Compilation Avancée

Implémentation d'un garbage collector pour la *Mini Zam*

CARBONNEAU Danaël
28709878
MALENFER François
28706664

24 mars 2024

Table des matières

1 Introduction

1.1 Présentation du projet

La Mini-ZAM est une machine virtuelle pour le langage OCaml[Ler+], réalisée dans le cadre d'un précédent projet de l'UE de Compilation Avancée lors d'une précédente année.

Il s'agit d'une machine à pile, inspirée de la ZAM (ZINC Abstract Machine), qui interpète un *bytecode*. Dans le cas de la Mini-ZAM, il y en a 27. Chaque instruction interprétée modifie l'état interne de la machine virtuelle afin d'exécuter le programme OCaml associé.

1.1.1 La question de la mémoire dynamique

Pour des raisons d'efficacité, plutôt que d'implémenter cette machine virtuelle en OCaml, il est davantage souhaitable de le faire en C[KR06], un language de plus bas niveau.

Certaines instructions nécessitent de manipuler des valeurs allouées dans un tas : c'est notamment le cas pour les valeurs qui ne sont pas des entiers (blocks sur le tas), les environnements d'exécution, et les fermetures (fonctions accompagnées de son environnement).

Il se pose alors la question de la gestion de la mémoire : en C, elle n'est pas gérée de manière automatique, il faut donc trouver un mécanisme pour la gérer dans la machine virtuelle. Faire des `malloc` à chaque nouveau besoin de mémoire dynamique n'est pas envisageable car aucune instruction de passée à la machine virtuelle ne peut nous indiquer quand libérer ces zones mémoires (la gestion de la mémoire en OCaml se fait de manière automatique, il est donc attendu que notre bytecode fasse de même).

Il en résulte que notre Mini-ZAM écrite en langage C doit être munie d'un **Garbage Collector** (ramasse-miettes), qu'il nous faudra donc implémenter.

1.1.2 La notion de Garbage Collector

Un Garbage Collector est un algorithme qui va permettre de gérer automatiquement la mémoire d'un programme : il le fait en permettant de réutiliser la mémoire précédemment allouée mais qui ne pourra plus être utilisée.

Il existe plusieurs algorithmes différents permettant de faire cette détection de mémoire inutilisée et son son recyclage : Comptage de référence, Mark&Sweep, Mark&Compact, Stop&Copy, etc. Chaque algorithme présente des avantages et des défauts. Dans ce projet, nous étudierons

l'implémentation d'un Garbage Collector suivant l'algorithme du **Mark&Compact** pour une implémentation dans le langage C de la Mini-ZAM (fournie).

1.2 Prise en main de la MiniZam

1.2.1 Exemple de bytecode

Dans un premier temps, afin de mieux comprendre le bytecode de la mini-ZAM, nous avons traduit à la main un petit programme en OCaml en bytecode permettant d'être exécuté par la Mini-ZAM.

```
1 let rec f n =
2   if n > 1000 then 0
3   else (1 +( f (n+3)))
4
5 let _ = f 3
```

Figure 1 – Programme OCaml à traduire en bytecode pour la Mini-ZAM

1.2.2 `APPLY` et `APPTERM`

Dans l'implémentation fournie de la Mini-ZAM, pour les codes `APPLY` et `APPTERM`, un tableau dynamique, c'est à dire alloué sur le tas à l'aide d'un `malloc` est utilisé pour placer comme souhaité l'environnement, le PC [1], et le nombre d'arguments en plus sur la pile : afin de prendre en main la manipulation fine des zones mémoire de notre machine virtuelle, nous avons optimisé l'interprétation de ces deux codes afin de ne plus utiliser de tableaux temporaires dans lesquels copier les valeurs présentes sur la pile.

Pour `APPLY`, on décale les n arguments, en partant du plus haut sur la pile, de 3 cases, puis on ajoute en dessous l'environnement, le PC, et le nombre d'arguments en plus (`extra_args`).

Pour `APPTERM`, on calcule une nouvelle base qui se trouve dans le tableau représentant la pile à $sp - n - (m - n)$, où sp est le pointeur de pile, n le nombre d'arguments de l'appel et (m-n) les variables globales de l'appelant (qu'il va falloir dépiler). À partir de cette base, on décale les n arguments au bon endroit sur la pile et on replace le pointeur de pile au bon endroit.

