Programmation Fonctionnelle Avancée
Structures de données fonctionnelles efficaces

Ralf Treinen

Université de Paris
UFR Informatique
Institut de Recherche en Informatique Fondamentale
letouzey@irif.fr

16 février 2022

Plan du cours : structures de données fonctionnelles efficaces

- Structures de données persistantes et immuables
- Queues et Dequeues
- Arbres Red-Black
- Exemples dans la librairie standard : Set et Map

Structures des données *persistantes* et *immuables*

- *Structure de donnée persistante* : Structure de donnée qui, lors d'une opération, préserve les versions précédentes.
- *Structure de donnée immuable* : Une valeur créée ne peut plus être modifiée.
  En OCaml, un tel type doit être défini sans référence, enregistrement avec champs mutables, table de hachage, array, etc.
- Les structures immuables sont persistantes. Et le contraire ?
- Est-ce que les structures immuables peuvent être efficaces ?
- NB : utilisez plutôt “immuable” que l'anglicisme “immutable”.

Pourquoi est-ce important ?

- La persistance est nécessaire lors de calculs *spéculatifs*, i.e. pouvant nécessiter de revenir à une version antérieure des données (exemple : un interpréteur de Prolog).
- Les structures immuables permettent un *partage* de sous-structures.
- Les structures immuables permettent un *raisonnement équationnel*.
- Les structures immuables sont faciles à utiliser en présence de plusieurs *threads*.

Exemples (list1.ml)

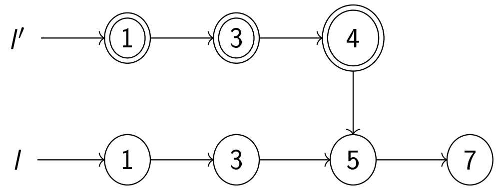
```
(* une insertion dans l'ordre, sans doublon *)
let rec insert x l = match l with
  | [] -> [x]
  | a::_ when x = a -> l
  | a::_ when x < a -> x::l
  | a::r -> a :: insert x r ;;

let l = [1;3;5;7];;
let l' = insert 4 l;;
(* l reste intacte *)
l;;
(* l et l' partagent la même sous-liste [5;7] *)
List.(tl (tl l) == tl (tl (tl l')));;
(* Exemple avec moins de partage que possible *)
l == insert 3 l;;
```


Ce qui se passe en mémoire

On simule la modification en place par

- une copie de la structure jusqu'à la modification
- l'introduction d'un nœud contenant la modification
- le partage du reste de la structure



Le prix à payer pour la persistance :

- une *occupation en mémoire* accrue,
- l'introduction d'un ramasse-miettes (garbage collector) pour récupérer la mémoire non utilisée.

Exemples (list2.ml)

```
(* Une insertion avec plus de partage *)
let rec insert_opt x l = match l with
  | [] -> [x]
  | a::_ when x = a -> l
  | a::_ when x < a -> x::l
  | a::r ->
      let r' = insert_opt x r in
      if r' == r then l else a::r'

let l = [1;3;5;7];;
l == insert_opt 3 l;;
```


Prouver des propriétés équationnellement

Si on n'utilise pas de structures de données mutables (enregistrements, tableaux, etc.), on peut prouver des propriétés de programmes par simple application du raisonnement équationnel :

- remplacement d'égaux par égaux
- induction bien fondée ou structurelle (i.e. preuve par récurrence)

Exemple

```
let rec append l1 l2 =
 match l1 with
  | [] -> l2
  | a::r -> a::(append r l2)
```

Prouvons que append est associative :

$$\forall l_1\ l_2\ l_3,\ \texttt{append (append } l_1\ l_2\texttt{)}\ l_3 = \texttt{append}\ l_1\ \texttt{(append}\ l_2\ l_3\texttt{)}$$

Preuve par induction structurelle

Prouvons

$$\forall l_1\ l_2\ l_3,\ \texttt{append (append } l_1\ l_2\texttt{)}\ l_3 = \texttt{append}\ l_1\ \texttt{(append}\ l_2\ l_3\texttt{)}$$

par induction structurelle sur l_1.

