Sorbonne Université, Master 1 STL
UE Compilation Avancée
Rapport de projet

Pablitto Bello Loïc Sylvestre

2 avril 2020

1 Introduction

On présente ici deux implémentations de *glaneurs de cellules*, de types Stop&Copy et Mark&Sweep, pour le *runtime* de la `mini-ZAM`[1]. Des solutions sont proposées afin de garantir en pratique la correction de notre programme et pour atteindre des performances raisonnables, à la fois en terme de temps d'exécution et du point de vue de l'occupation mémoire. Pour ce faire, nous avons suivi en détails les recommandations de l'enseignant : notamment la mise en place d'une *mémoire paginée*, le recours à de multiples *freelists* et la dérécursivation de l'algorithme de marquage. Parfois, des choix différents ont été opérés, et seront justifiés.

2 Lancement du projet

Le dossier tests contient l'ensemble des tests du projet. Le dossier src contient les fichiers sources. Nous avons totalement implémenté les unités de compilation suivantes, adossées au *runtime* de la `mini-ZAM` :

— `stop_and_copy.c|h`, implantant un Stop&Copy ;
— `mark_and_sweep.c|h`, `list.c|h` et `freelist.c|h`, implantant un Mark&Sweep.

La configuration des deux GC est personnalisable dans le *header* config.h ; on commentera notamment l'une des deux lignes suivantes en fonction du GC que l'on voudra utiliser :

1. Il s'agit d'une version simplifiée de la machinerie virtuelle OCaml (ZAM).

```
#define MARK_AND_SWEEP
#define STOP_AND_COPY
```

À partir de la racine du projet, on peut :

— lancer la batterie de tests avec la commande `make test`

— compiler le projet et produire un exécutable avec la commande `make` ; l'exécutable obtenu se nommera `minizam` et se trouve dans `src` ; on pourra s'en servir pour interpreter du *bytecode* exécutable par la `mini-Zam`, par exemple :

```
$ ./src/minizam tests/bench/list_4.txt -res
100000
```

— obtenir un fichier de *bytecode* exécutable par la `mini-Zam`, à partir d'un fichier source OCaml :

```
$ ./ml\_to\_bytecode.pl tests/bench/list_4.ml
```

3 Stop&copy

3.1 Organisation

Le Stop&Copy est fonctionnel et utilise le redimensionnement des *semispaces*. On lui alloue une mémoire initiale de 32KB, modifiable dans `domain.c`. Lors d'une allocation, la fonction `mlvalue *stop_and_copy_alloc(size_t)` est appelée, avec le nombre totale d'octets à allouer. S'il reste assez de place dans *from_space* pour faire une allocation, on incrémente le pointeur *heap_pointer* selon la taille désirée et retourne le pointeur vers la première case mémoire allouée dans le tas. Si le *from_space* n'a pas assez de place, la fonction `void run_gc()` est appelée, elle applique l'algorithme de Cheney en parcourant d'abord les racines (pile, `accu`, puis `env`). Elle parcourt ensuite le *to_space* afin d'ajuster les pointeurs. Elle échange *from_space* et *to_space* et replace le `heap_pointer` sur la première case allouable dans le nouveau `from_space`. Finalement, elle appelle la fonction `void resize_spaces()` qui se charge de redimensionner les *semispaces* si nécessaire.

3.2 Redimensionnement

Pour traiter le redimensionnement, on regarde d'abord si celui-ci est nécessaire. On alloue ensuite, un tas à la nouvelle taille après redimensionne-

ment, on peut maintenant copier les *mlvalues* de l'ancien tas vers le nouveau. Après ça, il faut réajuster tous les pointeurs qui pointent vers des éléments de l'ancien tas vers les éléments correspondants dans le nouveau. Pour cela, à chaque fois qu'il faut ajuster un pointeur, on récupère l'index de sa valeur dans l'ancien tas (sa position) et on pourra en déduire où est ce qu'on doit pointer dans le nouveau tas.

