Programmation Fonctionnelle
Cours 08

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

13 novembre 2014

Traits Impératifs

Références

Références

```
val ref   : 'a -> 'a ref
val (:=)  : 'a ref -> 'a -> unit
val (!)   : 'a ref -> 'a
```

- `ref` est un type polymorphe représentant une référence vers une case de mémoire:
  - on peut créer un type `ref` de n'importe quel autre type (fonctions, listes, sommes, ...)
  - l'identificateur est lié à une case mémoire, et cette liaison ne change pas lors d'une affectation !
- `:=` écrit une valeur sur la case mémoire (affectation)
  - la valeur précédente est écrasée
- `!` lit la valeur contenue dans la case mémoire

Exemples

```
# let x = ref 42;;          (*r est une reference vers un int*)
val x : int ref = {contents = 42}

# !x;;                                          (*deferencier*)
- : int = 42

# x:=2;;                                        (*affectation*)
- : unit = ()

# !x;;
- : int = 2

# let y =x;;                                    (*effet partage*)
val y : int ref = {contents = 2}

# x:= 234;;
- : unit = ()

# !x;;
- : int = 234

# !y;;
- : int = 234
```

Exemples

```
# let x = ref 17;;                (*distinction entre int et int ref*)
val x : int ref = {contents = 17}

# let y = 17;;
val y : int = 17

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

# x+1;;
Error: This expression has type int ref
but an expression was expected of type int

# y:=y+1;;
Error: This expression has type int but an expression
was expected of type 'a ref
```

Exercice (Mémoisation)

La mémoisation est une technique d'optimisation de code consistant à réduire le temps d'exécution d'une fonction en mémorisant ses résultats d'une fois sur l'autre.

- Écrire une fonction:

  ```
  val memo: ('a -> 'b) -> ('a -> 'b)
  ```

  qui prend en entrée une fonction `f: 'a -> 'b` et donne en sortie une mémoisation de `f`.

- C'est-à-dire `memo f` dispose d'une liste `tab` contenante les paires `(input, output)` d'entrées/sorties de `f` déjà calculées; lorsqu'on évalue une application `memo f input`:
  - d'abord, on cherche dans cette liste si `input` apparait dans une des paires de `tab`, si oui on donne l'`output` correspondant (sans exécuter `f`)
  - si non, on évalué `(f input)` et le résultat `output` est renvoyé comme résultat final après avoir ajouté à `tab` la nouvelle paire `(input, output)`.

Exercice (Mémoisation)

La mémoisation est une technique d'optimisation de code consistant à réduire le temps d'exécution d'une fonction en mémorisant ses résultats d'une fois sur l'autre.

- Écrire une fonction:

  ```
  val memo: ('a -> 'b) -> ('a -> 'b)
  ```

  qui prend en entrée une fonction `f: 'a -> 'b` et donne en sortie une mémoisation de `f`.

- C'est-à-dire `memo f` dispose d'une liste `tab` contenante les paires `(input, output)` d'entrées/sorties de `f` déjà calculées; lorsqu'on évalue une application `memo f input`:
  - d'abord, on cherche dans cette liste si `input` apparait dans une des paires de `tab`, si oui on donne l'`output` correspondant (sans exécuter `f`)
  - si non, on évalué `(f input)` et le résultat `output` est renvoyé comme résultat final après avoir ajouté à `tab` la nouvelle paire `(input, output)`.

Exercice (Mémoisation)

La mémoisation est une technique d'optimisation de code consistant à réduire le temps d'exécution d'une fonction en mémorisant ses résultats d'une fois sur l'autre.

- Écrire une fonction:

  ```
  val memo:  ('a -> 'b) -> ('a -> 'b)
  ```

  qui prend en entrée une fonction `f: 'a -> 'b` et donne en sortie une mémoisation de `f`.

