IN 213 **Solution** ENSTA IP PARIS

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 ?

Solution
Il y en a 3 : le registre de la VM, sa pile et l'environnement. Subtilement, le code n'en fait pas partie car nous n'avons pas les adresses comme constantes du langage, ni de fermetures ni d'opérateur d'extraction d'adresse (le `&` de C). Il est donc impossible d'avoir des adresses figurant dans les instructions de bytecode.

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`.

Solution
Maintenant, il va devoir lancer le GC si la mémoire demandée n'est pas disponible. À l'issue du GC, il va devoir vérifier à nouveau si la mémoire demandée est devenue disponible suite au compactage.

Si tel n'est pas le cas alors il devra procéder à une ré-allocation du tas. L'adresse de base obtenue suite à la ré-allocation étant différente de l'adresse initiale du tas, il faudrait un relogement comme pour une passe de GC normale.

Dans notre implémentation à base d'un tableau, la relocation pourra être également nécessaire si le tas actuellement utilisé est celui de la seconde moitié du tableau. En effet la taille d'un tas ayant doublé, les valeurs actuellement contenues dans la seconde moitié du tableau se retrouveraient, à l'issue de la réallocation, dans la première moitié, ce qui serait incorrect.

Mais bon, afin de simplifier, on considèrera que s'il n'y a toujours plus de mémoire à l'issue du GC, on échouera en levant une exception.

Puisque l'allocateur peut faire appel au GC, donc changer les racines de la VM (oui, il va appeler la fonction `transférer` sur les racines, qui sont les constituants de la VM, ce qui va retourner les nouvelles racines relogées), il doit désormais retourner, en plus de l'adresse du bloc alloué, l'éventuel nouvel état de la VM.

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.

Solution
Cette fonction prend en argument l'état actuel de la VM et retourne un état mis à jour suite aux copies de blocs et relogements d'adresses.

```
gc (etatvm) =
  Déterminer l'adresse de début du tas TO.
  Initialiser un index de destination de copie à cette adresse.
  (registre', pile', env') = copier les racines à partir de etatvm.
  Mettre à jour l'indice de prochain bloc libre dans la structure mémoire.
  Mettre à jour l'adresse de tas courant dans la structure mémoire.
  Construire et retourner l'état de VM avec registre', pile', env'
```

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

Solution

Cette fonction prend en argument une valeur (`VmBytecode.vm_val`) et retourne une valeur. En effet, une racine contient une valeur. Cette valeur peut être l'adressse d'un bloc, qui devra alors être déplacé. Il faudra bien alors récupérer la nouvelle adresse après copie et relogement pour mettre à jour cette racine. Et si la racine ne contient pas une adresse, alors elle doit être simplement copiée (sans déplacement).

```
copy_root (vracine) =
  Si vracine est int / bool /chaîne / code, retourner telle qu'elle.
  Sinon si c'est une adresse, il faut recopier le bloc vers lequel il
       pointe si ça n'a pas déjà été fait. Dans tous les cas, on retourne
       la nouvelle adresse (celle où l'on vient de copier le bloc ou celle
       où il avait déjà été copié).
  Sinon si c'est un environnement, comme c'est en fait une une liste
       d'adresses, il faut suivre / reloger tous ces pointeurs. On retourne
       une valeur d'environnement avec toutes ces adresses relogées.
```

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`) ?

Solution

Cette fonction prend donc en argument une adresse et retourne une adresse. L'adresse reçue en argument désigne la localisation d'un bloc. L'adresse en retour désigne la localisation de là où il a été déplacé dans le tas *TO*.

```
transfer_pointer (addr) =
  Vérifier si mem[addr] contient une «adresse distante».
  Si oui, retourner cette adresse.
  Si non
    Récupérer la taille du bloc (case «précédente» en mémoire par rapport à addr).
    Copier le bloc dans le tas TO et récupérer l'adresse a' où il a été copié.
    Modifier l'ancienne valeur à l'adresse avec cette adresse distante.
    Procéder récursivement sur le bloc copié : pour chaque case contenant une adresse a,
      mem[a] <- transfer_pointer (a)
    Retourner a' (i.e. l'adresse où le bloc a été copié).
```

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

Solution

C'est le cas si elle appartient au tas autre que le tas courant. Justement, dans la structure représentant la mémoire se trouve l'adresse du tas courant (champ `mem.heap_base`). Il y a 2 cas de figure pour qu'une adresse a soit «distante» :

- le tas courant est la première moitié (donc `mem.heap_base` $= 0$) et l'adresse a est $\geq$ à `mem.size`,
- ou le tas courant est la seconde moitié (donc `mem.heap_base` $=$ `mem.size`) et l'adresse a est $<$ à `mem.size`.

