Programmation – Tableaux

Jeremie.Detrey@ens-lyon.fr
Emmanuel.Jeandel@ens-lyon.fr

À rendre par mail pour le 18 octobre, 23h59

Contrairement aux DMs suivants, vous devrez faire ce devoir à la maison seul. Vous vous contenterez de rendre un seul fichier `votrenom.ml` *qui contiendra le code de toutes les fonctions demandées, ainsi que quelques explications lorsque celles-ci vous sembleront utiles. Il est en particulier important de bien indiquer par des commentaires quelle question vous traitez. Le fichier ne devrait normalement pas faire plus de 1000 lignes, puisque nous avons un corrigé qui en fait 300.*

Pensez à bien lire le sujet en entier avant de commencer. Les TDmen sont là pour vous aider, n'hésitez donc pas à les contacter en cas de difficulté.

Introduction

Le tableau est une structure de données de base qui existe dans beaucoup de langages de programmation. Il est sans doute inutile d'expliquer précisément en quoi consiste un tableau. Un tableau de taille n de type A correspond intuitivement à un ensemble de n cellules numérotées de 1 à n, chacune contenant un objet de type A.

2	7	1	8	2	8	1	8	2	8	4	5	9	0	4	5	2	3	5	3

FIG. 1 – Un tableau d'entiers[1] de taille 20.

Plusieurs opérations sont disponibles :
- créer un tableau de taille n (Dans notre contexte, un tableau a une taille fixée et on ne peut pas la changer) ;
- récupérer l'objet en $i^{\text{ème}}$ position ;
- changer la valeur d'une cellule d'un tableau ;

La troisième opération peut être comprise de plusieurs façons différentes. Cette opération change-t-elle le tableau, ou bien crée-t-elle un nouveau tableau ? Les deux possibilités sont envisageables et donnent lieu respectivement aux tableaux modifiables et aux tableaux fonctionnels.

Le but de ce devoir est de proposer plusieurs représentations possibles des tableaux modifiables et fonctionnels, et d'examiner comment on passe d'une structure à l'autre.

[1] Un seul chiffre manque. Lequel ?

1 Tableaux modifiables

Le concept de tableau modifiable est sans doute le plus connu. Pour cette structure, demander à changer le contenu d'une cellule d'un tableau revient à modifier le tableau (et non pas à en créer un nouveau).

Un module qui permet de représenter un tel tableau aura la signature suivante :

```
exception OutOfBounds;;

module type MTABLEAU = sig
  type 'a tableau
  val create : int -> 'a -> 'a tableau
  val length : 'a tableau -> int
  val get : 'a tableau -> int -> 'a
  val set : 'a tableau -> int -> 'a -> unit
end;;
```

Détaillons les différentes fonctions :

- `create n x` renvoie un tableau de taille `n` dont toutes les cases contiennent la valeur `x`.
- `length t` renvoie la taille du tableau `t`.
- `get t i` renvoie la valeur de la `i`ème case du tableau `t`. Si `i` est une valeur non autorisée (c'est-à-dire supérieure à la taille du tableau), la fonction lève l'exception `OutOfBounds` (on rappelle que les cases du tableau sont numérotées de 1 à n où n est la longueur du tableau).
- `set t i x` modifie le tableau `t` de sorte qu'à la `i`ème case du tableau soit désormais présente la valeur `x`. Cette fonction modifie ses arguments et ne renvoie rien.

Voilà une utilisation typique d'une telle structure de données :

```
(* A est un module de signature MTABLEAU *)

(* Création d'un tableau de chaînes de caractères
   de 6 cases initialisées avec la valeur "music" *)
let t = A.create 6 "music";;
A.length t;;    (* renvoie 6 *)
A.get t 2;;  (* renvoie "music" *)
A.set t 4 "klezmer";;
A.get t 4;;  (* renvoie "klezmer" *)
let m = t;;
A.set m 4 "grindcore";;
A.get t 4;;  (* renvoie "grindcore" *)
A.get m 4;;  (* renvoie "grindcore" *)
```

1.1 Le module `Array`

Le module `Array` de OCaml propose une représentation des tableaux modifiables. Les fonctions suivantes existent :

- `Array.make n x` renvoie un tableau du type `'a array` de taille `n` dont toutes les cellules contiennent `x` ;
- `Array.length t` renvoie la taille du tableau `t` de type `'a array` ;
- `Array.get t i` renvoie la valeur de la case numéro i du tableau
  Attention : Dans le module `Array`, les cases commencent à 0 et non à 1 ;
- `Array.set t i x` modifie `t` de sorte que la case numéro `i` du tableau soit `x`.

