Programmation Fonctionnelle
Cours 8
Traits impératifs

November 13, 2012

Table de matières

Références

Boucles

Enregistrements avec des champs modifiables

Tables de hachage

Arrays

Références

- Identificateurs en OCaml : liés à une valeur au moment de leur définition.
- Un identificateur ne peut pas être non-lié (contrairement au langage à affectation unique, comme Prolog ou Oz).
- La liaison d'un identificateur à sa valeur ne peut pas changer lors de l'exécution du programme, contrairement à des langages (principalement) impératifs comme C, Pascal, C++, etc.
- Pourtant, il y a une notion de référence et d'affectation en OCaml !

Exemples (ref1.ml)

```
let r = ref 42;;
r;;
r.contents;;
!r;;
r:= 57;;
!r;;
```

Exemples (ref2.ml)

```
let x = ref 17;;
let safe_x = x;;
x := 256;;
safe_x=x;;
```


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 lié à une case mémoire, et cette liaison ne change pas lors d'une affectation !
- C'est le contenu de la case mémoire qui change lors d'une affectation.
- Instance d'une construction plus générale (enregistrements avec des champs modifiables), voir plus tard.

- Il y a des 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 une seule fois, appelées des variables de type faible.
- Les variables de types faibles commencent sur le symbole _.

Exemples (ref3.ml)

```
(* polymorphisme faible *)

let x = ref [];;

x := [3;4;5];;

x;;

x := ["titi"; "toto"; "tata"];;
```

Exemples (ref4.ml)

```
(* Pour comparaison : polymorphisme fort *)

let id = function x -> x;;

id 42;;

id "toto";;
```


Fonctionnel vs. impératif

- Programmation fonctionnelle pure : calcul avec des valeurs (valeurs simples, listes, fonctions, etc.) non modifiables. Indépendance (à la non-terminaison près) de l'ordre d'évaluation.
- Style de programmation élégant (penser à l'addition de deux matrices avec `List.M p2`)
- Se prête très bien à la parallélisation (exécution sur plusieurs machines parallèles). Voir MapReduce développé par Google.

Fonctionnel vs. impératif

- Programmation impérative : calcul avec des variables à des valeurs modifiables.
- Forte dépendance de l'ordre dans lequel le programme est exécuté.
- Se prête mieux à la modélisation de systèmes qui changent leur état interne, et aussi pour certains algorithmes.

Fonctionnel vs. impératif

- Presque tous les langages de programmation préconisent un certain style de programmation (fonctionnel, impératif, à objet, logique, ...)
- Il y a très peu de langages qui sont purement et exclusivement impératif ou fonctionnel.
- OCaml : Le style de programmation préféré est la programmation fonctionnelle, pourtant il y a aussi les éléments de la programmation impérative.
- Conséquence pour nous : le premier choix est toujours la programmation fonctionnelle, mais il ne faut pas hésiter à utiliser des constructions impératives quand c'est pertinent.

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, ou boucle `while` qui est exécutée tant que condition donnée est vraie.

Exemples (for1.ml)

```
for i=1 to 10 do
  print_int i
done;;

let x = ref 1;;
for i = 0 to !x do
  print_int i;
  x:= !x+1
done;;
!x;;
```


Exemples (for2.ml)

```
for i = 10 to 1 do
  print_int i
done;;

for i = 10 downto 1 do
  print_int i
done;;
```


Exemples (while1.ml)

```
let x = ref 5326543;;
while !x <> 1 do
  if !x mod 2 = 0
  then x := !x/2
  else x := 3*(!x)+1;
  print_int !x;
  print_newline ()
done
```

```
open Graphics;;
open_graph " 500x500";;
exception Quit;;
let rec loop t =
  let eve = wait_next_event [Mouse_motion;Key_pressed]
  in
  if eve.keypressed
  then
    match eve.key with
      | 'b' -> set_color black; loop t
      | 'r' -> set_color red; loop t
      | 'g' -> set_color green; loop t
      | 'q' -> raise Quit
      | '0'..'9' as x -> loop (int_of_string (String.ma
      | _   -> loop t
  else begin
    fill_circle (eve.mouse_x-t/2) (eve.mouse_y-t/2) t;
    loop t
  end
in
try loop 5
with Quit -> close_graph ();;
```



```
(* variante : boucle et references *)
open Graphics;;
open_graph " 500x500";;
let quit = ref false and brush = ref 5;;
while (not !quit) do
  let eve = wait_next_event [Mouse_motion;Key_pressed]
  in
    if eve.keypressed
    then
      match eve.key with
        | 'b' -> set_color black
        | 'r' -> set_color red
        | 'g' -> set_color green
        | 'q' -> quit := true
        | '0'..'9' as x -> brush := (int_of_string (Str
        | _   -> ()
    else
      fill_circle (eve.mouse_x-(!brush)/2) (eve.mouse_y-
done;
close_graph ();;
```



```
(* encore mieux : boucle et exception *)
open Graphics;;
open_graph " 500x500";;
let brush = ref 5;;
exception Quit;;
try
  while true do
    let eve = wait_next_event [Mouse_motion;Key_pressed]
    in
      if eve.keypressed
      then
        match eve.key with
          | 'b' -> set_color black
          | 'r' -> set_color red
          | 'g' -> set_color green
          | 'q' -> raise Quit
          | '0'..'9' as x -> brush := (int_of_string (S
          | _   -> ()
      else
        fill_circle (eve.mouse_x-(!brush)/2) (eve.mouse_y-(!brush)/2) !brush;
  done
with Quit -> close_graph ();;
```


