size_glob [||]

let create_dummy_val i =
   if i mod 2 = 0 then T3 (i, i + 1, i + 2)
   else T5 (i, i + 1, i + 2, i + 3, i + 4)

let () =
    for i = 0 to size_glob do
      let arr1 = Array.make minor_heap_size (T3 (0, 0, 0)) in
      let arr2 = Array.make minor_heap_size (T3 (0, 0, 0)) in
      for j = 0 to minor_heap_size - 1 do
        arr1.(j) <- create_dummy_val i;
        arr2.(j) <- create_dummy_val (i + j);
        glob.(i) <- arr2
      done
    done
```

Figure 6.8 – Extrait du module `Alloc` n'allouant que des blocs de taille 3 et 5, dans le but de fragmenter le tas.

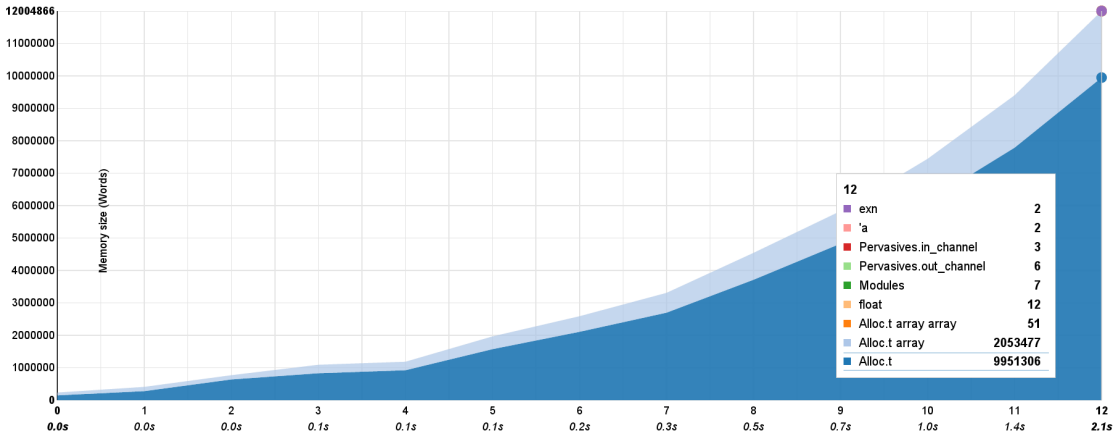

Figure 6.9 – Graphe mémoire obtenue sur le code de la figure 6.8. La majorité des valeurs allouées sont bien de type `Alloc.t`.

il met environ 1.97s.

En lançant notre analyse sur la distribution des blocs libres/vivants, on obtient le

résultat de la figure 6.10. Les résultats confirment qu'il y a bien eu essentiellement que des blocs de taille 3 et 5.

```
-------------------------------------------------------------------
| Block Size |    Free Blocks    |    Live Blocks    | Cumul Live |
-------------------------------------------------------------------
|          1 |  211841  19.28%   |       8  0.00%    |      0.0%  |
|          2 |      12  0.00%    |       3  0.00%    |      0.0%  |
|          3 |  712286  64.82%   |  988527  49.72%   |    49.72%  |
|          4 |       9  0.00%    |       0  0.00%    |    49.72%  |
|          5 |  174760  15.90%   |  999533  50.28%   |      100%  |
                     [...]
-----------------------------------------------------
|            |         1098917   |         1988120   ||  3087037
-----------------------------------------------------
```

Figure 6.10 – Tableau obtenu avec `ocp-memprof` sur la distribution des blocs libres et vivants sur le code de la figure 6.8. Les blocs de taille 3 et 5 sont bien mis en évidence sur ces résultats.

Dans cet exemple, le fait de n'allouer que des blocs de taille 3 et 5 et de maintenir un de ces tableaux, fragmente le tas. On se retrouve avec des blocs libres et parfois le fait d'allouer un bloc de taille 3 dans un espace de taille 5, va augmenter le taux de fragmentation.

Cet exemple étant écrit pour montrer la fragmentation et la durée de vie des valeurs dans le tas mineur, ici, nous savons qu'en augmentant la taille de ce dernier, la plupart des valeurs de notre tableau `arr1` ne vont pas survivre et être déplacées dans le tas majeur. Par conséquent, l'application n'aura pas de déplacement à effectuer et le tas ne sera alors pas fragmenté, ce qui fera que l'allocateur ne passera pas du temps à chercher un emplacement libre dans la freelist pour allouer les futures valeurs.

Dans l'analyse de certaines applications, certains blocs de taille bien précise sont rapidement mise en évidence avec ce tableau. Le temps passé par l'allocateur pour trouver un espace libre dans la mémoire est parfois non négligeable, car tous les blocs libres de toutes tailles confondues sont dans la même freelist.

L'allocateur d'OCaml ne faisant pas cette distinction, la recherche peut parfois être très coûteuse. Un allocateur avec une stratégie de malloc (voir section 1.5.5), diminuerait considérablement ce temps de recherche. Dans notre exemple, le fait que plus de 99% des blocs alloués ne sont que de taille 3 et 5, il suffirait alors de ne regarder que les freelists pour ces tailles là et de prendre les slots disponibles. De plus, cela éviterait la fragmentation, car l'allocateur renverra à chaque fois un bloc de la bonne taille.

Pour notre exemple de la figure 6.8, le fait d'augmenter la taille du tas mineur nous permet de gagner en temps d'exécution :

```
$time OCAMLRUNPARAM=s=4M ./alloc.asm
                0.69s user 0.04s system 92% cpu 0.799 total
```

Le temps d'exécution du programme passe de 1.97s à 0.69s.

6.3.2 Analyse du graphe mémoire

Dans la même idée que le profiler de tas du navigateur Google Chrome [34], nous nous sommes intéressés à la représentation du graphe mémoire d'une application OCaml, et ainsi mettre en évidence certains points importants à l'aide du graphe mémoire. Pour cela, différentes représentations sont possibles.

Dans le profiler de Google Chrome, une représentation sous forme de tableau permet d'afficher le graphe mémoire d'une application et de trier les valeurs selon différents critères. L'affichage par défaut permet de traquer les objets et leur utilisation mémoire, en fonction de leur type agrégé selon le nom de constructeur de type (*i.e.* `Array`, `Window`, etc.). Il est alors possible de visualiser la taille de l'objet lui-même (*shallow size*), la taille retenue par cet objet dans le graphe mémoire ; cette valeur représente la taille de la mémoire qui serait libérée en cas de collecte par le ramasse-miettes (*retained size*), la distance par rapport aux racines du programme et le nombre d'occurrences de cet objet dans la mémoire.

En changeant les vues, il est également possible d'afficher le graphe des dominateurs [32, 31]. Un objet D est un dominateur d'un objet `A` si tous les chemins depuis les racines du programme vers `A` passent par `D`. Cela signifie que retirer ce dominateur `D` de la mémoire va rendre `A` inaccessible et il sera ainsi collecté.

Un instantané de la mémoire contenant le graphe mémoire d'une application OCaml, il est alors possible d'afficher le même genre d'informations et ainsi avoir une vue de ce graphe.

Dans un premier temps, nous avons commencé par parcourir chacune des racines de notre programme et afficher la taille de chaque nœud dans le graphe, ainsi que la taille retenue par chacun des nœuds. Cette information est très utile lorsque l'on traque les fuites mémoire. Cela permet de comprendre le lien entre les valeurs dans le graphe mémoire et surtout, après avoir identifié les valeurs les plus allouées dans le programme, il est intéressant de comprendre, quels objets retiennent ces valeurs. Dans le chapitre 7, lors du profiling d'une application, ce graphe nous a permis de confirmer la correction d'un bug en mettant en évidence un nœud dans le graphe qui retenait 99,6% de la

mémoire.

Avec l'ajout des dominateurs, les informations obtenues pour une application OCaml seraient alors plus pertinentes. Le fait d'identifier les dominateurs de certains nœuds critiques, est alors une information pertinente pour comprendre la cause du maintien de ces nœuds en mémoire.

6.4 Durée de vie d'une valeur

La modification apportée aux en-têtes de nos blocs permettent de récupérer à moindre coût, le site d'allocation d'une valeur, son type et d'autres informations (voir section 5.1 pour plus de détails).

Dans la même idée que *Sleigh* en Java (section 4.3.1), notre travail pourrait être étendu pour rajouter une information sur l'âge des blocs. En effet, il serait possible de compter la date de relecture d'un bloc. Cela permettrait de mettre en valeur les sites d'allocations dont les valeurs ne sont pas lues depuis une longue durée. De plus, il pourrait y avoir un compteur pour le nombre d'écritures sur un bloc. De cette manière, une différence pourrait être apportée sur les blocs qui sont écrits mais non relus depuis un certains temps, et les blocs qui ne sont ni relus ni écrits.