Afin de vérifier que ces modifications sont correctes, il suffit de lancer nos tests avec cette nouvelle version de la Mini-ZAM, et de vérifier que ces derniers passent toujours.

1. Program Counter, tête de lecture du bytecode de la Mini-ZAM

```
1       BRANCH L2
2   L1: ACC 0
3       PUSH
4       CONST 1000
5       PRIM <=
6       BRANCHIFNOT L3
7       CONST 0
8       RETURN 1
9   L3: ACC 0
10      PUSH
11      CONST 3
12      PRIM +
13      PUSH
14      OFFSETCLOSURE
15      APPLY 1
16      PUSH
17      CONST 1
18      PRIM +
19      RETURN 1
20  L2: CLOSUREREC L1, 0
21      CONST 3
22      PUSH
23      ACC 1
24      APPLY 1
25      STOP
```

Figure 2 – Bytecode obtenu après traduction du programme

1.3 Fuites mémoire

En lançant la Mini-Zam sur des exemples à l'aide de valgrind[NS07], nous constatons qu'il y a des fuites mémoire : outre celles causées par les valeurs allouées pour l'interprétation du bytecode, nous en avons identifié deux dans le parser :

- Le fichier ouvert dans la fonction `parse` n'est pas fermé
- le champ label de la structure `lbl_list` n'est pas libéré (alors que la chaîne est allouée avec un `strdup`

En utilisant `fclose(f)` à la fin de la fonction `parse` et en libérant également le label dans la fonction `free_labels(lbl_list*l)`, nous n'avons plus que les fuites mémoire dues à l'interprète de bytecode, que nous allons gérer par la suite grâce au garbage collector.

2 Implémentation du Garbage Collector

Afin d'implémenter un garbage collector de type **Mark & Compact**, nous allons gérer la mémoire manuellement, ainsi, il nous faut ajouter dans notre état global (`caml_state`) un pointeur vers la zone mémoire qui sera notre tas.

2.1 Représentation des valeurs en mémoire

Dans la Mini-ZAM, les valeurs sont toutes représentées par des `mlvalue`, qui sont soit des pointeurs, soit un entier codé sur 63 bits (le bit de poids faible différencie les deux cas)

2.2 Allocation de mémoire dans la Mini-ZAM

Nous l'allouons dans son entièreté à l'initialisation de la machine virtuelle (avec une taille définie dans le fichier *config.h*). Il faut alors modifier la fonction `caml_alloc` (*alloc.c*) afin qu'elle retourne une zone de ce tableau : on maintient pour ça une variable globale `heap_free` qui retient la prochaine case libre du tas.

```
1 mlvalue *caml_alloc(size_t size)
2 {
3     mlvalue *addr_block = Caml_state->heap + heap_free;
4     heap_free += size;
5     return addr_block;
6 }
```

Figure 3 – Allocation dans notre tas

Suite à ces deux manipulations, utiliser valgrind sur nos tests nous montre qu'il n'y a plus aucune fuite mémoire : c'est du moins le cas vu de l'extérieur, les fuites étant désormais internes à un tableau que nous allons gérer manuellement.

On remarque, cependant, que pour un certain nombre de programmes [2], les tests ne marchent plus : nous n'avons, à ce stade, pas de GC, si un programme demande trop de mémoire par rapport aux 32Ko qui sont pour l'instant alloués, la machine virtuelle crash (à cause d'une erreur de segmentation).