Cas l_1=[] :

$$\begin{aligned} \text{append (append [] l2) l3} &= \text{append l2 l3} \\ &= \text{append [] (append l2 l3)} \end{aligned}$$

Cas l_1=a::r :

$$\begin{aligned} \text{append (append (a::r) l2) l3} &= \text{append (a::(append r l2)) l3} \\ &= \text{a::(append (append r l2) l3)} \\ &\overset{h.r.}{=} \text{a::(append r (append l2 l3))} \\ &= \text{append (a::r) (append l2 l3)} \end{aligned}$$

Exercice

```
(* reverse naive *)
  let rec rev = function
  | [] -> []
  | a::r -> (rev r)@[a];;

(* reverse efficace *)
let rec rev_append l l' = match l with
  | [] -> l'
  | a :: l -> rev_append l (a :: l');;
let rev' l = rev_append l []
```

- Prouvez : $\forall l$, rev' l = rev l
- Indication : on a besoin de prouver un énoncé plus général

Pour en savoir plus

Voir le cours de Sémantique : il donne les bases pour

- l'induction bien fondée
- le raisonnement équationnel sur les structures de données de premier ordre
- le λ-calcul, qui est à la base de tous les langages fonctionnels,
- etc.

Vous trouverez un traitement en profondeur avec des exemples détaillés (écrit pour SML) dans le livre

L.C. Paulson.
ML for the working programmer.
Cambridge University Press, 1996.

Les piles

- Aussi *LIFO* (pour *last in first out*)
- On peut ajouter des valeurs à la pile, et les sortir dans l'ordre inverse de l'insertion.

Exemples (stack.ml)

```
module Lifo = struct
  exception Empty
  type 'a t = 'a list
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push a l = a::l
  let pop = function
    | [] -> raise Empty
    | h::r -> (h, r)
end;;

Lifo.(empty |> push 42 |> push 17 |> pop |> fst);;
```


Analyse des piles fonctionnelles

- C'est purement fonctionnel, donc persistant.
- Toutes les opérations ont un coût constant (qui ne dépend pas du nombre d'éléments stockés dans la pile).
- C'est donc idéal.
- Peut-on toujours trouver une implémentation purement fonctionnelle avec un coût constant des opérations ?

Les files d'attentes

- Aussi *tampon*, *FIFO* (pour *first in, first out*)
- On peut ajouter des valeurs à la file, et les sortir *dans le même ordre*.
- En opposition à la structure de *pile* qui est *LIFO*.
- Plusieurs approches pour l'implémentation.

Solution fonctionnelle naïve

- Type abstrait pour les files
- Les opérations, par exemple add, envoient la nouvelle file comme résultat.
- Réalisation avec une liste, ajout de nouveaux éléments à la fin de la liste.

Exemples (queues1.ml)

```
module type FIFO = sig
  type 'a t
  exception Empty
  val empty : 'a t
  val is_empty : 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val remove : 'a t -> ('a * 'a t)
  (** leve l'exception Empty sur une file vide *)
end;;
```


Exemples (queues2.ml)

```
module FifoNaive : FIFO = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty f = (f = [])
  let add a f = f@[a]
  let remove = function
  | [] -> raise Empty
  | a::l -> (a, l)
end;;
```


Exemples (queues3.ml)

```
open FifoNaive

let q = empty |> add 1 |> add 2 |> add 3;;
let (x,r) = remove q;;
let (y,s) = remove r;;
let (x',r') = remove q;;
;; (* persistent ! *)
```

Solution fonctionnelle naïve

- Le type est bien persistant
- Problème : une séquence de n opérations peut avoir un coût de n^2 (car `add` appelle `append`)
- On doit faire mieux !