Pour cela, comme modification dans l'en-tête, il suffira d'un bit pour chacune des deux informations. En effet, après chaque dump, le bit correspondant à la lecture et/ou l'écriture serait remis à 0 si ce bloc aura été relu ou écrit. Notre outil pourra alors, entre chaque dump, compter le nombre de fois où ce bit n'a pas été modifié, donc ni relu ou écrit. Avec cette méthode, il n'y aura aucun impact au niveau du surcoût mémoire. En effet, on pourrait réduire le champs pour les identifiants de site d'allocation de 2 bits pour stocker ces informations.

Cependant, comme avec Sleigh, il y aura un impact au niveau des performances du programme. Il faudra alors aller modifier chaque endroit du compilateur faisant une lecture sur un bloc ou une écriture. Donc, chaque lecture, fera en plus une modification dans l'en-tête du bloc lu pour modifier le bit de lecture et pareil pour l'écriture. En plus de cela, après chaque dump, tous les compteurs doivent être remis à 0 par la runtime.

Une autre technique consisterait à stocker le compteur de «non-lecture», mais pour cela il faudrait plus qu'un bit pour obtenir une information utile. Cela éviterait néanmoins de remettre à 0 les compteurs de tous les blocs après chaque dump.

"La beauté est dans les yeux de celui qui regarde."
Oscar Wilde

"Le mauvais exemple est contagieux."
Sophocle

7

Affichage des données et cas d'étude

Dans ce chapitre, nous présentons trois cas d'études, où l'utilisation d'`ocp-memprof` a permis de rapidement résoudre des problèmes mémoire : que ce soit une fuite mémoire ou un problème d'allocations inutiles. Nous avons pu dans chacun de ces exemples apporter une correction.

Ces exemples ont aussi permis de valider certaines hypothèses que nous avions faîtes lors de la conception d'ocp-memprof. Ainsi, le nombre d'identifiants de points d'allocation est de 55 000 pour Alt-Ergo et de 64 000 pour Cumulus, et n'utilisent donc que 16 bits par rapport au 22 bits réservés dans l'en-tête. On observe aussi dans les deux exemples une compression des instantanés qui les ramène à 10 à 20% de la taille mémoire utilisée, et cela bien qu'ils contiennent le graphe complet des pointeurs dans le tas (graphe utilisé, entre autres, pour la reconstruction des types).

7.1 Visualisation des données

7.1.1 Visualisation du graphe de la consommation mémoire

La visualisation des informations présentes dans les instantanés nécessite la reconstruction et l'affichage d'informations pertinentes relatives aux graphes mémoire sauvegardés. L'outil `ocp-memprof` effectue la relecture des instantanés et produit des graphes

agrégés de différentes manières.

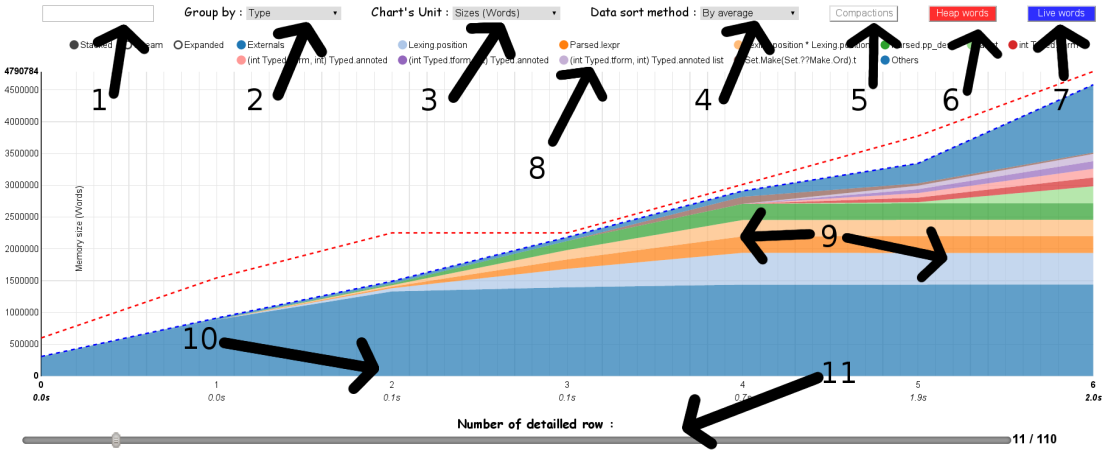

Figure 7.1 – Aperçu du graphe mémoire avec différentes options pour manipuler les informations et les afficher différemment.

Sur la figure 7.1, on peut voir l'affichage par défaut d'`ocp-memprof`, qui, ici, calcule les tailles en mémoire des blocs agrégés par leur type. En abscisse apparaissent les différents *GC* qui ont eu lieu, et donc les différentes images du tas qui ont été générées ; la taille en mots mémoire est en ordonnée.

L'affichage, délégué à la partie javascript de l'outil, est écrit en *js_of_ocaml* [90] et utilise la librairie Javascript d3.js [2], pour obtenir dynamiquement différentes vues. Les numéros sur la figure 7.1 sont expliqués ici :

1 Par défaut, seules les 20 premières valeurs sont affichées, par souci de lisibilité. La barre de recherche permet d'ajouter les valeurs qui ne sont pas présentes dans celles qui sont affichées. Intégrant un module d'auto-complétion, la barre de recherche permet de trouver rapidement ces valeurs non affichées.

2 `ocp-memprof` permet d'agréger les valeurs de différentes manières. Par défaut, les valeurs sont regroupées par leur type.

3 L'outil permet de changer l'unité d'affichage du graphe. Par défaut, celle utilisée par `ocp-memprof` est la taille mémoire (en mots) des valeurs. Il est également possible de trier les valeurs par leurs nombres d'occurrences de blocs en mémoire.

4 Il est possible de trier, selon l'unité choisie, les valeurs. Par défaut, `ocp-memprof` trie en faisant la moyenne des valeurs occupant le plus grand espace mémoire sur tous les tas qu'il aura examinés. Il est également possible de trier ces valeurs pour un tas en particulier et d'afficher le résultat sur toute la durée d'exécution du programme.

5 Le bouton *Compactions* permet d'afficher, le cas échéant, les compactions qui ont eu lieu. Cela permet de voir aussi à quel moment le ramasse-miettes a dû procéder à ces compactions

6 Le bouton *Heap words* permet de dessiner sur le graphe, la courbe de la taille du tas au fil de l'exécution du programme (correspond à la courbe en rouge sur la figure 7.1).

7 De la même manière, le bouton *Live words* permet d'afficher la courbe des valeurs vivantes dans le tas durant l'exécution du programme (correspond à la courbe en bleue sur la figure 7.1).

8 Cette zone représente la légende. On peut voir les valeurs, affichées selon l'agrégation choisie, avec leur code couleur

9 Cette zone représente le graphe des valeurs en mémoire, affiché selon les différentes options choisies.

10 L'axe des abscisses représente les différents GC qui ont eu lieu, avec en dessous le temps qui s'est écoulé depuis le début de l'application.

11 En plus de la barre de recherche pour ajouter des valeurs au graphe, il y a possibilité de rajouter ou de supprimer des valeurs en utilisant le *slider*.

La figure 7.2 présente un affichage différent du même graphe. Les valeurs sont agrégées

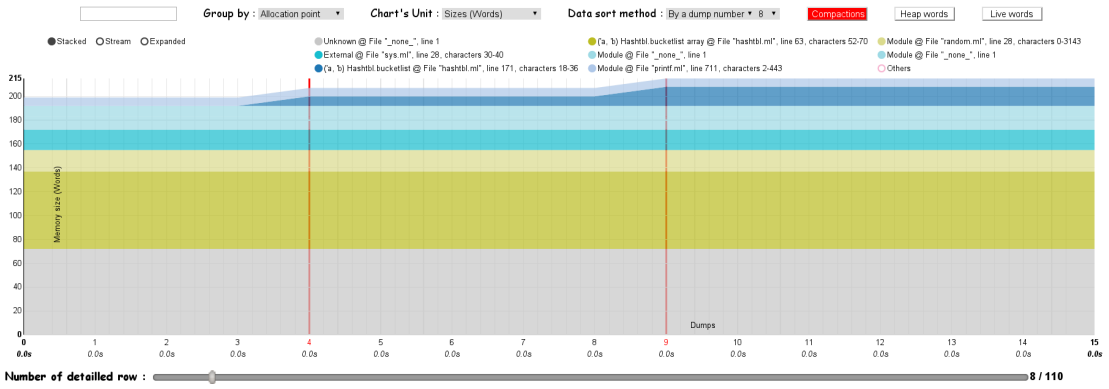

Figure 7.2 – Exemple d'affichage d'un graphe par locid, trié par rapport aux valeurs du 42^{me} tas.