Si on appelle `Array.get` et `Array.set` avec une valeur non valide (`i` supérieur ou égal à la taille du tableau), la fonction lève l'exception `Invalid_argument "index out of bounds"`.

Q1) Ecrire un module nommé `TArray` de signature `MTABLEAU` basée sur le module `Array`

```
module TArray : MTABLEAU = struct
    type 'a tableau = 'a array

    let create n x = Array.make n x

    (* Écrivez la suite *)
   end;;
```

Remarque N'oubliez pas les deux différences essentielles entre le module `Array` et celui qu'on vous demande de faire : les indices de nos tableaux commencent à 1 et non à 0. De plus, l'exception levée en cas d'erreur s'appelle `OutOfBounds` et non pas `Invalid_argument "index out of bounds"`.

1.2 Représentation par liste

Remarque Pour ceux qui ont fait l'initiation Caml, cette partie est très similaire à ce que vous avez fait pendant l'initiation.

On code les tableaux par des listes de références. Ainsi le tableau | 1 | 4 | 7 | 5 | 3 | est représenté par :

```
let t = [ ref(1); ref(4); ref(7); ref(5); ref(3) ];;
```

Q2) Écrire un module nommé `TRef` de signature `MTABLEAU` utilisant cette représentation des tableaux

```
module TRef : MTABLEAU= struct

    type 'a tableau = 'a ref list

    let rec create n x = (* ... *)

   (* ... *)

   end;;
```

1.3 Tableaux non sécurisés

Un petit plaisantin a fait un module `Blop` de signature `MTABLEAU` dans lequel il a oublié de vérifier, dans les fonctions `Blop.set` et `Blop.get`, si le programme n'essayait pas d'accéder à une case qui n'existait pas (donc d'un numéro supérieur à la longueur du tableau). On appelle de tels tableaux des tableaux non sécurisés.

Q3) Écrire un foncteur `Secure` qui transforme un module représentant des tableaux non sécurisés en un module représentant des tableaux qui n'ont pas ces petits problèmes.

```
module Secure (A : MTABLEAU) : MTABLEAU =
 struct
    .....
 end;;
```

2 Tableaux fonctionnels

Contrairement aux tableaux modifiables, on ne peut pas modifier un tableau fonctionnel : demander à changer la $i^{\text{ème}}$ cellule d'un tableau fonctionnel renvoie un nouveau tableau qui ne diffère du précédent que par cette cellule.

Un module qui permet de représenter un tel tableau aura la signature suivante :

```
exception OutOfBounds;;

module type FTABLEAU = sig
  type 'a tableau
  val create : int -> 'a -> 'a tableau
  val length : 'a tableau -> int
  val get : 'a tableau -> int -> 'a
  val set : 'a tableau -> int -> 'a -> 'a tableau
end;;
```

La différence par rapport à la partie précédente réside donc dans la fonction `set` qui, cette fois-ci, renvoie un tableau.

Écrivons un petit exemple :

```
  (* A est un module de signature FTABLEAU *)

  (* Création d'un tableau de chaînes de caractères
     de 6 cases initialisées avec la valeur "music" *)
  let t = A.create 6 "music";;
  A.length t;; (* renvoie 6 *)
  A.get t 2;;  (* renvoie "music" *)
  let m = A.set t 4 "klezmer";;
  A.get t 4;;  (* renvoie "music" *)
  A.get m 4;;  (* renvoie "klezmer" *)
```

2.1 Listes

Une façon simple de représenter de tels tableaux est de les représenter par des listes. Ainsi le tableau | 1 | 4 | 7 | 5 | 3 | est représenté par :

```
let t = [ 1; 4; 7; 5; 3 ];;
```

Q1) Ecrire un module nommé `Tlist` de signature `FTABLEAU` utilisant cette représentation des tableaux.

