TP13: OCaml impératif, tas binaire

Références

Une **référence** OCaml est une case mémoire pouvant stocker un objet, et pouvant être modifiée. C'est un concept proche des pointeurs C. Une référence contenant un objet de type `'a` est de type `'a ref`. La fonction `ref: 'a -> 'a ref` permet de créer une référence et de lui donner une valeur initiale:

```
let (r: int ref) = ref 32
```

On peut imaginer que le code ci-dessus s'exécute de manière analogue au code C suivant:

```
int* r = malloc(sizeof(int));
*r = 32;
```

Étant donné une référence `r`, on peut accéder à son contenu avec `!r`: C'est comme si l'on écrivait `*r` en C: on déréférence.
On peut aussi modifier le contenu d'une référence grâce à l'opérateur binaire `:=`, qui s'utilise comme suit:

```
let r = ref 0 ;;
r := 5 ;; (* change la valeur stockée dans r en 5 *)
r := !r + 1 ;; (* change la valeur stockée dans r en 6 *)
```

L'opérateur `:=` prend en paramètres une référence (à gauche) et une valeur (à droite). Il modifie le contenu de la référence, et renvoie `unit`. On peut donc enchaîner les `:=` comme on le ferait avec des print:

```
let r = ref 2 in
r := !r + 3;   (* r contient 5 *)
r := !r * !r;  (* r contient 25 *)
print_int !r   (* affiche 25  *)
```

Les références permettent de garder une mémoire persistant à travers les appels de fonction (comme les éléments stockés dans le tas en C). Par exemple, la fonction suivante permet de calculer la somme des éléments d'une liste:

```
let somme (l_0: int list) : int =
  let r = ref 0 in
  (* Ajoute à r la somme des éléments de l *)
  let rec somme_ref (l: int list) : unit =
    match l with
    | [] -> ()
    | x :: q ->
      r := !r + x;
      somme_ref q
  in
  somme_ref l
```

Notons que la fonction ci-dessus est récursive terminale !

Q1. Définir un type pour les arbres binaires, puis écrire une fonction calculant la liste des éléments d'un arbre dans votre ordre préféré parmi infixe, préfixe et postfixe. On passera par une fonction auxiliaire utilisant une `int list ref` pour stocker les étiquettes rencontrée au fur et à mesure:

```
let lister (a: 'a arbre) : (int * 'a) arbre =
  let r = ref [] in
  (* Ajoute tous les éléments de a à r, dans l'ordre [...]fixe *)
  let rec ajouter a =
    ...
  in ajouter a
```

Q2. Quelle est la complexité de votre fonction ?

Q3. Toujours sur le même type, écrire une fonction permettant de numéroter un arbre dans l'ordre choisi à la question précédente. On pourra passer par une référence stockant le prochain numéro à utiliser:

```
let numeroter (a: 'a arbre) : (int * 'a) arbre =
  let r = ref 0 in
  let rec numeroter_ref a =
    ...
  in numeroter_ref a
```

Tableaux

Les **tableaux** OCaml sont presque identiques aux tableaux C: on peut accéder à n'importe quelle case en $\mathcal{O}(1)$. Un tableau est délimité par `[| ... |]`, et on accède aux éléments avec la syntaxe `t.(i)` qui est l'équivalent de `t[i]` en C. Par exemple:

```
let t = [|"bla"; "truc"; "tata"; "titi"|]
let x = t.(0) (* x vaut "bla" *)
let y = t.(3) (* x vaut "titi" *)
```

Le module `Array` contient de nombreuses fonctions utiles sur les tableaux, notamment la fonction `Array.length: 'a array -> int` qui donne la longueur d'un tableau. Contrairement aux listes, cette opération est en $\mathcal{O}(1)$ pour les tableaux. Ainsi, à l'inverse du C, les fonctions manipulant des tableaux peuvent prendre en entrée le tableau seul, et obtenir la longueur avec `Array.length` sans surcoût:

```
(* Renvoie la somme des éléments de t *)
let somme (t: int array) : int =
  let n = Array.length t in
  (* Renvoie la somme des éléments de t entre
     i et n-1 inclus *)
  let somme_from (i: int) : int =
    if i >= n then 0 else
    t.(i) + somme_from (i+1)
  in
  somme_from 0
```

Q4. Sur le même principe que l'exemple ci-desus, écrire une fonction calculant le maximum d'un tableau.

On peut **modifier** un tableau, à l'aide de l'opérateur `<-`: écrire `t.(i) <- x` modifie `t.(i)` pour y écrire `x`. L'expression `t.(i) <- x` a pour type `unit`, c'est donc une instruction comme `print_int` ou `assert`: Par exemple:

```
let t = [|5; 3; 1; 6|] ;;
(* Remplace les deux premières cases de t par des zéros *)
let zero (t: int array) : unit =
  t.(0) <- 0;
  t.(1) <- 0
;;
zero t;;
t;; (* Affiche: [|0; 0; 1; 6|] *)
```

Notons la différence fondamentale avec les listes: quand on modifie une liste, on **renvoie** la nouvelle liste, et la liste initiale est inchangée. Pour un tableau, la modification se fait directement sur le tableau d'entrée, ce qui explique le type de sortie `unit`.

