OCaml impératif #2 : `arrays`, boucles

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Table des matières

Structure de donnée abstraite de tableau (rappel)

Tableau mutable

Une spécification possible :

- Constructeur *create* : paramètres : un entier n, une valeur e d'un certain type ; crée et renvoie un nouveau tableau de n éléments d'indices $\in [\![0, n-1]\!]$ et de valeur e
- Accesseur *get* : paramètres : tableau a, un indice i ; renvoie la valeur de l'élément de a à l'indice i
- Transformateur *set* : paramètres : un tableau a, un indice i, une valeur x ; écrit la valeur x à l'indice i de a

Implémentations possibles

C :

- *create* : `malloc` + initialisation
- *get*, *set* : immédiat & efficace
- pas de polymorphisme, à moins de « boxer » les éléments dans des `void *` et de connaître leur taille 🙈
- nécessite d'ajouter une fonction de libération

OCaml : avec une `'a list` de références

Exemple C

```c
int *create(size_t n, int e) {
    int *a = ckd_malloc(n * sizeof(int));
    for (size_t i = 0; i < n; i++) {
        a[i] = e;
    }
    return a;
}

void destruct(int *a) { free(a); }

int get(int *a, size_t i) { return a[i]; }

void set(int *a, size_t i, int x) { a[i] = x; }
```

Exemple OCaml

```
type 'a arr = 'a ref list
exception Out_of_bounds

let create n e = List.init n (fun _ -> ref e)

let rec get a i
    = match a with
    | [] -> raise Out_of_bounds
    | x::xs -> if i = 0 then !x else get xs (i - 1)

let rec set a i x
    = match a with
    | [] -> raise Out_of_bounds
    | y::ys -> if i = 0 then y := x else set ys (i - 1) x
```

Exemple OCaml : commentaires

- « Évidemment » inefficace
  - Détails ?
- On peut faire nettement (exponentiellement !) mieux dans le pire cas, en passant par des arbres (bientôt !)

Le type concret `'a array`

Type prédéfini `'a array`

Type des « tableaux » (mutables) polymorphes de valeurs de type `'a`

- Construction : fonction `Array.make` du module `Array`
- Lecture à l'indice `i` : `a.(i)` (ou `get`)
- Écriture à l'indice `i` : `a.(i) <- x` (ou `set`)
- Littéraux : `[|a1; a2; a3|]`

Avec `get`, `set` en temps constant (dans le modèle RAM gentil)

Tableaux en OCaml avec des `'a array`

```
type 'a arr = 'a array
let create n e = Array.make n e
let get a i = a.(i)
let set a i x : unit = a.(i) <- x
```

Démo.

`Array` : fonctions à connaître

- `Array.make : int -> 'a -> 'a array`
- `Array.length : 'a array -> int` : longueur (nombre d'éléments) ≥ 0 de son argument ; coût constant (équation : `Array.length (Array.make n _) = n`)
- `Array.copy : 'a array -> 'a array` : copie superficielle de son argument (**attention !** *cf.* plus bas)

Mais aussi :

- `Array.init : int -> (int -> ' a) -> 'a array` : création d'un `'a array` dont le $i$ème élément est donné par le second argument évalué sur i
- `Array.` `mem`, `exists`, `for_all`, `iter` : comme sur des `'a list`
- `Array.make_matrix : int -> int -> 'a -> 'a array` : création d'un `'a array` rectangulaire

Plus de détails (et de fonctions pratiques) : module `Array`

Attention : OCaml autorise la création de `'a array` de longueur nulle

Caveat aliasing

Les valeurs mutables OCaml sont manipulées *via* des références : risque d'*aliasing*

`Array.make` n'évalue son second argument qu'une seule fois

- Si celui-ci est « fonctionnel » : *fine*
- Si celui-ci est mutable : *aliasing* : tous les éléments référencent vers le même objet (**not fine**)

```
utop # let a = Array.make 4 (ref 3);;
val a : int ref array =
[|{contents = 3}; {contents = 3}; {contents = 3}; {contents = 3}|]
utop # a.(3) := 1;;
utop # a;;
- : int ref array =
[|{contents = 1}; {contents = 1}; {contents = 1}; {contents = 1}|]
```

Caveat aliasing *bis*

- Même problème pour construire des `'a array array` avec `make`
- Démo.