On parcours donc les racines (pile, `accu` et `env`), ainsi que le nouveau tas et ajuste les pointeurs. Il ne nous reste ensuite qu'à échanger les deux tas, libérer l'ancien et fixer la position de *heap_pointer*. Pour redimensionner le *to_space*, on se contente d'en créer un nouveau à la bonne taille, de les échanger et de libérer l'ancien.

3.3 Mémoire

Pour voir si le programme produit des fuites de mémoire, on utilise `Valgrind`, qu'on pourra par exemple lancer en exécutant :

```
$ valgrind src/minizam tests/bench/list_1.txt
```

`Valgrind` indiquera qu'il n'y a aucune fuite mémoire possible. On libère, en effet la mémoire utilisé par le Garbage Collector (tas) ainsi que la pile, en fin de programme, avec la procédure `void free_domain()` définie dans `domain.c`.

3.4 déboggage

En plus de `Valgrind` et `gdb`, nous avons programmé plusieurs fonctions pour le déboggage, afin par exemple d'afficher le contenu de la pile et des tas, des statistiques diverssent sur l'état du GC, ou de quoi vérifier s'il reste des pointeurs de *to_space* vers *from_space* à la fin de la fonction `run_gc()`.

Bien que nous ayons supprimé ces fonctions et affichages pour ce rendu, afin d'en faciliter la lecture, ce fut une étape importante dans l'écriture de ce GC.

4 Mark&sweep

4.1 Un marquage itératif

Le Mark&Sweep est une technique de *garbage collection* opérant en deux étapes : le marquage de l'ensemble des blocs atteignables à partir d'un ensemble de racines, puis la libération des blocs non-marqués. Le marquage

se définit très simplement de façon récursive. Pour autant, la pile d'appel de notre langage d'implantation (`C`) a une taille limitée qui contraindrait par là même le nombre de blocs alloués à un instant donné de l'exécution du programme. L'algorithme de DEUTSCH–SCHORR-WAITE [1] propose une approche élégante pour parcours le graphe des valeurs accessibles. Il permet en quelque sorte de *retrouver son chemin* en stockant une information supplémentaire dans les blocs. Comme cette technique nécessiterait de modifier – plus ou moins profondément – la représentation des valeurs dans le *runtime* de la `mini-ZAM`, on propose plus simplement de recourir à une *pile explicite*, implantée par un tableau, comme support d'une version itérative de l'algorithme de marquage.

4.2 Stockage des gros objets

Lors de l'exécution d'un programme, les objets de tailles importantes sont en général peu nombreux et ont des durées de vie longues. Il est donc, raisonnable d'allouer ces objets individuellement, par appel à la fonction `C` *malloc*. On propose de stocker ces objets dans une liste simplement chaînée. La phase de *sweep* se résume alors à un parcours de cette liste, avec libération des objets non marqués par appel à la fonction `C` *free*. Notre implémentation définit la taille `BIG_OBJECT_MIN_SIZE` au delà de laquelle une valeur allouée sera considérée comme faisant partie des « gros objets ».

4.3 Stockage des petits objets et *freelist*

Les allocations d'objets de petites tailles sont fréquentes dans les langages fonctionnelles — on pense aux *paires pointées* en `LISP`, ou aux fermetures en `Caml`, objets dont la durée de vie est courte le plus souvent. Classiquement, l'allocation d'un tel objet se résume au parcours d'une *freelist* – c'est à dire une liste de blocs disponibles – à la recherche d'un bloc de taille suffisante. Se pose la question de la structure de données à utiliser pour représenter la *freelist*. Lors de la libération d'un bloc, son header est conservé intact afin que l'*information de taille* puisse être à nouveau consulter en vue d'une future réutilisation. Pour autant, il se trouve que le champ 0 d'un objet ne transporte *a priori* pas d'information, son contenu étant obsolète une fois le bloc libéré. Ainsi, nous pouvons stocker dans ce champ 0 un pointeur vers le reste[2] de la *freelist*, en sorte que la structure d'une *freelist* ait un coût nul en mémoire. À noter que cette utilisation du champ 0 de chaque bloc libéré nécessite d'adapter la représentation des valeurs dans la `mini-ZAM` afin que

2. La *freelist* vide est représentée par l'entier 0.

tout bloc – et en particulier tout bloc vide – comprennent bien au moins deux mots mémoires : son header et un champ 0.