- C'est-à-dire `memo f` dispose d'une liste `tab` contenante les paires `(input, output)` d'entrées/sorties de `f` déjà calculées; lorsqu'on évalue une application `memo f input`:
  - d'abord, on cherche dans cette liste si `input` apparait dans une des paires de `tab`, si oui on donne l'`output` correspondant (sans exécuter `f`)
  - si non, on évalué `(f input)` et le résultat `output` est renvoyé comme résultat final après avoir ajouté à `tab` la nouvelle paire `(input, output)`.

Solution (Mémoisation)

```
1  let memo f =
2   let tab = ref [] in
3   let rec find_apply list x =
4    match list with
5     | (x',y)::_ when x=x' -> y
6     | _::list -> find_apply list x
7     | [] ->
8      let y = f x in
9      tab := (x,y):: !tab ;
10     y
11  in (fun x -> find_apply !tab x)
12
13 let rec fib x = match x with
14 | 0 | 1 -> 1
15 | x -> (fib (x-1)) + (fib (x-2))
```

Enregistrement à champs modifiables

Enregistrement à champs modifiables

```
# type personne = {
    nom:string;
    mutable age:int
  };;
type personne = { nom : string; mutable age : int; }
```

- `ref` est un cas particulier de enregistrement avec champs modifiables
- un enregistrement peut avoir à la fois des champs modifiables et non modifiables
  - les champs modifiables sont déclarés avec le mot clé `mutable`
  - lecture d'un champ (modifiable ou non) par `x.champ`
  - modification d'un champ modifiable par `x.champ <- y`
- `ref` est simplement une abréviation pour des enregistrements avec un seul champ du nom `content` qui est modifiable

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

Enregistrement à champs modifiables

```
# type personne = {
    nom:string;
    mutable age:int
  };;
type personne = { nom : string; mutable age : int; }
```

- `ref` est un cas particulier de enregistrement avec champs modifiables
- un enregistrement peut avoir à la fois des champs modifiables et non modifiables
  - les champs modifiables sont déclarés avec le mot clé `mutable`
  - lecture d'un champ (modifiable ou non) par `x.champ`
  - modification d'un champ modifiable par `x.champ <- y`
- `ref` est simplement une abréviation pour des enregistrements avec un seul champ du nom `content` qui est modifiable

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

Enregistrement à champs modifiables

```
# type personne = {
    nom:string;
    mutable age:int
  };;
type personne = { nom : string; mutable age : int; }
```

- `ref` est un cas particulier de enregistrement avec champs modifiables
- un enregistrement peut avoir à la fois des champs modifiables et non modifiables
  - les champs modifiables sont déclarés avec le mot clé `mutable`
  - lecture d'un champ (modifiable ou non) par `x.champ`
  - modification d'un champ modifiable par `x.champ <- y`
- `ref` est simplement une abréviation pour des enregistrements avec un seul champ du nom `content` qui est modifiable

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

Exercice (Listes simplement chainées)

- Mettre en ouvre la structure des listes simplement chainées en permettant une concaténation en temps constant:

```
val create : unit -> 'a list_ch
val empty : 'a list_ch
val app : 'a -> 'a list_ch -> 'a list_ch
val head : 'a list_ch -> 'a
val tail : 'a list_ch -> 'a list_ch
val concat : 'a list_ch -> 'a list_ch -> 'a list_ch
```

- Suggestion:

voir tableau...

- Quelles sont les différences avec une implantation purement fonctionnelle ?

Solution (Listes simplement chainées)

```
1   type 'a addr = Null | A of ('a ref)
2   type 'a cell = {elem : 'a; mutable next: ('a cell) addr}
3   type 'a list_ch = {mutable first : ('a cell) addr;
4                      mutable last : ('a cell) addr}
5
6   (*avant etait let empty = {first = Null; last = Null}
7   mais probleme compilation polymorfisme faible: empty: '_a list_ch*)
8   let create () = {first = Null; last = Null}
9
10  let app e list =
11   let c = {elem = e; next= list.first} in
12   {first =  A (ref c); last = list.last}
13
14  let head list = match list.first with
15  | Null -> failwith "head␣of␣empty␣list"
16  | A a -> (!a).elem
17
18  let tail list = match list.first with
19  | Null -> failwith "tail␣of␣empty␣list"
20  | A a -> {first = (!a).next; last = list.last}
21
22  let concat l1 l2 =
23   match l1.last with
24   | Null -> l2
25   | A a -> !a.next <- l2.first ; l1.last <- l2.last ; l1
```