Solution pour `'a array array`

Utiliser `Array.make_matrix`

```
utop # let a = Array.make_matrix 3 3 0;;
val a : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
utop # a.(0).(0) <- 1;;
utop # a;;
- : int array array = [|[|1; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

Solution générale

Forcer des créations d'objets distincts *via* une fonction `int -> 'a` utilisée dans `Array.init`

- Comparez `Array.init 4 (fun _ -> ref 3)` avec ci-dessus

Aliasing & copie

La fonction `Array.copy` est superficielle : elle copie uniquement les références

```
utop # let a = Array.make_matrix 2 2 0;;
utop # let b = Array.copy a;;
utop # a.(0).(0) <- 1;;
utop # b;;
- : int array array = [|[|1; 0|]; [|0; 0|]|]
```

Solutions

Rien de très général… dépend de la mutabilité ou non des éléments

Copies profondes *ad hoc*

Pour des `'a array array` (dans le dernier cas : non vide) sur un type immuable :

```
let copy2d a
    = Array.init (Array.length a) (fun i -> Array.copy a.(i))
let copy2d' a = Array.map Array.copy a
let copy2d'' a =
    let n = Array.length a in
    let b = Array.make n [||] in
    for i = 0 to (n - 1) do (* OwO what's this? *)
        let m = Array.length a.(i) in
        b.(i) <- Array.make m a.(i).(0) ;
        for j = 0 to (m - 1) do
            b.(i).(j) <- a.(i).(j)
        done ;
    done ;
    b
```

Table des matières

Boucles impératives

Il est possible en OCaml d'écrire des boucles :

- non bornées (« `while` »)
- bornées (« `for` »)

Occasionnellement utile, rarement agréable

Boucle `while`

Syntaxe & sémantique :

```
(while eb do st done : unit)
```

- `(eb : bool)` condition d'entrée/maintien dans la boucle
- `st` : expression du corps de boucle, évaluée à chaque itération
  - **devrait** être complètement impure et avoir type `unit` (sinon : valeur d'évaluation **perdue**)
  - souvent une expression « composée » donnée par une séquence d'expressions

Exemple : `while`

```
let gcd a b =
    let ra, rb = if a < b then
                 ref b, ref a else
                 ref a, ref b in
    while !rb > 0 do
        let t = !ra in
        ra := !rb ;
        rb := t mod !rb
    done ;
    !ra
```

Boucle `for`

Syntaxe & sémantique

```
(for i = s to e do st done : unit)
```

- `(i : int)` : compteur de boucle **immuable** prenant les valeurs `s`…`e` deux bornes incluses, avec des pas *croissants* de `1` lors des évaluations successives du :
- corps de boucle `st`, qui **devrait** être de type `unit`

```
(for i = s downto e do st done : unit)
```

- identique à ci-dessus, avec des pas *décroissants* de `1`

Exemples : `for`

```
copy2d''

utop # for i = 0 to 5 do print_int i ; print_string " " done;;
0 1 2 3 4 5 - : unit = ()

utop # for i = 5 downto 0 do
            float_of_int i |> print_float ; print_string " "
       done;;
5. 4. 3. 2. 1. 0. - : unit = ()
```

Sorties anticipées

Il n'y a pas de `return` en OCaml (rappel), et pas non plus de `break` ou de `continue` pour les boucles

- On peut utiliser des `exceptions` pour implémenter des `break`, `continue`, et sortie de fonction anticipée

Exemple

```
exception Okay

let exists p a
    = try
    for i = 0 to Array.length a - 1 do
        if p a.(i) then raise Okay
    done ;
    false
    with Okay -> true
```