2. list_1, list_2, list_3, list_4,

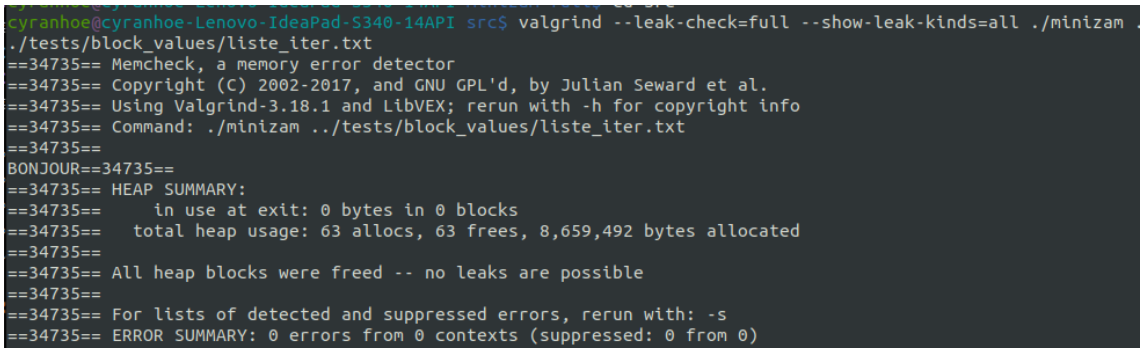

Figure 4 – Leak-check de valgrind nous montrant qu'il "n'y a plus" de fuite mémoire

2.3 Récupération de l'espace mémoire inutilisé

Pour notre garbage collector, nous allons implémenter un **Mark & Compact**. L'idée est la suivante :

- Depuis les racines du programme, on va opérer une phase de marquage : en suivant les pointeurs, on marque tous les blocs accessibles (ce qui revient à un parcours de graphe)
- Dans un second temps, nous allons opérer une phase de compactage : on va décaler vers le début du tas tous les blocks marqués (qui peuvent alors écraser les blocs qui ne le sont pas, ces derniers n'étant plus accessibles par le programme). Cette phase de compactage se déroule en 3 temps :
  - D'abord, on parcourt le tas une première fois afin de recalculer les adresses des blocks marqués
  - Puis, on parcourt nos objets marqués afin de mettre à jour les pointeurs
  - Enfin, on déplace tous les objets marqués à leur nouvelle adresse

2.3.1 Marquage

Notre parcourt du graphe de toutes les valeurs accessibles pour le programme à un instant T (où le GC est appelé) nous demande d'identifier des racines, à partir desquelles toutes les valeurs accessibles le sont encore pour le programme, et qui permettant l'accès à **toutes** les valeurs encore accessibles.

Nous avons identifié trois zones desquelles toutes les valeurs vivantes [3] du programme (uniquement) :

- La pile jusqu'au stack pointer

3. Une valeur vivante est une valeur accessible par le programme à l'instant T

- L'environnement, qui est un registre de la machine virtuelle et qui permet d'accéder aux valeurs courantes des variables locales
- L'accumulateur, qui est un registre de la machine virtuelle

Ainsi, en mettant l'environnement et l'accumulateur dans `Caml_state` afin d'y accéder de manière globale (et non plus localement à l'interprèteur de bytecode), les racines de notre parcours sont accessibles.

Notre marquage part donc d'une liste chaînée, que nous nommons *todo*, et qui contient initialement tous les pointeurs de la pile, l'accumulateur si ce dernier est un pointeur, et l'environnement [4]. Ces valeurs sont initialement marquées avec la couleur grise, pour indiquer que leur traitement est en cours.

Par la suite, on part de cette liste pour parcourir tous les blocks accessibles, les marquer en gris, et les y ajouter. Lorsqu'on a fini d'ajouter tous les descendants d'un nœud du graphe, on le marque en noir pour indiquer que son traitement est fini, et on continue tant que la liste n'est pas vide.

```
while (todo != NULL)
{
    tmp = todo;
    block = todo->block;
    todo = todo->next;
    free(tmp);
    size_block = Size(block);
    for (int64_t i = 0; i < size_block; i++)
    {
      if (Is_block(block[i]) && (Color((block[i])) == WHITE))
      {
        ptr_block = Ptr_val(block[i]);
        set_color((ptr_block - 1), GRAY);
        enfile(&todo, ptr_block);
        cpt_obj_mem++;
      }
    }
}
```

Figure 5 – Boucle de parcours des valeurs vivantes du programme

4. On identifie si les valeurs sont des pointeurs grâce à la macro `Is_block`

2.3.2 Calcul des adresses

Une fois la phase de marquage faite, on peut calculer les nouvelles adresses : pour cela, on utilise deux pointeurs : un vers le futur haut du tas, et un vers le haut du tas au cours du parcours.

