Paradigme impératif en OCaml

Andrey Martinez Cruz

1 Glossaire

1. Références
   (a) Création et déférencement
   (b) Assignation de référence
2. Le type array
3. Les boucles
   (a) while
   (b) for
   (c) paradigme fonctionnel vs paradigme impératif

2 Les références et les champs mutables

2.1 Création et déférencement

Par défaut, quand on fait une liason (locale ou globale) avec le mot clé **let**, la valeur qui est lié à la liaison est immutable et de ce fait, il est impossible de changer la valeur d'une liaison.

Le mot clé **ref** peut être rajouté avec la valeur dans la liaison pour dire que la liaison est mutable et peut être changé (sans crée une nouvelle liaison).

Voici un exemple simple :

```
# let a = ref 4;;
val a : int ref = {contents = 4}
```

Les liaisons qui sont des références se comportent plus comme des pointeurs que des variables, car pour accéder à la valeur d'une référence, il faut le déférencer soit en faisant la même approche qu'avec les types produits c'est à dire **nom.champ** ou utilisé l'opérateur de déférencement qui est le suivant : ! .

En reprenant notre exemple de tout à l'heure, on a le résultat suivant :

```
# !a;;
- : int = 4
# !a = a.contents;;
: - bool = true
```

2.2 Assignation de référence

Les types de références peuvent avoir leur valeurs réassigné via l'opérateur d'assignation :=.

En reprenant l'example de tout à l'heure, voici le résultat obtenu si on veut réassigner a avec la valeur 8 :

```
# a := 8;;
- : unit = ()
# !a;;
- : int = 8;;
```

Remarque :

```
# ( := );;
- : 'a ref -> 'a -> unit = <fun>
```

3 Array

Comparé aux type *list* d' OCaml, le type *array* est quant à lui mutable et possède une taille fixe. Pour contruire un array en OCaml, il faut utiliser la syntaxe suivante :

```
[| element_1; element_2; element_3; ...; element_n |]
```

Exemple :

```
# [| "Un"; "Deux"; "Trois"; "Quatre" |];;
- : string array = [|"Un"; "Deux"; "Trois"; "Quatre"|]
```

Comparé aux listes de OCaml, on peut accéder en temps constant à un élément d'un array et même modifier l'élément d'une case de celle-ci.

Exemple :

```
# let x = [|3; 5; 7; 8; 1; 6 |];;
val x : int array = [|3; 5; 7; 8; 1; 6|]
# x.(0);;
- : int = 3
# x.(0) <- 40;;
- : unit = ()
# x;;
- : int array = [|40; 5; 7; 8; 1; 6|]
```

Remarque que **array.(index)** est un moyen pour accéder à une cellule spécifique du tableau et le symbole ← permet de modifier une cellule spécifique du tableau.

4 Boucle

Lorsqu'on on programme avec un language fonctionnel, on temps à préférer la récursion qu'aux boucles pour exécuter certaines étapes d'un algorithme plusieurs fois. Cette section aura pour but de revoir les boucles en OCaml.

4.1 while

La syntaxe du boucle while en OCaml est la suivante :

```
while expression_booleenne do
    (*Faire quelque chose ici et doit être de type unit*)
done
```

Remarque qu'aprés **done**, cela retourne un type **unit**.

Exemple :

```
# let a = ref 1;;
val a : int ref = {contents = 1}
# while !a < 4 do
    print_endline "test";
    a := !a + 1
done;;
test
test
test
- : unit = ()
```

4.2 for

La syntaxe d'une boucle for qui incrément en OCaml est la suivante :

```
for nom = valeur_depart to valeur_limite_inclusive do
    (*Faire quelque chose ici et doit être de type unit*)
done
```

Sa variante qui décrémente le compteur est la suivante :

```
for nom = valeur_depart downto valeur_limite_inclusive do
    (*Faire quelque chose ici et doit être de type unit*)
done
```

Par exemple, si on veut afficher tous les éléments d'un array, on peut utiliser l'approche suivante :

```
# let tab = [| 4; 6; 7; 9; 9|];;
val tab : int array = [|4; 6; 7; 9; 9|]
# for i = 0 to (Array.length tab - 1) do
    Printf.printf "%d\t" tab.(i);
done;
print_newline ();;
4       6       7       9       9
- : unit = ()
```

Pour l'afficher dans le sens inverse, on peut utiliser l'approche suivante :

```
# for i = (Array.length tab - 1) downto 0 do
    Printf.printf "%d\t" tab.(i);
done;
print_newline ();;
9       9       7       6       4
- : unit = ()
```

4.3 Impératif vs fonctionnel

Dû que OCaml est language principalement fonctionnel, ce qu'on peut faire avec le paradigme impératif dans ce language sont limité (comparés aux languages impératifs où c'est l'inverse). Par exemple, il n'y a pas de mot clé pour arrêté une boucle (break) ou de continuer l'itération d'une boucle en évitant d'exécuter un bloc d'instruction (continue).

Par exemple, on peut simuler le comportement d'un break de la façon suivante :

```
# let tab = [| 2; 4; 6; 12; 10|];;
val tab : int array = [|2; 4; 6; 12; 10|]
# let est_croissant arr =
 try (* Difficile de mettre un break en OCaml .... *)
 for i = 0 to (Array.length arr - 2) do
  if arr.(i) > arr.(i + 1) then
  raise Exit
 done;
 true
 with
 | Exit -> false
 val est_croissant : 'a array -> bool = <fun>
 # est_croissant tab
```

```
- : bool = false
```

En bref, à moins que le contexte soit correct pour utiliser du code impératif, il est préférable de rester la majorité du temps avec du code venant du paradigme fonctionnel.

5 Source

Pour plus de détails et la source principalement utilisé : imperatif