Programmation fonctionnelle avancée

Notes de cours

Cours 8

18 novembre 2015

Sylvain Conchon

sylvain.conchon@lri.fr

Structures de files

Files

Une file est une structure où les éléments sont retirés dans l'ordre d'arrivée, ce qui correspond exactement à la notion usuelle de file d'attente.

On peut également associer une priorité aux éléments, qui ne sont alors plus retirés selon l'ordre d'arrivée ; on parle de file de priorité.

Files impératives

Signature

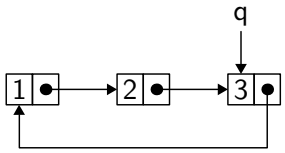
La signature minimale pour les files impératives est la suivante :

```
type 'a t
val create : unit -> 'a t
val is_empty : 'a t -> bool
val push : 'a -> 'a t -> unit
val pop : 'a t -> 'a
```


Listes chaînées

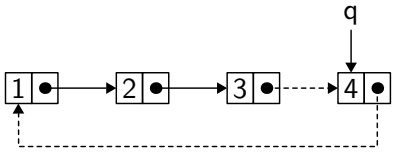
On représente une file q contenant les éléments 1, 2, 3 (insérés dans cet ordre) par une liste chaînée cyclique où chaque élément pointe vers le suivant dans la file et où le dernier pointe vers le premier.

Pour pouvoir insérer et extraire en temps constant, il suffit alors de conserver un pointeur sur le dernier élément de la file, ici 3.



Ajout d'un élément

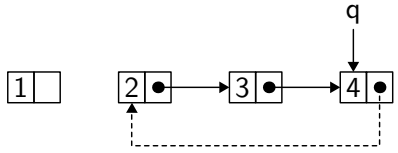
Ainsi, pour ajouter un nouvel élément, 4, il suffit de l'insérer comme le suivant de 3, c'est-à-dire entre les éléments 3 et 1, et de pointer désormais sur 4.



Accès et suppression du premier élément

On accède au premier élément, ici 1, en suivant le pointeur contenu dans le dernier élément.

Pour supprimer le premier élément, il suffit de faire pointer le dernier sur le second, ici 2, qui s'obtient en suivant deux pointeurs.



Implémentation

On commence par introduire un type `'a cell` pour représenter les cellules de la liste chaînée :

```
type 'a cell = { elt : 'a; mutable next : 'a cell }
```

On représente alors une file par une référence sur la dernière cellule.

Un problème se pose pour la représentation de la file vide, pour laquelle on ne veut justement pas avoir de cellule de liste contenant un élément.

On choisit donc de représenter une file par une référence sur une valeur de type `'a cell option`, où `None` représente la liste vide.

```
type 'a t = (('a cell) option) ref
```


Fonctions `create` et `is_empty`

Il est alors immédiat de réaliser les fonctions `create` et `is_empty`.

```
let create () =
  ref None
```

```
let is_empty q =
  !q = None
```


Insertion

Si la file `q` est vide, on crée une liste cyclique à un élément et on modifie `q` pour qu'elle pointe sur cet élément.

Si en revanche `q` n'est pas vide, on crée une nouvelle cellule `c` contenant `x`, que l'on insère entre le dernier et le premier élément, puis on modifie `q` pour pointer désormais sur cette nouvelle cellule.

```
let push x q = match !q with
  | None ->
      let rec c = { elt = x; next = c } in
      q := Some c
  | Some last ->
      let c = { elt = x; next = last.next } in
      last.next <- c;
      q := Some c
```


Extraction

Pour extraire le premier élément d'une file `q`, on commence par tester si `q` est vide. Dans ce cas, on lève une exception.

On traite le cas particulier où `q` ne contient qu'un élément (lorsque le dernier élément pointe sur lui-même)

```
let pop q = match !q with
  | None ->
      invalid_arg "pop"

  | Some last when last.next == last ->
      q := None;
      last.elt

  | Some last ->
      let first = last.next in
      last.next <- first.next;
      first.elt
```

Files persistantes

Signature

La signature minimale pour les files persistantes est la suivante :

```
type 'a t
val empty : 'a t
val is_empty : 'a t -> bool
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> 'a * 'a t
```

On voit notamment que les opérations `push` et `pop` renvoient la version modifiée de la structure.

Structure à deux listes

Pour réaliser efficacement l'ajout et le retrait d'un élément, on utilise deux listes :

- la première liste contient les éléments en tête de file, dans l'ordre
- la seconde les éléments en queue de file, en ordre inverse.

Ainsi, l'ajout comme le retrait se font en tête de liste.