4.4 Un allocateur paginé

Si la *freelist* est vide, alors nous devons faire appel à un autre allocateur, servant en quelque sorte de *bootstrap*. Pour cela, nous avons organisé la mémoire en pages de 64 Kilooctets, l'ensemble des pages étant stocké dans une unique liste simplement chaînée. Nous pouvons alors proposer un allocateur puisant de manière contiguë des blocs dans une page, tant qu'il reste de la place dans celle-ci. Si l'allocation échoue faute de place, le dernier bloc disponible dans la page est ajouté à la *freelist*[3] ; une nouvelle page est créée par appel à la fonction `C` *malloc*, puis l'allocation demandée est alors réalisée au début de cette nouvelle page. Cette approche diffèrent de la stratégie proposée dans l'énoncé du projet, consistant à charger entièrement une page fraîche dans la *freelist*. Elle semble cependant bien adapter à la gestion de *freelists* multiples.

4.5 Recherche dans une freelist

Nous avons implémenté une stratégie first fit de recherche d'un bloc de taille suffisante dans une *freelist*. Parallèlement, nous avons expérimenté plusieurs méthodes d'insertion de blocs libres dans la *freelist* : ajout en tête, *freelist* ordonnée suivant la taille des objets, *freelist* triée par adresse. Cette dernière approche (utilisée dans le runtime de `Caml-light`[4] est bien adaptée pour la fusion des blocs libres contigus (*coalescing*). Nous avons cependant choisi de ne pas réaliser de fusion de blocs ; estimant préférable de conserver la taille exacte des blocs dans le header, pour éviter d'alourdir la structure de la freelist. En outre, nous proposons un système à *multiples freelists* qui rend à la fois mal-aisée et peu pertinente la fusion des blocs contigus.

4.6 Balayage de l'espace paginé

Lors de la phase de balayage, il faut être en mesure de parcourir l'ensemble des objets du tas. C'est pourquoi notre implémentation maintient un liste simplement chaînée des objets alloués. Cette structure de liste n'est pas nécessaire et pourrait être retiré par la suite. En effet, nous pourrions réaliser

3. Les blocs de un mot font exception : ils ne sont pas ajoutés à une *freelist* puisque toute valeur allouée dans notre le *runtime* de la `mini-ZAM` occupe au moins deux mots.

4. https://github.com/camllight/camllight/blob/master/sources/src/runtime/freelist.c

un parcours des pages suivant l'ordre contigu des blocs, de manière analogue à ce qui a été mis en œuvre dans le Stop&Copy. Il faudrait quoi-qu'il en soit prendre garde à correctement *enjamber les miettes* et donc, d'une manière ou d'une autre, stocker une information supplémentaire (*eg.* un pointeur) pour chaque bloc.

4.7 Multiples *freelists*

Notre GC Mark&Sweep s'appuie sur un tableau de 64 *freelists*. Les *freelists* 0 à 15 accueillent des blocs de tailles respectives 1 à 16 mots. Ce choix d'implantation accroît considérablement les performances du `Mark&Sweep`, du point de vue du temps d'exécution. En effet, l'insertion et la recherche dans une *freelist* sont ainsi réalisées en temps constant pour les valeurs de petites tailles qui, comme dit précédemment, sont allouées massivement au cours de l'exécution du programme. Par ailleurs, les *freelists* 15 à 64 sont organisées de telle manière qu'un bloc de taille n soit associé à la *freelist* $i = \lfloor \log_2 n \rfloor$. Ce choix tient de l'intuition suivante : le nombre de blocs alloués de taille n décroît significativement à mesure que n grandit.