On peut créer un tableau avec la fonction `Array.make`:

```
(* Array.make n x renvoie un tableau de taille n, où chaque case vaut x *)
Array.make : int -> 'a -> 'a array
```

Q5. Écrire une fonction `range: int -> int list` telle que `range n` renvoie le tableau `[|0; 1; ...; n-1|]`.

On peut transformer un tableau en un arbre binaire de recherche en temps $\mathcal{O}(nh)$, où n est la taille du tableau et h la hauteur de l'arbre construit, en ajoutant un par un chaque élément. Si l'on a implémenté des arbres auto-équilibrants comme les ARN, cette construction se fait donc en $\mathcal{O}(n \log n)$. Cependant, pour un tableau trié, il existe un algorithme plus efficace:

Algorithme 1 : `construire`(T)

Entrée(s) : T un tableau trié
Sortie(s) : ABR contenant les éléments de T

$n \leftarrow$ taille de T;
si $n = 0$ **alors**
  retourner ***l'arbre vide***
$m \leftarrow \lfloor \frac{n}{2} \rfloor$;
$G \leftarrow$ `construire`$(T[0..m-1]$;
$D \leftarrow$ `construire`$(T[m+1..n-1]$;
retourner $\boldsymbol{N}(T[m], G, D)$

L'arbre ainsi construit sera de hauteur logarithmique, et la construction prend un temps $\mathcal{O}(n)$!

Q6. Implémenter l'algorithme précédent en OCaml. On pourra écrire une fonction récursive prenant en entrée le tableau T ainsi que deux bornes a, b, et construisant un ABR à partir des cases $T[a..b]$.

Q7. Justifier la complexité en $\mathcal{O}(n)$.

Boucles

Dans les questions précédentes, pour manipuler des tableaux, nous étions obligés de passer par des fonctions auxiliaires prenant en entrée un indice permettant d'itérer sur les cases du tableau, ce qui est assez pénible. En OCaml, la manière habituelle de manipuler les tableaux est d'utiliser les **boucles** !

Boucle pour La syntaxe générale est:

```
for i = DEBUT to FIN do (* les bornes DEBUT et FIN sont incluses *)
  CODE
done
```

où `DEBUT` et `FIN` sont des expressions de type `int` et `CODE` une expression de type `unit`.

Une boucle for est une **expression** OCaml, de type `unit`, et a donc toujours comme valeur `()`. On peut l'enchaîner avec des print, des affectations de références, des modifications de tableaux, etc...

Par exemple, pour calculer la factorielle:

```
let factorielle n =
  let res = ref 1 in
  for i = 1 to n do
    res := !res * i
  done;
  res
```

Notons que pour avoir une valeur modifiable au fil des boucles, on doit passer par des références. Essayez de réécrire le code précédent en ne faisant pas de `r` une référence pour comprendre pourquoi les variables OCaml classiques ne suffisent pas.

Q8. Implémenter le tri par sélection en utilisant des boucles:

Boucles tant que Syntaxe générale:

```
while CONDITION do
   CODE
done
```

où `CONDITION` et `CODE` sont des expressions OCaml. `CONDITION` doit être de type `bool`, `CONDITION` doit être de type `unit`, et la boucle est alors bien typée, et de type `unit`. Par exemple, voici une implémentation de l'exponentiation rapide:

```
(* renvoie a puissance n *)
let expo_rapide (a: float) (n: int) =
  let res = ref 1. in
  let x = ref a in
  let nn = ref n in
  while !nn > 0 do
    if !nn mod 2 = 1 then begin
      res := !x *. !res
    end;
    x := !x *. !x;
    nn := !nn / 2
  done;
  !res
```

Notons que tout le corps de la boucle est **une** grosse expression OCaml.

Q9. Implémenter au choix un des deux algorithmes suivants en utilisant des boucles:

- Recherche par dichotomie
- Tri par insertion

Q10. (**Optionnel**) Écrire une fonction permettant de transformer une liste en un tableau, et une fonction faisant l'inverse.

Tas binaire

Dans toute cette partie, interdiction d'utiliser la récursivité !

Implémentons les tas binaires vus en cours. On s'intéresse à des tas-min, où la racine contient l'élément minimal. On rappelle que dans l'implémentation par tableau, un tas est représenté par un tableau donnant ses éléments dans l'ordre du parcours en largeur. Alors, le nœud numéro i a pour enfants $2i+1$ et $2i+2$, et comme parent $\lfloor \frac{i-1}{2} \rfloor$ (sauf la racine, qui n'a pas de parent).