par leurs (*locids*) et triées par rapport au tas numéro 8. On peut voir également les différentes compactions ayant eu lieu aux instantanés 4 et 9 (traits rouges verticaux). On affiche ici, seulement les 8 premières valeurs sur les 110 disponibles.

7.1.2 Visualisation du graphe de la mémoire sous forme de tableau

Il existe plusieurs façons d'exploiter le graphe mémoire selon l'information que nous voulons mettre en avant. Nous nous sommes concentrés dans un premier temps sur l'exploration du graphe en partant des racines, et le calcul au fur et à mesure la taille du nœud (*shallow size*), qui représente la quantité de mémoire allouée pour stocker l'objet lui-même, ainsi que la taille retenue par celui-ci (*retained size*), c'est-à-dire la taille de l'objet ainsi que la taille des objets accessibles, directement ou indirectement, depuis cet objet.

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
⊖ heap_roots		0	0.0%	5344396	100.0%
⊕ finalised_values		0	0.0%	3022569	56.6%
⊕ globals		0	0.0%	1223042	22.9%
⊕ stack		0	0.0%	1030645	19.3%
⊕ dyn_globals		0	0.0%	66777	1.2%
⊕ hook		0	0.0%	1247	0.0%
⊕ c_globals		0	0.0%	116	0.0%

Figure 7.3 – Vue globale du graphe mémoire générée par `ocp-memprof`, avec l'ensemble des racines du programme. On peut voir les tailles des différentes catégories de racines ainsi que la taille retenue par chacune d'elles.

Sur la figure 7.3, on peut observer le graphe mémoire d'une application OCaml. La ligne *heap_roots* représente l'ensemble des racines du programme. En descendant dans le graphe, on observe les différentes catégories de racines de notre programme. On retrouve, par exemple, les racines globales, la pile, les racines globales en C, etc. Sur cette figure, on peut remarquer que les valeurs à finaliser retiennent 56,6% de la mémoire. Lors d'un profiling, il peut alors être intéressant de descendre dans le graphe pour tenter de comprendre les valeurs que retiennent les racines. Ici, nous optons pour l'approche où nous partons des racines en les triant par la quantité de mémoire retenue par chacune d'elle. Une autre approche intéressante serait de partir d'une valeur en particulière et de remonter dans ce graphe. Sur la figure 7.4, on peut observer qu'en descendant dans le graphe, un certain nombre de valeurs de type `'a Eliom_comet_base.channel_data Lwt_stream.t -> unit` sont allouées en mémoire. Chacune d'entre elles retient 2,2% de la mémoire.

Dans la suite de ce chapitre, nous présentons plusieurs exemples, où l'utilisation d'ocp-memprof a permis de rapidement résoudre des problèmes mémoire pour des applications réelles telles que Alt-Ergo, un parser pour SMT-LIB et Cumulus.

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
⊖ heap_roots		0	0.0%	5344396	100.0%
⊖ finalised_values		0	0.0%	3022569	56.6%
⊖ root29	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	116254	2.2%
⊖ 0	'a Lwt.thread_repr	2	0.0%	116247	2.2%
⊕ 0	'a Lwt.thread_state	2	0.0%	116245	2.2%
⊖ 1	exn	2	0.0%	2	0.0%
⊕ root30	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	116254	2.2%

Figure 7.4 – Vue du graphe mémoire généré par `ocp-memprof`. Ce graphe est généré pour la même application que celui de la figure 7.3. Un zoom est effectué sur une des racines pour observer les valeurs atteignables depuis une racine donnée.

7.2 Cas d'étude

7.2.1 Expérimentation sur Alt-Ergo, un prouveur SMT

Alt-Ergo [1] est un démonstrateur automatique basé sur la technologie SMT (Satisfiability Modulo Theories). Il est utilisé pour montrer la validité de formules logiques issues de la vérification de programmes.

A l'origine se trouve une formule produite par Cubicle [29], lors de la modélisation du protocole de cohérence de cache FLASH, et contenant des conjonctions imbriquées de 999 éléments. L'exécution d'Alt-Ergo sur cette formule provoque une consommation mémoire très importante.

Comprendre ce qui se passe en mémoire

Pour identifier la cause de cette allocation mémoire, nous avons exécuté `ocp-memprof` sur Alt-Ergo avec la formule générée. Après avoir compilé Alt-Ergo avec notre compilateur modifié, on lance `ocp-memprof` sans modifier le code source de l'application profilée :

```
$ ocp-memprof --exec ./alt-ergo.opt -type-only formula.mlw
```

Cette exécution va forcer la génération des instantanés de la mémoire après chaque collection majeure. On obtient alors 16 fichiers de dump prêts à être profilés.

Alt-Ergo met environ 10 secondes (26 secondes avec la génération des instantanés) à prouver la formule. À la figure 7.5, on observe les graphes de la consommation mémoire d'Alt-Ergo sur la formule, après notre analyse. Sur l'axe des ordonnées, on remarque que plus de 1,5Go de données sont allouées et jamais libérées par le programme.

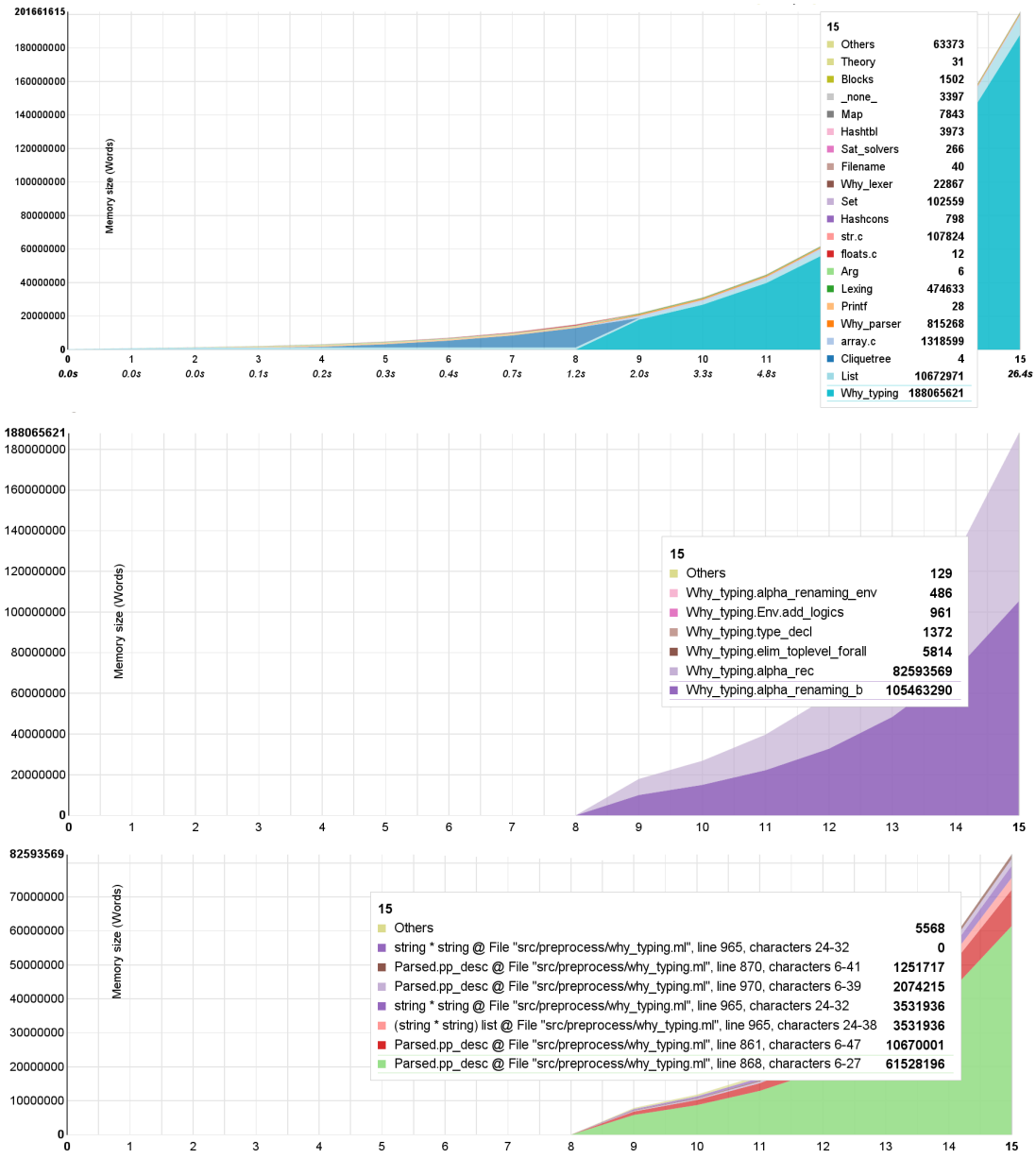

Figure 7.5 – Graphes mémoires générés à l'aide d'`ocp-memprof` sur une exécution d'Alt-Ergo. Le premier graphe est le graphe global trié par modules, le second est obtenu en cliquant sur la courbe Why_typing et le troisième est obtenu en cliquant sur Why_typing.alpha_renaming_b.

