Programmation Fonctionnelle

Emmanuel Chailloux

29 juin 2021

Plan

Plan du cours 4:

- Appels récursifs : suite de Fibonacci
- Arbres binaires
  - fonctions `mem` et `size`
  - de syntaxe abstraite (AST)
  - de recherche

Appels arborescents : suite de Fibonacci

La suite de *Fibonacci* est définie comme :

$$\begin{cases} fib(0) & = & 1 \\ fib(1) & = & 1 \\ fib(n) & = & fib(n-1)\ \ fib(n-2) \end{cases}$$

La fonction suivante calcule le n-ième terme de cette suite :

```
1  let rec fib (n:int) : int  =
2  match n with
3  | 0 | 1 -> 1
4  | _ -> let a = fib(n-1) in
5           let b = fib(n-2) in
6             a + b
```

`fib` : trace des appels

```
 # #trace fib;;
fib is now traced.
# fib 4;;
fib <-- 4
  fib <-- 3
    fib <-- 2
      fib <-- 1
      fib --> 1
      fib <-- 0
      fib --> 1
    fib --> 2
    fib <-- 1
    fib --> 1
  fib --> 3
  fib <-- 2
    fib <-- 1
    fib --> 1
    fib <-- 0
    fib --> 1
  fib --> 2
fib --> 5
- : int = 5
```

$fib(4)$ appelle $fib(3) + fib(2)$
$fib(5)$ appelle $fib(4) + fib(3)$
donne 8
$fib(6)$ appelle $fib(5) + fib(4)$
donne 13

l'imbrication des appels de $fib(n)$ est n.
Il y aura au plus n cadres d'appel empilés sur la pile d'évaluation.

Arbres binaires (1)

Définition de type:

A la manière des listes, les structures d'arbres binaires étiquetés peuvent se définir par 2 constructeurs : un pour un arbre vide, et un second qui prend une étiquette et deux arbres, et construit le nœud d'étiquette avec un fils gauche et un fils droit.

En OCaml on définit alors un type somme (union discriminante) avec deux constructeurs : aura le type suivant :

```
1  type 'a btree =
2      Empty
3    | Node of 'a * ('a btree) * ('a btree)
```

- `Empty` est un constructeur constant
- `Node` est un constructeur paramétré par un triplet : (étiquette, fils gauche, fils droit)

comme les listes, un `btree` est homogène : il ne contient que des étiquettes d'un même type.

Arbres binaires (2)

Construction:

Un arbre est une expression du type `btree`, dans l'exemple un `int btree`, c'est-à-d-ire un arbre binaire étiqueté par des entiers :

```
# let n2 = Node (2, Empty, Empty)
  and n4 =  Node(4, Empty, Empty) ;;
val n2 : int btree = Node (2, Empty, Empty)
val n4 : int btree = Node (4, Empty, Empty)
# let n3 = Node(3,Empty,n4) ;;;;
val n3 : int btree = Node (3, Empty, Node (4, Empty, Empty))
# let n1 =  Node (1,n2,n3) ;;
val n1 : int btree =
Node (1, Node (2, Empty, Empty), Node (3, Empty, Node (4, Empty, Empty)))
```

Arbres binaires (3)

Accès: :

L'accès aux éléments d'un nœud d'un arbre s'effectue par filtrage :

```
#  let combien_de_fils_directs bt = match bt with
  | Empty -> 0
  | Node(_,Empty, Empty) -> 0
  | Node(_,Empty, _) -> 1
  | Node(_,_, Empty) -> 1
  | _ -> 2
val combien_de_fils_directs : 'a btree -> int = <fun>

# combien_de_fils_directs Empty;;
- : int = 0
# List.map combien_de_fils_directs  [n1;n2;n3;n4] ;;
- : int list = [2; 0; 1; 0]
```

```
# type 'a btree2 = E | N of {etiq:'a; fg: 'a btree2; fd:'a btree2} ;;
```

`mem` : appartenance d'un élément à un arbre binaire (1)