Par exemple, la file contenant les éléments 1, 2, 3, 4, 5, dans cet ordre d'arrivée, peut être représentée par les deux listes :

`[1; 2]` et `[5; 4; 3]`

ou

`[1]` et `[5;4;3;2]`

ou même encore par

`[1;2;3;4;5]` et `[]`

qui sont toutes des représentations équivalentes de la même file.

Implémentation

Le type des files persistantes est simplement un synonyme pour une paire de listes.

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

La première liste représente la sortie de la file, la seconde l'entrée.

Valeur empty et fonction is_empty

La file vide empty et la fonction is_empty sont immédiates.

```
let empty = [], []

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


Ajout d'un élément

Pour ajouter un élément x, il suffit de l'ajouter en tête de la seconde liste.

```
let push x (o, i) =
  (o, x :: i)
```


Retrait d'un élément

Le retrait est plus délicat. Trois cas se présentent.

Premier cas : la file est vide.

Deuxième cas : la première liste, représentant la sortie, contient au moins un élément.

Troisième cas : tous les éléments se trouvent dans la liste des entrées.

```
let pop = function
  | [], [] -> invalid_arg "pop"
  | x :: o, i -> x, (o, i)
  | [], i ->
      match List.rev i with
      | x :: o -> x, (o, [])
      | [] -> assert false
```


Files de priorité persistantes

Files de priorité

On considère maintenant des files dans lesquelles les éléments se voient associer des priorités.

Dans de telles files, dites files de priorité, les éléments sortent dans l'ordre fixé par leur priorité et non plus dans l'ordre d'arrivée.

Signature

La signature minimale pour les files de priorité impératives est la suivante :

```
type 'a t
val empty : 'a t
val is_empty : 'a t -> bool
val add : 'a -> 'a t -> 'a t
val get_min : 'a t -> 'a
val remove_min : 'a t -> 'a t
```

La fonction `get_min` renvoie l'élément le plus prioritaire de la file et la fonction `remove_min` le supprime.

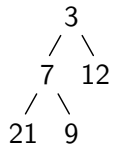
La structure de tas

Pour réaliser une file de priorité efficace, il faut recourir à une structure de données plus complexe que pour une simple file.

Une solution consiste à organiser les éléments sous la forme d'un tas (heap en anglais).

Un tas est un arbre binaire où, à chaque nœud, l'élément stocké est plus prioritaire que les deux éléments situés immédiatement au-dessous.

L'élément le plus prioritaire est donc situé à la racine. Ainsi, un tas contenant les éléments $\{3, 7, 9, 12, 21\}$, ordonnés par petitesse, peut prendre la forme suivante :



Implémentation

Le type `t` est simplement le type des arbres binaires.

```
type 'a t = Empty | Node of 'a t * 'a * 'a t
```

Le tas vide et la fonction `is_empty` sont immédiats.

```
let empty = Empty

let is_empty h =
  h = Empty
```

Fonction get_min

La structure de tas donne un accès immédiat au plus petit élément.

```
let get_min = function
 | Empty -> invalid_arg "get_min"
 | Node (_, x, _) -> x
```


Ajout d'un élément

Toute la subtilité de cette structure tient dans une fonction merge qui fusionne deux tas.

En supposant avoir écrit cette fonction, il est facile d'écrire les fonctions add

```
let add x h = merge (Node (Empty, x, Empty)) h
```


Suppression d'un élément

De même, la suppression du plus petit élément d'un tas consiste simplement à fusionner les deux tas fils de la racine.

```
let remove_min = function
  | Empty -> invalid_arg "remove_min"
  | Node (a, _, b) -> merge a b
```


Fusion de deux tas

```
let rec merge ha hb = match ha, hb with
  | Empty, h | h, Empty ->
      h
  | Node (la, xa, ra), Node (lb, xb, rb) ->
      if  xa <= xb then
        Node (ra, xa, merge la hb)
      else Node (rb, xb, merge lb ha)
```

Récursivement, on effectue une rotation des sous-arbres de la droite vers la gauche, de manière à assurer l'auto-équilibrage.

Files de priorité impératives

Signature

La signature minimale pour les files de priorité impératives est la suivante :