Le premier graphe est une vue dont les valeurs sont agrégées par module. En cliquant sur le module prenant le plus de place en mémoire, ici `Why_typing`, on obtient le second graphe, représentant le graphe agrégé par les fonctions qui allouent dans ce module. Enfin, en cliquant sur la fonction allouant le plus, c'est-à-dire `alpha_renaming_b`, on obtient

le dernier graphe. Celui-ci, représente les valeurs agrégées par leur point d'allocation. Les blocs de type `pp_desc`, allouées à la ligne 868 du fichier `src/process/why_typing.ml`, occupent environ 470Mo de mémoire à eux seuls.

Dans le code source, après analyse des graphes

L'extrait de code d'Alt-Ergo responsable de cette allocation excessive est montré à la figure 7.6. La fonction `alpha_renaming_b`, vue dans le deuxième graphe de la figure 7.5, est une fonction récursive effectuant de l'alpha renommage sur les formules analysées, afin d'éviter la capture des variables. Mais très souvent, il n'y a pas de problème de capture, et la fonction reconstruit une valeur `PPinfix(ff1, op, ff2)` structurellement égale à l'argument `f` de la fonction. Ceci fait que Alt-Ergo alloue énormément sur cette exemple contenant des conjonctions imbriquées de 999 éléments.

```
1  let rec alpha_renaming_b s f =
2  ...
3  | PPinfix(f1, op, f2) ->
4    let ff1 = alpha_renaming_b s f1 in
5    let ff2 = alpha_renaming_b s f2 in
6    PPinfix(ff1, op, ff2)      (* line 868 *)
7  ...
```

Figure 7.6 – Code source d'Alt-Ergo de la fonction d'alpha renommage, responsable de l'allocation observée sur la figure 7.5.

Correction du code source

Pour régler ce problème, il suffit avant de reconstruire une nouvelle valeur, de vérifier, en utilisant l'égalité physique, si cette valeur est différente de celle passée en argument. Si ce n'est pas le cas, il n'y a alors pas besoin de faire le renommage, et on peut alors retourner la même valeur de façon sûre. À la figure 7.7, le même code que celui de la figure 7.6 avec le test de l'égalité physique en plus.

À la figure 7.8, on ré-exécute Alt-Ergo sur la même formule et on relance `ocp-memprof` pour observer le comportement mémoire. Alt-Ergo met 1,8 secondes (2 secondes avec la génération des instantanés) pour prouver la même formule et surtout, il utilise moins de 35Mo de mémoire.

Pour cet exemple, nous avons diminué l'occupation mémoire de façon drastique, et par la même occasion, amélioré les performances d'Alt-Ergo.

```
let rec alpha_renaming_b s f =
  ...
  | PPinfix(f1, op, f2) ->
    let ff1 = alpha_renaming_b s f1 in
    let ff2 = alpha_renaming_b s f2 in
      (* no renaming performed by recursive calls ? *)
    if ff1 == f1 && ff2 == f2 then f
    else PPinfix(ff1, op, ff2)
  ...
```

Figure 7.7 – Code source d'Alt-Ergo de la fonction d'alpha renommage, après application du patch avec le test de l'égalité physique.

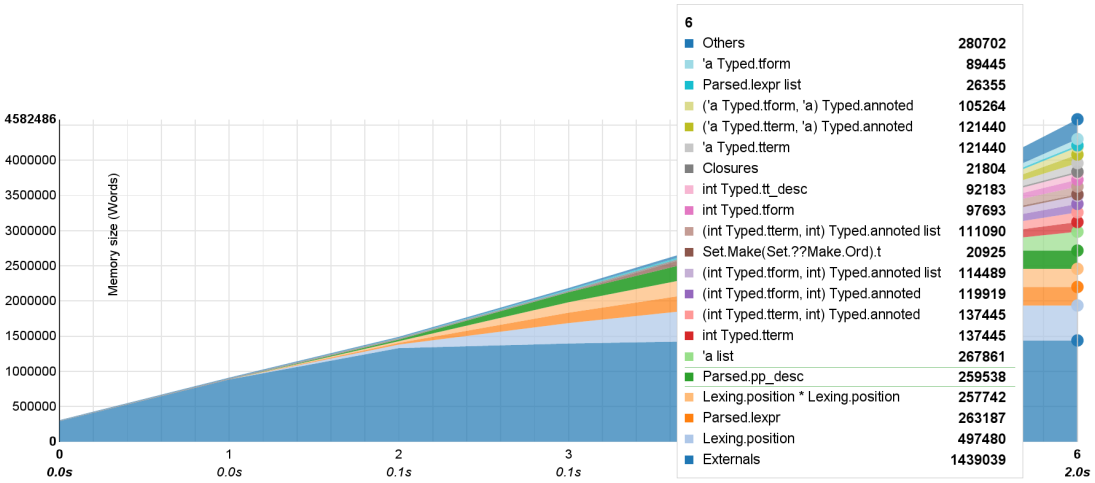

Figure 7.8 – Graphe mémoire généré à l'aide d'`ocp-memprof` sur une exécution d'Alt-Ergo, avec le code patché de la figure 7.7, sans la reconstruction de types.

7.2.2 Expérimentation sur un parser pour *SMT-LIB*

Lors de l'écriture d'un parser pour *SMT-LIB* [16], nous avons remarqué une consommation mémoire anormale. Sur la figure 7.9, un extrait du code qui lit et parse un fichier SMT-LIB.

Nous avons profilé ce programme à l'aide d'`ocp-memprof`. Les premiers résultats obtenus sont visibles sur la figure 7.10. Ce que l'on peut remarquer est qu'il y a en effet beaucoup de mémoire consommée : à la fin de l'exécution du programme, la mémoire atteint 162 073 918 mots mémoire, soit plus de 1,2Go. De plus, les structures prenant le plus de place représentent des bouts de l'AST qui donc donc maintenus en mémoire et jamais libérés au cours de l'exécution du programme. Après le dernier dump, demandé

```
1       let file = Sys.argv.(1) in
2       Heapdump.dump "opening";
3       let in_chan = open_in file in
4       Heapdump.dump "opening_file";
5       let lexbuf = Lexing.from_channel in_chan in
6       let parsed = Smtlib_parse.commands Smtlib_lex.token lexbuf
            in
7       Heapdump.dump "after_parsing";
8       ignore (parsed);
9       close_in in_chan;
10      Heapdump.dump "closing";
11      [...]
```

Figure 7.9 – Extrait de code de la fonction d'entrée qui parse un fichier du langage SMT-LIB.

manuellement à la ligne 11 (qui déclenche une collection majeure complète), l'AST n'est plus utilisé, mais n'est toujours pas libéré par le ramasse-miettes.

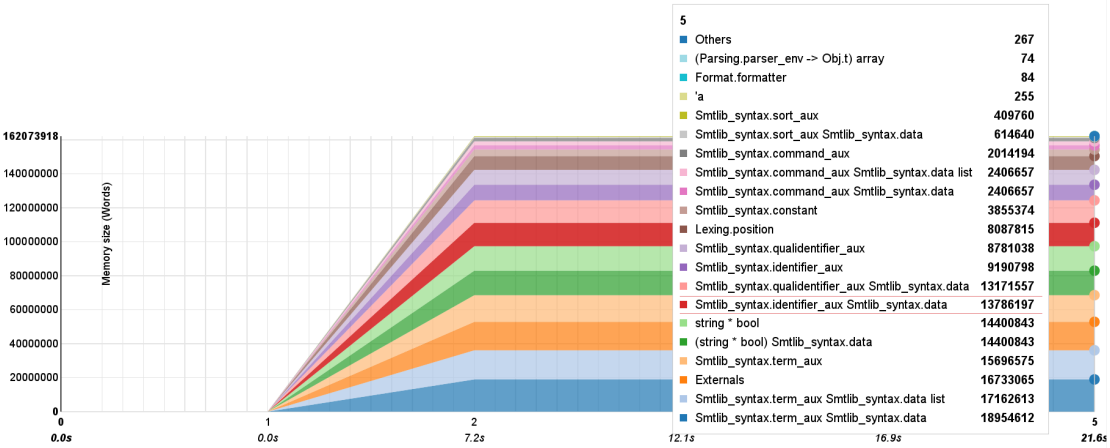

Figure 7.10 – Graphe mémoire obtenu avec `ocp-memprof` à partir de l'extrait de code de la figure 7.9.

