Sorbonne Université, Master 1 STL
UE Compilation Avancée

Rapport de projet

Bello Pablito
Sylvestre Loïc

Dans ce projet, nous avons implanté deux récupérateurs automatiques de mémoire, de types Stop&Copy et Mark&Sweep, pour une implémentation simplifiée de la machinerie virtuelle OCaml, ou mini-ZAM par la suite.

Organisation des fichiers / Lancement du projet

Le dossier **tests** contient l'ensemble des tests du projet.

Le dossier **src** contient les fichiers sources du projet. Voici les fichiers ajoutés à ceux fournis avec la minizam :

- **mark_and_sweep.h / .c** : fichiers du mark & sweep
- **stop_and_copy.h / .c** : fichiers du stop & copy
- **freelist.h / .c** : fichier pour la free list du mark & sweep
- **list.h / .c** : contient une structure de liste utile pour le mark & sweep

La configuration des gc se fait dans le fichier **config.h**, on commentera notamment l'une des deux lignes suivantes :

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

en fonction du GC que l'on veut utiliser.

A 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 exécuter manuellement un fichier .txt contenant du bytecode, par exemple ici le fichier list_4.txt : **src/minizam tests/bench/list_4.txt**
- Pour obtenir un fichier .txt contenant du bytecode à partir d'un fichier .ml, on exécutera : **./ml_to_bytecode.pl monfichier.ml**

Stop & copy

Organisation

Le stop & copy est fonctionnel et utilise le redimensionnement des semi space, 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 n)** 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).
Ensuite elle parcourt le **to_space** afin d'ajuster les pointeurs.
On échange ensuite **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 semi spaces si nécessaire.

Redimensionnement

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

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()** qui est dans **domain.c**.

Débogage

En plus de Valgrind et gdb, nous avons fait plusieurs fonctions pour le débogage : afficher le contenu de la pile et des tas, affichages divers, vérifier s'il reste des pointeurs de **to_space** vers **from_space** à la fin de la fonction **run_gc()**.

Néanmoins, nous avons supprimé toute ses fonctions et affichages afin que le code soit plus facilement lisible, mais ce fut une étape importante.

Mark & sweep

Le mark&sweep est une technique de *garbage collection* qui opère sur le tas en deux étapes. Le marquage de l'ensemble des blocs atteignables à partir d'un ensemble de racines, puis la libération de tous les blocs non-marqués. Nous déclenchons le GC à chaque fois que l'occupation mémoire du programme a augmenté de 50 %.

Stockage des gros objets

Lors de l'exécution d'un programme, les objets de tailles importantes sont en général peu nombreux et ont des durée de vie longues. Il est donc, raisonnable d'allouer ces objets individuellement, par appel à la fonction ***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 ***free***. Notre implémentation définit une constante `BIG_OBJECT_MIN_SIZE` indiquant la taille au delà de laquelle une valeur allouée sera considérée comme faisant partie des "gros objets".

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 de petite taille consiste à trouver un pointeur vers un zone mémoire de taille suffisante. Ce pointeur est stocké dans une ***freelist***, ie. une d'objets préalablement libérées. Se pose la question de la structure de données à utiliser pour représenter la freelist. Il se trouve que, lors de la libération d'un objet inaccessible depuis les racines du GC, on conserve intacte le ***header*** de ce bloc afin de pouvoir en déterminer la tailler en vue d'une future réutilisation. En revanche, le ***champ 0*** d'un objet ne transporte plus d'information pertinente. On propose donc de placer dans ce champ un pointeur vers l'objet suivant dans la ***freelist***, la ***freelist*** vide étant alors représentée par un pointeur nul. À noter que cette utilisation du ***champ 0*** des objets pour encoder la structure de la ***freelist***, nécessite d'adapter la représentation des valeurs dans la ***mini-ZAM*** afin que des ***blocs vides*** comprennent bien au moins deux mots mémoires : son ***header*** et un ***champ 0***.

Un allocateur paginé

Si la ***freelist*** est vide, alors nous devons faire appel à un autre allocateur, servant en quelque sorte au ***bootstrap*** pour celle-ci. La gestion d'un tas sous forme d'un unique tableau, comme c'est le cas dans le **Stop&Copy**, n'est pas adapté à cette situation. En effet, la multitude de petits objets alloués dans cet espace mémoire provoquerait un phénomène d'***émiettement***, et par ailleurs, le redimensionnement d'un tel tableau s'avérerait coûteux. A contrario, l'appel à la fonction ***malloc*** à chaque allocation de petits objets occasionnerait un *overhead* qu'il convient d'éviter. C'est pourquoi nous avons mis en œuvre un espace

organisé en ***pages*** de 64 Koctets, pour une meilleure localité spatiale, l'ensemble des ***pages*** étant stocké dans une unique liste simplement chaînée. Nous pouvons alors proposer un allocateur puisant de manière contiguë des pointeurs dans une page, tant qu'il reste de la place dans celle-ci. Si l'allocation échoue faute de place suffisante, le dernier bloc disponible dans la page est ajouté à la ***freelist*** ; une nouvelle page est créée par appel à la fonction ***malloc***, puis l'allocation demandée est alors réalisée au début de cette nouvelle page. Cette approche diffèrent de la stratégie demandée, préconisant de charger entièrement la page fraîche dans la freelist. Notre choix permet en effet d'introduire aisément une multitude de ***freelist*** à notre mark&sweep.