```
type 'a t
val create : 'a -> 'a t
val is_empty : 'a t -> bool
val add : 'a -> 'a t -> unit
val get_min : 'a t -> 'a
val remove_min : 'a t -> unit
```


Tas comme arbres binaires complets

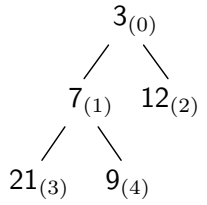
On note qu'il existe plusieurs tas contenant les mêmes éléments.

Cependant, pour des raisons d'efficacité, il est préférable de choisir un tas le moins haut possible.

Les tas construits comme des arbres binaires complets, c'est-à-dire où tous les niveaux sont remplis sauf peut-être le dernier, ont justement une hauteur minimale.

Arbres binaires complets comme des tableaux (1/2)

Un arbre binaire complet peut être représenté dans un tableau en numérotant les nœuds de l'arbre de haut en bas et de gauche à droite, à partir de 0. Sur le tas précédent cela donne l'étiquetage suivant :



Cette numérotation permet de représenter le tas dans un tableau à 5 éléments de la manière suivante :

0	1	2	3	4
3	7	12	21	9

Arbres binaires complets comme des tableaux (2/2)

De manière générale,

- la racine de l'arbre occupe la case d'indice 0
- les racines des deux sous-arbres du nœud stocké à la case i sont stockées respectivement aux cases $2i+1$ et $2i+2$
- le père du nœud i est stocké en $\lfloor (i-1)/2 \rfloor$

Tableaux redimensionnables

Comme ne connaît pas a priori la taille de la file de priorité et on ne peut donc pas fixer à l'avance la taille maximale du tableau.

Une solution consiste à supposer que l'on dispose d'un module `A` de tableaux redimensionnables, dont la signature est :

```
type 'a t
val length : 'a t -> int
val make : int -> 'a -> 'a t
val resize : 'a t -> int -> unit
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> unit
```


Fonctions `create`, `is_empty` et `get_min`

```
let create v = A.make 0 v

let is_empty h = A.length h = 0

let get_min h =
  if A.length h = 0 then invalid_arg "get_min";
  A.get h 0
```

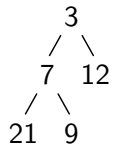

Insertion d'un élément (1/2)

L'insertion d'un élément `x` dans un tas `h` consiste à ajouter `x` dans l'arbre que représente `h` de manière à conserver la structure de tas.

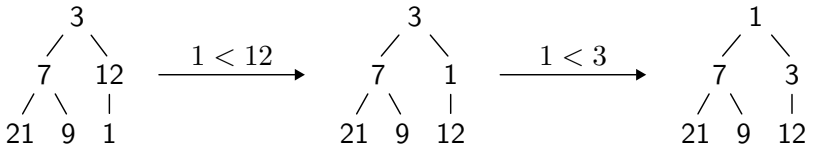
Une solution consiste à ajouter `x` tout en bas à droite de l'arbre et à le faire remonter tant que c'est nécessaire.

Insertion d'un élément (2/2)

Par exemple, considérons de nouveau le tas suivant :



L'ajout de l'élément 1 dans ce tas est réalisé en trois étapes :



Remonter un élément

On écrit une fonction `move_up` qui insère un élément `x` dans un tas `h`, en partant de la position `i`. Cette fonction suppose que l'arbre de racine `i` obtenu en plaçant `x` en `i` est un tas.

On utilise l'algorithme suivant : tant que `x` est plus petit que son père, on échange leurs deux valeurs et on recommence.

```
let rec move_up h x i =
  if i = 0 then A.set h i x
  else
    let fi = (i - 1) / 2 in
    let y = A.get h fi in
    if y > x then begin
      A.set h i y;
      move_up h x fi
    end
    else A.set h i x
```


La fonction add

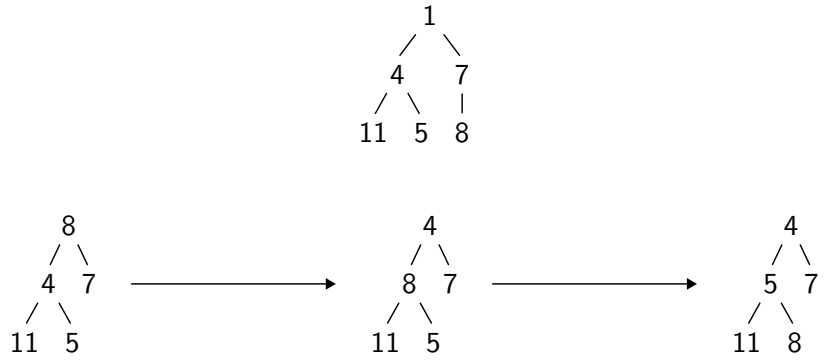
Pour implémenter add, on va donc étendre le tableau d'une case, y mettre la valeur `x`, puis à faire remonter `x` jusqu'à la bonne position.