Filtrage et récurrence:

fonctionne pour toutes les arbres binaires

4 cas pour `mem z bt` :

- si l'arbre `bt` est vide, alors `z` n'appartient pas à l'arbre ;
- si l'arbre `bt` est de la forme `Node(e,bt1,bt2)` alors, il y a 3 possibilités :
  - l'éqiquette `e` du nœud est égale à `z` et donc `z` appartient à `bt`
  - `z` appartient à `bt1`
  - `z` appartient à `bt2`

$$\begin{cases} (mem\ z\ Empty) & = & false \\ (mem\ z\ (Node(y, bt1, bt2)) & = & (z == y)\ || \\ & & (mem\ z\ bt1)\ || \\ & & (mem\ z\ bt2) \end{cases}$$

mem : appartenance d'un élément à un arbre binaire (2)

Définition:

```
# let rec mem (z:'a) (bt:'a btree) : bool = match bt with
  | Empty -> false
  | Node(y, bt1, bt2) -> (z=y) || (mem z bt1) || (mem z bt2)        ;;
val mem : 'a -> 'a btree -> bool = <fun>

# mem 1 n1 ;;
- : bool = true
# mem 1 n2 ;;
- : bool = false
# mem 1 Empty ;;
- : bool = false
# mem 3  n1 ;;
- : bool = true
```

avec $n1$:

```
  Node (1, Node (2, Empty, Empty), Node (3, Empty, Node (4, Empty, Empty)))
```

`size` : nombre de nœuds d'un arbre binaire (1)

Définition 1:

analogue de la fonction `List.length`

$$\begin{cases} (size\ Empty) & = & 0 \\ (size\ (Node, x, bt1, bt2)) & = & 1 + (size\ bt1) + (size\ bt2) \end{cases}$$

que l'on traduit en :

```
1  # let rec size (bt:'a btree) : int =
2     match bt with
3    | Empty -> 0
4    | Node(_, bt1, bt2) -> 1 + (size bt1) + (size bt2) ;;
5  val size : 'a btree -> int = <fun>
6
7  # List.map size [n1; n2; n3; n4; Empty]
8    ;;
9  - : int list = [4; 1; 2; 1; 0]
```

`size` : nombre de nœuds d'un arbre binaire (2)

Définition 2: : récursion terminale

La difficulté provient du calcul de `1 + size(bt1) + size(bt2)`, il devient alors nécessaire d'avoir d'une part :

- un accumulateur pour le comptage
- mais aussi un accumulateur des sous-arbres restant à traiter

alors on obtient :

```
1  # let rec size_aux (bt:'a btree) (bts:('a btree) list) (r:↩
       int) =
2     match bt with
3     | Node (_,bt1,bt2) -> size_aux bt1 (bt2::bts) (r+1)
4     | Empty -> ( match bts with
5                  | [] -> r
6                  | bt::bts -> size_aux bt bts r
7                ) ;;
8  val size_aux : 'a btree -> 'a btree list -> int -> int = <↩
       fun>
9  # let size (bt:'a btree) : int = size_aux bt [] 0 ;;
10 val size : 'a btree -> int = <fun>
11 # List.map size [n1; n2; n3; n4; Empty]  ;;
12 - : int list = [4; 1; 2; 1; 0]
```

```
TRACE (size_aux) :

aux n1 [] 0 ->
aux n2 [n3] 1 ->
aux Empty [Empty;n3] 2
aux Empty [n3] 2 ->
aux n3 [] 2 ->
aux Empty [n4] 3 ->
aux n4 [] 3 ->
aux Empty [Empty] 4 ->
aux Empty [] 4 -> = 4
```

size : nombre de œuds d'un arbre binaire (3)

Définition 3: : récursion terminale

L'idée est de simplifier le nombre d'arguments de la size_aux en intégrant le premier arbre à traiter bt dans la liste des arbres à traiter.

```
# let size (bt:'a btree) : int =
    let rec size_aux (bts:('a btree) list) (r:int) =
    match bts with [] -> r
    | (Empty::bts) -> (size_aux bts r)
    | ((Node(x,bt1,bt2))::bts) -> (size_aux (bt1::bt2::bts) (r+1)) in
      (size_aux [bt] 0) ;;
val size : 'a btree -> int = <fun>
# List.map size [n1; n2; n3; n4; Empty]  ;;
- : int list = [4; 1; 2; 1; 0]
```