5 Modification de la machinerie virtuelle

5.1 Blocs vides

Nous avons été amené à modifier la représentation des valeurs dans la `mini-Zam`, afin que les blocs vides occupent non pas un mais deux mots, le champ 0 permettant alors de stocker un éventuel *forwarding pointer* lors d'un Stop&Copy, ou chaîner les blocs de la ou des *freelists*. S'est également poser la question de modifier l'information de taille des bloc vide, contenu dans le header. Des instructions de la `mini-Zam` comme VECTLENGTH ou RESTART calcule en effet cette information de taille ; il faut donc veiller à ne pas changer le sens du programme. Notre choix d'implantation consiste à supposer que l'interprète de la `mini-Zam` ne consulte jamais la taille d'un bloc vide ; ce qui nous permet de fixer à 1 l'information de taille contenue dans le *header* des blocs vides. Le parcours du tas par ordre contigu des blocs est ainsi faciliter, notamment dans le Stop&Copy.

5.2 Accès aux racines depuis le GC

Nous avons globalisé les registres accu et env ainsi que le pointeur de pile, afin de pouvoir les utiliser comme racines. La `mini-ZAM` ne permettant

pas de définir de nouvelle racine globale, l'interprétation des instructions CLOSURE CLOSUREREC et GRAB a été modifiée, sur le modèle des codes fournis, en sorte que les valeurs manipulé par l'interprète soient bien toutes accessibles à partir des racines. À ce propos, nous avons été amené à résoudre un bug particulièrement subtile dans le Stop&Copy. En l'occurence, nous les allocations réalisés par l'interprète consistaient en des appels de fonctions. Par exemple, l'appel `make_closure(pc - 3, env)` alloue une fermeture et copie notamment le registre `env` dans le `champ 1` de celle-ci. Naturellement, l'allocation peut provoquer un déclenchement du GC, ce qui dans le cas du Stop&Copy a pour conséquence le *déplacement* des racines. Or, du fait de l'appel par valeur, le second argument de `make_closure` se trouve être en fait une copie obsolète de l'adresse contenue dans le registre `env`, d'où une erreur de segmentation intervenant plus ou moins tardivement dans l'interprète. Une fois la raison de ce bug détectée – non sans difficulté – nous sommes parvenu à un comportement correcte en recourant à des macros d'allocation, et non plus des fonctions, astuce qui avait été d'ailleurs suggérée par l'enseignant.

6 Conclusion

Nous proposons une implantation opérationnelle d'un Stop&Copy et d'un Mark&Sweep. Cette exercice à permis de prendre conscience des avantages et inconvénients de ces deux algorithmes. Le Stop&Copy consomme plus de mémoire, le tas étant partagé en deux *semispaces* ; il est cependant très rapide en pratique. Le Mark&Sweep consomme moins de mémoires, mais les blocs sont disséminés en de multiples emplacements. Bien que conceptuellement simple, cet algorithme difficile est à programmer de façon correcte. Il en outre excessivement lent et limite le nombre de mots alloué à un instant de l'exécution du programme, le marquage récursif pouvant en effet provoquer un débordement de la pile de notre langage d'implémentation (`C`). Suivant les recommandations données dans le sujet, nous avons cependant pu réaliser une implantation efficace du Mark&Sweep, avec marquage itératif, mémoire paginée, allocateur spécialisé pour les *gros objets*, et un ensemble de freelist agencé suivant des critères experimentaux. Nous sommes maintenant en mesures de combiner ces deux algorithmes en vue d'implémenter un GC générationnel, qui est historiquement – et en pratique – bien adapter au *runtime* d'OCaml.

Références

[1] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8) :501–506, 1967.