Donc, en OCaml, cela se traduit par :

```
(a >= mem.size && mem.heap_base = 0) ||
(a < mem.size && mem.heap_base = mem.size)
```

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`) ?

Solution

```
copy_block (from_addr, size) =
  Copier la taille du bloc : mettre cette taille à la prochaine case libre en mémoire.
  Incrémenter (de 1) le pointeur de prochaine case libre pour la copie.
  Mémoriser sa valeur a : ce sera l'adresse retournée.
  Pour chaque case i du bloc (de 0 à size -1),
    mem[prochaine case libre] <- mem[from_addr + i]
    prochaine case libre = prochaine case libre + 1
  Retourner a.
```

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.

Solution

```
(* Compteur global permettant, lors de la phase de copie, de savoir où l'on
   doit copier dans le tas destination. C'est plus simple de l'avoir en
   global plutôt que de devoir se trimbaler cet index partout.
   ATTENTION: doit être réinitialisé en début de GC, à la valeur de base
   du tas destination de la copie. *)
let next_free_in_to = ref 0 ;;


(* from_ addr = addresse de bloc. On récupère la taille dans la case
   précédente. *)
let copy_block from_addr block_size =
  (* Copie de la taille du bloc. *)
  mem.data.(!next_free_in_to) <- VmBytecode.VMV_int block_size ;
  next_free_in_to := !next_free_in_to + 1 ;
  (* On mémorise l'adresse du nouveau bloc relogé. *)
  let new_addr = !next_free_in_to in
  (* Copie du contenu du bloc. *)
  for i = 0 to block_size - 1 do
    mem.data.(!next_free_in_to) <- mem.data.(from_addr + i) ;
    next_free_in_to := !next_free_in_to + 1
  done ;
  new_addr
;;


(* Transfère la mémoire se trouvant à [addr] dans le tas TO si ça n'a pas
   déjà été fait et met un pointeur distant dans le bloc dans le tas FROM.
   Pour savoir si le bloc se trouvant à [addr] a déjà été transféré, il
   suffit de regarder s'il contient une adresse distante, donc si dans
   mem[addr] on trouve une VMV_ addr désignant une adresse dans le bloc TO. *)
let rec transfer_pointer addr =
  (* On va regarde en mémoire à l'adresse [addr]... *)
  let is_foreign =
    (match mem.VmBytecode.data.(addr) with
    | VmBytecode.VMV_addr a ->
        (* Est-ce que l'adresse est dans le bloc TO ? *)
        if (a >= mem.size && mem.heap_base = 0) ||
           (a < mem.size && mem.heap_base = mem.size) then Some a (* Oui. *)
```

```
        else None                                                    (* Non. *)
    | _ -> None    (* Pas une adresse, donc bloc à recopier. *)) in
  match is_foreign with
  | Some a -> a
  | None -> (
      (* Ne pointe pas vers une adresse distante. Récupération de la  taille
         du bloc. *)
      let block_size =
        (match mem.data.(addr - 1) with
        | VmBytecode.VMV_int s -> s
        | _ -> raise (Failure "copy_val : unexpected non-int block size")) in
      (* Il faut faire une copie du bloc. *)
      let new_addr = copy_block addr block_size in
      (* Modifier l'ancienne valeur à l'adresse avec cette adresse distante. *)
      mem.VmBytecode.data.(addr) <- VmBytecode.VMV_addr new_addr ;
      (* Puis, on récurse sur le bloc copié. *)
      for i = 0 to block_size - 1 do
        match mem.data.(new_addr + i) with
        | VmBytecode.VMV_addr a ->
            mem.data.(new_addr + i) <- VmBytecode.VMV_addr (transfer_pointer a)
        | _ -> () (* Pas un pointeur, pas besoin de suivre. *)
      done ;
      (* Retourner l'adresse où le bloc a été relogé. *)
      new_addr
    )
;;