On peut se demander alors pourquoi une référence est toujours gardée sur cette structure à la fin de l'exécution, malgré le fait qu'elle ne soit plus utile.

Le graphe mémoire obtenu ne nous donnant pas plus d'information sur l'origine de ce problème, notre intuition s'est portée sur le module `Parsing`. En examinant de plus près ce module de la librairie standard OCaml, on remarque alors la fonction

```
val clear_parser : unit -> unit
```

Une structure interne (un environnement) de type `parser_env` dans ce module, garde

une référence vers l'AST, et pour le libérer, après la phase de parsing terminée, il faut invoquer explicitement cette fonction `clear_parser` pour supprimer cette référence.

Nous avons utilisé une seconde visualisation qui nous a permis de confirmer cela. Comme expliqué précédemment, l'outil permet de parcourir le graphe mémoire depuis les racines, tout en les classant par la taille retenue par chacune d'elle. On peut ensuite naviguer à travers les fils et retrouver la même information de taille retenue par chacun de fils.

En utilisant l'outil sur le dernier dump obtenu, on remarque sur la figure 7.11 que 99,6% de la mémoire est retenue par une globale dans le module `Parsing`. En zoomant sur ce module, puis sur la globale responsable, on retrouve bien la valeur de type `Parsing.parser_env`, cette structure interne du module `Parsing`, qui était la référence qui retenait l'AST en mémoire.

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
heap_roots		0	0.0%	1258692	100.0%
globals		0	0.0%	1258692	100.0%
Parsing		0	0.0%	1253833	99.6%
root2	Parsing.parser_env	17	0.0%	1253827	99.6%
root0	exn	2	0.0%	2	0.0%
root1	exn	2	0.0%	2	0.0%
root3	'a	2	0.0%	2	0.0%
Format		0	0.0%	4332	0.3%
Smtlib_parse		0	0.0%	245	0.0%
Pervasives		0	0.0%	117	0.0%
Smtlib_main		0	0.0%	79	0.0%
Printf		0	0.0%	58	0.0%
Sys		0	0.0%	23	0.0%
Smtlib_syntax		0	0.0%	3	0.0%
Array		0	0.0%	2	0.0%
dyn_globals		0	0.0%	0	0.0%
stack		0	0.0%	0	0.0%
c_globals		0	0.0%	0	0.0%
finalised_values		0	0.0%	0	0.0%
hook		0	0.0%	0	0.0%

Figure 7.11 – Tableau représentant le graphe mémoire, trié selon la taille totale des valeurs retenues par les racines. On remarque dans ce tableau que 99,6% de la mémoire est retenue par la globale de type `parser_env` du module `Parsing`.

Après avoir modifié notre code pour vider cette structure, après la phase de parsing (figure 7.12), on obtient le graphe mémoire de la figure 7.13, qui était celui attendu.

Conclusion Ce que l'on peut conclure de cette expérience est que l'API du module `Parsing` impose de nettoyer son état interne quand on a fini de s'en servir. Cela est dû au fait qu'OCaml est un langage impur.

```
1        let file = Sys.argv.(1) in
2        Heapdump.dump "opening";
3        let in_chan = open_in file in
4        Heapdump.dump "opening_file";
5        let lexbuf = Lexing.from_channel in_chan in
6        let parsed = Smtlib_parse.commands Smtlib_lex.token lexbuf
             in
7        Heapdump.dump "after_parsing";
8        ignore (parsed);
9        Parsing.clear_parser ();
10       Heapdump.dump "after_clearing";  (* nettoyage *)
11       close_in in_chan;
12       Heapdump.dump "closing";
13       [...]
```

Figure 7.12 – Extrait de code similaire à la figure 7.9, avec l'ajout de la fonction de nettoyage de la structure internet au module `Parsing`.

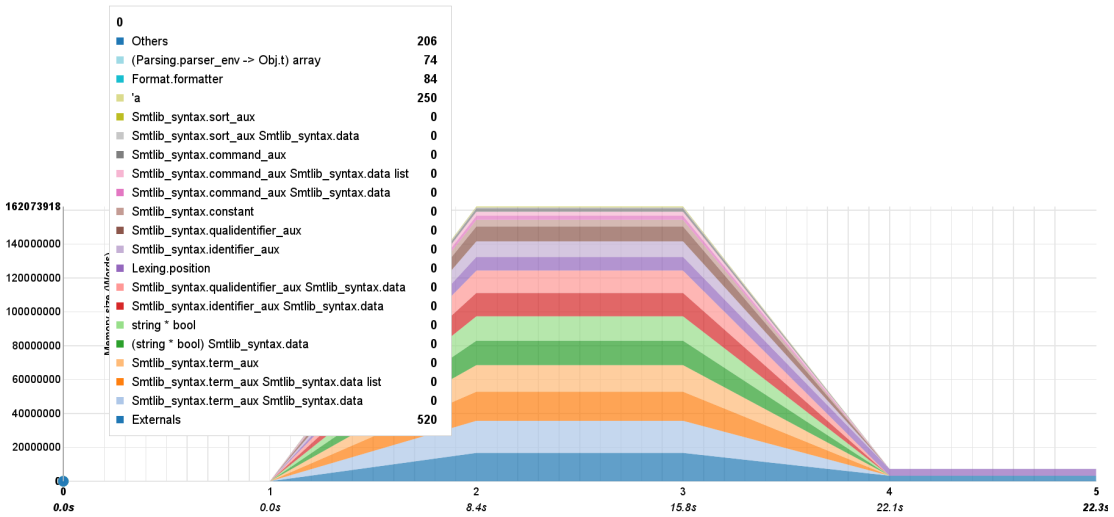

Figure 7.13 – Graphe mémoire obtenu avec `ocp-memprof` sur l'extrait de code de la figure 7.12. On remarque la chute de la mémoire après nettoyage de la structure interne du module `Parsing` à l'aide de la fonction `clear_parser`. Le ramasse-miettes a pu nettoyer l'AST qui n'était plus référencé.

7.2.3 Expérimentation sur Cumulus, un serveur web en Ocsigen

Nous présentons un exemple de détection d'une fuite mémoire apparaissant dans un serveur HTTP en OCaml. L'agrégateur de liens Cumulus[1] est un site web basé sur le *framework* Eliom du projet Ocsigen. Après plusieurs semaines d'exécution, le

1. http://cumulus.mirai.fr/

processus serveur en production montre une occupation mémoire anormalement élevée. Le programme ayant été initialement compilé avec notre compilateur, nous pouvons lui envoyer le signal `SIGHUP` pour obtenir des instantanés du tas mémoire.

Une fois ces tas rapatriés sur une machine de développement, nous pouvons les analyser avec `ocp-memprof` et obtenir des statistiques sur l'occupation mémoire (figure 7.14). L'utilisateur averti d'Eliom identifiera rapidement que la majorité de la mémoire est constituée de nœuds et d'attributs d'arbres XML ou HTML, mais aussi des chaînes de caractères et des clôtures.

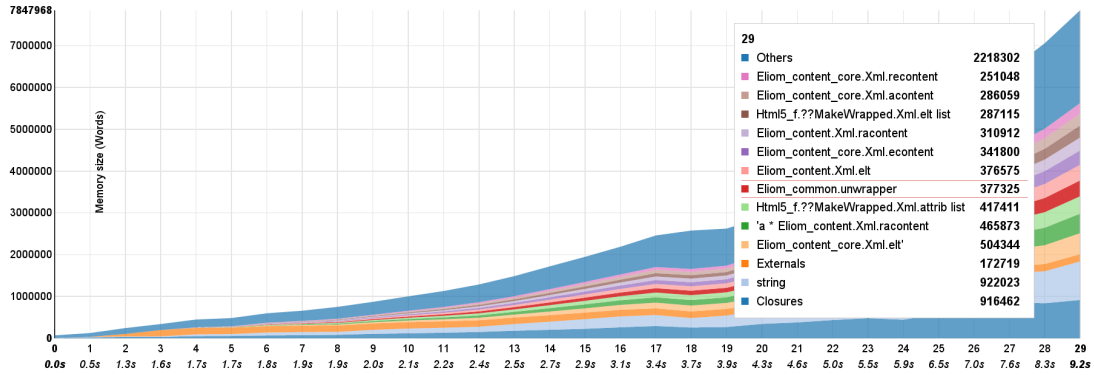

Figure 7.14 – Graphe mémoire généré à l'aide d'`ocp-memprof` sur une utilisation de l'application Cumulus.

Malheureusement, il n'est pas immédiat de savoir quels fragments de code de Cumulus sont responsables de l'allocation de ces arbres XML. Ces arbres sont en effet des types abstraits alloués à l'aide de fonctions exportées par des modules d'Eliom, la majorité des points d'allocation est donc située dans le code source d'Eliom.

