Stage de Master 2 Recherche en Informatique

Analyse post-mortem du graphe de pointeurs mémoire d'application OCaml

Auteur :
Albin Coquereau

Maître de stage :
Fabrice Le Fessant

Organisme d'accueil :
INRIA / OCamlPro

Secrétariat M2R NSI, LRI - Bâtiment 650
courrier électronique : alexandre.verrecchia@u-psud.fr

15 mars – 28 août 2015

Table des matières

Liste des figures

Résumé

Le travail que je vais ici présenter consiste en la fouille d'empreinte mémoire d'application OCaml, grâce au graphe de pointeur issu du ramasse-miettes. Ceci dans le but d'en extraire des données pertinentes facilitant l'analyse de la consommation mémoire et la recherche d'éventuelles fuites mémoire de l'application étudiée.

Ces graphes de pointeurs peuvent atteindre des tailles de l'ordre du million de noeuds, il est donc important d'utiliser des méthodes de parcours et de réduction efficaces pour en extraire les informations pertinentes. Pour cela j'ai utilisé des méthodes de compression d'arêtes, de transformation de graphe en arbre, grâce aux arbres de domination et aux composantes fortement connexes. J'ai aussi utilisé plusieurs méthodes d'affichage, graphiques et tableaux de données.

Les résultats que j'obtiens grâce au différentes analyses permettent de distinguer les structures et types ayant une forte empreinte mémoire, en distinguant la quantité de mémoire directement retenue ou partagée. Ils ont permis de mettre en évidence des fuites mémoire décelées dans des cas réels d'applications OCaml.

Termes généraux

Fouille de Graphe, Arbre de Domination, Composantes Fortement Connexes

Mots clefs

OCaml, Profilage du tas, Ramasse-miettes, Instantané Mémoire

Chapitre 1

Introduction

Lors de chaque lancement de programme, une partie de la mémoire vive est utilisée pour l'exécution, comme toute ressource elle est limitée, il est donc important de la gérer au mieux pour éviter tout gaspillage. Elle peut être gérée soit manuellement par le programmeur lors de la création du logiciel soit automatiquement via un outil de récupération de mémoire automatique, appelé ramasse-miettes. Malgré tous les efforts des programmeurs et l'efficacité de ces outils, il peut exister en mémoire des données inutilisées qui n'ont pas été libérées, on appelle cela des fuites mémoire. Ces fuites mémoire peuvent ralentir l'exécution des applications, pouvant même provoquer une fin d'exécution prématurée.

Grâce à des outils dits de profiling, copiant la mémoire du système à chaque lancement du ramasse-miettes, nous pouvons connaître la consommation de chaque objet, de chaque module, via ces copies, instantanés, du contenu de la mémoire. Ces données peuvent être en quantité importante, et deviennent donc difficilement compréhensibles, il faut donc avoir des analyses facilitant la lecture de ces données.

Ce rapport fait état des travaux effectués durant mon stage de fin de Master recherche, Nouveaux Systèmes Informatiques, de l'Université Paris Sud XI. Ce stage qui a duré 5 mois, a eu lieu au sein de l'entreprise OCamlPro, spécialiste des solutions en OCaml, en lien avec l'Institut National de la Recherche en Informatique Appliquée, sous la responsabilité de Fabrice Le Fessant. Ce stage a porté sur l'analyse de graphe de pointeur d'application OCaml issu de l'outil de profiling mémoire "OCaml Memory Profiler" d'OCamlPro (Memprof).

Objectifs

L'objectif de ce stage est d'améliorer l'exploitation des données issues des instantanés mémoire de Memprof, et en particulier à partir du graphe de pointeurs mémoire issus du ramasse-miettes. Il existe beaucoup de travaux d'étude de ces graphes pour des langages comme Java ou Haskell, réduction du graphe, comparaison d'instantanés, etc. Le but de ce stage est donc d'essayer d'adapter de tels travaux à OCaml via l'outil Memprof et de tirer le plus d'informations pertinentes des graphes. Ces graphes de pointeurs peuvent être de taille importante, nous devrons donc adapter au mieux les solutions pour traiter d'importantes quantités de données, via des techniques de réduction ou de transformation de graphes.

Plan

Après avoir étudié la structure mémoire des données en OCaml et parcouru l'état de l'art de ce type d'analyses pour les profileurs existants, nous proposerons de nouvelles analyses à effectuer sur un ou plusieurs graphes de pointeurs recueillis par Memprof, puis

les implémenterons en exploitant au maximum la bibliothèque d'algorithmes OCamlGraph, nous finirons par en étudier l'efficacité à fournir des informations exploitables et utiles sur des cas réels d'applications OCaml.

CHAPITRE 2

Contexte

La première partie de mon stage a été la découverte de la représentation mémoire en OCaml et son ramasse-miettes ainsi que l'outil de profiling Memprof. Je présenterai dans cette partie ces notions nécessaires à la compréhension du reste du rapport.

OCaml est un langage de programmation multi-paradigmes, alliant les programmations fonctionnelle, impérative et objet, c'est un langage statiquement typé dont un des avantages est d'éviter les problèmes de type à l'exécution. Nous allons présenter la représentation mémoire de ces données, puis le fonctionnement de son ramasse-miettes, et enfin présenter l'outil de profiling utilisé au cours de ce stage.

2.1 Représentation mémoire

En OCaml une valeur en mémoire peut représenter deux choses, soit une valeur entière immédiate soit un pointeur, cette distinction est effectuée via un bit en mémoire, l'espace d'adressage des pointeurs est donc aligné sur 4 ou 8 octets.

Une valeur entière peu représenter un entier ou un constructeur constant, `true` et `false`, liste vide `[]` , et `()` de type unit.

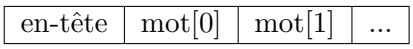


Figure 2.1 – Bloc mémoire OCaml

La représentation des blocs mémoires est une en-tête de taille fixe (32/64bits) suivie de un ou plusieurs mots (4/8 octets). Cette en-tête contrairement à celles de Java, ne nous donne que peu d'informations, du fait du typage statique du langage peu d'informations de type sont utiles à l'exécution ; l'en-tête contient donc uniquement des informations utiles minimum et des informations pour le ramasse-miettes.

2.2 Ramasse-miettes

On appelle ramasse-miettes les outils de récupération automatique de la mémoire, comme expliqué auparavant (voir section 1), la mémoire est une ressource limitée qui doit être gérée manuellement ou automatiquement. Dans le langage C, la mémoire est allouée manuellement, lors de la création d'une structure en mémoire il faut faire une allocation de la taille de la structure et libérer cette mémoire lorsque la structure n'est plus utile.

D'autres langages comme Java, OCaml, possèdent un système allouant et libérant la mémoire de manière automatique, on appelle ce système un ramasse-miettes ou GC pour "garbage collector". Ce type d'outils évite les erreurs humaines dues par exemple aux oublis de libération, mais ne peut déterminer à coup sûr si une donnée n'est plus utile et donc libérable, pouvant causer des fuites mémoires. Voici deux des techniques de récupération

automatique de la mémoire :

— **Comptage de références**
Chaque bloc possède un compteur, lorsque qu'une référence est créée vers un bloc, le compteur est incrémenté lorsque qu'un autre objet le pointe et est décrémenté lorsque cette référence est supprimée. Lorsque ce compteur tombe à 0 le bloc n'est plus utilisé et peut être libéré.
— **Balayage**
Ce type de ramasse-miettes fonctionne par atteignabilité, la mémoire est considérée comme un graphe avec plusieurs racines. Un bloc est considéré comme vivant tant qu'il est atteignable depuis les racines, dès lors où il ne l'est plus, il est considéré comme mort, et peut donc être libéré. Sur la figure 2.2 les blocs rouges n'ont pas de lien avec les racines, ils ne sont pas atteignables et peuvent être libérés.