2.2 Listes d'association

On peut également représenter des tableaux fonctionnels par une liste d'associations, c'est à dire par une liste de couples (i, x), signifiant "l'élément dans la case i est x". Lorsqu'on veut savoir quel est l'élément en position i, on cherche la première occurence de $(i, \dots)$ dans la liste.

Ainsi, un même tableau peut être représenté de plusieurs façons. Les deux listes suivantes

```
let l = [ (1,1) ; (2,4); (3,7); (4, 5); (5, 3) ];;
let m = [ (2,4) ; (2,5) ; (3,7); (4, 5); (1,1); (5, 3); (3,9) ];;
```

représentent le même tableau, qui est le tableau pris en exemple depuis maintenant plusieurs questions.

Q2) Ecrire un module nommé `TAlist` de signature `FTABLEAU` utilisant cette représentation des tableaux. (La fonction `List.assoc` peut être utile)

Remarque Il y a plusieurs façons de tenir compte du fait que les tableaux ont une taille fixée. Nous vous laissons entière liberté sur ce point.

On vous conseille toutefois de définir le tableau par :

```
type 'a tableau = int * (  (int * 'a) list )
```

au lieu de :

```
type 'a tableau =   (int * 'a) list
```

de sorte que le tableau précédent sera représenté par

```
let t = (5, [ (1,1) ; (2,4); (3,7); (4, 5); (5, 3) ]);;
```

afin d'indiquer sa taille.

2.3 Fonctions

Une des manières les plus naturelles[1] de représenter un tableau est de le représenter par une fonction de type `int -> 'a`. Ainsi le sempiternel tableau | 1 | 4 | 7 | 5 | 3 | peut être représenté par l'une des deux fonctions suivantes :

```
let f n = if n <= 2 then  n*n-1*6+6+4-(5-1)
          else if n <= 5 then 1+(6+(6+(4*n)-5*n)-1*n)
          else raise OutOfBound;;
```

[1] Mais aussi une des moins efficaces

```
let g n = if n <= 5 then List.nth [1;4;7;5;3] (n-1)
          else raise OutOfBound;;
```

Q3) Ecrire un module nommé `FunA` de signature `FTABLEAU` utilisant cette représentation des tableaux. (Commencez par écrire la fonction `length`, qui n'est pas la plus simple.)

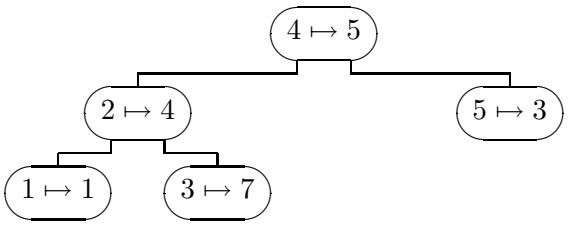
2.4 Arbres binaires de recherche

Il s'agit sans doute de la structure la plus utilisée pour représenter ces tableaux. La structure est un arbre binaire de couples `(i,x)` (signifiant que l'élément en position i est l'élément `x`) vérifiant la contrainte suivante : En tout noeud `(i,x)`, les éléments `(j,x)` qui ont une position antérieure ($j < i$) sont situés dans le fils droit du noeud, ceux qui vérifient $j > i$ sont dans le fils gauche.

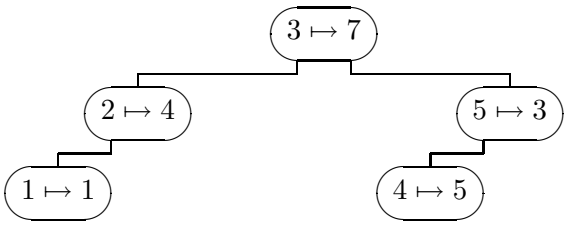
On définit donc le type comme suit :

```
type 'a couple = (int * 'a)
type 'a tableau = Node of  'a couple * 'a tableau * 'a tableau | Leaf
```

Voici deux exemples d'arbres binaires de recherche représentant un tableau[2].




```
let t = Node((4,5),
            Node((2,4) ,
               Node((1,1), Leaf, Leaf),
               Node((3,7), Leaf, Leaf)
                ),
            Node((5,3) , Leaf, Leaf)
            );;
```




```
let t = Node((3,7),
            Node((2,4) ,
               Node((1,1), Leaf, Leaf),
               Leaf
               ),
            Node((5,3) ,
               Node((4,5), Leaf, Leaf),
               Leaf
               )
            );;
```

[2]Jeu : de quel tableau s'agit-il ?

