

Algorithmique et Approche Fonctionnelle de la Programmation

Cours 8 : Les Arbres

16 novembre 2009

Louis Mandel

Université Paris-Sud 11

`louis.mandel@lri.fr`

D'après des transparents de Sylvain Conchon

Structures d'arbre

La structure d'arbre est très utilisée en informatique, que ce soit pour :

- ▶ présenter un ensemble d'objets ou d'informations élémentaires organisé en une structure hiérarchique
- ▶ en faciliter l'accès ou la recherche
- ▶ modéliser de nombreux problèmes

Les structures d'arbres constituent un complément indispensable de la structure de liste

- ▶ liste = organisation **linéaire** de l'information où tous les objets sont au même niveau
- ▶ arbre = organisation **hiérarchique** de l'information en **plusieurs niveaux**

Exemples de hiérarchies

- ▶ Le sommaire d'un **livre** reflète une hiérarchie
 - I. chapitres
 - 1. sections
 - a. sous-sections
 - paragraphes
- ▶ L'organisation des **fichiers** d'un système d'exploitation en répertoires et sous-répertoires correspond à une structure d'arbre

Terminologie

- ▶ **A** est la **racine** de l'arbre
- ▶ **A, B, ..., J** sont les **nœuds** de l'arbre
- ▶ **A** a pour **fil** **B, C** et **D**
- ▶ **E, F, ..., J** sont des **feuilles**, *i.e.* des nœuds sans fils
- ▶ **A, B, C** et **D** sont des **nœuds internes**
- ▶ **A** a trois **sous-arbres**
- ▶ **ACG** est une **branche**

Arbres binaires

- ▶ Les **arbres binaires** sont des arbres tels que tout nœud a au plus **deux** fils
- ▶ Ces arbres sont simples à mettre en œuvre et utilisés dans de nombreuses modélisations
- ▶ On peut donner une définition récursive d'un arbre binaire :
 1. un **arbre** est une **feuille**
 2. ou un **arbre** est un nœud interne qui peut être décomposé en la racine et les **fils gauche et droit**, qui sont également des **arbres**
- ▶ On peut de plus associer une **valeur** à chaque feuille et chaque nœud interne

Arbre vide

La notion d'**arbre vide** est utilisée pour uniformiser la représentation des arbres binaires, elle permet de ne pas distinguer les feuilles des nœuds internes

- ▶ un arbre vide se comporte comme une feuille à laquelle il n'est pas associé de valeur
- ▶ une feuille avec valeur peut alors se représenter comme un nœud interne dont tous les fils sont des arbres vides
- ▶ lorsqu'un arbre est non vide, il est caractérisé par sa racine à laquelle est associée une valeur et ses fils, qui sont **eux-mêmes des arbres**

Arbres binaires en Ocaml

Le type arbre suivant permet de définir des **arbres binaires** dont les feuilles et les nœuds contiennent des **entiers**

```
# type arbre = Vide | Noeud of int * arbre * arbre;;
```

```
type arbre = Vide | Noeud of int * arbre * arbre
```

```
# let a =  
    Noeud(10,  
          Noeud(2, Noeud(8, Vide, Vide), Vide),  
          Noeud(5, Noeud(11, Vide, Vide), Noeud(3, Vide, Vide)));;
```

```
val a : arbre =  
    Noeud (10, Noeud (2, Noeud (8, Vide, Vide), Vide),  
          Noeud (5, Noeud (11, Vide, Vide), Noeud (3, Vide, Vide)))
```

Arbres binaires polymorphes

On définit une structure d'arbre binaire **polymorphe** en ajoutant un paramètre de type

```
# type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre;;  
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

```
# let b = Noeud(10,Noeud(5,Vide,Vide),Vide);;  
val b : int arbre = Noeud (10, Noeud (5, Vide, Vide), Vide)
```

```
# let c = Noeud('f',Vide,Noeud('a',Vide,Noeud('g',Vide,Vide)));;  
val c : char arbre =  
    Noeud ('f', Vide, Noeud ('a', Vide, Noeud ('g', Vide, Vide)))
```

Taille d'un arbre binaire

La fonction `taille` renvoie le **nombre de nœuds** d'un arbre binaire

```
# let rec taille a =  
  match a with  
  | Vide -> 0  
  | Noeud (_, g, d) -> 1 + taille g + taille d;;  
val taille : 'a arbre -> int = <fun>
```

```
# let _ = taille b;;  
- : int = 2
```

```
# let _ = taille c;;  
- : int = 3
```

Profondeur d'un arbre binaire

La fonction **profondeur** renvoie la longueur de la plus grande branche d'un arbre binaire

```
# let rec profondeur a =  
  match a with  
  | Vide -> 0  
  | Noeud (_, g, d) -> 1 + max (profondeur g) (profondeur d)  
;;  
val profondeur : 'a arbre -> int = <fun>
```