Solution impérative

- type abstrait pour les files
- les opérations `add`, `remove` prennent une file en argument et la modifient. Pas besoin d'envoyer la file modifiée comme résultat car la file garde son identité même après modification
- réalisation avec une liste (simplement) chaînée
- type *pas* persistant
- opérations en temps constant

Exemples (queues4.ml)

```
module type FIFOIMP = sig
  type 'a t
  exception Empty
  val create : unit -> 'a t
  val is_empty : 'a t -> bool
  val add : 'a -> 'a t -> unit
  val remove : 'a t -> 'a
  (* raises Empty if the queue is empty *)
end;;
```



```
module FifoImp = struct
  exception Empty
  type 'a cell = Vide | Cons of 'a * 'a cell ref
  type 'a t = {mutable first: 'a cell;
               mutable last : 'a cell}
  let create () = {first = Vide; last = Vide}
  let is_empty f = (f.first = Vide)
  let add a f = match f.last with
    | Vide -> (* assert (f.first = Vide) *)
      f.first <- Cons (a, ref Vide);
      f.last <- f.first
    | Cons (_, r) ->
      r := Cons (a, ref Vide);
      f.last <- !r
  let remove f = match f.first with
    | Vide -> raise Empty
    | Cons (a, r) ->
      if f.last = f.first then f.last <- !r;
      f.first <- !r ;
      a
end;;
```

Exemples (queues6.ml)

```
open FifoImp;;
let f = create();;

add 3 f;;
f;;
add 4 f;;
f;;
remove f;;
f;;

(* pas persistant *)
```


Files fonctionnelles *efficaces*

- Idée : représenter une file comme une paire de piles, une pile de sortie et une pile d'entrée.
- Le sommet de la pile de sortie est l'élément suivant à sortir, le sommet de la pile d'entrée est le dernier élément entré (c'est exactement l'opposé des zippers de listes !)
- Add : empiler l'élément sur la pile d'entrée
- Remove : supprimer le sommet de la pile de sortie
- Quoi faire quand la pile de sortie est vide ?
- On renverse la pile d'entrée vers la pile de sortie, on utilisant une fonction de *reverse* à coût linéaire (voir le transparent dans la Section *raisonnement équationnel*).


```
module FifoDL : FIFO = struct
  exception Empty
  type 'a t = 'a list * 'a list

  let empty = ([],[])
  let is_empty = function ([],[]) -> true | _ -> false

  let add x (l_in, l_out) = (x::l_in, l_out)

  let remove (l_in, l_out) = match l_out with
  | a::l -> (a, (l_in, l))
  | [] -> match List.rev l_in with
      | [] -> raise Empty
      | a::l -> (a, ([], l))
end;;
```


Analyse de coût amorti

- Le module FifoDL fournit des opérations de coût non homogène : `add` a un coût constant, alors que `remove` peut avoir un coût linéaire quand la liste de sortie est vide.
- L'analyse de complexité dit que, dans le pire des cas, la complexité d'une suite de n opérations est bornée par

$$n * O(n) = O(n^2)$$

- C'est la même borne de complexité obtenue pour FifoNaive ! Est-ce que les deux solutions sont équivalentes ?
- Non, car une suite de n remove dans FifoDL n'utilise jamais un temps $O(n^2)$: si un des remove inverse la liste ($O(n)$), les autres $n-1$ ont coût constant !

Analyse de coût accumulé : la méthode du banquier

- Calculer, pour une séquence quelconque de n opérations,

$$\Sigma_{i=1}^{i=n} t(i)$$

où $t(i)$ est le temps d'exécution de la i-ème opération.
- On défini d'abord un *coût amorti* $a(i)$ de la i-ème opération. Il s'agit d'un *artefact* qui sert seulement à l'analyse de complexité. L'astuce est de trouver une définition de $a(i)$.
- Notre $a(i)$ doit avoir les propriétés suivantes :
  - $\forall i : a(i) \geq 0$
  - $\forall n : \Sigma_{i=1}^{i=n} a(i) \geq \Sigma_{i=1}^{i=n} t(i)$
- On peut avoir $a(i) > t(i)$ ou $a(i) < t(i)$ pour la i-ème opération considérée isolément.

Analyse de coût accumulé : la méthode du banquier

- Quand $a(i) > t(i)$ on imagine avoir fait un *gain* de $a(i) - t(i)$, quand $a(i) < t(i)$ on imagine une *perte* de $t(i) - a(i)$.
- Dans notre exemple, une operation `add` fait un gain. On imagine que ce gain est stocké sous forme d'un crédit avec l'élément ajouté. On peut se servir d'un crédit pour des opérations futures chères (renversement d'une liste).
- En général, le crédit accumulé après n opérations est

$$\Sigma_{i=1}^{i=n}(a(i) - t(i)) = \Sigma_{i=1}^{i=n} a(i) - \Sigma_{i=1}^{i=n} t(i) \geq 0$$

- On n'est donc jamais dans le rouge !