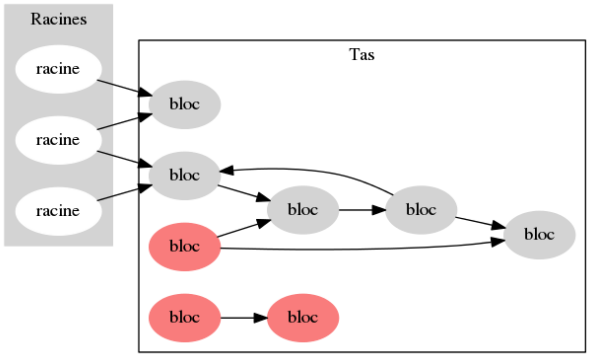


Figure 2.2 – Graphe de pointeurs

Le ramasse-miettes d'OCaml [6][4] est incrémental et générationnel, il fonctionne sur la technique de balayage. Il est incrémental car ne stoppe pas l'exécution durant le balayage et est générationnel car sa mémoire est divisée en deux parties, une où se situent les objets à courte durée de vie et une deuxième partie plus grande où se situent les objets qui survivent à plusieurs lancements du ramasse-miettes. Fonctionnant sur la technique de balayage, il existe un graphe de pointeurs issu de chaque lancement du ramasse-miettes que nous allons pouvoir utiliser.

2.3 Outils de profiling

Les fuites mémoires étant un problème connu et difficilement décelable, il y a un besoin d'outils d'analyse de la mémoire, outils dits de profiling. Outils aussi utiles dans les langages à gestion manuelle qu'à gestion automatique de la mémoire, il en existe aujourd'hui pour les langages très utilisés tels que Java, mais peu pour des langages tels que OCaml. Ces outils sont pourtant très prisés par les développeurs car ils peuvent permettre de détecter d'éventuelles fuites mémoire.

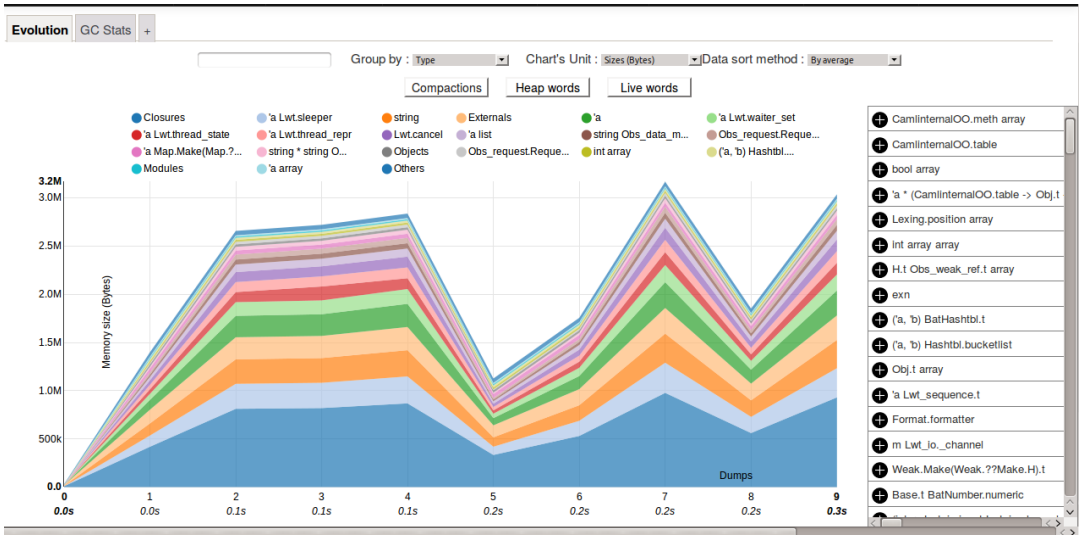


Figure 2.3 – Exemple de graphe donné par Memprof

Suite au travaux de thèse de Cagdas Bozmam [1], plus d'informations peuvent être recueillies des blocs de donnée mémoire OCaml. En effet lors de la compilation un identifiant est ajouté dans l'en-tête de chaque bloc de données. Cet identifiant nous donne des informations sur la localisation de l'allocation, le type de la donnée, etc.

De ces travaux est né Memprof [22] qui est un outil de profiling pour les applications OCaml, développé par OCamlPro, il permet d'obtenir graphiquement une courbe d'évolution 2.3 de la consommation mémoire, ainsi que des informations sur le ramasse-miettes. Il permet aussi pour un instantané donné d'obtenir plus d'informations sur les tailles retenues par les racines du ramasse-miettes2.4. Le graphe de pointeurs est ici uniquement utilisé pour donner des informations sur les tailles des données retenues par chaque racine.

Evolution | GC Stats | **Dump #239**

Title	Kind	Shallow Size	Shallow Size	Retained Size	Retained Size
heap_roots		0	0.0%	68209	100.0%
globals		0	0.0%	419	0.6%
dyn_globals		0	0.0%	0	0.0%
stack		0	0.0%	67797	99.4%
glob5	'a array	65536	96.1%	65543	96.1%
glob6	'a array	65536	96.1%	65543	96.1%
glob0	'a array	1001	1.5%	1001	1.5%
glob2	'a array	1001	1.5%	1001	1.5%
glob3	'a array	1001	1.5%	1001	1.5%
glob1	Camlstack.bytes	252	0.4%	252	0.4%
glob4	Camlstack.bytes	252	0.4%	252	0.4%
glob7	Huffman.node	7	0.0%	7	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 2.4 – Tableau de données mémoire d'un instantané par Memprof

CHAPITRE 3

Pistes

Grâce à ces connaissances, il nous est maintenant possible de chercher dans la littérature des documents pouvant être utiles à notre sujet : Étude de graphe mémoire d'application OCaml. Notre recherche s'est tout d'abord concentrée sur des documents traitant des graphes de pointeurs mémoire issus de ramasse-miettes. Nous avons ensuite approfondis leurs approches par des documents sur l'étude de graphes plus généralistes. Enfin nous nous sommes penché sur des documents traitant de l'affichage des données contenues dans les instantanés mémoires. Notre recherche s'est aussi effectuée sur les outils de profiling existants.

Dans cette partie nous allons présenter quatre pistes majeurs pour l'analyse de graphe de pointeurs, la première qui sera essentielle pour les autres sera la réduction du graphe, ensuite nous présenterons des technique de modifications de graphe. Puis des méthodes de catégorisation des données en mémoire, enfin nous présenterons différentes approches d'affichage pour les graphes de pointeurs.

3.1 Réduction de graphe

La taille des graphes de pointeurs peuvent varier en fonction de l'application, sa complexité, ses données, allant de quelques centaines de noeuds à plusieurs millions. La question du passage à l'échelle se pose donc. Nous avons pu trouver plusieurs solutions possible pour réduire le graphe à son essentiel.

