Plan du cours : structures de données fonctionnelles efficaces

- Structures de données persistantes
- Raisonnement équationnel
- Queues et Dequeues
- Arbres Red-Black
- Exemples dans la librairie standard : Set et Map
- Tableaux fonctionnels de Paulson
- Pour en savoir plus

Description des objectifs

Réaliser des structures de données *fonctionnelles*

- afin de pouvoir prouver des propriétés des programmes equationnellement
- et garder facilement la version avant la modification
- on ne modifie pas en place la structure de donnée.

On veut faire cela *de façon efficace*

Prouver des propriétés équationnellement

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

- remplacement d'égaux par égaux
- induction bien fondée

Exemple

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

Prouvons que append est associative.

$$\forall l_1\ l_2\ l_3.\mathrm{append}(\mathrm{append}\ l_1\ l_2)\ l_3 = \mathrm{append}\ l_1\ (\mathrm{append}\ l_2\ l_3)$$

Preuve par induction structurelle

Par induction structurelle sur l_1.
Cas l_1=[].

append (append [] l2) l3 = append l2 l3
= append [] (append l2 l3)

Cas l_1=a::r.

append (append (a : :r) l2) l3 = append (a : :(append r l2)) l3
= a : :(append (append r l2) l3)
$=_{h.r.}$ a : :(append r (append l2 l3)
= append (a : :r) (append l2 l3)

Exercice

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


(* reverse efficace *)
let rec rev_append  = function
    ([],l) -> l
  | (a :: l,l') -> rev_append (l, (a :: l'));;

let rev' l = rev_append (l,[]);;
```

Prouvez `rev' l=rev l`, pour toute liste `l`.

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.

Structures des données *persistantes*

Ce principe de raisonnement est correct pour les structures de données dites *persistantes* :

Structure de données persistante

Structure de donnée que, lors d'une modification, préserve les versions précedentes.

Les structures purement fonctionnelles sont immuables : on ne peut les modifier, mais seulement les copier :
elles sont donc automatiquement persistantes.

Voyons dans la suite quelques exemples significatifs de ces structures de données.

Les listes chainées I

```
(* une liste d'entiers *)
let l = [1;3;5;7];;

(* une fonction d'insertion dans l'ordre *)
let rec insert x =
   function [] -> [x]
   | a::r  -> if x > a then a::(insert x r)
                      else x::a::r;;
```


Les listes chainées II

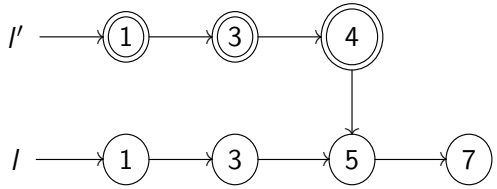
```
(* insertion  *)
let l' = insert 4 l;;
|| val l' : int list = [1; 3; 4; 5; 7]

(* l reste intacte *)
l;;
|| - : int list = [1; 3; 5; 7]
```


Ce qui se passe en mémoire

On simule la modification en place par

- une copie de la structure jusqu'á la modification
- l'introduction d'un noeud 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 recuperer la mémoire non utilisée.

Files fonctionnelles efficaces

Les files : spécification

Interface

```
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;;
```


Files fonctionnelles naîves

Implémentation

```
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;;
```

Inefficace

La concatenation de deux listes (@) est linéaire dans la taille de la première : l'insertion de n éléments peut avoir un coût *quadratique* !

Files impératives

Interface

```
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;;
```

Attention : on est obligés d'utiliser `create` pour obtenir une file vide différente à chaque appel !

Files impératives I

Implementation

```
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
```

Files impératives II

```
let add a f = match f.last with
  | Vide -> (* f.first doit etre Vide aussi *)
    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;;
```


Files imperatives

Temps d'insertion et extraction *constant*, mais ce n'est pas une structure de donnée persistante !

