Gestion mémoire : réalisation d'un GC pour IMP

Le but de ce TP est de programmer la partie gestion de mémoire automatique du langage IMP vu à la séance précédente. Il s'agira donc de programmer un GC copiant.

Les fichiers desquels vous devez partir sont dans dans l'archive associée à la séance de TD et sont les suivants :

- `Makefile` :
- `compile.ml(i)` : le squelette de la compilation des expressions d'IMP
- `impast.ml` : le type des arbres de syntaxe abstraite, et leur imprimeur
- `implex.mll` : l'analyseur lexical
- `impparse.mly` : l'analyseur syntaxique
- `mainCompile.ml` : la boucle principale de compilation
- `mainRun.ml` : la boucle principale de l'interprète de bytecode
- `printByteCode.ml` : les fonctions permettant d'afficher du bytecode textuellement
- `vmBytecode.mli` : les définitions de types liées au bytecode
- `vmExec.ml` : la mécanique d'exécution de la machine virtuelle
- `mem.ml(i)` : la gestion de la mémoire (fichier dans lequel vous allez travailler)
- `*.imp` : quelques exemples de programmes en IMP

Compiler et recompiler

La commande `make` permet de reconstruire 2 exécutables

- `impc` : le compilateur du langage IMP vers du bytecode
- `imprun` : l'interprète de fichiers de bytecode

Note : La toute première fois, après récupération des fichiers, vous devrez créer un fichier (vide) `.depend` dans le répertoire où vous aurez mis les fichiers. Ceci permettra une gestion automatisée des dépendances de compilation (quoi recompiler quand un fichier a été modifié). Vous invoquerez alors `make depend` afin que ces dépendances soient calculées initialement. Lorsque vous modifierez vos fichiers, si vous utilisez une fonction d'une unité de compilation non encore utilisée dans un fichier, la compilation pourra échouer car les dépendances entre fichiers ne seront peut-être pas à jour (en fait, c'est très peu probable car le squelette impose déjà les dépendances nécessaires). Dans ce cas, re-invoquez `make depend`.

Les GCs copiants

Comme vu en cours, un GC copiant nécessite 2 tas (*FROM* et *TO*) dont 1 seul est utilisé à la fois (le tas *FROM*). Les blocs sont alloués dans *FROM*. Lorsque ce dernier est plein, le GC est déclenché.

1. Phase de copie : le GC parcourt le tas *FROM*, à partir des racines, et recopie les blocs qu'il rencontre dans le tas TO (initialement vide).
2. Phase d'échange : le GC échange *FROM* et *TO*, et l'exécution reprend.

Puisque les blocs ont été déplacés de *FROM* vers *TO*, les pointeurs contenus dans les blocs doivent être mis à jour vers leur nouvelle adresse. Pour ce faire, on laisse une «adresse distante» dans l'ancien bloc après son déplacement. Si un pointeur vers l'ancien bloc est rencontré plus tard, le pointeur est mis à jour avec l'adresse distante du bloc.

En pseudo-code, l'algorithme de GC copiant peut s'écrire :

```
collecte_par_recopie () {
  pour chaque pointeur racine p :
    p := transférer (p) ; (* Récupérer la nouvelle racine. *)
}

transférer (p) {
  si p pointe vers un bloc contenant une adresse distante q, alors :
     renvoyer q ;
  sinon :
     soit q = copie (p, TO) ;
     écrire q comme adresse distante
               à la place du bloc pointé par p ;
     (* Parcours des champs après copie, donc depuis TO. *)
     soit flds les champs de q contenant des pointeurs ;
     pour chaque champ qf de flds,
        qf := transférer (qf) ;
     renvoyer q ;
}
```

1 Travail à faire

1.1 Réflexion préliminaire

Q1 Quelles sont les racines du GC dans le cas de IMP ?

