Programmation Fonctionnelle Avancée

Cours 6

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Plan

Corrigé commenté du TP 5

On fait un corrigé commenté du TP (~15 à 20 minutes)

Plan

Programmation impérative

Au niveau le plus bas, un ordinateur est une machine *impérative* :

- On dispose d'un tableau continu d'octets (la mémoire)
- Le processeur calcule en copiant des données de la mémoire vers les registres, puis en écrivant les résultats des registres vers la mémoire
- Le processeur réserve une région de la mémoire pour chaque périphérique (carte graphique, réseau, disques, ...). Les E/S la zone mémoire sont propagées au périphérique (cf. DMA, *Direct Memory Access*)

Pour calculer sur un ordinateur, il faut donc, à un moment, faire des effets de bords. Nous allons voir :

- Comment faire des effets de bords en OCaml (aujourd'hui)
- Comment tirer partie des deux mondes (dans la suite)

Plan

Référence

La notion la plus simple est la notion de référence représentée par le type `'a ref`. Il s'agit d'une case mémoire *modifiable* dont le contenu est de type `'a`. Les opérations sont :

- `ref e` : crée une nouvelle case dont le contenu est donné par `e`
- `!r` : extrait le contenu de la référence `r`
- `r := e` : met à jour le contenu de `r` avec la valeur de `e`

```
let x = ref 42
let () =
  Printf.printf "%d\n" !x;  (* affiche 42 *)
  x := !x + 1;
  Printf.printf "%d\n" !x   (* affiche 43 *)
```

Utilité

On peut vouloir créer une fonction `uid()` qui renvoie un entier unique (ex: créer des noms de fichiers distincts, donner des IDs unique à des données, ...)

```
let cpt = ref 0
let uid () =
    let c = !cpt in
    cpt := c + 1;
    c

let x = uid ()   (* 0 *)
let y = uid ()   (* 1 *)
let z = uid ()   (* 2 *)
```

Quel problème potentiel avec cette fonction ?

La référence est accessible, du code erroné pourrait faire:

```
let () = cpt := 0   (* réinitialisation! le compteur n'est plus unique *)
```

Peut-on faire mieux ?

Référence locale

On peut utiliser un `let in`

```
let uid =
  let cpt = ref 0 in
  fun () -> let c = !cpt in
            cpt := c + 1;
            c
```

- `cpt` est une variable locale à l'expression `fun () -> ...`
- `cpt` est visible depuis la fonction `fun () -> ...`
- L'expression renvoie une fonction qui est stockée dans la variable globale `uid`
- Une fois que la fonction est « créé » `cpt` n'est plus visible de l'extérieur

C'est un exemple simple de coopération entre programmation impérative (une référence) et fonctionnelle (on définit une fonction en plein milieu d'une expression, ici un `let … in`)

Références et ordre d'évaluation

Attention, l'ordre d'évaluation en OCaml n'est spécifié que pour certaines expressions :

- `let x = e1 in e2` : e1 est évalué avant e2
- `e1; e2` : e1 est évalué avant e2
- `e1 || e2` et `e1 && e2` : e1 est évalué avant e2
- `if e1 then e2 else e3` : e1 est évalué avant e2/e3
- `match e with p1 -> e1 | ... | pn -> en` : e est évalué avant celui des ei pour lequel pi réussit

Dans tous les autres cas, l'ordre n'est pas spécifié !

```
let x = ref 0
let a, b = (!x,  (x:= !x+1; !x)) (* a ? b ? *)
(* on suppose que l'on a encore jamais appelé uid() *)
let c = (uid ()) - (uid ())  (* 1 - 0 ou 0 - 1  ? *)
```

Ne jamais écrire un tel code ! Les résultats dépendent d'un ordre d'évaluation non spécifié, l'ordre peut changer selon les versions d'OCaml.

Plan

Enregistrements mutables


```
(* définition de type *)
type r = { l1: t1;  …; ln : tn }

(* définition de valeur *)
let r = { l1 = e1; …; ln = en }
```

Dans la définition de type, on peut qualifier un champ de `mutable` pour dire qu'on peut le mettre à jour.

```
type point = { mutable x : float; mutable y : float }

let move p i j =
  p.x <- p.x +. i;
  p.y <- p.y +. j

let p = { x = 0.0; y = 0.0 }
let () = move p 1.0 2.0
let () = Printf.printf "%f, %f\n" p.x p.y (* 1.0, 2.0 *)
```

Référence vs enregistrement mutable ?

Les références ne sont qu'un cas particulier d'enregistrement mutable

```
type 'a ref = { mutable contents : 'a  }

let ref x = { contents = x }
```

- `!x ≡ x.contents`
- `x:=e ≡ x.contents<- e`

Attention au partage


```
let x = ref 0 (* création d'une ref *)
let y = ref 0 (* création d'une autre ref *)
let z = x     (* x et z sont la même ref *)

let () =
    x := 1;
    Printf.printf "%d, %d, %d\n" !x !y !z

(* affiche 1, 0, 1 *)
```

L'utilisation des structures mutable permet d'observer le partage des objets en mémoire, chose qu'on ne pouvait pas faire en prog. fonctionnelle pure.

