TP 4 : Mutabilité en OCaml

15 novembre 2023

1 Rappels : Traits Impératifs en OCaml

Les instructions de programmation impératives en OCaml correspondent à des objets de type `unit`. Comme en programmation impérative classique, on peut écrire en OCaml :

- des séquences d'instructions séparées par un point virgule ;
- des instructions conditionnelles ;
- des boucles `for`, répétant une instruction pour plusieurs valeurs d'une variable itérée ;
- des boucles `while`, s'exécutant tant qu'une condition est vérifiée.

On a vu que les éléments de syntaxe connus pour l'instant ne permettaient pas de modifier la valeur d'une variable en mémoire : lorsque l'on crée plusieurs déclarations successives d'une variable `v`, plusieurs "variables" différentes sont créées avec une valeur stockée en mémoire. Les affectations successives ne changent pas la valeur d'une variable `v` mais la variable à laquelle fait référence le nom `v`.

Ne pas pouvoir modifier la valeur d'une variable limite les possibilités par rapport aux langages impératifs ; en particulier, cela empêche d'utiliser convenablement les boucles `while` dont la fin de l'exécution dépend de la modification de la valeur d'une variable.

Les structures de données vues pour l'instant sont **persistantes**. on va voir comment définir des structures de données modifiables.

2 Enregistrements Modifiables et Références

On a défini dans le TP précédent les *enregistrements* (ou *produits nommés*), qui sont des types définis par la donnée de champs qui prennent une valeur d'un type donné. On pouvait par exemple déclarer un type pour les nombres complexes avec un champ pour la partie réelle et un pour la partie imaginaire :

```
type complexe = {re : float ; im : float}
```

Il est possible de définir des enregistrements avec des champs modifiables : pour cela, il suffit de rajouter le mot-clé `mutable` devant le nom du champ. Pour rendre les deux champs du type précédent modifiable, on peut utiliser l'écriture suivante :

```
type complexe = {mutable re : float ; mutable im : float}
```

Pour définir un élément du type donné, on utilise la même syntaxe que sans la mutabilité :

`let z = {re = 1. ; im = 1.}`

Pour modifier la valeur d'un champ, on utilise l'opération suivante : `z.re <- valeur`

par exemple, si l'on souhaite changer la valeur de la partie réelle de `z` pour l'incrémenter d'une unité, on utilise la commande suivante :

```
z.re <- z.re +. 1. ;;
```

Il est également possible de définir des fonctions qui agissent sur des objets de types mutables en les modifiant, et sans calculer une valeur en fonction de celles-ci.

Exercice 1

Ecrire des fonctions `conjugue` et `carre` qui prennent en entrée un complexe mutable et modifient sa valeur pour la remplacer par son conjugué et son carré respectivement.

2.1 Références

Les enregistrements sont un type structuré. Pour utiliser la mutabilité sur des données simples, non-structurées, on utilise les **références** : de façon similaire aux pointeurs en C, les références sont les données d'endroits en mémoire dont le contenu peut être modifié.

La syntaxe pour définir une référence `x` contenant la valeur `val` est la suivante :

```
let x = ref val ;;
```

Exercice 2

Taper les commandes suivantes :

```
let x = ref 5 ;;
x ;;
!x ;;
x := !x + 1 ;;
!x ;;
```

On observe plusieurs choses :

- `x` est une variable de type `int ref`, tandis que `!x` est une variable de type `int` : le `!` précédant un nom de référence sert à récupérer son contenu.
- L'opération `:=` peut être utilisée pour changer la valeur contenue dans la référence. Si l'on souhaite l'exprimer en fonction de son ancienne valeur, on notera la référence `x` à gauche et on utilisera sa valeur `!x` à droite.

On remarque la ligne `val z : int ref = {contents = 5}` après l'exécution de la première ligne. La syntaxe semble signifier qu'une référence entière correspond à un enregistrement de champ `contents` prenant une valeur entière (et de façon plus générale, une valeur de type `'a` pour une référence de type `'a ref`). C'est le cas : en réalité, les références sont des enregistrements polymorphes dont le type `'a ref` est prédéfini dans OCaml.

2.2 Egalité physique et structurelle

Exercice 3

Taper les commandes suivantes. Que peut-on dire ?

```
let v1 = ref 5 ;;
let v2 = ref 5 ;;
v1 = v2 ;;
v1 == v2 ;;

let v3 = v1 ;;

v1 = v3 ;;
v1 == v3 ;;

v1 := !v1 + 1 ;;
v1 ;;
v3 ;;
```

On remarque ici plusieurs choses.

1) Les opérateurs booléens `=` et `==` ne testent pas la même chose :

- `=` teste **l'égalité structurelle**
- `==` teste **l'égalité physique**

L'égalité structurelle est vérifiée lorsque les contenus sont identiques. En revanche, pour des types mutables, l'égalité physique est vérifiée seulement lorsque les deux variables mutables correspondent à un même espace en mémoire. Ce n'est pas le cas de `v1` et `v2` qui, s'ils ont le même contenu, sont définis indépendamment. Pour des types non-mutables, le comportement de l'opérateur `==` peut dépendre de l'implémentation.

2) Lorsque l'on déclare `let v3 = v1 ;;`, on définit une référence `v3` comme étant égale à `v1`, d'est-à-dire qu'elle pointe vers le même espace en mémoire que `v1` : ce sont des *alias*. Cela signifie quel'égalité physique est vérifiée ; cela veut également dire qu'une modification du contenu dans `v1` est également une modification du contenu dans `v3`.