Recherche dans la freelist

Nous avons implémenté une stratégie first fit, pour rechercher un bloc de taille suffisante dans une freelist. Parallèlement, nous avons expérimenté plusieurs méthodes d'insertion d'un bloc libre dans la freelist : ajout en tête, insertion dans une freelist ordonnée suivant la taille des objets, ou insertion dans une freelist triée par adresse. Cette dernière approche (qui a été utilisé dans le runtime de Caml-light https://github.com/camllight) est bien adapter pour la fusion des blocs libres contigus (***coalescing***). Ce mécanisme n'est cependant pas encore fonctionnel dans notre implémentation.

Phase de Sweep dans un espace paginé

Lors de la phase de ***sweep***, il faut être en mesure de parcourir l'ensemble des objets alloué dans le tas. C'est pourquoi notre implémentation maintient actuellement un liste simplement chaînée des objets alloués. Cette structure de liste n'est pas nécessaire. Nous aurions pu réaliser un parcours des pages suivant l'ordre contigu des blocs, de manière analogue à la seconde phase du **Stop&Copy**. La gestion de *corner cases* est cependant délicat, en particulier les transitions entre pages et les zones de 1 mots qui occasionnent des trous.

Freelist multiples

Notre GC Mark&Sweep s'appuie sur un tableau de freelist, chaque freelist k étant destinée à stocker des objets de taille comprise entre **(k*`RANGE`)** et **((k+1) * `RANGE`)** avec `RANGE` une constante. L'association de chaque objet une freelist unique en fonction de la taille de celui-ci,est réalisée par la macro :

```
#define SelectFreeList(sz) (&Caml_state->freelist_array[sz/RANGE])
```

Parallèlement, le nombre de freelist est définit par :

```
#define NB_FREELIST (BIG_OBJECT_MIN_SIZE / RANGE)
```

Modification de la machinerie virtuelle

Blocs vides

Nous avons la représentation des valeurs dans la VM, 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 ***freelist***. C'est également poser la question de modifier l'*information de taille* des bloc vide, contenu dans le *header*. Des instructions de la VM comme **VECTLENGTH** ou **RESTART** consultent 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 VM ne consulte jamais la taille d'un bloc vide ; et nous fixons à 1 l'*information de taille* contenu dans le *header* des blocs vides. Cela facile ainsi le parcours du tas par bloc contigu dans le **Stop&Copy** et dans l'espace paginé mis en œuvre dans notre **Mark&Sweep**

Accès aux racines depuis le GC

Les registres **accu** et **env** ainsi que le pointeur de pile ont transformés en variable globale afin de pouvoir être utilisés comme racine par nos récupérateurs de mémoire. La **mini-ZAM** ne permettant pas de définir de nouvelle racine globale, l'interprétation des instruction **CLOSURE CLOSUREREC** et **GRAB** a été modifié à partir d'un code fourni, en sorte que les valeurs utilisées par le programme soient bien toute accessible à partir des racines. Cette exercice nous amené à devoir résoudre un bug particulièrement subtile dans le Stop&Copy : Nous effectuions en effet l'allocation d'une valeur dans le tas depuis l'interprète par des appels de fonction. Par exemple **make_closure(pc - 3, env)** qui alloue une fermeture, puis copie le registre **env** dans le **champ 1** de celle-ci. Or l'allocation peut naturellement provoquer un déclenchement du gc, qui dans le cas du Stop&Copy a pour conséquence de changer l'adresse des racines. Du fait de l'appel par valeur, le second argument de **make_closure** se trouve être alors une copie obsolète de l'adresse contenu dans le registre **env**, d'où une erreur de segmentation dans l'interprète, ou lors d'un futur déclenchement de GC. Une fois la cause de ce bug détecté - non sans mal ! - nous avons pu parvenir à un comportement correcte en recourant à des macros, à la place des fonctions d'allocation, comme cela avait été suggéré par l'enseignant.

Conclusion

Nous proposons une implantation opérationnelle d'un GC **Stop&Copy** et l'implémentation inachevée d'un **Mark&Sweep** avec allocation dans un espace paginé et ***freelist multiples***. *Nous pouvons maintenant envisager de combiner ces deux algorithmes en vue d'implémenter un GC générationnel, tel que celui utilisé dans les premières implémentations du* runtime *Caml-Light.*