

TP09

February 20, 2022

1 Enregistrements et champs mutables

On a vu qu'on pouvait définir des sortes de tuples dont les membres étaient nommés avec les enregistrements et leur champ. Ainsi, au lieu d'écrire

```
let e = ("Louise", "Michel", 1830)
```

on peut définir un type enregistrement :

```
type personne = { prenom : string; nom : string; annee_naissance : int }
```

```
et ensuite let e = { prenom = "Louise"; nom = "Michel"; annee_naissance = 1830 }
```

ce qui permet d'accéder directement au valeur d'un champ avec `e.prenom` par exemple.

Il y a de plus, un point très important des enregistrements : **on peut déclarer un champ mutable pour indiquer qu'on peut changer sa valeur**. C'est un point critique qui ouvre beaucoup de possibilités.

Commençons avec un exemple élémentaire :

```
[1]: type test = { a : int; mutable b : int }  
let t = { a = 3; b = 2}
```

```
[1]: type test = { a : int; mutable b : int; }
```

```
[1]: val t : test = {a = 3; b = 2}
```

Pour changer la valeur d'un champ mutable, on utilise `<-` comme pour un `array` :

```
[2]: t.b <- 4;  
t
```

```
[2]: - : test = {a = 3; b = 4}
```

Si on essaie avec un champ non mutable, cela produit un erreur.

```
[3]: t.a <- 2
```

```
File "[3]", line 1, characters 0-8:
1 | t.a <- 2
   ~~~~~~
Error: The record field a is not mutable
```

Il se trouve que vous manipulez déjà un type enregistrement avec champ mutable : **les références** !

En effet, on peut les redéfinir nous même :

```
[4]: type 'a ref = { mutable contents : 'a }
```

```
[4]: type 'a ref = { mutable contents : 'a; }
```

```
[5]: let ref x = { contents = x }
```

```
[5]: val ref : 'a -> 'a ref = <fun>
```

```
[6]: let (:=) r x = r.contents <- x
```

```
[6]: val ( := ) : 'a ref -> 'a -> unit = <fun>
```

```
[7]: let (!) r = r.contents
```

```
[7]: val ( ! ) : 'a ref -> 'a = <fun>
```

```
[8]: let x = ref 3 in
x := !x + 1;
!x
```

```
[8]: - : int = 4
```

On comprend bien pourquoi on utilise `:=` et pas `<-` pour les références, c'est parce que `:=` n'est qu'un raccourci sur `.contents <-`.

On peut voir au passage comment on peut définir des opérations avec des symboles en utilisant les parenthèses. On a alors droit d'utiliser n'importe quel assemblage des caractères `!$%&*+-./:~?@^<>=|`.

C'est assez pratique, on peut ainsi définir par exemple un enregistrement pour les `vec3` comme on a pu les voir en C :

```
[9]: type vec3 = { x : float; y : float; z : float}

let (+~) v1 v2 = { x = v1.x+.v2.x; y = v1.y+.v2.y; z = v1.z+.v2.z}

let a = { x = 1.; y = 2.; z = 3. }
let b = { x = -1.; y = 1.; z = 2. }
```

```
[9]: type vec3 = { x : float; y : float; z : float; }
```

```
[9]: val ( +~ ) : vec3 -> vec3 -> vec3 = <fun>
```

```
[9]: val a : vec3 = {x = 1.; y = 2.; z = 3.}
```

```
[9]: val b : vec3 = {x = -1.; y = 1.; z = 2.}
```

```
[10]: a +~ b
```

```
[10]: - : vec3 = {x = 0.; y = 3.; z = 5.}
```

2 Tableaux dynamiques

On va se servir d'un champ mutable pour définir un type 'a dynarray de tableaux dynamiques, c'est-à-dire de tableaux dont la taille pourra changer en ajoutant des éléments dedans à la manière des list de Python.

Pour cela, on va d'abord commencer à définir des fonctions sur les array (qui existent pour beaucoup dans la bibliothèque standard mais c'est surtout l'occasion de s'entraîner).

2.1 Préliminaires - fonctions de manipulations de tableaux

```
[2]: let array_blit (tab1:'a array) (debut1:int)
      (tab2:'a array) (debut2:int) (nb:int) : unit =
  (* Copie les nb éléments de tab1 à partir de l'indice debut1
   dans le tableau tab2 à partir de l'indice debut2.
   On appellera toujours cette fonction en sorte qu'il n'y ait pas d'accès
   érronés en mémoire *)
  (* BEGIN SOLUTION *)
  for i = 0 to nb-1 do
    tab2.(debut2 + i) <- tab1.(debut1 + i)
  done
  (* END SOLUTION *)
```