Grammaire et paternes

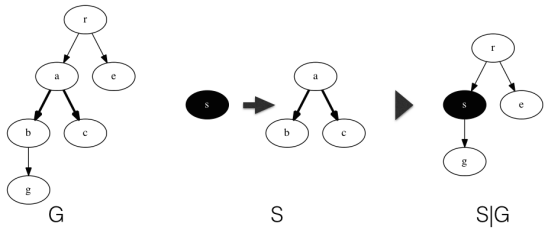


Figure 3.1 – Exemple de réduction par grammaire

La première technique que nous avons pu rencontrer dans le texte de Maxwell et Back [11], est la réduction de graphe grâce a des grammaires[21][8]. Une grammaire est une règle de transformation entre un groupe de noeud spécifique vers un autre noeud, comme dans l'exemple de la figure 3.1. Le but est ici de réduire le graphe en remplaçant des ensembles de noeuds en un seul.

Une technique sensiblement identique viens du document de Chis et Mitchell [9] traitant de l'efficacité des structure de données Java par rapport a leur empreinte mémoire. Le but de cette technique étant d'agréger des ensembles de noeuds en patterns et d'en étudier les occurrences.

Les grammaires et patterns sont des sous ensembles du graphe, représentant par exemple une structure de données (tableau, table de hachage) dans le langage étudié. Ces patterns peuvent être connus statiquement, calculés à partir de tests antérieurs. Ou bien dynamiquement, parcourir une première fois l'instantané du tas pour trouver des patterns récurrents et leurs compactions, puis le parcourir une seconde fois pour appliquer les réductions aux patterns.

Nous nous sommes intéressé à une possible adaptation d'un tel travail pour OCaml. Car en effet une telle réduction pourrai réduire le graphe de manière drastique pour une application contenant peu de structure de donnée de grande taille et donc de nombreux blocs, réduit a peu de pattern.

La représentation mémoire de Java est très complète, comme expliqué auparavant beaucoup d'informations sont disponibles, en lisant un bloc mémoire java il est simple de connaître son type, ce qui n'est pas le cas en OCaml ou il nous faut aller chercher cette information grâce au Locid (identifiant de point d'allocation). Statiquement certains patterns sont déjà connu, les listes par exemple, qui sont représentées en mémoire sous forme de chaîne de blocs pourraient être agrégées en un seul bloc. il en est de même pour les Map et table de hachage, ou l'on pourrait contracter la structure avec son contenu.

Contraction d'arêtes

Une autre méthode étudiée est celle de la contraction d'arêtes. Si une arête correspond au critère de réduction, elle est supprimée et les deux noeuds qui y sont reliés fusionnent. Cette technique peut permettre une réduction importante d'un sous graphe composé de chaîne, structure récursive par exemple.

Une piste de réduction serait l'agglomération par type, basé sur la contraction d'arètes, cela consisterait à contracter les blocs voisins de même type en un seul bloc. Cela serait moins efficace que la solution précédente en terme de taille de réduction mais offrirait peu de perte d'informations.

Nous avons aussi pensé à une réduction basé sur le même modèle qui consisterait à contracter les noeuds n'ayant qu'un seul parent, c'est à dire contracter toutes les parties de graphe en forme d'arbre, ne laissant que les noeuds du graphe possédant plusieurs parents,

ce qui ici serait interprété comme du partage.

Bien que la réduction par grammaires et patterns semble donner des résultats intéressants en Java, le temps de découverte des patterns inefficaces en OCaml ou réduisant la taille de manière intéressante nous semble trop importante. Nous allons préférer nous pencher sur la contraction d'arêtes car elle est plus rapide à mettre en place et peut offrir un ratio de réduction plus important.

3.2 Modification de graphes

Un graphe de pointeur n'est pas toujours évident à manipuler, les cycles et le partage (plusieurs parents) de certains noeuds rendent la fouille de graphe complexe. une solution proposé par The Runtime of Object Ownership[14] est de transformer un graphe en arbre[11], réduisant le nombre d'arêtes et supprimant les cycles.

Plusieurs idées découlent de cette transformation, supprimer le partage nous permettrait de visualiser plus facilement un chemin d'une racine à un noeud, remonter une information de taille ou de type deviendrait trivial. Une autre idée serait de ne s'intéresser qu'au noeud partagé, ici un bloc mémoire pointé plusieurs fois.

Arbre de Domination

La première technique de transformation étudiée est l'arbre de domination d'un graphe utilisé par l'outil MAT[23]. Les arbres de domination sont construit selon la relation suivante :

On dit qu'un noeud n domine un noeud m si chaque chemin qui atteint le noeud m passe par n.

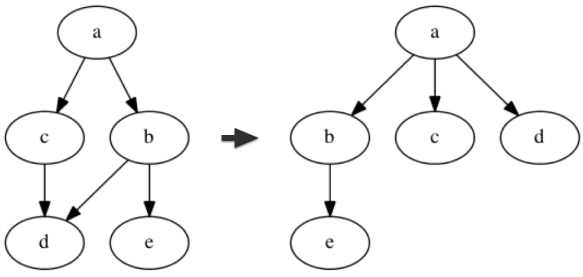


Figure 3.2 – Exemple d'arbre de dominateur

Sur un graphe orienté contenant du partage, il existe plusieurs parents sur certains noeuds, on obtient un arbre ou chaque noeud qui était partagé devient fils de son dominant, comme dans l'exemple 3.2.

Composantes fortement connexes

Une autre approche est de transformer le graphe grâce aux composantes fortement connexes, cette transformation ayant pour but de grouper les cycles de noeuds en un seul [15]. Cette approche peut permettre une réduction importante du graphe si celui ci comporte de nombreuse composantes fortement connexes réduite à un seul noeud, de plus la suppression des cycles simplifie le parcours du graphe au niveau algorithmique.

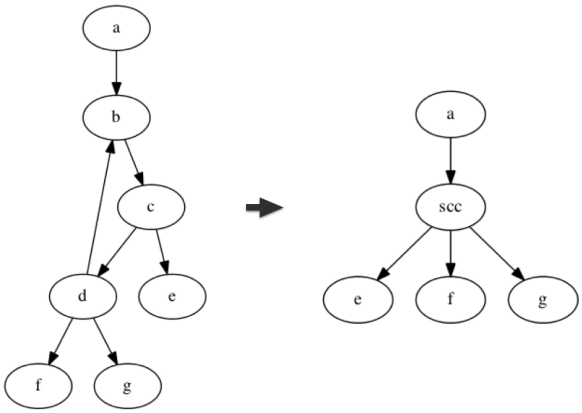


Figure 3.3 – Exemple de composantes fortement connexes

Cette technique permet de garder les informations de partage sur les noeuds et d'identifier les possibles cycles de pointeur.

Arbre couvrant

Nous avons aussi penser au calcul d'arbre couvrant, puis décider d'écarter cette solution car ici nos arêtes ne sont pas pondérés, l'arbre minimum serait donc un arbre du plus court chemin vers chaque noeuds. Ce genre d'informations a tout de même des avantages, l'arbre n'ayant pas de partage ni de cycle et étant de profondeur minimum, il pourrait permettre des parcours et calculs simplifiés sur des ensembles de données, par exemple calculer la taille des types. Mais il n'y aurait plus de notion de chemin pertinente, c'est pour cela que nous n'avons pas poursuivi dans cette voie.