De manière générale, ce problème de localisation des valeurs ayant un type abstrait est difficile à résoudre simplement avec des statistiques sur les points d'allocation. Il peut être utile de parcourir le graphe mémoire —qui peut être entièrement reconstruit à partir de l'instantané— pour, par exemple, identifier tous les chemins entre les globales et les blocs représentant les nœuds XML.

L'approche que nous adoptons consiste alors à regarder le graphe mémoire afin d'identifier les racines retenant une partie importante de la mémoire. Sur la figure 7.15, on observe le tableau des tailles retenues, avec le pourcentage, des racines de notre application, sur notre dernier instantané. L'observation qui est faite est que 88,1% de la mémoire est retenue par des finaliseurs. En les regardant de plus près (figure 7.16), on se rend compte qu'il y a une quantité non négligeable de valeurs de type `Eliom_comet_base.channel_data Lwt_stream.t -> unit` : la fuite mémoire semble donc venir du module `Eliom_comet_base`.

La fuite n'étant pas triviale à traquer et corriger, il y a cependant une correction

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
heap_roots		0	0.0%	27159360	100.0%
finalised_values		0	0.0%	23938386	88.1%
stack		0	0.0%	2112583	7.8%
globals		0	0.0%	1040276	3.8%
dyn_globals		0	0.0%	66752	0.2%
hook		0	0.0%	1247	0.0%
c_globals		0	0.0%	116	0.0%

Figure 7.15 – Tableau représentant le graphe mémoire de Cumulus avec la taille retenue par chacune des racines du programme.

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
heap_roots		0	0.0%	27159360	100.0%
finalised_values		0	0.0%	23938386	88.1%
root221	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	111730	0.4%
root222	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	111730	0.4%
root224	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110579	0.4%
root225	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110579	0.4%
root228	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110577	0.4%
root223	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110571	0.4%
root226	'a Eliom_comet_base.channel_data Lwt_stream.t -> unit	5	0.0%	110571	0.4%

'a Eliom_comet_base.channel_data Lwt_stream.t -> unit @ File "src/lib/eliom_comet.server.ml", line 736, characters 25-58

Figure 7.16 – Tableau représentant le graphe mémoire de Cumulus. Nous avons identifié les racines retenant plus de 88.1% de la mémoire. Chaque nouvelle connexion créée une valeur qui n'est jamais réclamée par le ramasse-miettes et donc les finaliseur associé à ces valeurs ne disparaissent jamais non plus.

possible dans le cas de Cumulus. En effet, après une recherche plus approfondie dans le code source de l'application, sur la figure 7.17, la fonction `of_react` prend un argument optionnel `scope` pour préciser quel genre de canal doit être utilisé par la fonction `Eliom_comet.Channel.create`. La valeur `Lwt_stream` associée à ce finaliseur ne semble jamais être récupérée par le ramasse-miettes.

```
1  let (event, call_event) =
2    let (private_event, call_event) = React.E.create () in
3    let event = Eliom_react.Down.of_react private_event in
4    (event, call_event)
```

Figure 7.17 – Extrait de code dans Cumulus responsable de la fuite mémoire.

En changeant la valeur par défaut de ce scope par une autre valeur fournie par ce

module, on obtient le code de la figure 7.18. Il n'y a alors plus qu'un seul canal qui est créé et tous les clients passent par celui-ci, contrairement à l'autre méthode qui créait un canal par client.

```
1  let (event, call_event) =
2    let (private_event, call_event) = React.E.create () in
3    let event = Eliom_react.Down.of_react
4                ~scope :Eliom_common.site_scope private_event in
5    (event, call_event)
```

Figure 7.18 – Extrait de code dans Cumulus, avec la fuite mémoire corrigée.

En relançant notre outil d'analyse sur le graphe, on obtient la figure 7.19. La méthode de création des canaux étant modifiée, la valeur retenue par ces racines passent de 88,1% à 0%.

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
heap_roots		0	0.0%	1561757	100.0%
globals		0	0.0%	770831	49.4%
stack		0	0.0%	721487	46.2%
dyn_globals		0	0.0%	67994	4.4%
hook		0	0.0%	1247	0.1%
c_globals		0	0.0%	116	0.0%
finalised_values		0	0.0%	82	0.0%

Figure 7.19 – Tableau représentant le graphe mémoire de Cumulus. Après avoir fixé la fuite mémoire, la valeur retenue par les racines correspondant aux finaliseurs redescend à 0%. En effet, il n'y a plus qu'un seul canal de communication créé maintenant.

On relance alors l'application pour observer la consommation mémoire et on obtient le graphe de la figure 7.20. L'application passe d'une consommation mémoire de plus de 60Mo pour seulement quelques dizaines de rechargements de la page, à moins de 4Mo pour plusieurs centaines de rechargements de la page ! Mais le plus important à noter ici est que dans la version initiale, la mémoire continuait d'augmenter, symptôme qu'il y avait bien une fuite mémoire. Or, maintenant, la mémoire reste constante comme nous pouvons l'observer sur cette figure.

7.3 Bilan

Les résultats que nous avons obtenus sur ces trois cas d'études sont très concluants et encourageants pour la suite. Dans chacune de ces applications, nous avons constaté une réduction de la consommation mémoire et parfois une amélioration du temps d'exécution.

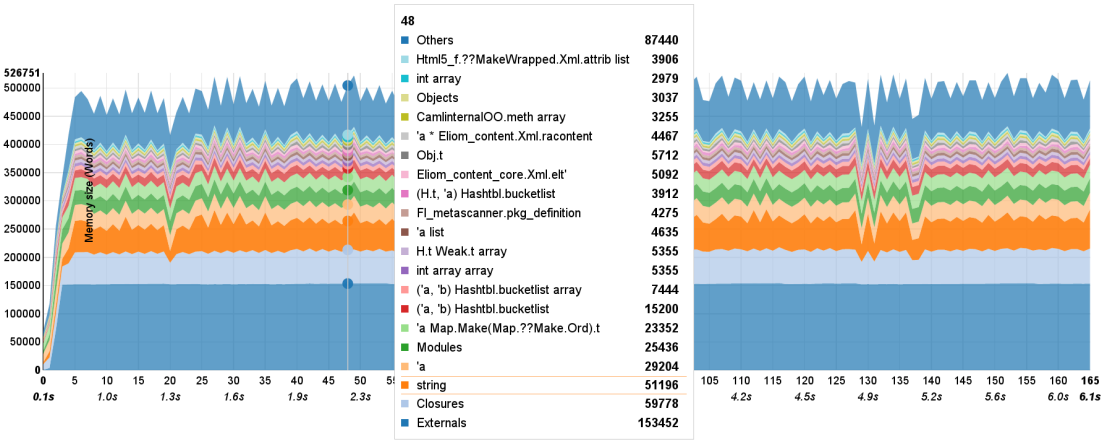

Figure 7.20 – Graphe mémoire généré à l'aide d'`ocp-memprof` sur l'application Cumulus, après réparation de la fuite mémoire. La mémoire reste constante tout au long de l'exécution de l'application.

Voici la démarche que nous suivons pour profiler une application dont l'utilisation mémoire serait anormalement élevée ou tout simplement pas celui attendu.

Dans un premier temps, il faut recompiler l'application avec notre compilateur. Cela permet d'introduire les identifiants de sites d'allocation dans les en-têtes des blocs.

Ensuite, pour comprendre ce qui se passe en mémoire, il faut produire le graphe de l'utilisation mémoire. Comme nous l'avons vu dans les exemples précédents, cela nous donne très rapidement une idée de ce qui se passe et des valeurs allouées et toujours vivantes. Dans certains cas (voir Alt-Ergo dans la section 7.2.1), il est immédiat et facile d'apporter une correction. Le tri effectué par défaut par `ocp-memprof` permet de mettre en avant, les valeurs prenant le plus de place en mémoire. C'est pour cela que notre attention va d'abord se focaliser sur ces valeurs. En triant les valeurs par module, puis par fonction et enfin par site d'allocation, cela nous permet de comprendre, par itération, où celles-ci ont été allouées. Dans la vue la plus précise (par site d'allocation), on peut alors se plonger dans le code source afin d'y apporter améliorations ou corrections.

Dans d'autres cas, le graphe mémoire ne permet pas de corriger les problèmes directement (voir le parser pour SMT-LIB à la section 7.2.2). Cependant, il permet tout de même d'observer le comportement de la mémoire et donne des intuitions pour des futurs corrections. Dans notre deuxième cas d'étude, cela nous a permis de comprendre que l'arbre de syntaxe abstraite était toujours en mémoire : il fallait donc nettoyer manuellement l'état interne du module `Parsing`.