La liste bts de la fonction size_aux correspond à une pile dans laquelle on ajoute les éléments à traiter. Cela correspond à la pile des appels d'une fonction récursive (voir fib). La différence vient que cette liste est allouée dans le tas où un récupérateur automatique de mémoire sévit, alors que les appels récursifs s'empilent dans la pile d'exécution.

height : hauteur d'un arbre

Les algorithmes sur les arbres peuvent être dépendants du nombre de nœuds ou de la hauteur de l'arbre.

$$\begin{cases} (heightEmpty) & = & 0 \\ (height(Node(x, bt1, bt2))) & = & 1 + (max(height\ bt1)(height\ bt2)) \end{cases}$$

```
1  # let rec height (bt : 'a btree) : int = match bt with
2    | Empty -> 0
3    | Node (_,bt1,bt2) -> 1 + max (height bt1) (height bt2) ;;
4  val height : 'a btree -> int = <fun>
5  # height n1;;
6  - : int = 3
```

search : recherche dans un arbre (1)

Définition:

on cherche dans un arbre une (ou des) valeur vérifiant une certain propriété :
La signature de la fonction `search` est : `(p:'a -> bool) -> (bt:'a btree) :'a`.
3 cas dans le processus de recherche :

- si l'arbre est vide, la recherche échoue
- sinon l'arbre est de la forme `Node(x,bt1,bt2)`
  - si (p x), alors la valeur de la fonction est x
  - sinon il faut chercher dans `bt1`, puis dans `bt2` si besoin

Se pose la question de comment savoir si la recherche dans `bt1` a permis de trouver l'élément, et s'il faut la poursuivre sur `bt2`.

Encore une fois on peut utiliser le type `'a option` ou déclencher une exception quand on l'a trouvée.

search : recherche dans un arbre (2)

Implantation: : avec `option`

Rappel: type `option`

```
1    type 'a option = None | Some of 'a
```

```
1  # let rec search (p: 'a -> bool) (bt:'a btree) : 'a option = match bt with
2      | Empty -> None
3      | Node (x,bt1,bt2) ->
4        if (p x) then Some x
5        else  let r = search p bt1 in
6          if r = None then search p bt2 else r
7  val search : ('a -> bool) -> 'a btree -> 'a option = <fun>
8
9  # List.map ( search (fun x -> x mod 2 = 0)) [n1;n2;n3;n4;Empty];;
10 - : int option list = [Some 2; Some 2; Some 4; Some 4; None]
```

search : recherche dans un arbre (3)

Implantation: : avec exceptions

```
# let rec search (p: 'a -> bool) (bt:'a btree) : 'a  = match bt with
    | Empty -> raise Not_found
    | Node (x,bt1,bt2) ->
      if (p x) then x
      else
      try  search p bt1
      with Not_found -> search p bt2
val search : ('a -> bool) -> 'a btree -> 'a = <fun>

# List.map ( search (fun x -> x mod 2 = 0)) [n1;n2;n3;n4;Empty];;;;
Exception: Not_found.
# List.map ( search (fun x -> x mod 2 = 0))  [n1;n2;n3;n4];;
- : int list = [2; 2; 4; 4]
```

search : recherche dans un arbre (4)

