Programmation Fonctionnelle
Cours 9
Traits impératifs

Delia Kesner

Table de matières

Références

Une référence d'OCaml est un espace mémoire modifiable vers une valeur d'un type donné. On peut voir les références comme des tableaux de taille 1. Cela permet aussi de retrouver la notion de variable modifiable disponible dans d'autres langages de programmation. Les typage de référence suit les principes suivants:

- Si `u` est un type OCaml, alors `u ref` est le type d'une référence contenant une valeur de type `u`.
- `ref` est donc un type polymorphe : on peut créer un type `ref` de n'importe quel autre type (fonctions, listes, types sommes, ...).
  Autrement dit `ref : 'a -> 'a ref`.
- Trois opérations:
  - Allocation: `ref` représente une référence vers une case de mémoire.
  - Écriture: `:=` écrit une valeur sur la case mémoire.
  - Lecture: `!` lit la valeur contenue dans la case mémoire.

Les types des opérations I

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

Premier exemple I

```
let r = ref 42;; (* r est une reference vers un entier *)
# val r : int ref = {contents = 42}
!r;;              (* lire *)
# - : int = 42
r:= 57;;          (* ecrire *)
# - : unit = ()
!r;;
# - : int = 57
```

Effet de partage I

```
let x = ref 17;;
# val x : int ref = {contents = 17}
!x;;
# - : int = 17
let y = x;;
# val y : int ref = {contents = 17}
!y;;
# - : int = 17
x := 256;;
# - : unit = ()
!x;;
# - : int = 256
!y;;
# - : int = 256
```

Erreurs de typage I

```
let x = ref 17;;
# val x : int ref = {contents = 17}
!x+1;;
# - : int = 18
x+1;;
x+1;;
^
Error: This expression has type int ref
       but an expression was expected of type int
let y = ref 17;;
# val y : int ref = {contents = 17}
!y;;
# - : int = 17
(!x)+(!y);;
# - : int = 34
```

Erreurs de typage II

```
x+(!y);;
x+(!y);;
^
Error: This expression has type int ref
       but an expression was expected of type int
x+y;;
x+y;;
^
Error: This expression has type int ref
       but an expression was expected of type int
let z = ref ((!x)+(!y));;
# val z : int ref = {contents = 34}
!z;;
# - : int = 34
```

Références et identités

- On peut définir une référence vers une case mémoire seulement en spécifiant une valeur initiale de la case mémoire.
- L'identificateur est ensuite lié à cette case mémoire, et cette liaison ne change pas lors d'une autre affectation !
- C'est le *contenu* de la case mémoire qui change lors d'une affectation.

Distinction entre `int` et `ref int` I

```
let x = ref 17;;
# val x : int ref = {contents = 17}
let y = 42;;
# val y : int = 42

x < y;;
x < y;;
    ^
Error: This expression has type int
       but an expression was expected of type int ref

!x < y;;
# - : bool = true
```

Distinction entre `int` et `ref int` II

```
!x < !y;;
!x < !y;;
      ^
Error: This expression has type int
       but an expression was expected of type 'a ref
```

Références sur les fonctions I

```
let f = ref (function x -> x+1);;
# val f : (int -> int) ref = {contents = <fun>}

f 10;;
f 10;;
^
Error: This expression has type (int -> int) ref
       This is not a function; it cannot be applied.

(!f) 10;;
# - : int = 11
```

Références sur les fonctions II

```
f := function x -> x+x;;
# - : unit = ()

(!f) 10;;
# - : int = 20
```

Génération de différents nombres entiers I

```
let gensym =
let r = ref 0 in    (* la reference est uniquement accessible localement *)
                    (* par la fonction gensym *)
                    (* elle est initialitee a 0 lors du premier appel seulement *)
  fun () -> r := !r+1; !r;;
# val gensym : unit -> int = <fun>

gensym ();;
# - : int = 1
gensym ();;
# - : int = 2
gensym ();;
# - : int = 3
```

Génération de différents nombres entiers II

```
(* Idem, mais avec une possibilite de remise-a-zero *)
let gensym, reset =
  let r = ref 0 in
  (fun () -> r := !r+1; !r), (fun () -> r := 0);;
# val gensym : unit -> int = <fun>
val reset : unit -> unit = <fun>

gensym ();;
# - : int = 1
gensym ();;
# - : int = 2
reset ();;
# - : unit = ()
```

Génération de différents nombres entiers III

```
gensym ();;
# - : int = 1
gensym ();;
# - : int = 2
```

Références et inférence de types

- Quelques subtilités dans l'inférence de types qui sont dûes à la combinaison de références et du polymorphisme en OCaml.
- Résolue en OCaml par des variables de types qui ne peuvent être instanciées qu'une seule fois, appelées des *variables de type faible*.
- Les variables de types faibles commencent sur le symbole _.