Q4) Ecrire un module nommé `FArbre` de signature `FTABLEAU` utilisant cette représentation des tableaux.

Remarque Il y a plusieurs façons de faire `create`. Écrivez un ou deux mots d'explications sur le choix que vous avez privilégié.

3 Foncteurs

On va maintenant construire des foncteurs permettant de passer d'un des modèles à l'autre.

Q1) Écrire un foncteur `MtoF` qui convertit un module de type `MTABLEAU` en un module de type `FTABLEAU`.

Q2) Écrire un foncteur `FtoM` qui convertit un module de type `FTABLEAU` en un module de type `MTABLEAU`.

On va maintenant expliquer une méthode permettant de transformer un tableau modifiable en un tableau fonctionnel. L'idée de base est la suivante : dans la vraie vie, on n'a souvent besoin que de la dernière "version " d'un tableau fonctionnel. Par exemple, si on a le code suivant :

```
(* t est un tableau fonctionnel *)
let l = create 5 "x";;
let t = set l 3 "a";;
let v = set t 2 "b";;
let y = set l 1 "z";;
(* ... *)
```

on observe statistiquement que le reste du programme va davantage s'attacher à `v` et `y` qu'aux deux autres tableaux.

On suppose disposer d'un module `A` représentant des tableaux modifiables. Notre modèle des tableaux fonctionnels est donc le suivant : un tableau fonctionnel est soit un tableau du type `A.tableau`, soit un *patch* sur un tableau fonctionnel. Plus précisément, le type sera :

```
type 'a tab = Basis of 'a A.tableau | Patch of int * 'a * 'a tableau
 and  'a tableau = 'a tab ref
```

Montrons le fonctionnement sur un exemple suivi

- `let l = create 5 "x"`. On se contente de créer un tableau de type `'a A.tableau` et on "l'encapsule" en un tableau du type `'a tableau`. De sorte que `l` est en fait égal à `ref(Basis(m))` où `m` est un tableau de type `'a A.tableau` rempli de `"x"`.
- `let t = set l 3 "a"`. Comme `l` va servir statistiquement moins que `t`, il serait idiot de recopier tout `l` dans `t`. On va en fait remplacer `t` par `ref(Basis(m))`, après avoir modifié la 3ème case du tableau `m`. Pour retrouver `l`, il suffit de dire que `l` est comme `t` dans lequel on n'aurait pas changé la 3ème valeur, donc dans lequel la valeur est toujours `"x"`. On va donc faire `l := Patch(3, "x", t)`
- `let v = set t 2 "b"`. Même principe, on va en fait remplacer `v` par `ref(Basis(m))`, après avoir modifié la 2ème case du tableau `m`. Maintenant `t := Patch(2, "x", v)`. Comme on a toujours `l := Patch(2, "x", t)`, tout va bien se passer.

Vous avez compris ? Bien. Sinon, n'hésitez pas à demander à votre TDman.

Il reste un dernier cas à traiter

– `let y = set l 1 "z"`. Comme `l` est maintenant un patch et non pas un vrai tableau, on ne peut pas faire la même chose qu'avant. Il y a plusieurs façons de régler le problème. Dans cette situation, le plus simple est de créer un nouveau tableau `n`, et de recopier dans `n` le contenu de `l`, sans oublier évidemment de changer le premier élément en `"z"`. Quant à `l`, on peut au choix le laisser comme tel, soit dire maintenant qu'on l'obtient comme un patch sur `y`.

Vous avez tout compris ? Il vous reste maintenant à répondre à la dernière question.

Q3) Ecrire un foncteur `MtoFun` qui convertit un module de type `MTABLEAU` en un module de type `FTABLEAU` en utilisant le procédé décrit ci-avant.

```
module MtoFun (A:MTABLEAU): FTABLEAU =
struct

 type 'a tab = Basis of 'a A.tableau | Patch of int * 'a * 'a tableau
 and  'a tableau = 'a tab ref

 let create n x = ref(Basis(A.create n x))
  ...
  ...

 end;;
```