Implantation: avec liste d'attente

```
# let search (p:'a -> bool) (bt:'a btree) : 'a =
    let rec loop (bts:'a btree list) =
    match bts with
    | [] -> raise Not_found
    | (Empty::bts) -> (loop bts)
    | (Node(x, bt1, bt2))::bts ->
          if (p x) then x
          else (loop (bt1::bt2::bts))
    in
    (loop [bt]) ;;
val search : ('a -> bool) -> 'a btree -> 'a = <fun>

# List.map ( search (fun x -> x mod 2 = 0)) [n1;n2;n3;n4;Empty];;
Exception: Not_found.
# List.map ( search (fun x -> x mod 2 = 0)) [n1;n2;n3;n4];;
- : int list = [2; 2; 4; 4]
```

mem : appartenance d'un élément à un arbre binaire (3)

retour sur `mem`**:**

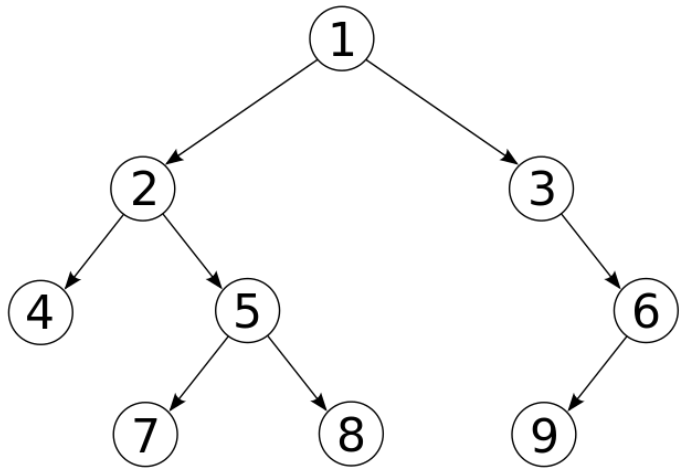
en utilisant `search`**:** :

```
let mem (x:'a) (bt:'a btree) : bool =
  try let e =  search (fun y -> x = y) bt in true
  with Not_found -> false ;;
val mem : 'a -> 'a btree -> bool = <fun>

# List.map (mem 2) [n1;n2;n3;n4;Empty];;
- : bool list = [true; true; false; false; false]
```

Arbres binaires : parcours (1)

Différents parcours:



Dans cet arbre binaire, [1]

- parcours en profondeur
  - préfixe : étiquette puis fils_gauche puis fils_droit
    1, 2, 4, 5, 7, 8, 3, 6, 9
  - postfixe : fils_gauche puis fils_droit puis étiquette
    4, 7, 8, 5, 2, 9, 6, 3, 1
  - infixe : fils_gauche puis étiquette puis fils_droit
    4, 2, 7, 5, 8, 1, 3, 9, 6
- parcours en largeur : 1, 2, 3, 4, 5, 6, 7, 8, 9

1. wikipédia

Arbres binaires : parcours (2)

en largeur (ou bfs pour breadth first search):

utilisation d'une file d'attente à la manière de `size` en récursif terminal. Il faut conserver les sous-arbres restant à traiter.

Mais il ne faut pas utiliser une *pile* (comme dans size) mais une file d'attente (où le premier entré est bien le premier sorti) (voir cours 3 et prochain transparent).

module Fqueue : fichier fqueue.ml

```
1  type 'a t = { debut : 'a list ; fin : 'a list }
2
3  exception Empty_queue
4
5  let create () = { debut = []; fin = []  }
6  let is_empty q = q.fin = [] && q.debut = []
7
8  let rec top_pop  q =
9    match q.debut with
10   | [] -> begin
11       match List.rev q.fin with
12       | [] -> raise Empty_queue
13       | x::xs -> x, { debut = xs ; fin = [] }
14     end
15   | [x] -> x, { debut = List.rev q.fin; fin = [] }
16   | x::xs -> x, { q with debut = xs }
17
18 let top q = let (t,_) = top_pop q in t
19 let pop q = let (_,p) = top_pop q in p
20 let push elem q = { debut = q.debut ; fin = elem::q.fin }
21 let add = push
22 let to_list q = q.debut @ List.rev q.fin
23 let from_list l = {debut = l ; fin = [] }
```