Remarque. Les opérations `=` et `==` étant différentes, elles ont chacune leur négation :

- La négation de `=` est `<>`
- La négation de `==` est `!=`

Exercice 4

Ecrire une fonction `affichage_syracuse` qui prend en entrée un entier n et un seuil s et affiche la liste des termes de la suite de Syracuse à partir de n avant le premier terme inférieur au seul s (le programme aucun terme si n et inférieur à s).

3 Tableaux

Les tableaux sont un autre outil important de la programmation impérative en OCaml. Un tableau de taille n est un ensemble de n éléments, indexés entre 0 et $n-1$, tous de même type. Le type des tableaux est le type prédéfini `'a array`.

Un tableau est noté comme la liste de ses éléments entre des crochets + barres, séparés par des points-virgules. Le tableau contenant les éléments 1, 2, 5, 8 se note par exemple :

`[|1 ; 2 ; 5 ; 8 |]`

et est un objet de type `int array`.

On accède à l'élément d'indice i d'un tableau `t` avec la notation `t.(i)`. Les tableaux sont des structures mutables : on peut modifier la valeur à l'intérieur d'une case avec l'opération `<-`.

Le module `Array` contient plusieurs fonctions à utiliser sur des tableaux, ou pour en créer. On a notamment :

- La fonction `Array.length` qui prend en entrée un tableau et renvoie sa longueur ;
- La fonction `Array.make n v` qui permet de créer un tableau à n élément valant initialement tous la valeur `v` ;
- La fonction `Array.init n f` qui permet dé créer un tableau à n élément tel que l'élément d'indice i vaut `f(i)`

Exercice 5

Ecrire une fonction `maximum t` qui prend en entrée un tableau d'entiers et calcule sa valeur maximum.

Exercice 6

Ecrire une fonction `swap t i j` qui intervertit les éléments d'indices i et j dans le tableau t.

`Exercice 7`

Ecrire une fonction `somme t` qui calcule la somme de tous les éléments de t.

`Exercice 8`

Ecrire une fonction `trie` qui teste si un tableau est trié.

`Exercice 9`

Ecrire une fonction `array_to_list t` qui transforme un tableau en liste.

Exercice 10

Ecrire une fonction **pascal n** qui renvoie les lignes 0 à n du triangle de Pascal (sous la forme d'un tableau de tableaux).

3.1 Matrices

On peut utiliser la structure de tableau d'OCaml pour représenter les matrices, qui sont des tableaux en deux dimensions : ceux-ci prennent la forme de tableaux de tableaux, qui sont des éléments de type `('a array) array`, ou plus simplement `'a array array`. Chaque élément du tableau principal correspond à une ligne de la matrice.

Exemple. Si l'on souhaite représenter la matrice $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, on définit le tableau suivant :

```
let t = [| [|1;2|] ;
           [|3;4|] |]
;;
```

Pour un indice i, `t.(i)` désigne le i-ème élément du tableau principal (où les éléments sont

numérotés à partir de 0). Cet élément est un tableau correspondant à la $i+1$ ème ligne de la matrice représentée. On obtient la valeur d'indice j dans ce tableau avec `(t.(i)).(j)`.

Cet élément correspond à l'élément à la $i+1$ème ligne et à la $j+1$ème colonne de la matrice. On peut simplement le noter `t.(i).(j)`.

Comme pour les tableaux unidimensionnels, on peut directement créer une matrice de taille $n \times m$ contenant une valeur v avec la commande :

```
let m1 = Array.make_matrix n m v ;;
```

Attention

Une commande `let m1 = Array.make n (Array.make m v) ;;` peut créer un objet similaire (tableau de taille n contenant des tableaux de taille m contenant tous la valeur v). Néanmoins, dans ce cas, les lignes (c'est-à-dire les éléments du tableau principal) sont *égales physiquement* : elles correspondent au même objet en mémoire.
Par conséquent chaque modification sur l'une des lignes entraîne une modification sur les autres, de telle sorte que les n lignes sont toujours identiques.

Exemple. Avec les lignes de programme suivantes :

```
let m1 = Array.make_matrix 3 4 5 ;;

let m2 = Array.make 3 (Array.make 4 5) ;;

m1.(0) = m1.(1) ;;
m1.(0) == m1.(1) ;;
m2.(0) = m2.(1) ;;
m2.(0) == m2.(1) ;;

m1.(0).(0) <- 12 ;;
m2.(0).(0) <- 12 ;;

m1 ;;
m2 ;;
```

On voit que l'on a une égalité physique pour les tableaux avec la deuxième version que l'on a pas pour la première, même si les valeurs observées dans la matrice sont identiques dans les deux cas. Aussi, on voit qu'une modification d'un élément du tableau se propage à toutes les lignes dans la deuxième version, ce qui n'est pas le cas dans la première.

Exercice 11

Ecrire une fonction `trace m` qui prend en entrée une matrice et calcule sa trace (somme des termes diagonaux).

Exercice 12

Ecrire une fonction `addmat m1 m2` qui prend en entrée deux matrices de mêmes tailles et calcule leur somme.

Exercice 13

Ecrire une fonction `transpose m` qui transpose une matrice m donnée en entrée.

Exercice 14

Ecrire une fonction `symetrique m` qui prend en entrée une matrice carrée `m` et vérifie qu'elle est symétrique.

Exercice 15

Ecrire une fonction `multmat m1 m2` qui prend en entrée deux matrices de mêmes tailles et calcule leur produit.