Ces derniers commencent par pointer vers le début du tas, puis, à chaque fois qu'on rencontre un block marqué en Noir (valeur encore vivante), on incrémente le futur haut du tas. Afin de retenir les nouvelles adresses calculées, nous utilisons un tableau à deux entrées, une ligne pour l'ancienne, et une pour la nouvelle adresse, afin de garder l'association en mémoire.

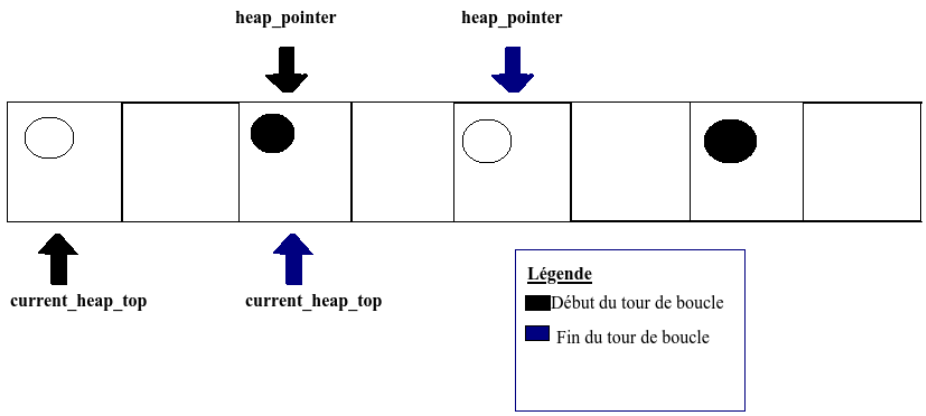

Figure 6 – Schéma représentant le tas au moment d'un tour de boucle du calcul des adresses

2.3.3 Application des adresses

Avant de pouvoir faire le décalage des blocs dans le tas, il nous faut cependant changer, partout où un block est pointé, l'ancienne adresse par la nouvelle à l'aide du tableau construit à l'étape précédente. Pour cela, on parcourt la pile, le tas (où on met à jour les adresses pour les blocks marqués en noir uniquement), et on met également à jour, au besoin, les pointeurs dans l'accumulateur et dans la variable globale env.

2.3.4 Compactage du tas

Une fois les nouvelles adresses calculées, on va pouvoir, enfin, faire "glisser" tous nos blocks encore vivants vers le début du tas. Pour cela, on se sert du fait que dans notre tableau de correspondances d'adresses, ces dernières sont croissantes : on peut alors commencer par le début du tableau, et copier chaque block dans l'adresse de destination calculée précédemment.

Le traitement étant terminé, on repasse tous les blocks vus à la couleur blanche (pour le prochain appel au GC)

2.4 Quand déclencher le Garbage Collector ?

Maintenant que nous avons un algorithme permettant de récupérer la mémoire non utilisée, il nous faut choisir quand le déclencher : la manière la plus évidente est que, tant que nous avons suffisamment de mémoire, il n'est pas nécessaire de "nettoyer" le tas des valeurs inutiles. Nous pouvons alors considérer qu'il n'est utile d'appeler le garbage collector que lorsqu'une allocation n'est pas possible dans le tas car nous avons dépassé sa capacité. Pour cela, il suffit, dans la fonction `caml_alloc`, de faire un test, avant l'allocation (telle que décrite en 2.2), afin de vérifier si on ne va pas dépasser la taille maximale du tas, auquel cas il devient nécessaire de faire appel au garbage collector pour *essayer* de récupérer de la place pour de nouvelles allocations.

