# Programmation TP 3 : Tableaux, mémoire

{ jrobert, fbouchez, apardon } @ens-lyon.fr
http://perso.ens-lyon.fr/florent.bouchez/
2 octobre 2007

## Tableaux, mémoire

Le tableau est une structure de données de base qui existe dans beaucoup de langages de programmation. 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 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 $i$-ème position,
- changer la valeur d'une cellule d'un tableau.

Cette dernière opération peut être comprise de plusieurs façons différentes :
- soit elle change le tableau,
- soit elle renvoie un nouveau tableau.

On parlera de tableau *modifiable*, dans le premier cas, et *fonctionnel* dans le second cas.

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

**Q 0.0** Téléchargez l'archive qui se trouve sur la page des tds, et ouvrez-là :

```
mkdir tp03; cd tp03; tar xjf ../fichiers03.tar.bz2
```

Elle contient trois fichiers. Les fichiers `check.cmo` et `check.cmi` sont des fichiers compilés pour OCaml. Le fichier `types.mli` contient, entre autres, une signature pour les tableaux modifiables, et une pour les tableaux fonctionnels.

Quelle est la différence entre ces deux signatures ?

Pour la suite, ouvrez un nouveau fichier, dans lequel vous travaillerez, et chargez le module `Check` pour avoir accès à ses définitions : `#load "check.cmo";; open Check;;`

# 1 Tableaux modifiables

Un module qui permet de représenter un tableau modifiable a la signature suivante (définie dans le module que vous venez de charger) :

```
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 même 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, ou négative, ou nulle), 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 que la valeur `x` soit désormais stockée à la `i`-ème case du tableau. 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 "tonk" *)
let t = A.create 6 "tonk";;
A.length t;;     (* renvoie 6 *)
A.get t 2;;      (* renvoie "tonk" *)
A.set t 4 "tchak";;
A.get t 4;;      (* renvoie "tchak" *)
let m = t;;
A.set m 4 "pouet";;
A.get t 4;;      (* renvoie "pouet" *)
A.get m 4;;      (* renvoie "pouet" *)
```

## 1.1 Représentation par des listes de références

Les tableaux modifiables sont déjà implantés dans OCaml, dans le module `Array`. Nous allons les coder avec des listes de références. Ainsi le tableau | 1 | 4 | 7 | 5 | 3 | est représenté par la liste :

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

**Q 1.1** Écrivez un module nommé `MRefList` de signature `MTABLEAU` utilisant cette représentation des tableaux.

```
module MRefList : MTABLEAU= struct
  type 'a tableau = 'a ref list
  let rec create n x = (* ... *)
  (* ... *)
end;;
```

**Q 1.2** Vérifiez sa validité à l'aide du foncteur `MCheck`, (cf. `check.mli`). La syntaxe est un peu bizarre : `let module M = MCheck(MRefList) in M.test false 100;;`
- `100`, c'est pour faire 100 tests,
- `false`, c'est pour ne pas tout afficher ; à mettre à `true` si votre module s'avère être faux.

## 2 Tableaux fonctionnels

Contrairement aux tableaux modifiables, on ne peut pas modifier un tableau fonctionnel : demander à changer la $i$-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 a la signature suivante :

```
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;;
```

Un exemple d'utilisation d'un tel module :

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

### 2.1 Fonctions

La façon la plus naturelle de représenter un tableau fonctionnel est d'utiliser les listes ; comme c'est trop facile, on ne le fera pas dans ce TP… Une des manières les plus élégantes[1] de représenter un tableau est de le représenter par une fonction de type `int -> 'a`. Ainsi le tableau | 1 | 4 | 7 | 5 | 3 | peut être représenté par l'une des deux fonctions suivantes :

[1] Mais aussi une des moins efficaces

```
let f n = if n>0 and n <= 2 then  n*n-1*6+6+4-(5-1)
          else if n>0 and n <= 5 then 1+(6+(6+(4*n)-5*n)-1*n)
          else raise Outofbounds;;
let g n = if n>0 and n <= 5 then List.nth [1;4;7;5;3] (n-1)
          else raise Outofbounds;;
```

**Q 2.1** Écrivez et vérifiez un module nommé `FFun` de signature `FTABLEAU` utilisant cette représentation des tableaux. (Commencez par écrire la fonction `length`, qui n'est pas la plus simple.)