```
let add x h =
  let n = A.length h in
  A.resize h (n + 1);
  move_up h x n
```


Suppression du plus petit élément

Idée principale. On supprime la racine de l'arbre, on la remplace par l'élément tout en bas à droite du tas (l'élément occupant la dernière case du tableau), puis on fait descendre cet élément dans le tas jusqu'à sa place.

Supposons par exemple que l'on veuille supprimer le plus petit élément du tas suivant :



La fonction remove_min

```
let remove_min h =
  let n = A.length h - 1 in
  if n < 0 then invalid_arg "remove_min";
  let x = A.get h n in
  A.resize h n;
  if n > 0 then move_down h x 0
```

La fonction move_down fait descendre une valeur à partir d'une position donnée.

Descente d'un élément

```
let rec move_down h x i =
  let j = smallest_node h x i in
  if j = i then
    A.set h i x
  else begin
    A.set h i (A.get h j);
    move_down h x j
  end
```

La fonction smallest_node détermine si une valeur x que l'on voudrait mettre au nœud i doit ou non descendre dans le tas.

La fonction smallest_node

```
let min h l r =
 if (A.get h r) < (A.get h l) then r else l

let smallest_node h x i =
  let l = 2 * i + 1 in
  let n = A.length h in
  if l >= n then
    i
  else
    let r = l + 1 in
    let j = if r < n then min h l r else l in
    if(A.get h j) < x then j else i
```


Tableaux redimensionnables

Un tableau redimensionnable est un tableau dont on peut changer la taille à tout moment, avec une opération resize, aussi bien pour l'agrandir que pour la diminuer.

La signature de tels tableaux est :

```
type 'a t
val length : 'a t -> int
val make : int -> 'a -> 'a t
val resize : 'a t -> int -> unit
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> unit
```

Principe

L'idée derrière la réalisation d'un tableau redimensionnable est très simple :

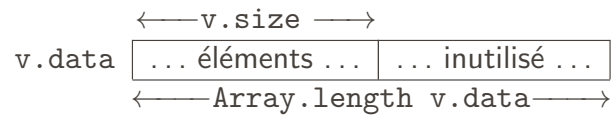
- On utilise un tableau usuel pour stocker les éléments et, lorsqu'il devient trop petit, on en alloue un plus grand dans lequel on recopie les éléments du premier.
- Pour éviter de passer notre temps en allocations et en copies, on s'autorise à ce que le tableau de stockage soit trop grand, les éléments au-delà d'un certain indice n'étant pas significatifs.

Implémentation

Le type des tableaux redimensionnables est donc :

```
type 'a t = {
  mutable size: int;
  mutable data: 'a array;
}
```

Si `v` est un tel tableau redimensionnable, on peut donc schématiser ainsi la situation :



On maintiendra donc toujours l'invariant suivant :

$$0 \leq \texttt{v.size} \leq \texttt{Array.length v.data}$$

Redimensionnement (1/2)

Si la nouvelle taille demandée excède la capacité du tableau `v.data`, on alloue un nouveau tableau, suffisamment grand, on y recopie les éléments de `v.data` puis on affecte `v.data` à ce nouveau tableau.

Lorsque l'on souhaite diminuer la taille du tableau `v`, il suffit de modifier `v.size`.

Redimensionnement (2/2)

Petite difficulté : il ne faut pas conserver de pointeurs sur les éléments qui ne sont plus significatifs, afin que le GC puisse les récupérer lorsque cela est possible.

Dès lors, il nous faut une valeur du bon type pour remplacer les éléments qui disparaissent. Pour cela, on va exiger une valeur par défaut lors de la création du tableau, qu'on conservera dans un troisième champ de la structure.

On a donc au final le type suivant :

```
type 'a t = {
  default : 'a;
  mutable size: int;
  mutable data: 'a array;
}
```

Fonctions `length` et `make`

```
let length a = a.size

let make n d =
 { default = d; size = n; data = Array.make n d }
```

On va maintenir l'invariant suivant sur le type `t` : tous les éléments à partir de l'indice `size`, c'est-à-dire tous les éléments non significatifs, ont la valeur par défaut stockée dans le champ `default`.

Accès et affectation

Pour accéder au `i`-ième élément du tableau redimensionnable `a`, il convient de vérifier la validité de l'accès, car le tableau `a.data` peut contenir plus de `a.size` éléments.

```
let get a i =
  if i < 0 || i >= a.size then invalid_arg "get";
  a.data.(i)
```

L'affectation est analogue :

```
let set a i v =
  if i < 0 || i >= a.size then invalid_arg "set";
  a.data.(i) <- v
```


Fonction `resize`

```
let resize a s =
  if s <= a.size then
    Array.fill a.data s (a.size - s) a.default
  else begin
    let n = Array.length a.data in
    if s > n then begin
      let n' = max (2 * n) s in
      let a' = Array.make n' a.default in
      Array.blit a.data 0 a' 0 a.size;
      a.data <- a'
    end
  end;
  a.size <- s
```