Pour obtenir cette taille limite, il est important de se souvenir que la taille du tas (`Heap_size`) est allouée en octets, et non en nombre de `mlvalues`. Nos valeurs étant codées sur 64 bits, pour obtenir le nombre de cases disponibles dans le tableau, il nous faut diviser la taille du tas par 8 (il y a 8 octets dans un entier 64 bits). Il en résulte le test suivant :

```
1  mlvalue *caml_alloc(size_t size)
2  {
3    if (heap_free + size >= (Caml_state->heap_size) / 8)
4    {
5      mark_and_compact();
6    }
7    mlvalue *addr_block = Caml_state->heap + heap_free;
8    heap_free += size;
9    nb_alloc++;
10   return addr_block;
11 }
```

Figure 7 – Allocateur déclenchant le GC au besoin

3 Tests de notre Garbage Collector

3.1 Premiers tests

Dans un premier temps, nous avons voulu confronter notre GC à des programmes plutôt simples vis à vis de la mémoire (dans le banc de test fournis avec la minizam, il s'agit de tous excepté *bench/list_1, bench/list_2, bench/list_3,* et *bench/list_4.*).

Afin qu'un appel au GC soit nécessaire, nous avons mis une taille de tas réduite à 512 octets, soit 64 mlvalues.

On voit (à l'aide d'un print fait dans l'appel au GC) alors que pour

- *appterm/facto_tailrec*
- *appterm/fun_appterm*
- *bench/list_5*
- *block_values/insertion_sort*
- *rec_funs/facto*

Le GC est appelé, parfois à plusieurs reprises, et le résultat final reste correct.

```
[error] appterm/fun_appterm
  expected: 1
  got: Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
1
```

Figure 8 – Test effectué sur *appterm/fun_appterm* avec un tas de 512 octets

On peut de ces premiers tests inférer que notre GC conserve bien l'intégrité des programmes.

3.2 Le cas de *bench/list_1*

Ce programme est **très demandant** en taille de tas : malgré des appels au GC, si le tas n'est pas suffisamment grand, alors il y aura une erreur de segmentation (notre tas ne gère pas encore proprement les dépassements, et crashe lorsque sa capacité maximale est atteinte).

Expérimentalement, nous avons mesuré qu'avec un tas de 256 Kio, le programme pouvait tourner correctement à l'aide de notre GC.

```
[error] bench/list_1
  expected: 10000
  got: Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
Appel au GC
10000
```

Figure 9 – Test effectué sur *bench/list_1* avec un tas de 256 Kio

3.3 Les cas de *bench/list_2*, *bench/list_3*, et *bench/list_4*.

Ces trois programmes on mis notre GC à rude épreuve : ils demandent une grande quantité de mémoire, et beaucoup d'appels au GC, ce qui peut prendre un certain temps. Nous avons , pour pouvoir réaliser nos tests dans un temps réaliste modifié ces programmes : la liste sur laquelle les fonctions sont appelées fait 10000 éléments dans nos tests.

Il nous faut alors une taille de tas très importante pour que le programme puisse fonctionner (même avec le garbage collector). Expérimentalement, nous observons qu'un tas de 256 Kio (similaire au cas de *bench/list_1*) au minimum est nécessaire pour que les tests *bench/list_2* et *bench/list_3* fonctionnent.

```
[OK] bench/list_1
[OK] bench/list_2
[OK] bench/list_3
```

Figure 10 – Tests effectués sur *bench/list_2* et *bench/list_3* avec un tas de 256 Kio

Pour que le *bench/list_4* passe, il nous faut un tas de 512 Kio.

```
[OK] appterm/fun_appterm
[OK] bench/list_4
[OK] bench/list_5
```

Figure 11 – Test effectué sur *bench/list_*avec un tas de 512 Kio

3.4 Conclusions sur cette première phase expérimentale

Ces différents tests nous amènent à nous dire que la taille du tas devrait pouvoir s'adapter à la taille du code, et qu'on ne voudrait pas allouer directement et pour tous les programmes un tas aussi grand que celui utilisé pour *bench/list_4*.