(* Copie des racines. Retourne la valeur de la nouvelle racine et déclenche
   Si nécessaire la copie et le relogement de pointeur si la racine en est
   un. *)
let copy_root v =
  match v with
  | VmBytecode.VMV_int _ | VmBytecode.VMV_bool _ | VmBytecode.VMV_string _
  | VMV_code_addr _ ->
      (* Remarque : il n'y a pas d'adresses dans le code, pas de pointeurs,
         donc rien à copier ni reloger. *)
      v
  | VMV_addr addr ->
      (* La racine est un pointeur, il faut recopier le bloc vers lequel  il
         pointe si ça n'a pas déjà été fait. *)
      VmBytecode.VMV_addr (transfer_pointer addr)
  | VMV_env env ->
      (* Un environnement est en fait une liste d'adresse. Donc il faut
         suivre reloger tous ces pointeurs. *)
      VMV_env (List.map transfer_pointer env)
;;


(* Déclenchement du GC. *)
```

```
let gc vm_state =
  Printf.printf "*******␣GC␣*******\n" ;
  (* Copy. *)
  (* On doit recopier tous les blocs vivants dans l'autre tas. On détermine
     donc l'adresse de l'autre tas, qui, à la fin, deviendra le tas courant.
     Comme la mémoire est coupée en 2 parties égales de taille size, les
     adresses de début de tas sont soit 0 soit size. *)
  let new_base = if mem.heap_base = 0 then mem.size else 0 in
  Printf.printf "*␣@FROM:␣%d␣@TO:␣%d\n" mem.heap_base new_base ;
  (* Réinitialise l'index de copie au début du tas destination de la copie. *)
  next_free_in_to := new_base ;
  (* On copie tout à partir des racines. *)
  let register' = copy_root vm_state.VmBytecode.register in
  let stack' = List.map copy_root vm_state.VmBytecode.stack in
  let env' = List.map transfer_pointer vm_state.VmBytecode.env in
  (* On met à jour la structure mémoire : l'indice du prochain bloc libre et
     la base du tas. *)
  mem.next_free <- !next_free_in_to ;
  mem.heap_base <- new_base ;
  Printf.printf "*␣Next␣free␣block:␣%d\n#␣living␣blocks:␣%d\n"
    mem.next_free (mem.next_free - new_base) ;
  Printf.printf "*******␣END␣GC␣*******\n" ;
  (* On retourne le nouvel état de la VM où les racines ont été mises à
     jour. *)
  { VmBytecode.register = register' ;
    VmBytecode.code = vm_state.VmBytecode.code ;
    VmBytecode.stack = stack' ;
    VmBytecode.env = env' }
;;


(* Allocation d'un bloc mémoire de taille donnée. S'il n'y a plus assez de
   mémoire dans le tas, déclenche un GC. *)
let new_block vm_state alloc_size =
  (* L'allocation effective sera de 1 de plus pour pouvoir mémoriser
     la taille du bloc. *)
  let vm_state' =
    if (mem.next_free - mem.heap_base) + alloc_size + 1 >= mem.size then (
      (* Besoin de déclencher un GC car plus assez de mémoire. *)
      let state = gc vm_state in
      (* On vérifie s'il reste bien de la mémoire après le GC, sinon on
         échoue. *)
      if (mem.next_free - mem.heap_base) + alloc_size + 1 >= mem.size then
        raise (Failure "Really␣no␣more␣memory") ;
      state
    )
    else vm_state in
  (* L'adresse du bloc est celle juste après son champ de taille. C'est cette
     adresse que l'on retourne à "l'utilisateur". *)
  let tmp = mem.next_free + 1 in
```

```
  (* On mémorise la taille du bloc. *)
  mem.data.(mem.next_free) <- VmBytecode.VMV_int alloc_size ;
  (* On fait progresser l'indice de prochain bloc libre. Comme on a pris une
     case pour mettre la taille du bloc, il ne faut pas oublier le + 1. *)
  mem.next_free <- mem.next_free + alloc_size + 1 ;
  (* On retourne l'adresse du bloc alloué. *)
  (tmp, vm_state')
;;
```

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.