Errata Corrige (Listes simplement chainées)

- Attention ! en cours j'avais demandé la liste vide :

```
let empty = {first = Null ; last = Null};;
```

- ceci donne un erreur de compilation (mais pas d'interprétation) car la liste vide (dans cette structure avec references) a type `'_a list_ch` :
  - le paramètre `'_a` est une variable de type faible: elle peut être instanciée qu'une seule fois, par exemple:

```
app 3 empty;;
app "Hello" empty;;
```

    donne un erreur car on instance le type de `empty` d'abord comme `int list_ch` et puis comme `string list_ch`.
  - les variables de type faible sont temporairement acceptées par l'interpréteur mais ils sont refusés par le compilateur
- la meilleure solution est de concevoir mieux l'interface de `list_ch` en remplaçant la valeur `empty` par la fonction `create: unit -> list_ch`, qui crée une liste vide à chaque fois qu'il est nécessaire.

Boucles et Tableaux

Boucles

```
for identif = expr_start to expr_end do expr done

for identif = expr_start downto expr_end do expr done

while expr_cond do expr done
```

- utiles quand il y a du code à itérer qui fait des effets de bord, au lieu de renvoyer un résultat:
  - `expr` est censé être de type `unit`, sinon warning
- boucles `for` pour un nombre fixe d'itérations, ou boucle `while` qui est exécutée tant que condition donnée est vraie.
- à utiliser avec modération, préférez la récurrence aux boucles !

Exemple (la factorielle)

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

# 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>

# 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>
```

Impératif vs fonctionnel

`f(1) = f(1)`

- dans un cadre purement fonctionnel la valeur `f(1)` dépend seulement de la définition de `f`:
  - le résultat d'une évaluation ne dépend pas du moment où l'évaluation est faite,
  - comme en mathématiques: une proposition est vraie (ou fausse) pour toujours,
  - en particulier, l'égalité ci-dessus est toujours vraie

- en présence de primitives impératives, la valeur de `f(1)` peut dépendre de l'état de la mémoire, p.ex.:

```
let x = ref 1
let f y = (x:=!x+y); !x
```

  - la valeur de `f(1)` mute dans le temps,
  - en particulier, l'égalité ci-dessus est toujours fausse.

Impératif vs fonctionnel

`f(1) = f(1)`

- dans un cadre purement fonctionnel la valeur `f(1)` dépend seulement de la définition de `f`:
  - le résultat d'une évaluation ne dépend pas du moment où l'évaluation est faite,
  - comme en mathématiques: une proposition est vraie (ou fausse) pour toujours,
  - en particulier, l'égalité ci-dessus est toujours vraie

- en présence de primitives impératives, la valeur de `f(1)` peut dépendre de l'état de la mémoire, p.ex.:

```
let x = ref 1
let f y = (x:=!x+y); !x
```

  - la valeur de `f(1)` mute dans le temps,
  - en particulier, l'égalité ci-dessus est toujours fausse.

Impératif vs fonctionnel

`f(1) = f(1)`

- dans un cadre purement fonctionnel la valeur `f(1)` dépend seulement de la définition de `f`:
  - le résultat d'une évaluation ne dépend pas du moment où l'évaluation est faite,
  - comme en mathématiques: une proposition est vraie (ou fausse) pour toujours,
  - en particulier, l'égalité ci-dessus est toujours vraie

- en présence de primitives impératives, la valeur de `f(1)` peut dépendre de l'état de la mémoire, p.ex.:

```
let x = ref 1
let f y = (x:=!x+y); !x
```

  - la valeur de `f(1)` mute dans le temps,
  - en particulier, l'égalité ci-dessus est toujours fausse.

Impératif vs fonctionnel

`f(1) = f(1)`

- dans un cadre purement fonctionnel la valeur `f(1)` dépend seulement de la définition de `f`:
  - le résultat d'une évaluation ne dépend pas du moment où l'évaluation est faite,
  - comme en mathématiques: une proposition est vraie (ou fausse) pour toujours,
  - en particulier, l'égalité ci-dessus est toujours vraie

- en présence de primitives impératives, la valeur de `f(1)` peut dépendre de l'état de la mémoire, p.ex.:

```
let x = ref 1
let f y = (x:=!x+y); !x
```

  - la valeur de `f(1)` mute dans le temps,
  - en particulier, l'égalité ci-dessus est toujours fausse.

Impératif vs fonctionnel

`f(1) = f(1)`

- dans un cadre purement fonctionnel la valeur `f(1)` dépend seulement de la définition de `f`:
  - le résultat d'une évaluation ne dépend pas du moment où l'évaluation est faite,
  - comme en mathématiques: une proposition est vraie (ou fausse) pour toujours,
  - en particulier, l'égalité ci-dessus est toujours vraie

- en présence de primitives impératives, la valeur de `f(1)` peut dépendre de l'état de la mémoire, p.ex.:

```
let x = ref 1
let f y = (x:=!x+y); !x
```

  - la valeur de `f(1)` mute dans le temps,
  - en particulier, l'égalité ci-dessus est toujours fausse.

Style fonctionnel

- style de programmation élégant
  (penser au tri rapide mis en ouvre au début du cours)
- se prête très bien à la parallélisation – exécution sur plusieurs machines parallèles
  (voir `MapReduce` développé par Google).
- se prête aussi bien à la vérification automatique de la correction des programmes
  (voir l'assistant de preuve `Coq`)

Style fonctionnel

- 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 (et à objet).
- 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.

Tableaux (arrays)

```
# let t = [|1;3;6|];;
val t : int array = [|1; 3; 6|]

# t.(2);;
- : int = 6

# t.(2)<- 9;;
- : unit = ()
```

- tableau de longueur fixe de valeurs du même type
- les éléments du tableau peuvent être modifiés et lus en temps constant
- le module `Array` contient plusieurs fonctions
  - `Array.make: int -> 'a -> 'a array`
  - `Array.length: 'a array -> int`
  - `Array.make_matrix :int ->int ->'a ->'a array array`

Tableaux (Exemples)

```
# let t = Array.make 6 'a';;
val t : char array = [|'a'; 'a'; 'a'; 'a'; 'a'; 'a'|]

# t.(2)<-'b';;  (*array est un type modifiable : trait imperatif*)
- : unit = ()

# t;;
- : char array = [|'a'; 'a'; 'b'; 'a'; 'a'; 'a'|]

# let u = t;;
val u : char array = [|'a'; 'a'; 'b'; 'a'; 'a'; 'a'|]

# t.(4)<- 'c';;
- : unit = ()

# t;;
- : char array = [|'a'; 'a'; 'b'; 'a'; 'c'; 'a'|]

# u;;                          (*attention au partage de references*)
- : char array = [|'a'; 'a'; 'b'; 'a'; 'c'; 'a'|]
```

Exercice (Crible)

- Écrire un programme qui mets en ouvre le crible d'Ératosthène.
- depuis Wikipedia:
  "il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers."

Solution (Crible)

```
1   (*cree une liste contenant les elements d'un tableau tab*)
2   (*qui satisfont une condition f*)
3   let filter tab f =
4    let result = ref [] in
5    for i = (Array.length tab -1) downto 0 do
6     let e = tab.(i) in
7     if f e then result := e :: !result
8     else ()
9    done ;
10   !result
11
12  (*cree la liste des nombres premieres plus petits *)
13  (* au egal a n*)
14  let crible n =
15   let tab = Array.init (n-1) (fun x->x+2) in
16   for i = 0 to (n-2) do
17    let p = tab.(i) in
18    if p>0 then
19     for j=i+1 to (n-2) do
20      let q = tab.(j) in
21      if (q > 0 && q mod p = 0) then tab.(j)<-0
22      else ()
23     done
24    else ()
25   done ;
26   filter tab (fun x-> x>1)
```