Pour détecter ce type de problème, notre deuxième visualisation depuis les racines est un atout important. En effet, le fait de pouvoir observer la taille retenue depuis les

différentes racines, nous a permis de confirmer notre intuition sur cet exemple. 99% de la mémoire était atteignable depuis la structure interne du module `Parsing`. Un nettoyage de celui-ci nous a donc permis de corriger cette fuite mémoire.

Enfin, grâce à cette deuxième visualisation, la navigation dans le graphe mémoire peut être très instructive : elle permet de descendre dans le graphe mémoire et de voir la taille retenue par chacun des nœuds de ce graphe. Dans notre dernier cas d'étude, cela nous a permis de comprendre que des finaliseurs étaient toujours en attente, car les valeurs qui leur étaient associées n'étaient jamais collectées. Il sera intéressant d'afficher, dans cette vue, les valeurs associées à ces finaliseurs, pour essayer de comprendre par la suite ce qui les retient en mémoire.

"Le futur appartient à ceux qui croient à la beauté de leurs rêves."
Eleanor Roosevelt

"Travailler sur soi ne finit pas."
Daniel Desbiens

Conclusion et perspectives

Dans l'introduction de cette thèse, nous avons vu que le typage statique fort d'OCaml permettait à sa bibliothèque d'exécution de ne pas avoir besoin de beaucoup d'informations de types à l'exécution. Cela impliquait une représentation très compacte entraînant de meilleures performances. De ce fait, l'absence quasi-totale d'informations de type à l'exécution devenait un problème pour le profiling des applications OCaml.

Nous avons proposé dans cette thèse, dans le cadre du langage de programmation OCaml, une méthode économique et efficace d'ajout d'informations dans les en-têtes des blocs mémoires. L'absence de surcoût mémoire de notre méthode permet de profiler des exécutables dont le comportement est très similaire à celui des applications originales, non instrumentées : les informations que cette méthode permet d'obtenir sont d'autant plus fiables.

Perspectives Maintenant qu'il y a beaucoup d'informations à exploiter, plusieurs possibilités s'offrent à nous.

Le travail de **reconstruction de type** est toujours en cours de développement et n'est pas encore totalement exploité. Celui-ci peut être complété en tentant de retrouver des informations de types encore manquantes, en partant des racines du programme [47]. En effet, d'après nos expérimentations, retrouver par exemple les instances des foncteurs nous donne une information très pertinente. Dans le cadre du projet ANR BWare, nous avons pu grâce à cela éliminer différentes instances des `Map` et faire remonter celles qui étaient vraiment très coûteuses en mémoire. Une reconstruction quasi-complète des types, nous permettrait donc une meilleure compréhension du programme profilé.

Une possibilité d'ajouter des phases durant le profiling est un travail qui peut également être intéressant. Par exemple, dans le cadre d'un compilateur, il serait possible d'annoter le programme source pour séparer la phase de parsing, typage, optimisation. génération de code, etc. Il serait alors possible de profiler les dumps générés durant une phase plutôt que tous. Pour une meilleur compréhension du comportement mémoire, les

phases pourraient être également affichées sur le graphe et, en augmentant la lisibilité du graphe, permettraient de mieux comprendre les augmentations ou les diminutions de la mémoire durant certaines phases.

Au niveau du **graphe mémoire**, nous n'en exploitons aujourd'hui qu'une petite partie. Nous avons développé une visualisation avec des tableaux (à la Chrome) pour mettre en avant la taille de la mémoire retenue depuis les racines. Ce travail montre en effet que dans certains cas, il est plus simple de partir de cette information pour commencer le profiling. En effet, une fois les racines, puis les nœuds suspects identifiés, il est alors parfois plus facile de comprendre l'origine du problème (voir section 7.2.3. Ce travail peut être complété par le graphe des dominateurs. En effet, chaque objet n'est dominé que par un seul dominateur. Il peut être intéressant d'identifier les points d'accumulation de la mémoire rapidement. Toutes les informations nécessaires pour calculer ce graphe sont présentes dans les instantanés.

De plus, d'après notre expérience, il peut parfois être intéressant de partir depuis une valeur suspecte dans le graphe pour remonter jusqu'aux racines. Le graphe pouvant être assez grand, et plusieurs chemins pouvant mener à ce même nœud, ce travail n'est pas si trivial que cela. Il faut trouver la bonne façon de faire pour que l'information pertinente obtenue ne soit pas perdue dans les différents chemins dans le graphe.

Dans certains cas, il peut être intéressant de pouvoir récupérer des informations sur un instantané ou bien même d'en comparer plusieurs. Pour cela, une possibilité serait d'embarquer un toplevel OCaml, qui permettrait de recharger plusieurs tas et de faire ce travail. Comparer des instantanés ou bien même naviguer dans le graphe mémoire donne des informations intéressantes. L'interface graphique limitant cette navigation, avec le toplevel, l'utilisateur pourra alors implanter ces propres fonctions pour interroger ou manipuler le graphe mémoire. TryOCaml[62], une des contributions de cette thèse, non décrite dans ce mémoire, est un toplevel embarqué compilé entièrement en JavaScript. Celui-ci pourrait être embarqué sur la page HTML générée pour permettre de faire cela, directement depuis l'interface graphique.

Une autre idée serait d'utiliser les instantanés dans le cadre d'un **débogueur**. Il permettrait de récupérer toutes sortes d'informations sur le programme comme d'afficher certaines valeurs, leur tag, leur taille et de pouvoir naviguer dans la mémoire. Aujourd'hui, l'interface que nous avons choisie n'affiche qu'une partie des informations et ne permet pas de récupérer toutes celles qui sont dans les instantanés.

Enfin, dans le cadre de notre travail sur le **profiling continu** en cours de prototypage, il reste encore à comprendre les informations qui sont pertinentes lors d'un profiling. Ce travail mérite une attention particulière, car il permet de comprendre l'his-

toire d'une valeur depuis son allocation dans le tas mineur jusqu'à sa promotion dans le tas majeur et son nettoyage par le ramasse-miettes. Ces informations, sauvegardées dans des tableaux, peuvent être complétées par d'autres informations complémentaires permettant, par exemple, de marquer et suivre certaines valeurs, ou de distinguer certains sites d'allocation.

Bibliographie

[1] Alt-ergo, an open source smt solver. http://alt-ergo.ocamlpro.com/.

[2] D3.js, a javascript library for manipulating documents based on data. http://d3js.org/.

[3] The glorious glasgow haskell compilation system user's guide, version 7.8.3. https://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html/.

[4] The haskell 2010 report. Technical report, 2010.

[5] The jikes research virtual machine. http://jikesrvm.org/, 2011.

[6] The swift programming language. https://developer.apple.com/swift/, 2014.

[7] Shail Aditya and Alejandro Caro. Compiler-directed type reconstruction for polymorphic languages. In *In Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 74–82, 1993.

[8] Edward Aftandilian and Samuel Z. Guyer. Gc assertions : using the garbage collector to check heap properties. In *Proceedings of the 2008 ACM SIGPLAN (ASPLOS '08)*, MSPC '08, pages 36–40, New York, NY, USA, 2008. ACM.

[9] Edward. Aftandilian, Samuel Z. Guyer, Martin Vechev, and Eran Yahav. Asynchronous assertions. *SIGPLAN Not.*, 46 :275–288, October 2011.

[10] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management : improving region-based analysis of higher-order languages. *SIGPLAN Not.*, 30 :174–185, June 1995.

[11] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2) :171–183, February 1989.

[12] Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm : an efficient runtime for detecting defects in deployed systems. *SIGPLAN Not.*, 43 :143–162, October 2008.

[13] J.L. Baer and M Fries. On the efficiency of some list marking algorithms. *North-Holland Publ. Co., Amsterdam - New York - Oxford, 1977*, 77 :751–756, 1977.

[14] Chris Bailey. From java code to java heap. http://www.ibm.com/developerworks/library/j-codetoheap/, 2012.

[15] Chris Bailey, Andrew Johnson, and Kevin Grigorenko. Debugging from dumps, diagnose more than memory leaks with memory analyzer. 2011.

[16] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2010.

[17] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. ACM.

[18] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks : Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10) :169–190, October 2006.

[19] Bruno Blanchet. Escape analysis : correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 25–37, New York, NY, USA, 1998. ACM.

[20] Michael D. Bond and Kathryn S. McKinley. Bell : bit-encoding online memory leak detection. *SIGPLAN Not.*, 41 :61–72, October 2006.

[21] Cagdas Bozman, Michel Mauny, Fabrice Le Fessant, and Thomas Gazagnaire. Study of ocaml programs' memory behavior. OCaml Users and Developers, 2012.