```
open FifoImp;;

let f = create();;

add 3 f;;
|| - : unit = ()

f;;
|| - : int FifoImp.t =
|| {first = Cons (3, {contents = Vide}); last = Cons (3, {contents = Vide})
|| }

add 4 f;;
|| - : unit = ()

f;;
|| - : int FifoImp.t =
|| {first = Cons (3, {contents = Cons (4, {contents = Vide})});
||  last = Cons (4, {contents = Vide})}

remove f;;
|| - : int = 3

f;;
|| - : int FifoImp.t =
|| {first = Cons (4, {contents = Vide}); last = Cons (4, {contents = Vide})
|| }
```


Files fonctionnelles *efficaces* I

Implémentation

```
module FifoDL : FIFO = struct
  exception Empty

  type 'a t = 'a list * 'a list

  let empty = ([], [])

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


Files fonctionnelles *efficaces* II

```
  let add x (l1, l2) = (x::l1, l2)

  let remove (l1, l2) = match l2 with
  | a::l -> (a, (l1, l))
  | [] -> match List.rev l1 with
      | [] -> raise Empty
      | a::l -> (a, ([], l))
end;;
```

Est-ce efficace ? `remove` appelle `List.rev` !

Analyse de coût amorti

Le module FifoDL fournit des opérations de coût non homogène : `add` a coût constant, alors que `remove` peut avoir un coût linéaire quand la première liste est vide.
L'analyse de complexité en moyenne dit que, dans le cas le pire, 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 :

- une suite de n insert dans FifoNaive donne temps $O(n^2)$
- 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 amorti

Idée

On s'intéresse à borner *directement* le coût d'une *suite* de n opérations, et pas en multipliant par n le cas le pire de chaque opération prise en isolation.

Si pour tout n, on a que $T(n)$ est une borne supérieure pour le coût d'une suite de n opérations, on appelle *coût amorti* d'une opération la valeur $T(n)/n$.

Attention Le coût amorti d'une opération peut être très différent du coût individuel de cette opération : c'est tout l'intérêt de la méthode.

Analyse de coût amorti

Il y a trois classes de méthodes pour cela :

- aggregate
- accounting
- potential

Pour en savoir plus

Cormen, Leiserson, Rivest. Introduction to Algorithms. MIT Press and McGraw-Hill, 2001. Chapter 18.

Analyse de coût amorti : accounting

Dans la méthode accounting, on assigne un *crédit* $\overline{c}_i$ à chaque opération op_i et on montre que pour chaque séquence de n opérations on a

$$\sum_{i=1}^{n} \overline{c}_i \geq \sum_{i=1}^{n} c_i$$

où c_i est le coût réel de l'i-ème opération.

En général, on s'imagine de stoquer la différence entre le membre gauche et droit de l'équation dans la structure de donnée, pour payer pour les futures opérations chères.

Analyse de coût amorti pour FifoDL

Avec la méthode "accounting", on a les données suivantes :

- coût réel de add : 1
- coût réel de remove : 1 si la première liste est non vide, *len* si la première liste est vide et la deuxième a longueur *len*
- crédit pour add : 3
- crédit pour remove : 0

Après avoir payé pour chaque opération, on se retrouve avec chaque élément sur la prémière liste ayant 1 crédit, et chaque élément de la deuxième liste en ayant 2.

Dans le cas le pire une suite de n opération utilisera $3 * n = O(n)$ crédits, ce qui donne une complexité amortie de $O(n)/n = O(1)$.
On a donc bien gagné en utilisant FifoDL.
Note aux dévéloppeurs Cette structure est la plus efficace connue, si on n'a pas besoin de la persistance !