Arbres binaires : parcours (3)

en largeur en utilisant une file d'attente:

début du fichier `main.ml`

```
1  open Fqueue
2
3  type 'a btree =
4    Empty
5    | Node of 'a * ('a btree) * ('a btree)
6
7  let bfsearch (p:'a -> bool) (bt:'a btree) : 'a =
8    let rec loop (bts:('a btree) t) =
9      if (is_empty bts) then raise Not_found
10     else (
11       match (top bts) with
12       | Empty -> (loop (pop bts))
13       | Node(x,bt1,bt2) -> (
14          if (p x) then x
15          else (loop (add bt2 (add bt1 (pop bts))))
16       )
17     ) in
18       (loop (add bt (create()))) ;;
19 (* val bfsearch : ('a -> bool) -> 'a btree -> 'a = <fun> *)
20
21   (* ... *)
```

Arbres binaires : parcours (4)

suite du fichier main.ml: :

```
1
2   (* ... *)
3
4   let n1 =
5     let f9 = Node(9,Empty,Empty) in
6     let f8 = Node(8,Empty,Empty) in
7     let f7 = Node(7,Empty,Empty) in
8     let f4 = Node(4,Empty,Empty) in
9     let n6 = Node(6,f9,Empty) in
10    let n5 = Node(5,f7,f8) in
11    let n3 = Node(3,Empty,n6) in
12    let n2 = Node(2,f4,n5) in
13      Node(1,n2,n3);;
14
15  let est_multiple n x = let _ = print_int x in
16    let _ = print_string " " in x mod n = 0
17
18  let main () =  bfsearch  (est_multiple 8) n1
19
20  main()
```

Arbres binaires : parcours (5)

```
% ocamlc -c -i fqueue.ml
type 'a t = { debut : 'a list; fin : 'a list; }
exception Empty_queue
val create : unit -> 'a t
val is_empty : 'a t -> bool
val top_pop : 'a t -> 'a * 'a t
val pop : 'a t -> 'a
val top : 'a t -> 'a t
val push : 'a -> 'a t -> 'a t
val add : 'a -> 'a t -> 'a t
val to_list : 'a t -> 'a list
val from_list : 'a list -> 'a t
$  ocamlc -o main.exe fqueue.cmo main.ml
$ ./main.exe
1 2 3 4 5 6 7 8
$ ocamlopt -c fqueue.ml
$ ocamlopt -o main.exe fqueue.cmx main.ml
$ ./main.exe
1 2 3 4 5 6 7 8
```

Exemple : un évaluateur d'expressions (1)

On cherche à définir un mini-langage d'expressions et son évaluateur :

```
expr ::= entier | booléen
       | expr + expr
       | expr == expr
       | if expr then expr else expr
```

pour cela on définit le type suivant :

```
1  type expr =
2    I of int
3  | B of bool
4  | Add of expr * expr
5  | Eq of expr * expr
6  | If of expr * expr * expr ;;
```

```
1  # type resultat =
2      RI of int
3    | RB of bool ;;
```

Exemple : un évaluateur d'expressions (2)

- primitives d'addition et d'égalité :

```
# let add e1 e2 =  match (e1,e2) with
    RI i1, RI i2 -> RI(i1+i2)
  | _ -> failwith "add";;
val add : resultat -> resultat -> resultat = <fun>
# let eq e1 e2 =  match (e1,e2) with
    RI i1, RI i2 -> RB (i1 = i2)
  | RB b1, RB b2 -> RB (b1 = b2)
  | _ -> failwith "eq"
val eq : resultat -> resultat -> resultat = <fun>
```

- l'interprète des expressions :

```
# let rec eval e = match e with
    I i -> RI i
  | B b -> RB b
  | Add (e1,e2) -> add  (eval e1) (eval e2)
  | Eq (e1,e2) ->  eq (eval e1) (eval e2)
  | If (e1, e2, e3) ->
     (match (eval e1) with
        RB b -> if b then eval e2 else eval e3
        | _ -> failwith "If" ) ;;
val eval : expr -> resultat = <fun>
```