```
[2]: val array_blit : 'a array -> int -> 'a array -> int -> int -> unit = <fun>
```

```
[12]: let t1 = Array.init 5 (fun i -> i) in
let t2 = Array.init 6 (fun i -> i+1) in
array_blit t1 3 t2 1 2;
assert(t2 = [|1;3;4;4;5;6|]);
let t1 = [| 'a'; 'b'; 'c' |] in
let t2 = [| 'd'; 'e' |] in
array_blit t1 1 t2 0 2;
assert(t2 = [| 'b'; 'c' |]);
"Valide"
```

```
[12]: - : string = "Valide"
```

```
[3]: let array_extend (tab:'a array) (debut:int) (nb:int) : 'a array =
  (* Renvoie un NOUVEAU tableau contenant le même contenu que tab sauf qu'on
  ↪ a rajouté
  nb éléments à partir de l'indice debut. La valeur des éléments rajoutés
  ↪ n'est pas signifiante.
  On suppose que tab est non vide.
  Si debut vaut la longueur de tab, les nouveaux éléments sont rajoutés après
  ↪ la recopie de tab.
  On utilisera array_blit *)
  (* BEGIN SOLUTION *)
  let n = Array.length tab in
  let tabn = Array.make (n + nb) tab.(0) in
  array_blit tab 0 tabn 0 debut;
  array_blit tab debut tabn (debut + nb) (n - debut);
  tabn
  (* END SOLUTION *)
```

```
[3]: val array_extend : 'a array -> int -> int -> 'a array = <fun>
```

```
[4]: let t = Array.init 5 (fun i -> i)
```

```
[4]: val t : int array = [|0; 1; 2; 3; 4|]
```

```
[5]: array_extend t 3 5
```

```
[5]: - : int array = [|0; 1; 2; 0; 0; 0; 0; 0; 3; 4|]
```

```
[32]: let t = Array.init 5 (fun i -> i) in
let t2 = array_extend t 3 5 in
assert(Array.length t2 = 10 && Array.sub t2 0 3 = [|0;1;2|] && Array.sub t2 8 2
↳ = [|3; 4|]);
let t = Array.init 5 (fun i -> char_of_int (i+97) ) in
let t2 = array_extend t 0 3 in
assert(Array.length t2 = 8 && Array.sub t2 3 5 = t);
let t2 = array_extend t 5 3 in
assert(Array.length t2 = 8 && Array.sub t2 0 5 = t);
"Valide"
```

```
[32]: - : string = "Valide"
```

```
[4]: let array_concat (tab1:'a array) (tab2:'a array) : 'a array =
(* Renvoie le tableau obtenu en plaçant les éléments de tab1 avant
ceux de tab2 *)
(* BEGIN SOLUTION *)
let n1 = Array.length tab1 in
let n2 = Array.length tab2 in
if n1 = 0 then tab2
else if n2 = 0 then tab1
else
    let tab = Array.make (n1+n2) tab1.(0) in
    array_blit tab1 0 tab 0 n1;
    array_blit tab2 0 tab n1 n2;
    tab
(* END SOLUTION *)
```

```
[4]: val array_concat : 'a array -> 'a array -> 'a array = <fun>
```

```
[34]: assert(array_concat [|1;2|] [|3|] = [|1;2;3|]);
assert(array_concat [|]| [|]| = [|]|);
assert(array_concat [|]| [|1|] = [|1|]);
assert(array_concat [|1|] [|]| = [|1|]);
"Valide"
```

```
[34]: - : string = "Valide"
```

```
[9]: let array_delete (tab : 'a array) (debut : int) (nb : int) : 'a array =
(* Renvoie un NOUVEAU tableau obtenu en supprimant nb éléments de tab
à partir de l'indice debut. On suppose que nb > 0 et donc que tab est non
↳ vide. *)
(* BEGIN SOLUTION *)
let n = Array.length tab in
```

```

let t = Array.make (n - nb) tab.(0) in
array_blit tab 0 t 0 debut;
array_blit tab (debut + nb) t debut (n - nb - debut);
t
(* END SOLUTION *)

```