Les informations intéressantes à retenir de nos tests sont d'une part que notre GC semble fonctionner, mais qu'il prend un certain temps (surtout si on manipule une grosse mémoire), et que certains programmes ont besoin de beaucoup plus de place **utilisée** que d'autres, ce qui nous ouvre vers trois enjeux : étendre la taille du tas au fil de l'exécution, bien choisir quand déclencher le GC et améliorer sa complexité

4 Compilation et lancement des tests

À la compilation, nous avons quelques *warnings* : mis à part les formats d'affichage [5], dans la fonction *apply_new_addr* (du fichier *alloc.c*), qui correspond à la seconde passe de la seconde phase de notre algorithme. Ces derniers s'expliquent par le fait que dans notre représentation des valeurs, les `mlvalue` sont aussi des `mlvalue *`. Dans notre tas, les cases sont soit des pointeurs, soit des entiers (soit des headers, encodés sur un entier), avec une représentation des types uniformes (un pointeur est un entier représentant une adresse...). Lorsqu'on met à jour les adresses, on met dans les cases (contenant un type `mlvalue`) du tas des valeurs qui sont des pointeurs (`mlvalue *`). Le compilateur nous avertit, mais c'est une manière de faire qui fonctionne et a le comportement attendu.

Afin de lancer les tests tels que nous les avons conduits, il faut changer dans le fichier *src/config.h* la taile du tas afin d'observer les comportements attendus. Afin d'obtenir les affichages (attention, lorsqu'ils sont présents, le script indique que les tests ne passent pas...), il suffit d'y écrire `#define DEBUG`.

5. Et une valeur "non utilisée" dans les macros permettant de push et de pop dans la pile (ce warning n'a aucune incidence sur le fonctionnement du GC)

5 Critiques et améliorations

Ainsi, d'après ce que nous avons retenu de la phase expérimentale, trois axes de critique ressortent de notre implémentation d'un Garbage Collector :

Une première remarque est que, dans le cas où malgré un appel à ce dernier, il n'y a pas assez de place en mémoire, à ce stade, si ce cas de figure se présente, notre machine virtuelle crashe (ce qui est cohérent).

Une seconde est que la stratégie choisie pour le délenchement du Garbage Collector fait que, de manière très inéquitable, une allocation (non choisie ni prévisible par le programmeur qui veut faire tourner son code sur la Mini-ZAM) va durer beaucoup plus longtemps que les autres car elle devra faire appel au GC, là où les autres ne font qu'une simple arithmétique sur des pointeurs.

Enfin, nos tests nous ont fait remarquer que notre GC est plutôt lent : notre implémentation a une grande complexité, qui pourrait être améliorée.

5.1 Ré-allocation dynamique du tas

Concernant la première remarque, une solution au problème du tas trop petit malgré un passage du GC, que nous avons implémentée, est de faire une **réallocation** : lorsque la taille maximale du tas est atteinte, on va chercher à reconstruire un tas plus grand, qui aurait alors la capacité nécessaire (ou pas, auquel cas on augmente la taille jusqu'à ce que ce soit bon).

Pour cela, il existe en C une fonction `realloc`, qui permet, si c'est possible, d'étendre la taille d'une zone allouée, et qui sinon alloue une autre zone mémoire et y copie le contenu de la zone précédemment allouée (avant de la libérer).

Cependant, dans notre zone mémoire, certaines de nos valeurs sont des pointeurs, qui réfèrent donc toujours à l'ancienne zone (qui n'est plus accessible, il y a des risques de *segmentation fault*). On va donc passer sur notre nouveau tas (ainsi que sur la pile et l'accumulateur et la variable d'environnement), pour mettre à jour toutes les adresses à l'aide de la formule suivante :

$$addr_{new} = addr_{old} + (heap_{new} - heap_{old})$$

5.1.1 Implémentation

Nous avons écrit une fonction realloc, appelée si l'appel au GC ne permet pas de récupérer assez de mémoire pour l'allocation, qui multiplie la taille du tas par 2, fait un appel à la fonction *realloc* de la libC, puis calcule le décalage des adresses, et parcourt tout le tas, la pile, et les variables globales du programme (`env` et `acc`) pour mettre à jour les pointeurs avec le calcul d'adresse décrit précédemment.

5.1.2 Tests

Comme pour l'implémentation du GC, pour tester la réallocation, nous avons commencé par des exemples simples (sans les bench sur de grandes listes), avec un très petit tas, afin de voir des petits programmes avoir besoin de réallouer leur tas.

```
[error] block_values/liste_iter
  expected: BONJOUR0
  got: Appel au GC
Appel à realloc
appel à realloc
Appel au GC
Appel à realloc
appel à realloc
BONJOUR0
```

Figure 12 – Test du GC avec réallocation automatique de mémoire pour *block_value/liste_iter*

Ce premier test nous permet de voir des exemples de programmes où suite à une réallocation du tas, le GC est à nouveau appelé, et fonctionne toujours !

