Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 4 – Corrigé
Représentations par pointeurs

1 Arbres

a. Les fonctions de calcul type `somme` s'écrivent de manière similaire aux versions sans pointeurs :

```
let rec somme = function
  | Vide -> 0
  | Noeud n -> n.val + (somme n.fils_g) + (somme n.fils_d) ;;
```

Le miroir d'un arbre peut ainsi s'écrire en "reconstruisant l'arbre" comme au TP 2 :

```
let rec miroir = function
  | Vide -> Vide
  | Noeud n -> Noeud { val = n.val ;
                       fils_g = miroir n.fils_d ;
                       fils_d = miroir n.fils_g } ;;
```

L'utilisation des pointeurs permet cependant une solution qui modifie directement l'arbre donné en entrée, la fonction renvoyant un type `unit`. Cette solution ne recopie aucune valeur et se contente d'échanger les liens `fils_g` et `fils_d` de chaque noeud. Après l'appel de la fonction, l'arbre initial est "perdu".

```
let rec miroir = function
  | Vide -> ()
  | Noeud n -> miroir n.fils_g ; miroir n.fils_d ;
               let g = n.fils_g
               in n.fils_g <- n.fils_d ;  n.fils_d <- g ;;
```

L'insertion dans un ABR peut aussi se faire "en place". Remarquons que la fonction obtenue ne permet pas d'insérer dans un arbre vide.

```
let nouveau_noeud x = Noeud { val = x ; fils_g = Vide ; fils_d = Vide } ;;

let rec insere_ABR x = function
  | Vide -> failwith "erreur"
  | Noeud n -> if (x < n.val)

               then (if (n.fils_g = Vide)
```

```
            then n.fils_g <- nouveau_noeud x
            else insere_ABR x n.fils_g)

      else (if (n.fils_d = Vide)
            then n.fils_d <- nouveau_noeud x
            else insere_ABR x n.fils_d) ;;
```

Ces fonctions permettent ainsi de traiter des grandes quantités de données sans les dupliquer.

2 Files d'attente

b. Lorsqu'on réalise les files d'attente par des listes, il faut choisir où on place les "nouveaux arrivants". Ici nous les ajoutons "au début" de la liste, ce qui mène à une insertion en $\mathcal{O}(1)$ et à une suppression en $\mathcal{O}(n)$.

```
let insere f x = x :: f ;;

let rec supprime = function
  | [] -> raise File_vide
  | [a] -> a, []
  | t::q -> let a,f = supprime q in a,(t::f) ;;

let est_vide = function
  | [] -> true
  | _ -> false ;;

let listage f = f ;;
```

Inversement, la solution consistant à insérer "au bout" de liste mène à une insertion en $\mathcal{O}(n)$ et une suppression en $\mathcal{O}(1)$.

c. Pour réussir cette question, il fallait bien distinguer
- la structure de liste doublement chaînée `liste_double`
- et la structure `file` qui ne comporte que deux pointeurs vers une liste double,ent chaînée.

```
let est_vide = function
  |  { tete = Nil ; queue = Nil } -> true
  |  _ -> false ;;

let listage f =
  let rec aux = function
    | Nil -> []
    | Cellule {valeur = x ; lien_av = suite} -> x :: (aux suite)
  in aux f.tete ;;
```

On peut remarquer ci-dessus qu'il est possible d'effectuer un filtrage partiel (`{valeur = x; lien_av= suite}`). L'insertion se fait de manière différente selon la file originale `f` : si elle n'est pas vide, il suffit

d'ajouter une cellule au début (en n'oubliant pas de modifier le pointeur `lien_ar` de l'ex-première cellule `ct`) et de mettre à jour le pointeur `f.tete` ; si elle est vide, il faut en plus faire pointer `f.queue` vers la nouvelle et unique cellule.

```
let insere f x =
  let c = Cellule { lien_ar = Nil ; valeur = x ; lien_av = f.tete }
  in match f.tete with
    | Nil -> f.tete <- c ; f.queue <- c ; f
    | Cellule ct -> ct.lien_ar <- c ; f.tete <- c ; f ;;
```