3.3 Analyse du contenu

Le document de Mitchell et Sevitsky [13] propose de caractériser le contenu et l'usage de chaque blocs, pour obtenir des informations statistiques sur le contenu de la mémoire.

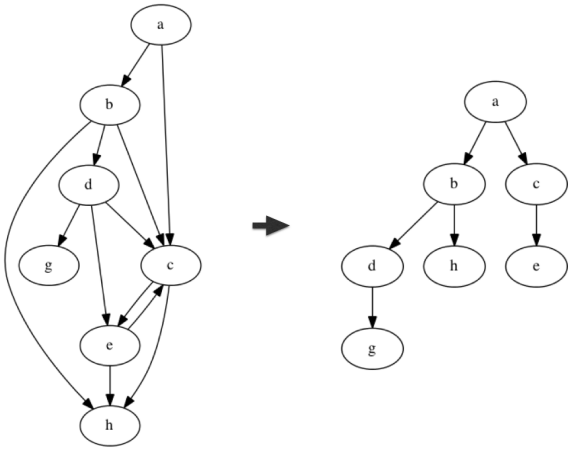


Figure 3.4 – Exemple d'arbre couvrant

Le document propose tout d'abord de diviser le contenu des blocs en quatre catégories :

- Primitif : élément primitif du langage sur 1-8 octets
- En-tete : données de l'objet utile pour l'exécution 12-16 octets
- Pointeur : références entre objets 4 octets
- Null : espace de pointeur non utilisé 4 octets

Cette catégorisation permet de chiffrer pour un contenu mémoire donnée la taille utilisée pour les données pures (primitives), les en-têtes ou les pointeurs. Si votre mémoire est remplie de bloc pointants sur d'autre blocs, vous aurez une majorité de votre consommation en pointeur et en-tête et peu en données réels, on peut imaginer un entier contenu dans un objet lui même contenu dans un objet etc.

Dans un second temps le document propose une catégorisation par utilité dans les collections, là encore quatre catégories sont proposées :

- tête : type de donnée représentant une collection
- tableau
- entrée : type de donnée récursive
- contenu : tout le reste

Cette catégorisation permet de trier chaque blocs par son rôle, sans le décomposé contrairement à la première catégorisation. Les proportions de chaque rôle peuvent nous intéresser par exemple si la taille du contenu est inférieur au reste, la structure de donnée peut être peu remplie ou est plus coûteuse en structure qu'en données.

La composition de ces deux catégorisations permet de définir cinq nouvelle catégories :

- donnée : vrai donnée en mémoire
- charge primitive : surplus de pointeur inter-connecté
- petit objet
- charge pointeur : structure trop peu remplie
- glue : connections entre les collections

Le document continu ensuite par une mise en oeuvre d'une telle modélisation, les résultats servants à montrer à partir ou jusqu'à quelle quantité de valeurs les structures de données sont efficaces, ainsi que la proportion de donnée réel dans les benchmarks étudiés.

La représentation en mémoire des données OCaml est beaucoup plus légère que celle de Java comme nous l'avons vu dans la partie 2.1 Nous avons essayé de transposer ce travail à OCaml, comme pour Java, les blocs ont une en tête, ici elle est fixe 32bit ou 64bits selon l'architecture. S'en suivent une liste de pointeurs ou de valeurs immédiates sur 32 bits, la catégorisation par contenu de bloc serait donc sensiblement identique à celle proposée dans le document. Pour ce qui est de la deuxième catégorisation il nous faut analyser les blocs plus en profondeur car les informations de type ne sont pas directement disponibles. On peut caractériser les contenus comme les valeurs immédiate en mémoire, les entrées sont les structures sous forme de List OCaml.

3.4 Visualisation

Une fois notre graphe modifié ou réduit il nous faut des moyens de l'afficher ou d'en extraire les informations souhaitées, des outils comme Valgrind[24] et Hprof[25] proposent des affichages de données sous forme de tableau dans la console. Le document Scalable Visualization of Object-Oriented Systems with Ownership Trees [10] et le document Visualizing Reference Patterns for Solving Memory Leaks in Java [18] nous apporte deux approches de visualisation graphique du graphe de pointeur. Le premier propose une forme de tableau ou l'on peut développer chaque noeuds du graphe en profondeur, des liens apparaissent lorsqu'il existe une relation entre deux éléments en largeur.

Le deuxième document propose quant a lui une visualisation sous forme d'arbre de noeuds. Chaque noeuds pouvant être développé pour en afficher ses fils. Comme pour le premier document chaque noeuds est représenté par le type de la donnée contenue, de plus ici des informations sur l'âge des données sont disponibles.

Actuellement, dans l'outil Memprof, l'affichage du graphe de pointeurs et des informations qu'il contient s'effectue sous forme de tableaux ou chaque noeuds peut être développé en un autre tableau. Cette représentation est basique et pour limité les coups, à une profondeur d'affichage limité, la solution proposée comporte plusieurs analyses.

CHAPITRE 4

Mise en oeuvre

Avant toute mise en oeuvre des pistes présentées auparavant, il nous a fallu nous familiariser avec l'outil Memprof. Ensuite une fois acquise une bonne compréhension d'une partie du code source, nous avons pu entreprendre le développement de solutions d'analyse de graphe de pointeurs.

4.1 Memprof

Comme nous l'avons présenté auparavant, Memprof est un outil d'analyse de consommation mémoire pour les applications OCaml, il requiert une modification du compilateur pour obtenir plus d'informations sur les blocs en mémoire. Lors d'un lancement du ramasse-miettes durant l'exécution d'une application, il sérialise le contenu de la mémoire et le tas en fichiers sur le disque.

Les fichiers contiennent donc toutes les données en mémoire lors du lancement du ramasse-miettes. Ces données ne sont pas forcément vivantes, nous avons donc à notre disposition un tableau de blocs représentant les racines du ramasse-miettes. A partir de ces blocs nous pouvons explorer leur pointeurs pour reconstruire le graphe par exploration. Une représentation de ce graphe était à ma disposition lors du début du stage, j'ai pu donc entreprendre rapidement de mettre en place mes analyses.

L'outil possède aussi des structures référençant les noms des racines et d'autres permettant de lier les blocs à des informations telles que le type, la taille, le point d'allocation (fichier, fonction, ligne). Il m'a donc été possible de lier chaque point du graphe à des informations plus explicites qu'un simple numéro de noeud. Ma première idée a été d'avoir une version graphique du graphe.

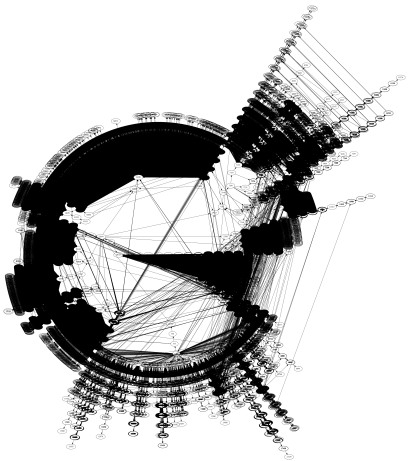


Figure 4.1 – Affichage basique du graphe de pointeur

4.2 Réduction de graphe

Comme nous pouvons le voir sur la figure 4.1 le graphe de pointeurs contient une quantité importante de points, de ce fait il nous est impossible d'en extraire de quelconques informations sans modifications. La première étape fut de supprimer les composantes fortement connexes pour faciliter le parcours du graphe. Nous avons ensuite plus facilement pu contracter certaines arêtes du graphe pour pouvoir l'afficher et l'analyser plus simplement.