```
[9]: val array_delete : 'a array -> int -> int -> 'a array = <fun>
```

```
[13]: assert(array_delete [|1;2;3;4;5;6|] 2 2 = [|1;2;5;6|]);
      assert(array_delete [|1;2;3;4;5;6|] 0 3 = [|4;5;6|]);
      "Valide"
```

```
[13]: - : string = "Valide"
```

2.2 Premier type de tableaux dynamiques

On définit le type suivant pour contenir un tableau dynamique. On va devoir realloquer les éléments, donc on aura besoin d'accéder en écriture au tableau qui les contient, le plus simple dans un premier temps est de faire une référence sur un tableau.

Note le type est un type synonyme, par défaut, OCaml va afficher 'a array ref dans ses résultats mais on peut forcer l'affichage de 'a dynarray en spécifiant les types dans les fonctions comme on va le faire en dessous.

```
[15]: type 'a dynarray = 'a array ref
```

```
[15]: type 'a dynarray = 'a array ref
```

Comme on va définir des tableaux en partant du tableau vide, comme en Python avec [], on peut se permettre de ne pas donner le type d'un des éléments. En fait, contrairement à ce qui semble la fonction suivante n'a pas un type magique, le 'a sera instancié le plus vite possible, ce qu'on peut voir avec le '_weak1 dans le type. **Attention** : vraiment comprendre ces notions de paramètres génériques faibles est trop complexe et dépasse le cadre du cours de CPGE.

```
[37]: let dynarray_init () = ref [|]|
```

```
[37]: val dynarray_init : unit -> 'a array ref = <fun>
```

```
[38]: let v = dynarray_init ()
```

```
[38]: val v : '_weak2 array ref = {contents = [|]|}
```

```
[39]: v := [|1; 2|]
```

```
[39]: - : unit = ()
```

```
[40]: (* Maintenant v est un int dynarray *)  
v
```

```
[40]: - : int array ref = {contents = [|1; 2|]}
```

On va maintenant définir des fonctions d'ajout et de suppression d'éléments dans le tableau. Pour cela, on va realloquer le tableau grâce aux fonctions de la partie précédente. **Attention** rappel dans `array_extend` on a supposé le tableau non vide, mais ici il faudra surement faire un cas.

```
[42]: let dynarray_append (tab : 'a dynarray) (x : 'a) : unit =  
      (* ajoute x à la fin du tableau tab *)  
      (* BEGIN SOLUTION *)  
      let n = Array.length !tab in  
      if n = 0 then tab := [|x|]  
      else begin  
          let t = array_extend !tab n 1 in  
          t.(n) <- x;  
          tab := t  
      end  
      (* END SOLUTION *)
```

```
[42]: val dynarray_append : 'a dynarray -> 'a -> unit = <fun>
```

```
[47]: let t = dynarray_init () in  
dynarray_append t 1;  
dynarray_append t 2;  
dynarray_append t 3;  
dynarray_append t 4;  
assert(!t = [|1;2;3;4|]);  
"Valide"
```

```
[47]: - : string = "Valide"
```

```
[16]: let dynarray_pop (tab : 'a dynarray) : 'a =  
      (* supprime le dernier élément de tab et le renvoie. *)  
      (* BEGIN SOLUTION *)  
      let n = Array.length !tab in  
      let x = !tab.(n-1) in  
      let t = array_delete !tab (n-1) 1 in  
      tab := t;  
      x
```

```
(* END SOLUTION *)
```

```
[16]: val dynarray_pop : 'a dynarray -> 'a = <fun>
```

```
[20]: let t = ref [|1;2;3;4|] in  
      let x = dynarray_pop t in  
      assert(x = 4 && !t = [|1;2;3|]);  
      "Valide"
```

```
[20]: - : string = "Valide"
```

Quelle est la complexité de l'ajout et de la suppression d'un élément en fonction de la longueur du tableau ?

Describe the task here!

2.3 Type efficace pour les tableaux dynamiques

Afin de résoudre les problèmes observés précédemment de complexité, on va définir un nouveau type pour les tableaux en permettant d'avoir un tableau contenant plus d'éléments que ce qui est vraiment utilisé. Ainsi, si on a un tableau de taille 10 mais qui ne contient que 9 éléments réels, on pourra en temps constant rajouter un élément. Pour cela, on va avoir besoin non seulement de gérer les éléments du tableau, mais aussi un paramètre `length` donnant la longueur logique, c'est-à-dire le nombre d'éléments significatifs du tableau.

