Master STL - M1 | **Compilation Avancée** | **Université Pierre et Marie Curie**
Module CA (MI190) | *Documents autorisés* | **Année 2005–2006**

Examen du 5 septembre 2006

1ère partie (10 points) - à traiter sur une copie séparée

1ère partie : description d'un récupérateur automatique de mémoire

On cherche à implanter une variante de l'algorithme de Stop & Copy qui utilise une partie de la mémoire `toSpace` (appelé ici `mirror space`) disponible pour y stocker une information sur les valeurs allouées, ici une information de typage. On appelera cet algorithme un Stop & Copy en place dans la mesure où il copie en place dans l'espace `fromSpace` appelé ici `values space`. Pour simplifier on n'allouera que des cellules de deux éléments : CAR pour la première case et CDR pour la second. La représentation des valeurs immédiates et des pointeurs est uniforme (32 bits chacun).

On présente cet algorithme dans un pseudo-C, vous aurez à le faire tourner sur un petit exemple. Soit une valeur O'Caml construite par le code suivant :

```
let l  =  ['b';'c';'d'] ;;
let a  =  match ( ['a'] , l )
          with  p  ->  ( fst p , snd p ) ;;
```

Premièrement, ce code alloue deux listes dans le tas, suivi d'une paire pour construire la paire de ces deux listes. A la fin de la construction de la apire une deuxième paire est créée pour relier les deux listes. Au final, la deuxième paire est une racine de l'ensemble des racines du GC, et la première peut être récupérée. Avec notre implantation, la zone libre (mirror space) du tas conserve une information de typage de l'objet allouée de manière alignée. La structure de ces valeurs est montrée à la figure 1.

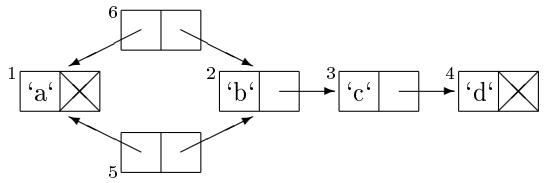

Figure 1: la représentation d'une valeur O'Caml

Nous allons partir de l'état du tas initital :

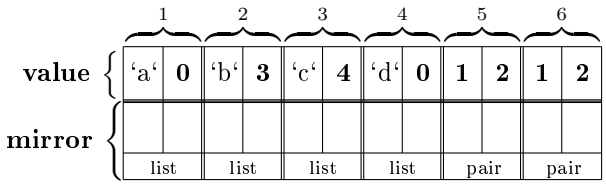

Figure 2: Tas avant Garbage Collecting

On utilise l'ensemble de racines suivant : `[3;6]`. On suppose par ailleurs qu'il existe quelquepart (dans la valeur ou dans le type) un bit pour chaque mot pour distinguer une valeur immédiate d'un pointeur.

Soient `VS` et `MS` les variables du *value-space* et du *mirror-space*. On note `VS[wh]`, la wh-ième cellule, `CAR[wh]`, la partie CAR de la wh-ième cellule, `CDR[wh]`, la partie CDR de la wh-ième cellule, `WIG[wh]` est le pointeur p_1 de la wh-ième cellule et `WWC[wh]` est le pointeur p_2 pointer de la wh-ième cellule.

La variable `free` dénote un pointeur vers la prochaine cellule libre. Comme indiqué chaque valeur est marquée pour être une valeur immédiate ou un pointeur, on a les fonctions suivantes :

```
int is_addr_car (addr);
int is_addr_cdr (addr);
```

La future position d'une cellule est sauvée dans le champ `WIG` (Where I Go), et l'adresse de la cellule à venir est sauvée dans le champ `WWC` (Which Will Come). Tout cela est réalisé par la fonction `Treat`.

```
void Treat(addr a) {
  if ( WIG[a] == NULL) {
    WIG[a] := free;  WWC[free] := a;
    free++;
  }
}
```

Pour chacune des étapes suivantes indiquer l'action du pseudo-code écrit et décrire l'état du tas sous la forme du tableau de la figure 2.

1. étape 1 :

```
  free=1;
  for (i=1;i<length(RS);i++) Treat(RS[i]);
```

2. étape 2

```
  for ( wh=1 ; wh < free ; wh++ ) {
    addr a = WWC[wh];
    if (is_addr_car(a))  Treat(CAR[a]);
    if (is_addr_cdr(a))  Treat(CDR[a]);
  }
```

3. étape 3

```
   for (i=1;i<length(RS);i++) RS[i]=WIG[RS[i]];
   for ( wh=1 ; wh < free ; wh++ ) {
     addr a = WWC[wh];
     if (is_addr_car(a))  CAR[a] = WIG[CAR[a]]
     if (is_addr_cdr(a))  CDR[a] = WIG[CDR[a]]
   }
```

4. étape 4

2 fonctions auxilaires

```
void move (addr src, addr dest) {
  VS[dest] = VS[src];
  MS[dest]->type = MS[src]->type;
  WWC[dest] = NULL;
  WIG[src] = NULL;
}

void move_chain (addr a) {
  while (a != NULL) {
    addr b = WWC[a];
    move(b,a);
    a = b;
  }
}
```

algorithme

```
for ( wh=1 ; wh < free ; wh++ ) {
  addr a = WWC[wh];
  if (a != NULL)  {
    if (WIG[wh] == a ) {
       WIG[wh] = NULL;
       WWC[wh] = NULL;
    }
    else
      if (WIG[wh]==NULL) move(a,wh);
      else {
        addr b = WIG[a];
        while ( b!=a && WIG[b]!=NULL)
          b=WIG[b];
        if (WIG[b]==NULL)
          move_chain(WIG[b])
        else {
          move(a,buffer);
          move_chain(a);
          move(buffer,WIG[a]);
        }
      }
  }
}
```