Pour supprimer les composantes fortement connexes j'ai voulu utiliser la bibliothèque OCamlGraph renfermant de nombreuses implémentations concernant l'étude de graphe, en particulier le module Components. Ce module utilise l'algorithme de Tarjan pour supprimer les composantes fortement connexes du graphe, dans un souci fonctionnel l'implémentation de l'algorithme est récursive et offre de très bonnes performances. Mais sur de grands graphes, de l'ordre du million de noeuds, la pile d'appel OCaml déborde, il nous a donc fallu transformer le module pour le rendre itératif. Après que mon maître de stage ait apporté les modifications nécessaires au module j'ai pu réduire le graphe. Lorsqu'un noeud n'a qu'un seul parent nous contractons ce noeud avec son parent, cette technique est basée sur la contraction d'arêtes, nous perdons plus d'informations qu'une contraction par type, mais cette approche va nous permettre d'avoir une première vision du graphe de pointeur.

Pour cette analyse nous avons effectué deux affichages : un graphique et un tableau. Pour le premier nous utilisons le langage de description pour générer un fichier convertible en fichier image. Après réduction, les noeuds du graphe peuvent être des ensembles de blocs mémoire, nous avons décidé d'afficher les dix noeuds retenant les plus grandes quantités mémoire. Nous affichons aussi les chemins des racines aux noeuds que nous avons choisis.

Le deuxième affichage choisi pour cette analyse est un tableau HTML, pour chaque noeud nous affichons les données qu'il contient, leurs types et leurs poids en mémoire.

4.3 Le partage

Suite à cette première analyse nous nous sommes intéressés au partage des données, en effet aujourd'hui l'outil propose d'afficher les tailles retenues par les racines sous forme d'un tableau 2.4, ces tailles sont calculés à partir des racines, il arrive donc que certaine d'entre elles soit comptabilisé plusieurs fois lorsque l'on a du partage sur les données. L'idée est donc de calculer les tailles retenues réelles de chaque racine et de chaque noeud. De plus nous pensons qu'il serait intéressant de savoir qu'une partie de la taille retenue par un noeud est partagée entre ses fils.

Pour calculer les tailles partagées nous avons utilisé les arbres de domination, appliquées à notre graphe les noeuds étant partagés par plusieurs arêtes, sont reliés à leur dominateur. En partant des feuilles du graphe nous remontons les informations de taille des blocs, le graphe ne comportant plus de cycles son parcours est d'autant plus simple. Nous comparons les fils de chaque noeud avant et après modification par les arbres de domination, les noeuds apparaissant après modification sont donc les noeuds qui étaient partagés. Nous avons

donc deux tailles possibles pour chaque noeud, la taille retenue et celle qui est partagée à partir de ce noeud.

Comme pour les composantes fortement connexes le module des arbres de domination d'OCamlGraph n'est pas passé à l'échelle, il a donc fallu modifier l'algorithme de DFS (parcours en profondeur) pour le rendre itératif. Cette modification comme la précédente à été proposée au mainteneur de la bibliothèque pour quiconque se retrouvant face au même problème.

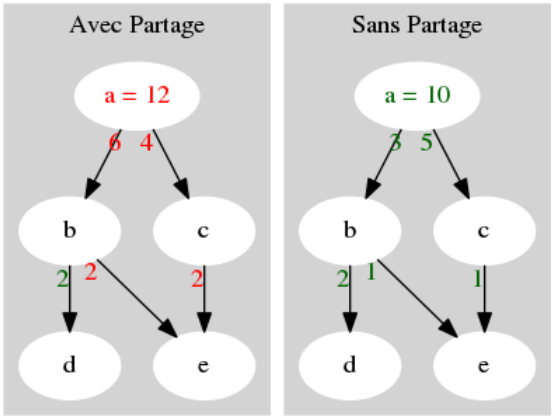


Figure 4.2 – Différence de taille retenue avec/sans gestion du partage

Ce calcul modifie de manière importante le graphe de pointeurs, nous avons donc voulu avoir des données sans perte d'informations. Nous avons décidé comme précédemment de faire remonter les informations de taille de blocs dans le graphe, mais cette fois le graphe n'est pas modifié. Il a donc fallu stocker les noeuds déjà visités pour ne pas boucler dans les cycles, et ne pas remonter les tailles sans modification dans le graphe. Si l'on remonte les tailles des objets partagés, on les remonte plusieurs fois, il nous faut donc diviser chaque taille par le nombre de de parents des noeuds (voir figure 4.2).

Durant ce parcours du graphe nous en profitons pour récupérer des informations sur les types, à chaque bloc nous ajoutons sa taille au type de données contenu dans se bloc. Nous obtenons donc un tableau de tailles par rapport aux types. La quantité mémoire retenue par chaque type présent dans la courbe d'évolution peut maintenant être affichée sous forme de tableau.

4.4 Les finaliseurs

Lors de la thèse de Cagdas Bozman une fuite mémoire à été décelée dans l'application Cumulus, elle était due au framework Eliom, qui facilite la programmation web en OCaml. En effet ce framework utilisait des finaliseurs, supprimant les objets lorsque qu'il est appelé, ce sont des outils difficiles à utiliser. Lors de la découverte, un patch avait été

proposé en modifiant le code pour ne plus les utiliser. Cependant lors d'un autre projet de développement web cette fuite est réapparue, j'ai donc réfléchi à une méthode pour trouver ce genre de fuites. Les finaliseurs ne peuvent pas se déclencher et supprimer l'objet concerné s'ils pointent eux-même sur cet objet.

La technique proposée est donc de parcourir en profondeur les fils du finaliseur, si l'on atteint l'objet finaliser alors il y a une erreur et une fuite certaine. L'objet a finaliser et ses fils ne seront jamais finalisés durant l'exécution. Pour aider les développeurs à modifier cette fuite nous leur proposons d'afficher la suite de blocs menant à cette erreur. Les développeurs peuvent donc analyser le cycle pour en déceler la raison.

CHAPITRE 5

Résultats

Je vais maintenant présenter mes résultats obtenus sur différentes applications OCaml, j'ai souhaité appliquer mes recherches sur différents types d'application pour avoir des résultats différents. Les tests seront donc effectués sur les applications Alt-Ergo, Cumulus et OCamlopt. Alt-Ergo est un SMT-solveur, logiciel de résolution de formules mathématiques, Cumulus est une application web minimaliste pour le partage de liens, elle est basée sur Eliom outil client/server en OCaml. J'ai aussi pu étudier d'autres applications comme ocamlopt le compilateur natif pour OCaml ainsi qu'une exécution d'un programme d'un client d'OCamlPro.

Le problème avec ce dernier fut la taille du tas dépassant la quantité de mémoire de mon poste de travail. Lors du lancement de l'analyse le tas est chargé en mémoire, ici il est impossible de poursuivre l'analyse. Mon maître de stage à donc modifié le chargement du tas en mémoire pour que Memprof puisse se lancer. Malgré cela je n'ai pu adapter ce nouveau modèle à mes études.