Analyse de coût amorti pour FifoDL : schéma de preuve

Quand on éxécute un *add*, on utilise un crédit pour payer le coût réel du add, et on place les 2 autres sur l'élément qui est inséré sur la pile.
Quand on éxécute un *remove* qui fait appel à `List.rev`, on a n éléments sur la deuxième liste, donc $2n$ crédits ; on utilise n crédits pour payer le List.rev, 1 crédit pour payer l'enlévement d'un élément, et on laisse les $n-1$ crédits restants sur la première liste, qui contient $n-1$ éléments.
Quand on éxécute un remove de coût unitaire, on utilise le crédit restant sur l'élément de la première liste pour en payer le coût.
L'invariant est : chaque élément sur la prémière liste a 1 crédit, chaque élément de la deuxième liste en a 2.

Les Dequeues

Il est possible d'adapter la même technique pour traiter les double ended queues, qui permettent d'inserer et enlever 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 équilibrés fonctionnels efficaces

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 noeud `T(l,v,r)`, la valeur `v` est plus grand que celle de tous les noeuds de `l`, et plus pétite que celle de tous les noeuds de `r`*

Recherche

Dans un arbre binaire de recherche, trouver un élément revient à faire

```
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
;;
```

Mais cette fonction peut avoir coût linéaire si l'arbre est dégénéré (reduit à une liste) !

Arbres Binaires de Recherche Equilibrés

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 noeuds est bornée par $O(\log\ n)$*

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

ABR Equilibrés

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 d'au maximum 1
- 2-3 trees : structure permettant 2 ou 3 fils aux noeuds 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 Equilibrés

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-que elles permettent de réaliser facilement :

- des ensembles ordonnés, ou des tables d'associations
- une implémentation fonctionnelle persistente et assez efficace ($log\ n$ contre n) 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 noeuds et les feuilles ont une couleur, Red ou Black.

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

Il satisfait en plus les conditions suivantes :

- le père d'un noeud rouge est noir
- tout chemin de la racine à une feuille (vide) contient le même nombre de noeuds noirs

Il s'ensuit que la profondeur de l'arbre est au maximum $2(log\ n)$, et on peut donc espérer des operations en temps $O(log\ n)$

Arbres Red-Black : recherche I

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

```
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;;
```

Ici `islt: 'a -> 'a -> bool` est une fonction qui retourne vrai si son premier argument est inferieur au deuxième, dans un ordre donné.

Arbres Red-Black : insertion I

L'insertion, c'est autre chose : 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 islt x y then T (color, ins x a, y, b)
        else if islt y x 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 noeud Rouge avec père Rouge.
Si le nouveau élément est coloré noir, on peut se retrouver, après l'insertion, avec un chemin ayant plus de noeuds noirs que les autres.

Arbres Red-Black : re-équilibrage I

On colorie Rouge les nouveaux noeuds, et on corrige les séquences Rouge-Rouge une à une en restaurant l'invariant en bas mais en faisant remonter une racine rouge. (voir le dessin fait au tableau) :

```
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);;
```

La nouvelle fonction insert (correcte) s'écrit comme suit :

Arbres Red-Black : re-équilibrage II

```
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  (* surement non vide *)
  | T (_, a, y, b) -> T (B, a, y, b)
  | _ -> assert false;;
```

Le point important à noter 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 (* surement non vide *)
  | T (_, a, y, b) -> T (B, a, y, b)
  | _ -> assert false
end
;;
```


Completer l'exemple

On peut ajouter facilement des fonctions qui retournent le plus grand ou plus pétit élément, ou la liste des éléments dans l'ordre. Pour ajouter une fonction qui rétire 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

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

Tableaux fonctionnels de Paulson (de Alfaro)

Luca de Alfaro a écrit une librairie de *functional extensible arrays* sur la base du code pour Map et Set de la librairie standard :

```
val append : 'a -> 'a t -> 'a t
(* append d v extends the vec v,
   adding the element d at the end *)

val setappend : 'a -> 'a -> int -> 'a t -> 'a t
(* setappend d0 d i v sets the element in position
   i of v to value d, if v is long enough. Otherwise,
   it extends v with as many elements d0 as needed,
   then appends the value d in position i. *)
```

Toutes les opérations se font en temps $O(\log n)$

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.