Analyse de coût accumulé pour FifoDL

- Coût réel :
  - coût réel de add : 1
  - coût réel de remove : 1 si la liste de sortie est non vide, *len* si la liste de sortie est vide et la liste d'entrée a longueur *len*
- Coût amorti :
  - coût amorti pour add : 2
  - coût amorti pour remove : 1
- Après avoir payé pour chaque opération, on se retrouve avec chaque élément sur la liste de sortie ayant 0 crédit, et chaque élément de la liste d'entrée en ayant 1.
- Dans le pire des cas, une suite de n opérations a un coût amorti accumulé de $2 * n = O(n)$, ce qui donne une complexité accumulée de $O(n)/n = O(1)$.
- On a donc bien gagné en utilisant FifoDL.

Analyse de coût amorti pour FifoDL : schéma de preuve

Il reste à montrer que pour tout n :

- $\Sigma_{i=1}^{i=n} a(i) \geq \Sigma_{i=1}^{i=n} t(i)$
- tout élément dans la pile d'entrée porte un crédit de 1

par induction sur la longueur de la liste d'opérations :

- *empty* (c.-à-d. $n = 0$) : trivial
- dernière opération *add* : on a un gain de 1, qu'on place comme un crédit sur l'élément ajouté à la pile d'entrée.
- dernière opération *remove* qui fait appel à `List.rev` : si la pile d'entrée est de longueur l on a un crédit de l (1 par élément) qu'on utilise pour renverser la liste (coût l).
- dernière opération *remove* de coût unitaire : pas de gain et pas de perte.

Les Dequeues

Il est possible d'adapter la même technique pour traiter les *double ended queues*, qui permettent d'insérer et supprimer en tête et en queue.

```
module type DEQUE = sig
  type 'a queue
  val empty : 'a queue
  val is_empty : 'a queue -> bool
  (* insert, inspect, and remove the front element *)
  val cons : 'a -> 'a queue -> 'a queue
  val removefirst : 'a queue -> 'a * 'a queue
  (* raises Empty if queue is empty *)
  (* insert, inspect, and remove the rear element *)
  val snoc : 'a queue -> 'a -> 'a queue
  val removelast : 'a queue -> 'a * 'a queue
  (* raises Empty if queue is empty *)
end
```


Arbres Binaires de Recherche

- Un arbre binaire est facile à définir en OCaml

```
type 'a abr =
| E
| T of 'a abr * 'a * 'a abr
```

- Un arbre binaire est appelé arbre de recherche s'il satisfait la propriété suivante :

  Pour tout nœud `T(l,v,r)`, la valeur `v` est plus grande que celle de tous les nœuds de `l`, et plus petite que celle de tous les nœuds de `r`.

- Autrement dit, un parcours *infixe* d'un arbre binaire de recherche donne les valeurs stockées dans l'arbre dans l'ordre croissant.

Recherche

- Trouvons un élément dans un arbre binaire de recherche :

```
let rec member x = function
| E -> false
| T (_, y, _) when x = y -> true
| T (g, y, _) when x < y -> member x g
| T (_, y, d) (* when x > y *) -> member x d
```

- Attention au coût linéaire si l'arbre est dégénéré (réduit à une liste) !

Arbres Binaires de Recherche Équilibrés

- Pour que les opérations soient efficaces, il faut que l'arbre soit *équilibré*.
- Il existent différentes définitions d'équilibre,
  mais pour ce qui nous concerne, l'important est cette propriété :

  La profondeur d'un arbre binaire équilibré contenant n nœuds est bornée par $O(\log n)$

- Grâce à cette propriété, la recherche qu'on a écrit plus haut s'effectue en temps logarithmique sur un arbre équilibré.