Q2 Par rapport à la semaine dernière, quelles modifications devront être apportées à l'allocateur de mémoire (fonction `new_block`) ? Pour souvenir, la semaine dernière, l'allocateur se contentait de ré-allouer de la mémoire en doublant sa taille en cas de manque de mémoire, puis recopiait le contenu de l'ancienne mémoire dans la nouvelle. La mémoire était implantée par un simple tableau de `VmBytecode.vm_val`.

La mémoire est représentée par la structure OCaml suivante que l'on trouve dans le fichier `vmBytecode.mli`.

```
type mem = {
  mutable size : int ;          (* Taille d'un seul tas. *)
  mutable next_free : int ;     (* Adresse prochain bloc libre . *)
  mutable heap_base : int ;     (* Adresse de début du tas courant. *)
  mutable data : vm_val array (* Les 2 tas, donc de taille size * 2. *)
} ;;
```

En mémoire un «bloc» de taille n est constitué de $n+1$ cases, la première servant à mémoriser la taille du bloc (dans une valeur `VMV_int`). Les cases suivantes constituent la mémoire réellement allouée au programme suite à la demande.

À chaque allocation, si la mémoire est disponible, l'adresse retournée est l'indice `next_free` (qui désigne le prochaine bloc libre) + 1 puisque la première case sert de taille de bloc. Puis `next_free` est incrémenté de la taille allouée + 1.

Le champ `heap_base` sert à savoir quelle moitié de la mémoire est actuellement le tas courant, plus précisément, son «adresse» (i.e. indice) de départ («de base»). Ainsi, ce champ peut valoir soit 0 soit `size`.

À l'exception de la vérification de taille de mémoire disponible, ce champ ne sert qu'en phase de GC. Autrement dit, lorsque l'allocateur retourne une adresse, elle est directement dans le bon tas.

Une variable global `mem` est définie dans le fichier `mem.ml` afin de ne pas devoir trimbaler cette mémoire en paramètre de toutes les fonctions, sachant que cette unique mémoire est partagée partout.

Q3 Donnez en pseudo-code la forme de la fonction principale `gc` qui est appelée lorsqu'il est nécessaire de déclencher une phase de GC.

Q4 On s'intéresse maintenant à la copie de racines. Quelle est, en pseudo-code, la forme de cette fonction (appelons-la `copy_root`) ?

Q5 On s'intéresse désormais à la fonction qui prend une adresse et se charge de **déclencher** la recopie d'un bloc vers lequel elle pointe, **s'il n'a pas déjà été recopié**, puis retourne la nouvelle adresse où se trouve ce bloc.

Attention, on ne s'intéresse pas à la fonction qui copie effectivement un bloc «valeur par valeur» : ce sera l'objet de la question **Q7**.

Quelle est, en pseudo-code, la forme de cette fonction (appelons-la `transfer_pointer`) ?

Q6 Comment sait-on si une adresse est «distante» ?

Q7 On s'intéresse maintenant à la fonction qui va copier un bloc «valeur par valeur», connaissant son adresse et sa taille (un `memcpy` à la C, donc). Quelle est, en pseudo-code, la forme de cette fonction (appelons-la `copy_block`) ?

Q8 Maintenant que vous avez le pseudo-code de toutes les étapes (fonctions) du GC, implémentez-les pour avoir un GC qui fonctionne.

Q9 Testez votre GC, par exemple avec le programme fourni `cause_gc.imp`. Ce programme utilise des tableaux (façon rapide de consommer de la mémoire) qu'il remplit et affiche de nombreuses fois. De ce fait, chaque tableau, après son utilisation devient «mort» et pourra donc être recyclé au prochain GC. Et comme le programme crée beaucoup de tableaux, au bout d'un moment ceux qui sont «morts» prennent trop de place en mémoire et le GC doit se déclencher pour libérer de l'espace mémoire.

1.2 S'il vous reste du temps ...

... n'hésitez pas à documenter votre programme avec des commentaires si vous n'avez pas pris soin de le faire en cours de rédaction, ou bien implémentez le redimensionnement du tas en cas de manque de mémoire persistant après un GC.