Q11. Écrire trois fonctions permettant d'accéder aux enfants et au parent d'un nœud:

```
parent: int -> int (* Précondition: l'entrée n'est pas la racine *)
gauche: int -> int
droite: int -> int
```

Q12. Implémenter une fonction ayant la spécification suivante:

```
(* est tas t i renvoie true si les i premières cases de t forment un tas *)
est_tas: 'a array -> int -> bool
```

Dans la suite, on écrira "le tas (t, i)" pour dire "le tas stocké dans les i premières cases du tableau t"

Q13. Écrire une fonction permettant d'échanger deux cases d'un tableau:

```
(* échanger t i j échange les cases t.(i) et t.(j) *)
echanger: ('a array) -> int -> int -> unit
```

On rappelle que la procédure d'insertion consiste à placer le nouvel élément sur une nouvelle feuille, puis à faire **remonter** cet élément vers la racine jusqu'à ce qu'il soit bien placé, i.e. qu'il soit plus grand que son parent.

Q14. Écrire une fonction `inserer: 'a array -> int -> 'a -> unit` telle que `inserer t i x` insère x dans le tas (t, i). On rappelle que pour un tas de taille i, la prochaine feuille se place dans la case i de t.

En particulier, si t est un tableau quelconque, on peut le transformer en tas en:

- insérant $t[0]$ dans le tas $(t, 0)$;
- insérant $t[1]$ dans le tas $(t, 1)$;
- ...
- insérant $t[k]$ dans le tas (t, k);
- ...
- insérant $t[n-1]$ dans le tas $(t, n-1)$;

Q15. Écrire une fonction `tasifier: 'a array -> unit` transformant un tableau en un tas.

Notons $n.e$ l'étiquette d'un nœud n. On rappelle l'algorithme utilisé pour extraire le min d'un tas:

Algorithme 2 : Extraction de la racine

Entrée(s) : T un tas
Sortie(s) : e étiquette de la racine de T. La racine a été supprimée, et T reste un tas

```
1  r ← racine de T ;
2  e ← r.e // valeur à renvoyer
3  f ← dernière feuille de T ;
4  Échanger r.e et f.e;
5  Réduire la taille de T de 1;
6  tant que r.e est plus grande que l'étiquette de l'un des enfants de r faire
7      n ← l'enfant de r avec la plus petite étiquette;
8      Échanger r.e et n.e;
9      n ← f;
10 retourner e
```

Plutôt que d'écraser simplement la valeur à la racine, on l'**échange** avec la valeur de la dernière feuille. Cette méthode permet de conserver globalement les valeurs contenues dans le tableau, et, si le tas est de taille i, la valeur extraite est placée à la case i du tableau t utilisé pour stocker le tas.

Q16. Implémenter une fonction déterminant si un nœud est bien placé par rapport à ses éventuels enfants. Attention, un nœud peut avoir 0, 1 ou 2 enfants:

```
(* est_bien_place t i j renvoie true si le noeud d'indice j
   dans le tas (t, i) est plus petit que son ou ses enfants. *)
est_bien_place: 'a array -> int -> int -> bool
```

Q17. Implémenter une fonction qui, étant donné un nœud mal placé (et possédant donc au moins un enfant), renvoie l'indice de son enfant ayant la plus petite étiquette:

```
(* bon_enfant t i j renvoie l'enfant de j dans le tas (t, i)
   ayant la plus petite valeur.
   Précondition: j est un noeud ayant au moins un enfant dans (t, i)
   dont l'étiquette est plus petite. *)
bon_enfant: 'a array -> int -> int -> int
```

Q18. Écrire une fonction d'extraction pour les tas binaires. **Attention**, après avoir échangé la racine et la feuille, la taille du tas n'est plus i mais $i-1$:

```
(* Extrait le minimum du tas de taille i stocké dans t.
   Renvoie la valeur extraite, et la place en t.(i) *)
let tas_extraire (t: 'a array) (i: int) : 'a = ...
```

On rappelle le principe du tri par tas. Pour trier un tableau T, dans un premier temps, on transforme T en tas, puis on extrait son minimum n fois. La procédure d'extraction plaçant le minimum du tas (T, i) à la case i de T, on obtient essentiellement une version plus efficace du tri par sélection, en $\mathcal{O}(n \log n)$!

Q19. A l'aide des fonctions précédentes, implémenter le tri par tas.

Tasification efficace

Il est possible de transformer un tableau en un tas en temps linéaire, via l'algorithme suivant:

Algorithme 3 : `tasifier`(T)

Entrée(s) : T un tableau
Sortie(s) : T a été modifié en un tas

```
1 n ← taille de T;
2 pour i = n − 1 à 0 faire
3     tamiser(T, i);
```

où `tamiser` fait descendre le nœud i vers les feuilles, et correspond à la boucle de la fonction d'extraction du min.

Q20. Implémenter la tasification efficace, et la tester

Q21. Étant donné un nœud i de profondeur p, combien de tours de boucle de la fonction `tamiser` sont-ils nécessaires pour placer i ?

Q22. Combien y a t-il de nœuds de profondeur p au plus ? En déduire une majoration du nombre total de tours de boucles effectuées par la fonction `tamiser` lors de l'exécution de `tasifier`, en fonction de h la hauteur du tas construit.

Q23. Montrer que la majoration obtenue à la question précédente est $\mathcal{O}(n)$.