5.1 Réduction de graphe

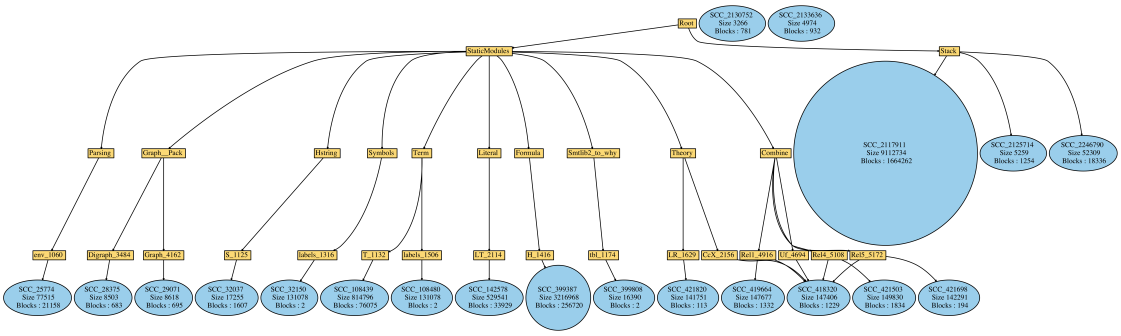


Figure 5.1 – Premier affichage des ensembles après réduction

Pour les premiers résultats concernant la visualisation du graphe de pointeur, j'ai donc lancé mon analyse combinant composantes fortement connexes et réduction de graphe sur des graphes de pointeurs issus d'exécutions des applications présentées précédemment. Le premier affichage fut simpliste (voir figure 5.1), ici les vingt plus gros ensembles de blocs sont affichés, leurs tailles sur le graphe étant proportionnelles à celle de l'ensemble en mémoire. Nous affichons aussi les chemins des racines aux blocs. Cette représentation nous montre quelle racine retient le plus de taille en mémoire.

Cet affichage n'apporte pas beaucoup de données, car seul le numéro de l'ensemble, sa taille et le nombre de blocs sont visibles. La seule information à en tirer est le ratio bloc/taille, par exemple cela peut mettre en avant un unique bloc retenant 20% de la

mémoire, on imagine un tableau.

Pour chaque ensemble j'ai tout d'abord souhaité savoir quel bloc retient le reste des données dans les ensembles, offrant de plus amples informations pour des types comme des tables de Hash, une entrée retient un ensemble de blocs. Sur la figure 5.2 les blocs d'entrée des ensembles sont représentés en bleu et le reste des données de l'ensemble en triangle rouge.

Nous avons aussi souhaité afficher un noeud (triangle inversé) contenant le reste des données non contenues dans nos ensembles, nous n'afficherons que les arêtes partant de nos ensembles vers ce nouveau noeud. Grâce à ces arêtes nous pouvons savoir quel ensemble n'est pas lié au reste, l'exécution de notre analyse sur Alt-Ergo à permis de mettre en évidence la présence de deux HashTable initialisées mais non utilisées.

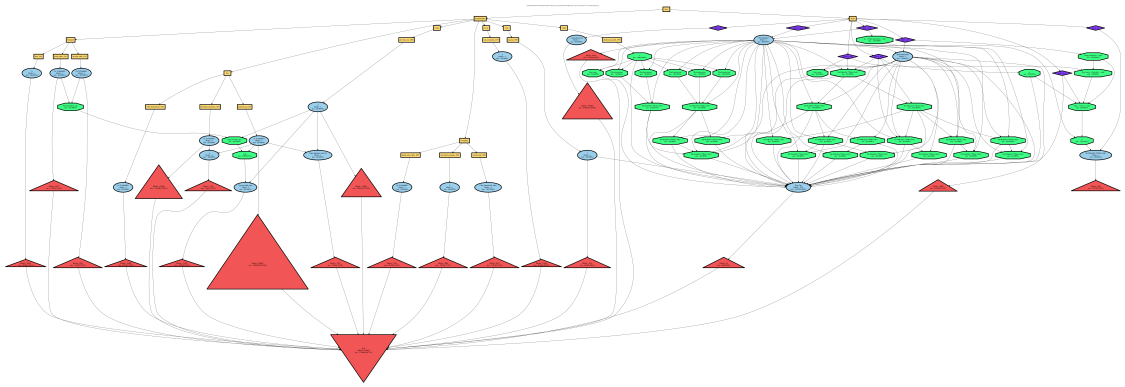


Figure 5.2 – Affichage des ensembles liés au reste

L'application de cette technique n'a pas été concluante pour Cumulus car le programme comporte beaucoup de partage et le programme se chargeant de convertir le fichier Dot en fichier image n'arrive pas à traiter les données.

5.2 Le partage

J'ai ensuite appliqué l'analyse de partage sur mes cas d'études. La plus pertinente est l'exécution sur Cumulus car l'application utilise de nombreuses bibliothèques extérieures sur les mêmes objets en mémoire. Dans un premier temps nous affichons donc sous forme de tableau HTML les tailles partagées et réelles retenues par chaque racine. Puis nous détaillons l'affichage des tailles, par type de racine, modules statique, dynamique, racine de la pile, etc.

Sur la figure 5.4 nous pouvons voir trois tailles différentes pour chaque racine, correspondant au différentes méthodes expliquées précédemment. La taille partagée sur la racine Root exprime ici le fait que deux racines, Static, Stack partage des données. Nous pouvons aussi pour chaque racine connaître la taille non partagée qu'elle retient, en faisant la différence entre ce nombre et la taille globale qu'elle retient nous pouvons calculer la

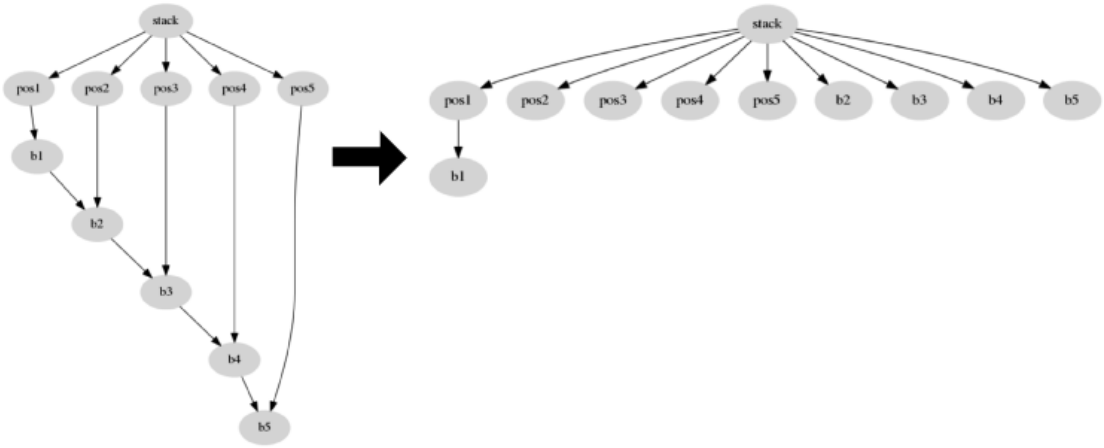


Figure 5.3 – Exemple de structure de données récursive dans la pile

part partagée.