Polymorphisme faible I

```
let x = ref [];;  (*premiere instantiation *)
# val x : '_weak1 list ref = {contents = []}

x := [3;4;5];;
# - : unit = ()

x;;
# - : int list ref = {contents = [3; 4; 5]}

x := [6;7;8;9;10];

x;;
# - : int list ref = {contents = [6; 7; 8; 9; 10]}
```

Polymorphisme faible II

```
x := ["titi"; "toto"; "tata"];; (*ne correspond pas a la premiere
    instantiation *)
x := ["titi"; "toto"; "tata"];; (*ne correspond pas a la premiere
    instantiation *)
      ^^^^^^
Error: This expression has type string
       but an expression was expected of type int
```

Polymorphisme fort I

```
let id = function x -> x;;
# val id : 'a -> 'a = <fun>

id 42;;
# - : int = 42

id "toto";;
# - : string = "toto"
```

Fonctionnel versus impératif

- Programmation fonctionnelle pure : calcul avec des valeurs (valeurs simples, listes, fonctions, etc.) non modifiables. Indépendance de l'ordre d'évaluation.
- Le résultat d'une évaluation ne dépend pas de l'état de la mémoire.
- Style de programmation élégant (penser à l'addition de deux matrices avec `List.Map2`).
- Se prête très bien à la *parallélisation* (exécution sur plusieurs machines parallèles).

Fonctionnel versus impératif

- Programmation impérative : calcul avec des variables à des valeurs modifiables.
- Forte dépendance de l'ordre dans lequel le programme est exécuté.
- La valeur d'une fonction dépend de l'état de la mémoire.
- Se prête mieux à la modélisation de systèmes qui changent leur état interne.

Stype Fonctionnel

- Presque tous les langages de programmation préconisent un certain style de programmation (fonctionnel, impératif, logique, à objet, ...)
- Très peu de langages qui sont purement et exclusivement impératifs ou fonctionnels
- OCaml: le style préféré est fonctionnel, pourtant il y a aussi des éléments de programmation impérative et à objet
- Dans ce cours on priviléria le style fonctionnel sauf quand des constructions impératives sont plus pertinentes

Construction de boucle

- Les boucles sont utiles quand il y a du code à itérer qui fait des effets de bord, au lieu de renvoyer un résultat.
- Boucles `for` pour un nombre fixe (calculé au début de la boucle) d'itérations.
  `for id = exp1 to exp2 do exp-princ done`
  `for id = exp1 downto exp2 do exp-princ done`
- Boucle `while` qui est exécutée tant que condition donnée est vraie (à utiliser avec modération! privilégier la récurrence).
  `while cond do exp done`

Premiers exemples de for I

```
for i=1 to 9 do
  print_int i
done;;
# 123456789- : unit = ()

for i = 9 downto 1 do
  print_int i
done;;
# 987654321- : unit = ()
```

Encore quelques boucles I

```
for i = 9 to 1 do
  print_int i
done;;
# - : unit = ()

let x = ref 4;;
# val x : int ref = {contents = 4}
for i = 0 to !x do (* valeurs calculees au debut de la boucle *)
  print_int i;
  x:= !x+1;
done;;
# 01234- : unit = ()
!x;;
# - : int = 9
```

Les trois styles de programmation I

```
let rec fact_rec n = match n with
  | 0 -> 1
  | n -> n*(fact_rec (n-1));;
# val fact_rec : int -> int = <fun>

fact_rec 4;;
# - : int = 24
```

Les trois styles de programmation II

```
let fact_for n =
   let j = ref 1 in
   for i= 2 to n do
       j:= !j * i
   done;
   !j;;
# val fact_for : int -> int = <fun>

fact_for 4;;
# - : int = 24
```

Les trois styles de programmation III

```
let fact_wh n=
let j = ref 1 in
let i = ref n in
   while !i > 0 do
       j:= !j * !i;
       i:= !i - 1;
   done;
   !j;;
# val fact_wh : int -> int = <fun>

fact_wh 4;;
# - : int = 24
```

Enregistrements avec des champs modifiables

- On peut déclarer le champ d'un enregistrement comme étant modifiable, en utilisant le mot clef `mutable`.
- Modification d'un champ avec la construction `x.c <- v`
- Un enregistrement peut avoir à la fois des champs modifiables et non modifiables.