Nous reviendrons sur cet aspect qui est source de bug.

Plan

type 'a array

OCaml possède un type pour les tableaux de taille fixe pour des éléments de type `'a`

```
let tab = [| 1; 5; 42; -3 |]

let x = tab.(0)

let () =
  tab.(0) <- x + 1;
  Printf.printf "%d\n" tab.(0) (* affiche 2 *)
```

- Un tableau littéral est délimité par `[|` et `|]`
- L'accès au $i^{ème}$ élément (à partir de 0) est noté `tab.(i)`
- La mise à jour du $i^{ème}$ élément est notée `tab.(i) <- e`

Comme pour les listes, les tableaux sont homogènes (on ne peut pas mélanger d'éléments de plusieurs types différents dans le même tableau)

Le module `Array`

Les fonctions sur les tableaux sont regroupées dans le module `Array`. Ce module regroupe

- Des fonctions de création de tableau
- Des fonctions de copie
- Des itérateurs
- Des fonctions de tri
- Des conversions, depuis et vers les listes

Créations de tableaux et fonctions de base

- `Array.make : int -> 'a -> 'a array` : `Array.make n v` crée un tableau de taille `n` dont toutes les cases contiennent `v`
- `Array.init : int -> (int ->'a) -> 'a array` : `Array.init n f` crée le tableau `[| f 0; f 1; … ; f (n-1)|]`
- `Array.length : 'a array -> int` : renvoie la longueur du tableau
- `Array.get :'a array -> int -> 'a` : renvoie le contenu de la $i^{ème}$ case
- `Array.set :'a array -> int -> 'a -> unit` : met à jour le contenu de la $i^{ème}$ case
- `Array.copy : 'a array -> 'a array` : copie le tableau donné en argument
- `Array.append : 'a array -> 'a array -> 'a array` : renvoie la concaténation de deux tableaux

Itérateurs

Tout ceux qu'on ~~aime~~ connaît :

- `Array.iter : ('a -> unit) -> 'a array -> unit`
- `Array.map : ('a -> 'b) -> 'a array -> 'b array`
- `Array.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`
- `Array.for_all : ('a -> bool) -> 'a array -> bool`
- `Array.exists : ('a -> bool) -> 'a array -> bool`

Tri de tableau

La fonction de tri est :

```
Array.sort : ('a -> 'a -> int) -> 'a array -> unit
(* Tri en place, le tableau est modifié par le tri *)
```

C'est une différence significative d'API avec les listes, ici le tableau est modifié. Si on veut conserver le tableau original, il faut utiliser `Array.copy`

Attention au partage !

```
let tab = Array.make 3 [| 0; 0; 0 |] (* une "matrice" de taille 3x3 ? *)
let pr_tab t =
      Array.iter (fun s ->
        Array.iter (fun i -> Printf.printf "%d " i) s;
              Printf.printf "\n") t

let () = tab.(0).(0) <- 42
let () = pr_tab tab
(*
  42 0 0
  42 0 0
  42 0 0
  C'est le même tableau interne qui est utilisé 3 fois !
  On écrira plutôt :
*)
let tab2 = Array.init 3 (fun _ -> Array.make 3 0)
(* la fonction est rappelée pour chaque case, un nouveau tableau est alloué
  à chaque fois *)
```

Plan

Boucle `while`

La boucle `while` est classique. On utilise les mots clés `do`et `done` pour délimiter le corp dela boucle:

```
let res = ref 0
let i = ref 0
let () =
  while !i < 10 do
    res := !res + i;
    i := !i + 1
  done
```

Comme la boucle doit utiliser une condition qui change (sinon c'est une boucle infinie), on doit utiliser des effets de bords pour modifier les valeurs testées (ici on modifie la référence `i`)

Boucle `for`

La boucle `for` permet d'aller d'une borne à une autre par pas de 1:

```
let () =
  for i = 0 to 42 do
    Printf.printf "%d\n" i;
  done;

  for j = 100 downto 0 do
    Printf.printf "%d\n" j;
  done
```

- La variable de boucle est de type `int`
- Ce n'est pas une référence modifiable par le programmeur mais sa valeur change
- La boucle va jusqu'à la borne sup incluse

Exemple de boucle `for` et tableau


```
let average tab =
    let total = ref 0.0 in
    for i = 0 to Array.length tab - 1 do
      total := !total +. tab.(i);
    done;
    !total /. (float (Array.length tab))

(* average: float array -> float *)
```

Pas de break :(

Le langage OCaml ne possède pas les mots clés `break` et continue. 😭

On peut les simuler grâce a des exceptions. 🤔

```
exception Break
exception Continue

let () =
  try
    for i in 0 to 100 do
      try
        ... (* le code ici peut contenir raise Continue
               ou raise Break *)

      with Continue -> ()
    done
  with Break -> ()
```

C'est pour montrer que c'est possible, mais en pratique on utilise plutôt les itérateurs prédéfinis qui s'arrêtent au bon moment.

Conclusion

- On a juste vu la syntaxe des traits impératifs de bases
- On verra au fur et à mesure comment mélanger impératif et fonctionnel et quels avantages de l'un par rapport à l'autre.