```
# let _ = profondeur b;;  
- : int = 2
```

```
# let _ = profondeur c;;  
- : int = 3
```

Relation en taille et profondeur

- ▶ profondeur \leq taille
- ▶ taille $\leq 2^{\text{profondeur}} - 1$
- ▶ un arbre binaire est dit **complet** si : taille = $2^{\text{profondeur}} - 1$

Miroir d'un arbre binaire

La fonction `miroir` retourne l'image miroir d'un arbre binaire

```
# let rec miroir a =  
  match a with  
  | Vide -> Vide  
  | Noeud (r, g, d) -> Noeud(r,miroir d,miroir g);;  
val miroir : 'a arbre -> 'a arbre = <fun>
```

```
# let _ = miroir c;;  
- : char arbre =  
Noeud ('f', Noeud ('a', Noeud ('g', Vide, Vide), Vide), Vide)
```

Parcours d'un arbre binaire

De nombreuses fonctions sur les arbres binaires consistent à les parcourir suivant un certain ordre

- ▶ **préfixe** : on traite d'abord la **racine**, puis on parcourt le sous-arbre **gauche**, et enfin le sous-arbre **droit**
- ▶ **infixe** : on parcourt d'abord le sous-arbre **gauche**, puis on traite la **racine**, et enfin le sous-arbre **droit**
- ▶ **suffixe** : on parcourt d'abord le sous-arbre **gauche**, puis le sous-arbre **droit**, et enfin la **racine**

Itérateurs

Comme pour les listes, on peut définir des itérateurs sur les arbres binaires

La fonction `fold_gdr` réalise par exemple un parcours `suffixe` d'un arbre binaire en appliquant une fonction `f` à la racine et aux résultats des sous-arbres gauche et droit

```
# let rec fold_gdr f acc a =  
  match a with  
  | Vide -> acc  
  | Noeud(r,g,d) ->  
    let vg = fold_gdr f acc g in  
    let vd = fold_gdr f acc d in  
    f r vg vd;;  
val fold_gdr : ('a -> 'b -> 'b -> 'b) -> 'b -> 'a arbre -> 'b = <fun>
```

Exemples

Les fonctions `taille`, `profondeur` et `miroir` peuvent être réécrites en utilisant l'itérateur `fold_gdr`

```
# let taille a = fold_gdr (fun _ x y -> 1+x+y) 0 a;;
```

```
val taille : 'a arbre -> int = <fun>
```

```
# let profondeur a = fold_gdr (fun _ x y -> 1 + max x y) 0 a;;
```

```
val profondeur : 'a arbre -> int = <fun>
```

```
# let miroir a = fold_gdr (fun r x y -> Noeud(r,y,x)) Vide a;;
```

```
val miroir : 'a arbre -> 'a arbre = <fun>
```

Arbres n-aires

- ▶ Les arbres **n-aires** ont un nombre de sous-arbres arbitraire
- ▶ On représente les arbres n-aires polymorphes à l'aide du type suivant

```
# type 'a arbre = Vide | Noeud of 'a * 'a arbre list;;
```

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre list
```

```
# let a =
```

```
    Noeud(10, [ Noeud(2, []);
```

```
                Noeud(5, [Noeud(11, []);Noeud(3, [])]);
```

```
                Noeud(8, [])]);;
```

```
val a : int arbre =
```

```
    Noeud (10,
```

```
        [Noeud (2, []); Noeud (5, [Noeud (11, []); Noeud (3, [])]); Noeud (8, [])])
```

Fonctions taille et profondeur

Les fonctions **taille** et **profondeur** pour les arbres n-aires se définissent par :

```
# let rec taille a =  
  match a with  
  | Vide -> 0  
  | Noeud(_,l) ->  
    1 + List.fold_left (fun acc x -> acc + taille x) 0 l;;  
val taille : 'a arbre -> int = <fun>
```

```
# let rec profondeur a =  
  match a with  
  | Vide -> 0  
  | Noeud(_,l) ->  
    1 + List.fold_left  
      (fun acc x -> max acc (profondeur x)) 0 l;;  
val profondeur : 'a arbre -> int = <fun>
```

Ensemble des valeurs d'un arbre

La fonction `liste_arbre` retourne une liste formée des éléments d'un arbre n-aire

```
# let list_arbre a =  
  let rec liste_rec acc a =  
    match a with  
    | Vide -> acc  
    | Noeud(r,l) -> List.fold_left liste_rec (r::acc) l  
  in liste_rec [] a;;  
val list_arbre : 'a arbre -> 'a list = <fun>
```

```
# let _ = list_arbre a;;  
- : int list = [8; 3; 11; 5; 2; 10]
```