Exemple d'enregistrement modifiable I

```
type point = {
  mutable x: float;
  mutable y: float
};;
# type point = { mutable x : float; mutable y : float; }
let p = {x=1.0; y=1.0};;
# val p : point = {x = 1.; y = 1.}
p.x <- 2.0;;
# - : unit = ()
p.y <- p.y +. p.x;;
# - : unit = ()
p;;
# - : point = {x = 2.; y = 3.}
```

Exemple d'enregistrement mixte I

```
type etudiant = {
  nom: string;
  mutable age: int
};;
# type etudiant = { nom : string; mutable age : int; }
let e = {nom="Garcia"; age=20};;
# val e : etudiant = {nom = "Garcia"; age = 20}
e.nom <- "Perez";;
e.nom <- "Perez";;
^^^^^^^^^^^^^^^^
Error: The record field nom is not mutable
e.age <- e.age + 2;;
# - : unit = ()
e;;
# - : etudiant = {nom = "Garcia"; age = 22}
```

Références et enregistrements avec des champs modifiables

- Une référence peut se voir simplement comme une abréviation d'un enregistrement avec un seul champ `content` qui est modifiable.
- Dit autrement, on peut définir les références à l'aide d'un champ modifiable.

Definition alternative du type ref "a la main" I

```
type 'a ref = {mutable content: 'a};;
# type 'a ref = { mutable content : 'a; }
let set x v = x.content <- v;;
# val set : 'a ref -> 'a -> unit = <fun>
let get x = x.content;;
# val get : 'a ref -> 'a = <fun>
let y = {content = 17};;
# val y : int ref = {content = 17}
set y 42;;
# - : unit = ()
get y;;
# - : int = 42
let z = y;;
# val z : int ref = {content = 42}
```

Definition alternative du type ref "a la main" II

```
get z;;
# - : int = 42
set z 50;;
# - : unit = ()
get z;;
# - : int = 50
get y;;
# - : int = 50
```

Autres types modifiables

- Les chaînes de caractès (string) sont modifiables : On peut changer le caractère à une position donnée d'une chaîne (voir le module `String` de la bibliothèque standard)
- Les tableaux
- Les tables de hachage.

Les tableaux (type `array`)

- Module `Array` dans la bibliothèque standard.
- Tableaux de longueur fixe dont les valeurs ont le même type
- Les éléments du tableau sont modifiés et lus en temps constant (contrairement aux listes)
- Création d'un tableau à une dimension : `Array.make n i` (tableau de longueur n, toutes les cases ont initialisées avec i)
- Longueur d'un tableau : `Array.length`
- Obtenir la valeur de la case d'un tableau : `a.(n)`
  Attention : la numérotation commence sur 0.
- Changer la valeur d'une case : `a.n <- x`

Premier exemples avec tableaux I

```
Array.make;;
# - : int -> 'a -> 'a array = <fun>
Array.length;;
# - : 'a array -> int = <fun>
List.length;;
# - : 'a list -> int = <fun>

let a = Array.make 5 42;;
# val a : int array = [|42; 42; 42; 42; 42|]
Array.length a;;
# - : int = 5
a.(4);;
# - : int = 42
a.(5);;
# Exception: Invalid_argument "index␣out␣of␣bounds".
```

Premier exemples avec tableaux II

```
a.(4) <- 17;;
# - : unit = ()
a.(4);;
# - : int = 17
```

Tableaux et partage I

```
let t = [|11;12;13;14;15;16|];;
# val t : int array = [|11; 12; 13; 14; 15; 16|]

t.(2) <- t.(5);;
# - : unit = ()

t;;
# - : int array = [|11; 12; 16; 14; 15; 16|]

let u = t ;;
# val u : int array = [|11; 12; 16; 14; 15; 16|]

u.(4) <- 17;;
# - : unit = ()
```