```
[21]: type 'a dynarray = {  
      mutable elements : 'a array;  
      mutable length : int  
    }
```

```
[21]: type 'a dynarray = { mutable elements : 'a array; mutable length : int; }
```

```
[22]: let dynarray_init () = { elements = [|]|; length = 0 }
```

```
[22]: val dynarray_init : unit -> 'a dynarray = <fun>
```

```
[23]: let dynarray_of_array (tab : 'a array) : 'a dynarray =  
      (* Prend un tableau tab et renvoie le dynarray le contenant : on  
      ↪initialisera sa  
      longueur à la longueur de tab *)  
      (* BEGIN SOLUTION *)  
      { elements = tab; length = Array.length tab }  
      (* END SOLUTION *)
```

```
[23]: val dynarray_of_array : 'a array -> 'a dynarray = <fun>
```

```
[24]: assert(dynarray_of_array [|1;2|] = { elements = [|1;2|]; length = 2 });  
      "Valide"
```

```
[24]: - : string = "Valide"
```

```
[25]: let get (tab : 'a dynarray) (i : int) : 'a =  
      (* Renvoie le ième élément du tableau contenu par tab *)  
      (* BEGIN SOLUTION *)  
      tab.elements.(i)  
      (* END SOLUTION *)
```

```
[25]: val get : 'a dynarray -> int -> 'a = <fun>
```

```
[26]: let set (tab : 'a dynarray) (i : int) (x : 'a) : unit =  
      (* Place la valeur x dans le tableau tab à l'indice i *)  
      (* BEGIN SOLUTION *)  
      tab.elements.(i) <- x  
      (* END SOLUTION *)
```

```
[26]: val set : 'a dynarray -> int -> 'a -> unit = <fun>
```

```
[27]: let d = dynarray_of_array [|1;2|] in  
      assert(get d 1 = 2);  
      set d 1 3;  
      assert(d.elements = [|1;3|]);  
      "Valide"
```

```
[27]: - : string = "Valide"
```

Pour ajouter des éléments au tableau, on va regarder s'il y a de la place `t.length < Array.length t.elements` et sinon on va **doubler** sa taille. Si le tableau contenait 3 éléments, on va en allouer 6 donc 3 de plus, ce qui va permettre de faire 2 ajouts derrière.

Attention : quand le tableau est vide, on ne pourra pas doubler sa longueur, on se contente de faire un tableau de taille 1 dans ce cas.

```
[35]: let rec dynarray_append (tab : 'a dynarray) (x : 'a) : unit =  
      (* ajoute x à la fin du tableau tab *)  
      (* BEGIN SOLUTION *)  
      let n = Array.length tab.elements in  
      if n = 0 then begin
```

```

    tab.elements <- [|x|];
    tab.length <- 1
end
else begin
    if tab.length = Array.length tab.elements
    then tab.elements <- array_extend tab.elements tab.length tab.length;
    tab.elements.(tab.length) <- x;
    tab.length <- 1 + tab.length
end
(* END SOLUTION *)

```

[35]: val dynarray_append : 'a dynarray -> 'a -> unit = <fun>

```

[39]: let t = dynarray_init () in
dynarray_append t 1;
dynarray_append t 2;
dynarray_append t 3;
dynarray_append t 4;
dynarray_append t 5;
assert(Array.sub t.elements 0 5 = [|1; 2; 3; 4; 5|] && t.length = 5);
"Valide"

```

[39]: - : string = "Valide"

```

[40]: let rec dynarray_pop (tab : 'a dynarray) : 'a =
    (* supprime le dernier élément de tab et le renvoie.
    On ne supprime rien physiquement, on se contente de changer la valeur. *)
    (* BEGIN SOLUTION *)
    let x = tab.elements.(tab.length-1) in
    tab.length <- tab.length - 1;
    x
    (* END SOLUTION *)

```

[40]: val dynarray_pop : 'a dynarray -> 'a = <fun>

```

[42]: let t = { elements = [|1;2;3;4;5|]; length = 4 } in
let x = dynarray_pop t in
assert(x = 4 && t.elements = [|1;2;3;4;5|] && t.length = 3);
"Valide"

```

[42]: - : string = "Valide"

Que pensez-vous de la complexité en pire cas de append ? Pourquoi est-ce tout de même préférable

à la méthode précédente quand on effectue de nombreux ajouts ?

[]: Si maintenant on souhaite effectuer des ajouts d'éléments au milieu du tableau, ↪
↪ quelle complexité obtient-on ?