Nous pouvons remarquer que la majeur partie de la mémoire est retenue par la pile (Stack), cette quantité est dominé directement par la pile, ce qui signifie que les données sont partagés.

Roots

[-] Click to collapse

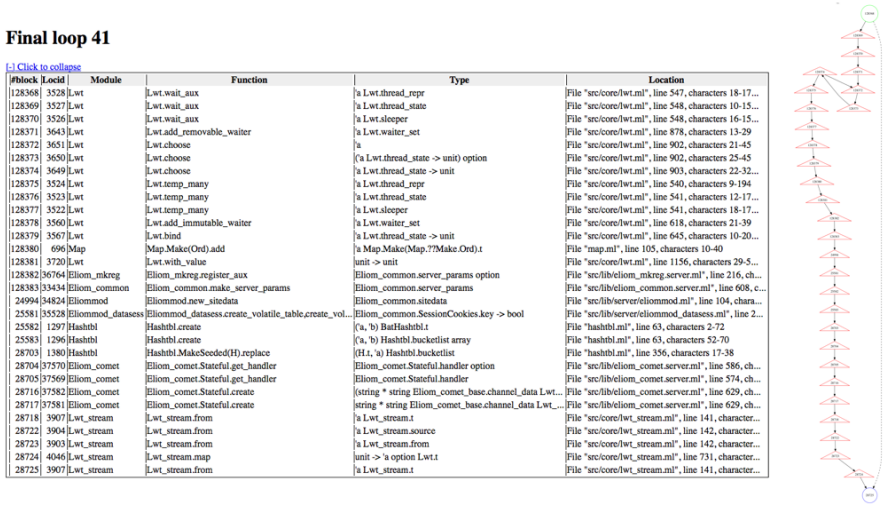
Name	Retained Size	%	Dominate Size	%	Shared Size	%
Root	16553718	100.0	207770	1.3	16547443.2	100.0
StaticModules	1516702	9.2	151357	0.9	1637007.5	9.9
DynamicModules	0	0.0	0	0.0	0.0	0.0
Stack	14829211	89.6	13785234	83.3	14910398.6	90.1
Cglobals	35	0.0	0	0.0	37.2	0.0
Finalizers	0	0.0	0	0.0	0.0	0.0
Hooks	0	0.0	0	0.0	0.0	0.0

Figure 5.4 – Exemple de l'affichage des tailles retenues par les racines

5.3 Les Finaliseurs

Les derniers résultats concernent l'étude des finaliseurs et la découverte d'éventuels cycles entraînant des fuites mémoire. Parmi nos applications, seul Cumulus utilise des finaliseurs, et lors de nos tests sur 73 finaliseurs plus de la moitié contenaient des cycles.

Ces 43 finaliseurs étaient identiques car ils correspondaient aux connections ouvertes durant l'exécution de Cumulus, et les ouvertures de connections sont effectuées à un seul endroit du code, le type est donc identique.



Final loop 41

[-] Click to collapse

#block	Locid	Module	Function	Type	Location
128368	3528	Lwt	Lwt.wait_aux	'a Lwt.thread_repr	File "src/core/lwt.ml", line 547, characters 18-17...
128369	3527	Lwt	Lwt.wait_aux	'a Lwt.thread_state	File "src/core/lwt.ml", line 548, characters 10-15...
128370	3526	Lwt	Lwt.wait_aux	'a Lwt.sleeper	File "src/core/lwt.ml", line 548, characters 16-15...
128371	3643	Lwt	Lwt.add_removable_waiter	'a Lwt.waiter_set	File "src/core/lwt.ml", line 878, characters 13-29
128372	3651	Lwt	Lwt.choose	'a	File "src/core/lwt.ml", line 902, characters 21-45
128373	3650	Lwt	Lwt.choose	('a Lwt.thread_state -> unit) option	File "src/core/lwt.ml", line 902, characters 25-45
128374	3649	Lwt	Lwt.choose	'a Lwt.thread_state -> unit	File "src/core/lwt.ml", line 903, characters 22-32...
128375	3524	Lwt	Lwt.temp_many	'a Lwt.thread_repr	File "src/core/lwt.ml", line 540, characters 9-194
128376	3523	Lwt	Lwt.temp_many	'a Lwt.thread_state	File "src/core/lwt.ml", line 541, characters 12-17...
128377	3522	Lwt	Lwt.temp_many	'a Lwt.sleeper	File "src/core/lwt.ml", line 541, characters 18-17...
128378	3560	Lwt	Lwt.add_immutable_waiter	'a Lwt.waiter_set	File "src/core/lwt.ml", line 618, characters 21-39
128379	3567	Lwt	Lwt.bind	'a Lwt.thread_state -> unit	File "src/core/lwt.ml", line 645, characters 10-20...
128380	696	Map	Map.Make(Ord).add	'a Map.Make(Map.??Make.Ord).t	File "map.ml", line 105, characters 10-40
128381	3720	Lwt	Lwt.with_value	unit -> unit	File "src/core/lwt.ml", line 1156, characters 29-5...
128382	36764	Eliom_mkreg	Eliom_mkreg.register_aux	Eliom_common.server_params option	File "src/lib/eliom_mkreg.server.ml", line 216, ch...
128383	33434	Eliom_common	Eliom_common.make_server_params	Eliom_common.server_params	File "src/lib/eliom_common.server.ml", line 608, c...
24994	34824	Eliommod	Eliommod.new_sitedata	Eliom_common.sitedata	File "src/lib/server/eliommod.ml", line 104, chara...
25581	35528	Eliommod_datasess	Eliommod_datasess.create_volatile_table,create_vol...	Eliom_common.SessionCookies.key -> bool	File "src/lib/server/eliommod_datasess.ml", line 2...
25582	1297	Hashtbl	Hashtbl.create	('a, 'b) BatHashtbl.t	File "hashtbl.ml", line 63, characters 2-72
25583	1296	Hashtbl	Hashtbl.create	('a, 'b) Hashtbl.bucketlist array	File "hashtbl.ml", line 63, characters 52-70
28703	1380	Hashtbl	Hashtbl.MakeSeeded(H).replace	(H.t, 'a) Hashtbl.bucketlist	File "hashtbl.ml", line 356, characters 17-38
28704	37570	Eliom_comet	Eliom_comet.Stateful.get_handler	Eliom_comet.Stateful.handler option	File "src/lib/eliom_comet.server.ml", line 586, ch...
28705	37569	Eliom_comet	Eliom_comet.Stateful.get_handler	Eliom_comet.Stateful.handler	File "src/lib/eliom_comet.server.ml", line 574, ch...
28716	37582	Eliom_comet	Eliom_comet.Stateful.create	(string * string Eliom_comet_base.channel_data Lwt...	File "src/lib/eliom_comet.server.ml", line 629, ch...
28717	37581	Eliom_comet	Eliom_comet.Stateful.create	string * string Eliom_comet_base.channel_data Lwt_...	File "src/lib/eliom_comet.server.ml", line 629, ch...
28718	3907	Lwt_stream	Lwt_stream.from	'a Lwt_stream.t	File "src/core/lwt_stream.ml", line 141, character...
28722	3904	Lwt_stream	Lwt_stream.from	'a Lwt_stream.source	File "src/core/lwt_stream.ml", line 142, character...
28723	3903	Lwt_stream	Lwt_stream.from	'a Lwt_stream.from	File "src/core/lwt_stream.ml", line 142, character...
28724	4046	Lwt_stream	Lwt_stream.map	unit -> 'a option Lwt.t	File "src/core/lwt_stream.ml", line 731, character...
28725	3907	Lwt_stream	Lwt_stream.from	'a Lwt_stream.t	File "src/core/lwt_stream.ml", line 141, character...