Pour la suppression, il faut traiter deux cas particulier : la file vide (déclenchement de l'exception `File_vide`) et la file réduite à une cellule (dans ce cas, il faut aussi faire pointer `f.tete` vers `Nil`).

```
let supprime f = match f.queue with
  | Nil -> raise File_vide
  | Cellule cq -> let x = cq.valeur in
                  match cq.lien_ar with
                    | Nil -> f.tete <- Nil ; f.queue <- Nil ; x,f ;
                    | (Cellule cp) as p -> f.queue <- p ; cp.lien_av <- Nil ; x,f ;;
```

3 Parcours colorés de graphes

d. On suppose que tous les sommets du graphes sont colorés à `Noir`. Lorsque la fonction `cycle` visite un sommet `Noir`, elle le colore en `Rouge`, s'appelle récursivement sur tous ses successeurs, puis recolore le somet à `Noir`. Si la fonction rencontre un sommet `Rouge`, c'est que ce sommet a déjà été visité, et donc qu'il y a un cycle. Dans le code ci-dessous, la fonction auxiliaire `ou : bool list -> list` calcule un "ou généralisé" sur un ensemble de valeurs booléennes.

```
let rec ou = function
  | [] -> false
  | t::q -> t || (ou q) ;;

let rec cycle s = match s.coul with
  | Rouge -> true
  | Noir -> s.coul <- Rouge ;
            let res = ou (map cycle s.arcs) in s.coul <- Noir ;
                                               res ;;
```

e. L'énumération des sommets se passe de la même manière, la couleur `Rouge` indiquant qu'un sommet a déjà été visité. La première idée est d'écrire la fonction ainsi :

```
let rec union = function
  | [] -> []
  | t :: q -> t @ (union q) ;;

let rec enumere s = match s.coul with
```

```
  | Rouge -> []
  | Noir -> s.coul <- Rouge ;
            let res = s::(union (map enumere s.arcs)) in s.coul <- Noir ;
                                                          res ;;
```

La fonction auxiliaire `union` a malheureusement avoir une complexité linéaire par rapport à la taille (et non au nombre) des listes passées en entrée, ce qui mène a une complexité quadratique pour `enumere` dans le cas le pire. Une solution linéaire peut s'obtenir en passant en argument la liste des sommets et en l'agrandissant au fur et à mesure (ruse analogue à celle de `miroir` dans le TP 1) :

```
let rec enumere_aux liste s = match s.coul with
  | Rouge -> liste
  | Noir -> s.coul <- Rouge ;
            let rec aux = function
               | [] -> (s::liste)
               | t::q -> enumere_aux (aux q) t
            in let res = aux s.arcs in s.coul <- Noir ;
                                       res ;;

let enumere2 = enumere_aux [] ;;
```

f.* Une première solution est d'écrire la fonction suivante qui ne s'arrête jamais. L'appel `do_list f l` applique une fonction `f : 'a -> unit` à tous les éléments d'une `'a list` et renvoie `()` (c'est donc quasiment la même chose que `map f l`).

```
let rec initialise s =
  s.coul <- Noir ;
  do_list initialise s.arcs ;;
```

Comment s'arrêter efficacement ? Dans ce modèle de graphe où la seule connaissance est le sommet "en cours de visite", la bonne solution est de faire des parcours colorés comme précédemment... on doit donc garantir une absence initiale d'une certaine couleur. La solution prioposée ci-dessous suppose qu'aucun sommet n'est `Jaune`, les colore tous en `Jaune`, puis les recolore en `Noir`.

```
let rec init_aux couleur s =
  if (s.coul = couleur) then ()
                        else s.coul <- couleur ;
                             do_list (init_aux couleur) s.arcs ;;

let initialise2 s = init_aux Jaune s ; init_aux Noir s ;;
```

g.* Pour rendre "plus sûrs" les parcours de graphe, il faut employer d'autres structures de données. On peut, en plus de la représentation par pointeurs utilisée ici, connaître une liste ou un tableau de tous les sommets.

Bonnes fêtes à tous !