Cependant, pour les programmes plus gourmands en mémoire tels que les programmes de bench sur de grandes listes, nous avons des erreurs de segmentation suite à la réallocation dont nous ne parvenons pas à trouver la cause... Une utilisation de Valgrind[NS07] nous permet cependant d'identifier que nous tentons toujours d'accéder à des adresses correspondant à l'ancien tableau, notamment par l'environnement.

Cette réallocation dynamique est donc encore à parfaire, mais marche sur le principe, et sur de petits exemples...

5.2 Stratégie de déclenchement du gc équitable

Un autre soucis de GC est qu'il va se pénaliser une allocation spécifique, qui sera alors beaucoup plus lente que les autres : on peut imaginer d'autres stratégies, comme un arrêt aléatoire du programme pour faire appel au Garbage Collector.

5.3 Amélioration de la complexité

Enfin, un problème sous-jacent à notre GC, déjà soulevé par le point précédent, est son temps de calcul : sur un tas de taille conséquente, déclencher l'algorithme prend du temps.

La partie de l'algorithme qui prend le plus de temps, et qui pourrait être améliorée, est l'association entre l'ancienne et la nouvelle adresse : avec notre tableau à deux entrées, en plus d'avoir une grande complexité spatiale (le tableau prend de la place en mémoire), rechercher la nouvelle adresse d'un block se fait en un temps linéaire au nombre de blocks vivants, ce qui est à faire pour chaque block. L'étape décrite en 2.3.3 se fait donc en $O(n^2)$, ce qui peut être long s'il y a beaucoup de valeurs vivantes dans le programme.

Plusieurs idées pourraient être mises en place pour améliorer cette partie : implémenter une structure pour associer une ancienne à une nouvelle adresse dont la recherche est plus efficace que $O(n)$, comme une table de hachage.

On pourrait également imaginer que les headers nous permettent de stocker la nouvelle adresse, mais il faudrait pour cela les représenter sur deux blocks, ce qui changerait beaucoup de choses sur notre manipulation des valeurs dans la Mini-Zam [6]

6 Conclusions

Pour conclure, ce projet nous aura permis d'en apprendre plus sur le fonctionnement interne d'un Garbage Collector, et plus largement de la manière dont la représentation en mémoire fonctionne en OCaml[Ler+], notamment dans la Mini-Zam.

Nous avons cependant été confrontés à beaucoup de difficultés lors de l'implémentation, qui nous ont permis de davantage comprendre les problèmes posés par la gestion de la mémoire (temps pris par l'algorithme, erreur de segmentation, problèmes de cast, etc.), mais aussi les limites d'un Mark & Compact, notamment concernant sa complexité temporelle.

Face à ces limites, on peut alors réfléchir à la pertinence de l'utilisation de cet algorithme pour une machine virtuelle. Notons, par ailleurs, que l'algorithme que nous avons implémenté n'est pas celui de la machine virtuelle d'OCaml[Ler+], qui est bien plus élaboré et fonctionne avec plusieurs zones de tas (c'est un hybride d'un algorithme incrémental et generationnel).

6. Les limites de cette solution sont alors que le tas sera plus rapidement rempli, ce qui augmentera le nombre de fois où on fait appel au GC, ce qui peut risquer de faire perdre du temps...

Références

[KR06] Brian W Kernighan et Dennis M Ritchie. *The C programming language*. 2006.

[NS07] Nicholas Nethercote et Julian Seward. "Valgrind : a framework for heavyweight dynamic binary instrumentation". In : *SIGPLAN Not.* 42.6 (juin 2007), p. 89-100. issn : 0362-1340. doi : 10.1145/1273442.1250746. url : https://doi.org/10.1145/1273442.1250746.

[Ler+] Xavier Leroy et al. "The OCaml system : Documentation and user's manual". In : *INRIA* 3 (), p. 42.