Enregistrements avec des champs modifiables

- On peut déclarer un champ d'un enregistrement comme étant modifiable, en mettant le mot clef `mut` `ble`.
- Un enregistrement peut avoir à la fois des champs modifiables et non modifiables.
- Modification d'un champ avec la construction `x.c <- v`
- Les références sont simplement une abréviation pour des enregistrements avec un seul champ du nom `content` qui est modifiable.

Exemples (mutable1.ml)

```
type point = {
  mutable x: float;
  mutable y: float
};;
let p = {x=1.0; y=1.0};;
p.x <- 2.0;;
p.y <- p.y +. p.x;;
p;;
```


Exemples (mutable2.ml)

```
(* definition du type ref a la main *)
type 'a ref = {mutable content: 'a};;
let set x v = x.content <- v;;
let get x = x.content;;
let y = {content = 17};;
set y 42;;
get y;;
```


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)
- Il y a un type de vecteurs à composants modifiables (voir le module `Array` de la bibliothèque standard)
- Tables de hachages.

Les tableaux (arrays)

- module `Array` dans la bibliothèque standard.
- Création d'un tableau à une dimension : `Array.create n i` (longueur n, toutes les cases ont initialisées avec i)
- Obtenir la valeur d'une case d'un tableau : `.(n)` Attention : la numérotation commence sur 0.
- Changer la valeur d'une case : `.n <- x`

Exemples (array1.ml)

```
let a = Array.make 5 42;;
a.(4);;
a.(5);;
a.(4) <- 17;;
a.(4)

let b = [|1; 2; 3; 4|];;
b.(1) <- b.(3);;
b;;
```


Les tableaux à deux dimensions

- Création : `Arr y.m ke_m trix m n`
- Accès : `.(i).(j)`
- Modification : `.(i).(j) <- x`
- Voir le module `Arr y` de la bibliothèque standard.

Exemples (array2.ml)

```
let a = Array.make_matrix 5 5 0;;
for i = 0 to 4 do
  for j = 0 to 4 do
    a.(i).(j) <- 5*i+j
  done
done;;
a;;
```


Exemples (array3.ml)

```
(* Attention au partage *)
let a = Array.make 5 (Array.make 5 0);;
for i = 0 to 4 do
  for j = 0 to 4 do
    a.(i).(j) <- 5*i+j
  done
done;;
a;;
```

Partage de structures

- 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 physiquement partagée entre deux valeurs, ou s'il s'agit de deux copies.

Programmation Fonctionnelle Cours 8 Traits impératifs
└Arrays

Exemples (array4.ml)

```
(* Plus de partage *)
let v = Array.make 5 0;;
let a = Array.make 5 v;;
for i = 1 to 4 do a.(i) <- Array.copy v done;;
for i = 0 to 4 do
  for j = 0 to 4 do
    a.(i).(j) <- 5*i+j
  done
done;;
```

Fonction de hachage

- Envoie les clefs vers une valeur de hachage.
- Exemple : somme des valeurs ASCII de tous les caractères de la clef modulo 1024.
- Difficulté : trouver une fonction qui
  - ne gaspille pas trop d'espace dans le tableau
  - ne produit pas trop de conflits entre clefs avec la même valeurs de hachage.
- Pour en savoir plus : voir un cours d'Algorithmique.

Tables de hachage en OCaml

- Module `H shtbl` de la bibliothèque standard.
- Polymorphe : les types des clefs et des valeurs sont des paramètres de type.
- Choix automatique d'une bonne fonction de hachage, ajustement automatique de la taille du tableau.

Exemples (hash1.ml)

```
let t = Hashtbl.create 10;;
Hashtbl.add t "john" 234;;
t;;
Hashtbl.add t "peter" 456;;
Hashtbl.find t "john";;
Hashtbl.find t "julie";;
Hashtbl.remove t "john";;
Hashtbl.find t "john";;
```


Exemples (hash2.ml)

```
(* partage *)
let t1 = Hashtbl.create 10;;
let t2 = Hashtbl.create 10;;
Hashtbl.add t1 "a" t2;;
Hashtbl.add t1 "b" t2;;
Hashtbl.find (Hashtbl.find t1 "b") "toto";;
Hashtbl.add (Hashtbl.find t1 "a") "toto" 42;;
Hashtbl.find (Hashtbl.find t1 "b") "toto";;
```