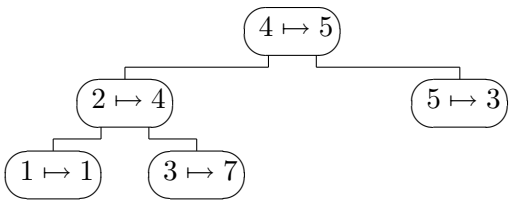
Attention, pour le vérifier, il faut utiliser le foncteur `FCheck` (avec un `F` comme fonctionnel).

## 2.2 Arbres binaires de recherche

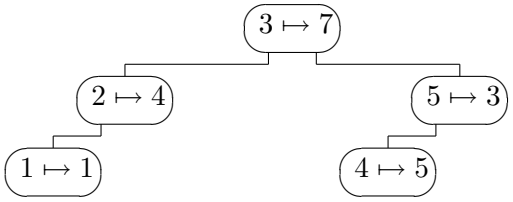
Il s'agit sans doute de la structure la plus utilisée pour représenter ces tableaux. La structure est un arbre binaire de couples `(i,x)` (signifiant que l'élément en position $i$ est l'élément `x`) vérifiant la contrainte suivante : En tout nœud `(i,x)`, les éléments `(j,x)` qui ont une position antérieure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

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

Voici deux exemples d'arbres binaires de recherche représentant le même tableau :




```
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));;
```

**Q 2.2** Écrivez et vérifiez un module nommé `FArbre` de signature `FTABLEAU` utilisant cette représentation des tableaux.

**Remarque** Il y a plusieurs façons de faire `create`. Essayez d'écrire la meilleure…

# 3 Foncteurs

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

**Q 3.1** Écrivez un foncteur `MtoF` qui convertit un module de type `MTABLEAU` en un module de type `FTABLEAU` (faites simple, cf. la suite).

**Q 3.2** Écrivez un foncteur `FtoM` qui convertit un module de type `FTABLEAU` en un module de type `MTABLEAU`.

Vérifiez leur validité à l'aide des foncteurs `MFCheck` et `FMCheck`.

On va maintenant expliquer une méthode plus élaborée, permettant de transformer un tableau modifiable en un tableau fonctionnel. L'idée de base est la suivante : statistiquement, on n'utilise que les dernières "versions" d'un tableau fonctionnel. Par exemple, si on a le code suivant :

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

le reste du programme va généralement davantage utiliser `v` et `y` que les 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 :

- `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` soit 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 donne donc la valeur `ref(Basis(m))` pour `t`, après avoir modifié la 3e case du tableau `m`. Pour retrouver `l`, il suffit de dire que `l` est comme `t` dans lequel on n'aurait pas changé la 3e valeur, donc dans lequel la valeur est toujours `"x"`. On fait donc `l := Patch(3, "x", t)`.
- `let v = set t 2 "b"`. Même principe, donne la valeur `ref(Basis(m))` pour `v`, après avoir modifié la 2e case du tableau `m`. Maintenant `t := Patch(2, "x", v)`. Comme on a toujours `l := Patch(2, "x", t)`, tout se passe bien.

Il reste un dernier cas :

- `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`.

(N'hésitez pas à demander des explications orales !)

**Q 3.3** Écrivez et vérifiez un foncteur `MtoFPatch` qui convertit un module de type `MTABLEAU` en un module de type `FTABLEAU` en utilisant le procédé décrit ci-avant.

```
module MtoFPatch (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;;
```

# 4 Question subsidiaire

Implantez un module `Check` de signature `Check.mli`, permettant de vérifier vos modules.