Tableaux et partage II

```
u;;
# - : int array = [|11; 12; 16; 14; 17; 16|]
t;;
# - : int array = [|11; 12; 16; 14; 17; 16|]
```

Creation partielle I

```
let f = Array.make 5;; (* tableau sans valeur initiale est une fonction *)
# val f : '_weak1 -> '_weak1 array = <fun>
let a = f 42;;
# val a : int array = [|42; 42; 42; 42; 42|]
let b = f 2;;
# val b : int array = [|2; 2; 2; 2; 2|]
let b = f "toto";;
let b = f "toto";;
          ^^^^^^
Error: This expression has type string
       but an expression was expected of type int
```

Les tableaux à deux dimensions

- Création partielle : `Array.make_matrix m n`
- Création totale : `Array.make_matrix m n i`
- Accès : `a.(i).(j)`
- Modification : `a.(i).(j) <- x`
- Voir le module `Array` de la bibliothèque standard.

Premier exemple tableaux à deux dimensions I

```
let a = Array.make_matrix 3 2 0;;
# val a : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
for i = 0 to 2 do
  for j = 0 to 1 do
    a.(i).(j) <- 3*i+j
  done
done;;
# - : unit = ()
a;;
# - : int array array = [|[|0; 1|]; [|3; 4|]; [|6; 7|]|]
let a = [|[|1; 1|]; [|2; 2|]; [|3; 3|]|];;
# val a : int array array = [|[|1; 1|]; [|2; 2|]; [|3; 3|]|]
```

Tableaux à deux dimensions et polymorphisme I

```
Array.make_matrix;;
# - : int -> int -> 'a -> 'a array array = <fun>
Array.make_matrix 3;;
# - : int -> '_weak1 -> '_weak1 array array = <fun>
Array.make_matrix 3 2;;
# - : '_weak2 -> '_weak2 array array = <fun>

let f = Array.make_matrix 3 2;;
# val f : '_weak3 -> '_weak3 array array = <fun>
let a = f 0;;
# val a : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
let b = f 6;;
# val b : int array array = [|[|6; 6|]; [|6; 6|]; [|6; 6|]|]
```

Tableaux à deux dimensions et polymorphisme II

```
let c = f "toto";;
let c = f "toto";;
          ^^^^^^

Error: This expression has type string
       but an expression was expected of type int
```

Encore tableaux et partage I

```
(* Tableau de 3 elements, donc chacun est un tableau de 2 elements *)
let a = Array.make 3 (Array.make 2 0);;
# val a : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]

for i = 0 to 2 do
  for j = 0 to 1 do
    a.(i).(j) <- 3*i+j
  done;
done;;
# - : unit = ()

a;;
# - : int array array = [|[|6; 7|]; [|6; 7|]; [|6; 7|]|]
```

Explications

Quand on utilise `Array.make 3 (Array.make 2 0)`, voici ce qui se passe :

- `Array.make 2 0` crée un tableau `[| 0; 0 |]`.
- `Array.make 3 (Array.make 2 0)` crée un tableau de 3 éléments, mais au lieu de créer un (sous)tableau distinct pour chaque position $0, 1, 2$, il partage la référence du même tableau `[| 0; 0 |]` pour les 3 positions.

Donc `a.(0)`, `a.(1)` et `a.(2)` pointent tous vers le même tableau `[| 0; 0 |]`. Toute modification sur un élément d'un sous-tableau affectera les autres sous-tableaux (car ils partagent la même référence).

Partage de références

- C'est **le** piège des structures modifiables (ou de la programmation impérative en général)
- Si une structure est modifiable alors il faut savoir si elle est physiquement partagée entre deux valeurs, ou s'il s'agit de deux copies.

Solution I

```
let a = Array.init 3 (fun _ -> Array.make 2 0);;
# val a : int array array = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]

for i = 0 to 2 do
for j = 0 to 1 do
a.(i).(j) <- 3*i+j
done;
done;;
# - : unit = ()

a;;
# - : int array array = [|[|0; 1|]; [|3; 4|]; [|6; 7|]|]
```