ABR Équilibrés

- Il y a un certain nombre de structures de données dans cette famille :
  - AVL : premier inventé, Adelson-Velskii et Landis, 1962 :
    La hauteur de deux sous-arbres diffère par 1 au plus.
  - 2-3 trees : structure permettant 2 ou 3 fils aux nœuds internes, et 1 ou 2 valeurs dans les feuilles
  - Red-Black trees : introduit par Rudolf Bayer, 1972
- Dans tous les cas, la recherche est faite comme pour les ABR, mais l'insertion et la suppression demandent du travail supplémentaire pour maintenir l'équilibre.

Importance des ABR Équilibrés

- Il est possible de donner une implémentation fonctionnelle de structures de données sophistiquées comme les arbres binaires de recherche équilibrés.
- Ces structures de données sont importantes parce qu'elles permettent de réaliser facilement :
  - des ensembles ordonnés, ou des tables d'associations
  - une implémentation fonctionnelle persistante et assez efficace ($O(log\ n)$ contre $O(1)$) de structures impératives comme les tableaux.
- Nous allons regarder ici une possible implémentation fonctionnelle des arbres Red-Black.

Arbres Red-Black

- Un arbre Red-Black est un arbre binaire de recherche dont les nœuds ont une couleur, Red ou Black.

```
type color = R | B
type 'a tree =
| E
| T of color * 'a tree * 'a * 'a tree
```

- On impose en plus les conditions suivantes :
  - le père d'un noeud rouge est noir
  - tout chemin de la racine à une feuille contient le même nombre de nœuds noirs
- Conséquence : la profondeur de l'arbre est au plus $2(log\ n)$, et on peut donc espérer des opérations en temps $O(log\ n)$.

Arbres Red-Black : recherche I

La recherche s'effectue en temps logarithmique, comme pour tout ABR équilibré.

```
let rec member x = function
| E -> false
| T (_, a, y, b) ->
  if x < y then member x a
  else if y < x then member x b
  else true
```

On peut éventuellement utiliser une fonction de comparaison dédiée plutôt que le < générique d'OCaml.

Arbres Red-Black : insertion I

L'insertion est plus délicate. Le code suivant est faux :

```
let colorinsert = R (* ou B ? *)

let rec ins x = function
| E -> T (colorinsert , E, x, E)
| T (color , a, y, b) as s ->
  if x < y then T (color , ins x a, y, b)
  else if y < x then T (color , a, y, ins x b)
  else s
```

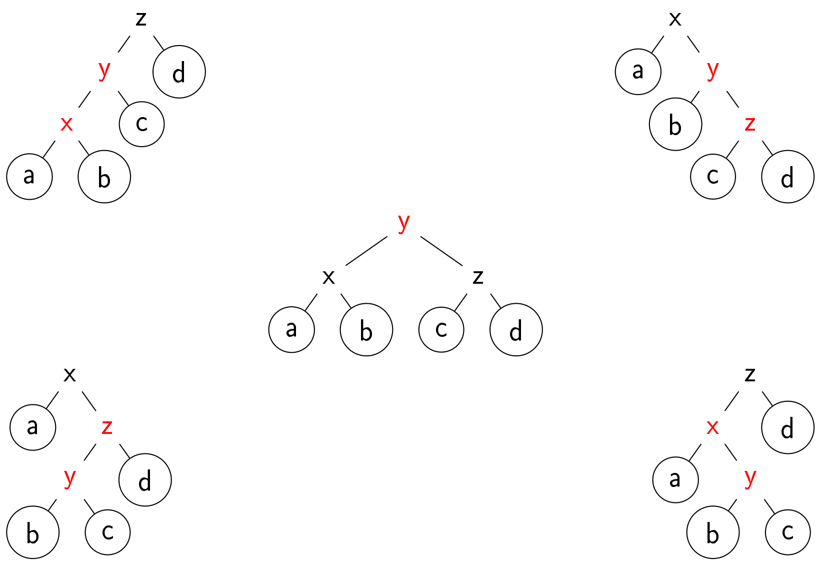
Si le nouveau élément est coloré rouge, on peut se retrouver, après l'insertion, avec un nœud Rouge avec père Rouge.
Si le nouveau élément est coloré noir, on peut se retrouver, après l'insertion, avec un chemin ayant plus de nœuds noirs que les autres.

