Programmation fonctionnelle avancée

Notes de cours

Cours 3

23 septembre 2015

Sylvain Conchon

sylvain.conchon@lri.fr

Arbres binaires

Arbres binaires

Les arbres binaires avec des informations de type 'a aux nœuds sont définis avec le type suivant :

```
type 'a arbre =
   Vide | Noeud of 'a * 'a arbre * 'a 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, ... , ...)
```

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>
```

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>


# taille a;;
- : int = 6
```

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>
```

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>
```

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>


# miroir a;;
- : int arbre =
  Noeud (10,
    Noeud (5, Noeud (3, Vide, Vide), Noeud (11, Vide, Vide)),
    Noeud (2, Vide, Noeud (8, Vide, Vide)))
```

Recherche dans un arbre binaire

La fonction `recherche`, de type `'a -> 'a arbre -> bool` recherche un élément dans un arbre binaire :

```
let rec recherche e = function
  | Vide -> false
  | Noeud(x,g,d) ->
      x=e || recherche e g || recherche e d
```

- Le temps de recherche dans le pire des cas est en $O(n)$.
- Il faut des hypothèses plus fortes sur la structure de l'arbre afin d'obtenir une meilleure complexité.

Arbres n-aires

- Les arbres n-aires ont un nombre arbitraire de sous-arbres
- 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
```

Arbres n-aires

- Les arbres n-aires ont un nombre arbitraire de sous-arbres
- 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,[...])
```

Taille et profondeur d'un arbre n-aire

Les fonctions taille et profondeur pour les arbres n-aires se définissent de la manière suivante

```
# 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>
```

Taille et profondeur d'un arbre n-aire

Les fonctions taille et profondeur pour les arbres n-aires se définissent de la manière suivante

```
# 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 n-aire

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 liste_arbre : 'a arbre -> 'a list = <fun>


# liste_arbre a;;

- : int list = [10; 2; 5; 11; 3; 8]
```

Arbre binaire de recherche

Arbres ordonnés (ou de recherche)

Un arbre binaire est ordonné (ou de recherche) par rapport à une relation d'ordre quelconque si :

- c'est l'arbre `Vide`
- ou c'est un arbre non-vide `Noeud(x,g,d)` et
  1. les éléments du sous-arbre gauche `g` sont inférieurs à la racine `x`
  2. la valeur `x` stockée à la racine de l'arbre est inférieure aux éléments du sous-arbre droit `d`
  3. les sous-arbres `g` et `d` sont eux-mêmes ordonnés

Recherche d'un élément

La structure ordonnée des arbres binaires de recherche permet d'effectuer la recherche d'un élément avec une compléxité en moyenne de $O(\texttt{log}n)$.

```
let rec recherche e = function
  | Vide -> false
  | Noeud (x, _, _) when x=e -> true
  | Noeud (x, g, _) when e<x -> recherche e g
  | Noeud (_, _, d) -> recherche e d
```

Ajout d'un élément

L'ajout d'un élément dans un arbre binaire de recherche peut se faire de deux manières :

- ajout aux feuilles : facile à définir
- ajout à la racine : utile si les recherches portent sur les éléments récemment ajoutés

Ajout aux feuilles

La fonction `ajout : 'a -> 'a arbre -> 'a arbre` est définie de la façon suivante :

```
let rec ajout e a =
  match a with
    | Vide -> Noeud(e,Vide,Vide)
    | Noeud(x, _, _) when e=x -> a
    | Noeud(x, g, d) when x<e -> Noeud(x, g, ajout e d)
    | Noeud(x, g, d) -> Noeud(x, ajout e g, d)
```

Ajout à la racine

L'ajout à la racine d'un élément `x` consiste à

- "couper" un arbre en deux sous-arbres (de recherche) `g` et `d`, contenant respectivement les éléments plus petits et plus grands que `x`
- construire l'arbre `Noeud(x,g,d)`

Couper un arbre en deux

La fonction `coupe : 'a -> 'a arbre -> 'a arbre * 'a arbre` réalise la coupure d'un arbre.

```
let rec coupe e a =
 match a with
  | Vide -> (Vide , Vide)
  | Noeud(x, g, d) when x=e -> (g , d)
  | Noeud(x, g, d) when x<e ->
    let (t1 , t2) = coupe e d in
    (Noeud(x, g, t1), t2)
  | Noeud(x, g, d) ->
    let (t1 , t2) = coupe e g in
    (t1 , Noeud(x, t2, ld))
```

Ajout à la racine

La fonction `ajout : 'a -> 'a arbre -> 'a arbre` est alors définie de la manière suivante :

```
let ajout e a =
  let (g , d) = coupe e a in Noeud(e,g,d)
```

Suppression d'un élément (1/2)

La suppression d'un élément `x` dans un arbre binaire de recherche consiste à :

- isoler le sous-arbre `Noeud(x, g, d)`
- supprimer le plus grand élément `y` de `g` (on obtient ainsi un arbre de recherche `h`)
- reconstruire l'arbre `Noeud(y, h, d)`

```
let rec enleve_plus_grand = function
  | Vide -> raise Not_found
  | Noeud(x, g, Vide) -> (x, g)
  | Noeud(x, g, d) ->
    let (y, d') = enleve_plus_grand d in
    (y, Noeud(x, g, d'))

val enleve_plus_grand : 'a arbre -> 'a * 'a arbre
```

Suppression d'un élément (2/2)

La fonction `suppression : 'a -> 'a arbre -> 'a arbre` est alors définie par :

```
let rec suppression e a =
 match a with
 | Vide -> Vide
 | Noeud(x, Vide, d) when x=e -> d
 | Noeud(x, g, d) when x=e ->
    let (y, g') = enleve_plus_grand g in
    Noeud(y,g',d)
 | Noeud(x, g, d) when e<x ->
    Noeud(x, suppression e g,d)
 | Noeud(x, g, d) ->
    Noeud(x, g, suppression e d)
```