[22] Cagdas Bozman, Michel Mauny, Fabrice Le Fessant, and Thomas Gazagnaire. Profiling the memory usage of ocaml applications without changing their behavior. OCaml Workshop, 2013.

[23] Çagdas Bozman, Grégoire Henry, Mohamed Iguernelala, Fabrice Le Fessant, and Michel Mauny. ocp-memprof : un profileur mémoire pour OCaml. JFLA, Vosges, France, 2015.

[24] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 85–96, New York, NY, USA, 2004. ACM.

[25] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 383–407, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In *ECOOP*, pages 383–407, 2011. test.

[27] James Clause and Alessandro Orso. Leakpoint : pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 515–524, New York, NY, USA, 2010. ACM.

[28] Sylvain Conchon, Jean-Christophe Filliâtre, Fabrice Le Fessant, Julien Robert, and Guillaume Von Tokarski. Observation temps-réel de programmes Caml. Vieux-Port La Ciotat, France, 2010. Hermann.

[29] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Cubicle : A parallel smt-based model checker for parameterized systems - tool paper. In *CAV*, pages 718–724, 2012.

[30] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system : a story of weak pointers and hashconsing in ocaml 3.10.2. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 13–22, New York, NY, USA, 2008. ACM.

[31] Cooper K. D., Harvey T. J, and Kennedy K. A simple, fast dominance algorithm. http ://www.hipersoft.rice.edu/grads/publications/dom14.pdf, 2001.

[32] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. A practical interprocedural dominance algorithm. *ACM Trans. Program. Lang. Syst.*, 29(4), August 2007.

[33] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection : Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM.

[34] Google Chrome Dev. Profiling memory performance. https://developer.chrome.com/devtools/docs/heap-profiling.

[35] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 1–14, New York, NY, USA, 1994. ACM.

[36] Damien Doligez. *Conception, réalisation et certification d'un glaneur de cellules concurrent*. PhD thesis, Université Paris 7, May 1995.

[37] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 70–83, New York, NY, USA, 1994. ACM.

[38] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 113–123, New York, NY, USA, 1993. ACM.

[39] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. *SIGPLAN Not.*, 35(5) :274–284, May 2000.

[40] ej-technologies GmbH. Jprofiler. "http://www.ej-technologies.com/products/jprofiler/overview.html".

[41] David A. Fisher. Copying cyclic list structures in linear time using bounded workspace. *Commun. ACM*, 18(5) :251–252, May 1975.

[42] David Gay and Alex Aiken. Memory management with explicit regions. *SIGPLAN Not.*, 33 :313–323, May 1998.

[43] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof : A call graph execution profiler. *SIGPLAN Not.*, 17(6) :120–126, June 1982.

[44] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. *SIGPLAN Not.*, 37 :282–293, May 2002.

[45] B. K. Haddon and W. M. Waite. *A compaction procedure for variable-length storage elements*. Computer J. v10 i2, 1967.

[46] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '01, pages 175–186, New York, NY, USA, 2001. ACM.

[47] Grégoire Henry, Michel Mauny, Emmanuel Chailloux, and Pascal Manoury. Typing unmarshalling without marshalling types. In *ICFP*, pages 287–298, 2012.

[48] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell : Being lazy with class. In *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*, pages 1–55. ACM Press, 2007.

[49] Richard Jones. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., 1996.

[50] Maria Jump and Kathryn S. McKinley. Cork : dynamic memory leak detection for garbage-collected languages. *SIGPLAN Not.*, 42 :31–38, January 2007.

[51] Toshiaki Kurokawa. A new fast and safe marking algorithm. *Lisp Bull.*, (3) :9–35, December 1979.

[52] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. Jml : notations and tools supporting detailed design in java. In *IN OOPSLA 2000 COMPANION*, pages 105–106. ACM, 2000.

[53] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1) :1–69, January 2006.

[54] Chin-Yang Lin and Ting-Wei Hou. A lightweight cyclic reference counting algorithm. In *Advances in Grid and Pervasive Computing*, pages 346–359. Springer, 2006.

[55] LRI. Why3. http://why3.lri.fr/.

[56] Simon Marlow and Simon Peyton Jones. The new ghc/hugs runtime system, 1998.

[57] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4) :184–195, April 1960.

[58] Nick Mitchell and et al. Leakbot : An automated and lightweight tool for diagnosing memory for diagnosing memory leaks in large applications, 2003.

[59] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[60] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 66–77, New York, NY, USA, 1995. ACM.

[61] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228, London, UK, UK, 1984. Springer-Verlag.

[62] OCamlPro. Try ocaml in your browser. http://try.ocamlpro.com/.

[63] OCamlPro. Typerex, a development environment for ocaml. http://typerex.org/.

[64] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[65] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 116–134, London, UK, 1999. Springer-Verlag.

[66] Quan Phan and Gerda Janssens. Path-sensitive region analysis for mercury programs. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '09, pages 161–170, New York, NY, USA, 2009. ACM.

[67] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark during sweep rather than mark then sweep. In *Proceedings of the Parallel Architectures and Languages Europe, Volume I : Parallel Architectures*, PARLE '89, pages 224–237, London, UK, UK, 1989. Springer-Verlag.

[68] Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward E. Aftandilian, and Samuel Z. Guyer. What can the gc compute efficiently ?: a language for heap assertions at gc time. *SIGPLAN Not.*, 45 :256–269, October 2010.

[69] Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward E. Aftandilian, and Samuel Z. Guyer. What can the gc compute efficiently ?: a language for heap assertions at gc time. *SIGPLAN Not.*, 45 :256–269, October 2010.

[70] Julien Robert and Guillaume Von Tokarski. Ocamlviz. Technical report, 2009.

[71] RTJS. Real time specification for java. http://www.rtsj.org/.

[72] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 3 :217–245, 1993.

[73] Niklas Röjemo and Colin Runciman. Lag, drag, void and use heap profiling and space-efficient compilation revisited. *SIGPLAN Not.*, 31 :34–41, June 1996.

[74] Manuel Serrano. Bee : an integrated development environment for the scheme programming language. *Journal of Functional Programming*, 10 :353–395, 2000.

[75] Manuel Serrano and Hans-J. Boehm. Understanding memory allocation of scheme programs. In *Proceedings of the fifth ACM SIGPLAN international conference on*

Functional programming, ICFP '00, pages 245–256, New York, NY, USA, 2000. ACM.

[76] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. Automatic removal of array memory leaks in java. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 50–66, London, UK, 2000. Springer-Verlag.

[77] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains : robust, distributed references supporting acyclic garbage collection. Rapport de recherche RR-1799, INRIA, 1992. Projet SOR.

[78] Mark Shinwell. Allocation profiling for x86-64 native code, 2013. https://github.com/mshinwell/ocaml/tree/4.01-allocation-profiling.

[79] Quest Software. Jprobe. http://www.quest.com/jprobe/.

[80] Jan Sparud. Fixing some space leaks without a garbage collector. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 117–122, New York, NY, USA, 1993. ACM.

[81] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[82] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[83] Kurokawa T. *New Marking Algorithms for Garbage Collection*. Proc. of 2nd. USA-JAPAN Computer Conference (Aug. 1975).

[84] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4) :724–767, July 1998.

[85] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[86] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132 :109–176, February 1997.

[87] David Ungar. Generation scavenging : A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5) :157–167, April 1984.

[88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *SIGPLAN Not.*, 23(11) :1–17, January 1988.

[89] Martin Vechev, Eran Yahav, and Greta Yorsh. Phalanx : parallel checking of expressive heap assertions. *SIGPLAN Not.*, 45 :41–50, June 2010.

[90] Jérôme Vouillon and Vincent Balat. From bytecode to javascript : the js of ocaml compiler.

[91] Philip Wadler. Fixing some space leaks with a garbage collector. *Softw. Pract. Exper.*, 17(9) :595–608, September 1987.

[92] E. P. Wentworth. Pitfalls of conservation garbage collection. *Softw. Pract. Exper.*, 20(7) :719–727, July 1990.

[93] P. R. Wilson. A simple bucket-brigade advancement mechanism for generation-bases garbage collection. *SIGPLAN Not.*, 24(5) :38–46, May 1989.

[94] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.

[95] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. Leakchaser : helping programmers narrow down causes of memory leaks. *SIGPLAN Not.*, 46 :270–282, June 2011.

[96] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. Leakchaser : helping programmers narrow down causes of memory leaks. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 270–282, New York, NY, USA, 2011. ACM.

[97] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 151–160, New York, NY, USA, 2008. ACM.

[98] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 151–160, New York, NY, USA, 2008. ACM.

[99] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. *SIGPLAN Not.*, 45 :160–173, June 2010.