Arbres Red-Black : rééquilibrage I

On colorie Rouge les nouveaux nœuds, et on corrige les séquences Rouge-Rouge une à une en restaurant l'invariant en bas mais en faisant remonter une racine rouge.

```
let bal = function
  | B, T (R, T (R, a, x, b), y, c), z, d
  | B, T (R, a, x, T (R, b, y, c)), z, d
  | B, a, x, T (R, T (R, b, y, c), z, d)
  | B, a, x, T (R, b, y, T (R, c, z, d)) ->
      T (R, T (B, a, x, b), y, T (B, c, z, d))
  | c, a, x, b -> T (c, a, x, b)
```


Rééquilibrage local : la fonction `bal`



Insert avec rééquilibrage

La nouvelle fonction insert (correcte, $O(log\ n)$) s'écrit comme suit :

```
let insert x s =
  let rec ins = function
  | E -> T (R, E, x, E)
  | T (color , a, y, b) as s ->
    if x < y then bal (color , ins a, y, b)
    else if y < x then bal (color , a, y, ins b)
    else s
  in
  match ins s with
  | T (_, a, y, b) -> T (B, a, y, b)
  | _ -> assert false (* ins s ne peut être vide *);;
```

Notez est que la racine est colorée Noir, ainsi même une violation Rouge-Rouge à la racine est corrigée.

Utilisation des arbre Red-Black pour Set I

Nous pouvons construire un module Set à partir de ces arbres :

```
(* Un type ordonne et la fonction de comparaison *)
module type ORDERED = sig
  type t
  val compare : t -> t -> int
end

module type SET = sig
  type elem
  type set
  val empty : set
  val insert : elem -> set -> set
  val member : elem -> set -> bool
end;;
```


Utilisation des arbre Red-Black pour Set II

```
module RedBlackSet (Element : ORDERED) :
      (SET with type elem = Element.t) =
struct

 type elem = Element.t
 let islt x y = (Element.compare x y) < 0

 type color = R | B
 type tree = E | T of color * tree * elem * tree
 type set = tree

 let empty = E
```


Utilisation des arbre Red-Black pour Set III

```
 let rec member x = function
   | E -> false
   | T (_, a, y, b) ->
      if islt x y then member x a
      else if islt y x then member x b
      else true

 let bal = function
   | B, T (R, T (R, a, x, b), y, c), z, d
   | B, T (R, a, x, T (R, b, y, c)), z, d
   | B, a, x, T (R, T (R, b, y, c), z, d)
   | B, a, x, T (R, b, y, T (R, c, z, d)) ->
      T (R, T (B, a, x, b), y, T (B, c, z, d))
   | a, b, c, d -> T (a, b, c, d)
```


Utilisation des arbre Red-Black pour Set IV

```
 let insert x s =
   let rec ins = function
   | E -> T (R, E, x, E)
   | T (color, a, y, b) as s ->
     if islt x y then bal (color, ins a, y, b)
     else if islt y x then bal (color, a, y, ins b)
     else s
   in
   match ins s with
   | T (_, a, y, b) -> T (B, a, y, b)
   | _ -> assert false
end
```

Compléter l'exemple

- On peut ajouter facilement des fonctions qui retournent le plus grand ou plus petit élément, ou la liste des éléments dans l'ordre.
- Pour ajouter une fonction qui retire un élément, il faut un peu plus de travail, voir par exemple :
  - `http://www.lri.fr/~filliatr/software.en.html`
  - `http://benediktmeurer.de/2011/10/16/red-black-trees-for-ocaml/`

Quelques exemples de la librairie standard

Set.Make	Ensembles
Map.Make	Associations

Ils sont paramétrés par un ordre sur les type de données des éléments, comme notre exemple précédent, et utilisent des AVL.

Pour en savoir plus

T.H. Cormen, C.E. Leiserson, and Stein. C. Rivest, R. L.
Introduction to algorithms.
MIT electrical engineering and computer science series. MIT Press, 2001.

Chris Okasaki.
Red-black trees in a functional setting.
J. Funct. Program., 9(4) :471–477, 1999.