Figure 5.5 – Affichage du cycle entraînant la fuite mémoire

Pour chacun de ces finaliseurs nous affichons graphiquement, le cycle entraînant la fuite mémoire ainsi que sous forme de tableau apportant de plus amples détails. Bien que les finaliseurs soient identiques, les cycles peuvent varier car notre algorithme de détection ne renvoie pas le plus court chemin, mais le premier chemin qu'il trouve. Grâce à ces informations 5.5 le programmeur peut donc suivre les allocations pour trouver la raison de la fuite mémoire dans le code source.

CHAPITRE 6

Conclusion et perspectives

Durant ce stage nous avons proposé trois approches différentes sur l'analyse des graphes de pointeurs d'application OCaml. Ces trois approches permettent d'obtenir de plus amples informations que celles actuellement disponibles via l'outil Memprof, nous permettons la visualisation du graphe sous forme graphique, et sous forme de tableau. Bien qu'elles ne permettent pas de déceler à coup sûr une fuite mémoire, les méthodes présentées offrent de nouvelles approches du graphe de pointeur, pouvant faciliter la recherche de fuite mémoire.

Ces travaux m'ont permis de perfectionner mes connaissances en OCaml ainsi qu'en gestion de la mémoire des programmes à ramasse-miettes. Il m'a aussi été possible d'utiliser des outils comme git et opam qui me seront utiles pour mes travaux de thèse futurs.

Ce stage de recherche de Master recherche m'a permis de travailler sur un projet liant l'aspect industriel et l'aspect recherche, ce qui est en accord avec mon projet professionnel futur. L'aspect recherche à été présent tout au cours de mon travail, de la recherche de méthodes à leurs applications. Effectuant ce stage au sein de la société OCamlPro j'ai pu y découvrir les liens entre recherche et industrie via par exemple des résultats de Memprof sur des cas réels d'application.

Il existe encore beaucoup de travail possible sur l'analyse mémoire d'applications OCaml, tout d'abord concernant les travaux effectués, il serait bon d'utiliser encore plus la bibliothèque OCamlGraph, que ce soit pour la réduction du graphe, la recherche du plus court chemin pour les finaliseurs et l'affichage grâce au langage DOT, via les modules Contraction, Path et Dot de la bibliothèque.

Dans un second temps des analyses écartées au début du stage pourraient être mises en oeuvre, telle que la comparaison d'instantanés mémoire, ou plus complexe, le calcul de la durée de vie de chaque bloc mémoire, cette technique pourrait être réalisable après modification du compilateur. Ou encore l'étude de la gestion de la mémoire cache, qui sera une des premières approches de ma thèse.

Remerciements

Tout d'abord, je tiens à remercier mon maître de stage, Mr Fabrice Le Fessant, chercheur INRIA et conseiller scientifique dans l'entreprise OCamlPro, de m'avoir accueilli au sein de l'entreprise OCamlPro ainsi que pour ses conseils avisés et la confiance qu'il m'a accordée au long de ce stage.

Je remercie également toute l'équipe d'OCamlPro pour leur accueil, et en particulier Mr Cagdas Bozman et Mr Grégoire Henry, qui m'ont beaucoup aidé au cours de mon stage en particulier à la compréhension de Memprof.

Enfin, je tiens à remercier les personnes qui m'ont conseillé et relu lors de la rédaction de ce rapport de stage : Alwine Lambert, et Léo Pelletier.

Bibliographie

[1] Çagdas. Bozman. *Profilage mémoire d'applications OCaml*. 2014.

[2] Cagdas Bozman, Thomas Gazagnaire, Fabrice Le Fessant, and Michel Mauny. Study of ocaml programs' memory behavior, sep 2012. Presented at OUD'2012, Copenhagen.

[3] Cagdas Bozman, Michel Mauny, Fabrice Le Fessant, and Thomas Gazagnaire. Profiling the memory usage of ocaml applications without changing their behavior, September 2013.

[4] 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.

[5] Jim Smith and Ravi Nair. *Virtual Machines : Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[6] Damien Doligez, Inria Rocquencourt, and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems, 1994.

[7] Michael D. Bond and Kathryn S. McKinley. Bell : Bit-encoding online memory leak detection. *SIGPLAN Not.*, 41(11) :61–72, October 2006.

[8] Jacek Kukluk. *Inference of Node and Edge Replacement Graph Grammars*. PhD thesis, Arlington, TX, USA, 2007. AAI3258594.

[9] 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.

[10] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings. 37th International Conference on*, pages 202–213, 2000.

[11] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 115–124, New York, NY, USA, 2010. ACM.

[12] Vladimir Šor, Nikita Salnikov-Tarnovski, and Satish Narayana Srirama. Automated statistical approach for memory leak detection : Case studies. In *Proceedings of the 2011th Confederated International Conference on On the Move to Meaningful Internet Systems - Volume Part II*, OTM'11, pages 635–642, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 245–260, New York, NY, USA, 2007. ACM.

[14] Nick Mitchell. The runtime structure of object ownership. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 74–98, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Derek Rayside and Lucy Mendel. Object ownership profiling : A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 194–203, New York, NY, USA, 2007. ACM.

[16] Niklas Röjemo and Colin Runciman. Lag, drag, void and use&mdash ;heap profiling and space-efficient compilation revisited. *SIGPLAN Not.*, 31(6) :34–41, June 1996.

[17] Colin Runciman and David Wakeling. Heap profiling of a lazy functional compiler. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 203–214, London, UK, UK, 1993. Springer-Verlag.

[18] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *in Proceedings of the ECOOP '99 European Conference on Object-oriented Programming*, pages 116–134. Springer-Verlag, 1999.

[19] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[20] Çagdas Bozman, Grégoire Henry, Mohamed Iguernelala, Fabrice Le Fessant, and Michel Mauny. ocp-memprof : un profileur mémoire pour OCaml. In David Baelde and Jade Alglave, editors, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Le Val d'Ajol, France, January 2015.

[21] Istvan Jonyer, Lawrence B. Holder, and Diane J. Cook. Mdl-based context-free graph grammar induction and applications. *International Journal on Artificial Intelligence Tools*, 13 :65–79, 2004.

[22] OCamlPro. Ocp-memprof. http://memprof.typerex.org/. septembre, 2015.

[23] Eclipse. Mat. https://eclipse.org/mat/. septembre, 2015.

[24] Valgrind Developers. Valgrind. http://valgrind.org/. septembre, 2015.

[25] Oracle. Hprof. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html. septembre, 2015.

[26] OCamlPro. Alt-ergo. /http://alt-ergo.ocamlpro.com/. septembre, 2015.

[27] OCamlPro. Opam. http://opam.ocamlpro.com/.

[28] Ocsigen. Eliom. http://ocsigen.org/eliom/.