Exemple : un évaluateur d'expressions (3)

des évaluation:

```
1  (* 3 + (if true then 10 else 20) *)
2  # let e1 = Add (I 3, If (B true, I 10, I 20));;
3  val e1 : expr = Add (I 3, If (B true, I 10, I 20))
4
5  # eval e1;;
6  - : resultat = RI 13
7
8  (* 3 + true *)
9  # let e2 = Add (I 3, B true);;
10 val e2 : expr = Add (I 3, B true)
11
12 # eval e2;;
13 Exception: Failure "add".
```

Exemple : un évaluateur d'expressions (4)

Ajout des variables : Letin et Var: :

- modification du type

```
#  type expr = I of int | B of bool | Add of expr * expr
    | Eq of expr * expr | If of expr * expr * expr
    | Var of string | Letin of string * expr * expr
```

- modification de l'évaluateur

```
# let rec eval env e = match e with
    I i -> RI i
  | B b -> RB b
  | Add (e1,e2) -> add  (eval env  e1) (eval env e2)
  | Eq (e1,e2) -> eq (eval env e1) (eval env e2)
  | If (e1, e2, e3) -> (match (eval env e1) with RB b -> if b then eval↩
       env e2 else eval env e3
                                                  | _ -> failwith "If")
  | Var s -> List.assoc s env
  | Letin (s,e1,e2) -> let v1 = eval env e1 in eval ((s,v1)::env) e2  ↩
      ;;
val eval : (string * resultat) list -> expr -> resultat = <fun>
```

Exemple : un évaluateur d'expressions (5)

des évaluation:

```
1  (* 3 + (let x = 3 + 4 in if x = 7 then 20 else 100) *)
2  # let e1 = Add(I 3, Letin ("x",(Add (I 3, I 4)), If ((Eq(Var "x", I 7), I 20,↩
       I 100)))) ;;
3  val e1 : expr =
4    Add (I 3, Letin ("x", Add (I 3, I 4), If (Eq (Var "x", I 7), I 20, I 100)))
5
6  # eval [] e1;;
7    - : resultat = RI 23
8
9  # let e2 = Add(I 3, Letin ("x",(Add (I 3, I 4)), If ((Eq(Var "y", I 7), I 20,↩
       I 100)))) ;;
10 val e2 : expr =
11   Add (I 3, Letin ("x", Add (I 3, I 4), If (Eq (Var "y", I 7), I 20, I 100)))
12 # eval [] e2;;
13   Exception: Not_found.
14 # eval [("y", I 7)] e2 ;;
15 - : resultat = RI 23
```

Arbres binaires de recherche (1)

Un arbre binaire de recherche `t` est un arbre binaire dans lequel tout sous-arbre `u` de `t` est :

- soit vide
- soit de la forme `Node(e,g,d)` et alors : pour toute étiquette `a` de `g`, $a < e$ et pour toutes étiquettes `b` de `d` , $e \leq b$; et `g` et `d` sont des arbres binaires de recherche.

Exemple:

```
1  let f1 = Node(1,Empty,Empty)
2  let f3 = Node(3,Empty,Empty)
3  let f8 = Node(8,Empty,Empty)
4  let n4 = Node(4,f3,Empty)
5  let n7 = Node(7,Empty,f8)
6  let n6 = Node(6,Empty,n7)
7  let n2 = Node(2,f1,n4)
8  let n5 = Node(5,n2,n6)
```

Arbres binaires de recherche (2)

Recherche (rapide) dans un ABR:

```
let rec mem (x:'a) (bt:'a btree) : bool = match bt with
  | Empty -> false
  | Node(e,bt1,bt2) ->
    if x = e then true
    else
     if x < e then mem x bt1
     else mem x bt2
val mem : 'a -> 'a btree -> bool = <fun>

# mem 8 n4;;
- : bool = false

# mem 8 n5;;
- : bool = true
```

Arbres binaires de recherche (3)

Parcours infixe:

```
# let rec to_list (bt: 'a btree) : 'a list = match bt with
    | Empty -> []
    | Node (e,bt1,bt2) -> let l1 = to_list bt1 in l1 @ (e::(to_list bt2))
 val to_list : 'a btree -> 'a list = <fun>

# to_list n5;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```