

LIFPF – Programmation fonctionnelle

TD3 – Structures inductives et types

Licence informatique UCBL – Printemps 2022–2023

Exercice 1 : Arbres binaires

1. Rappeler la définition des arbres binaires d'entiers (`int`) avec données dans les nœuds. Donner des exemples d'arbres contenant 0, 1, 2 et 3 `int`.
2. On rappelle que les arbres binaires de recherche sont des arbres binaires tels que :
 - Le fils gauche ne contient que des éléments *plus petits* que le nœud courant.
 - Le fils droit ne contient que des éléments *plus grands* que le nœud courant.Est-il possible de définir les arbres binaires de recherche uniquement avec la déclaration `type` ? Si oui, donner la définition, sinon expliquer pourquoi et comment contourner le problème.
3. Discuter des différences entre fonctions et constructeurs de type.

Exercice 2 : Généralisation des arbres binaires

1. Définir un type paramétré `'a abr` pour des arbres binaires. Le paramètre de type correspondra au type des éléments.
2. Définir la fonction `renverse` qui renverse un arbre c'est-à-dire qui inverse les fils gauche et droit de chacun des nœuds de l'arbre. On prendra soin de bien écrire le type des paramètres et du résultat de la fonction. Ces types devront être *les plus généraux possibles*.
3. Pourquoi n'a-t-on pas que des paramètres de type différents pour chaque type dans la fonction précédente ?
4. On regarde à présent le cas particulier où le paramètre de type `'a` est une paire (`'b * 'c`). Donner une fonction `assoc` qui prend un élément `e` de type `'b` et renvoie *la liste des éléments* de type `'c` associés à `e` dans l'arbre.
5. Contrairement à la fonction `renverse`, la fonction `assoc` accède à la valeur des éléments. Pourtant son type reste paramétré, pourquoi ?
6. De même qu'en λ -calcul, OCaml peut prendre des fonctions en arguments. Donner le code de la fonction `insere` qui insère un élément dans un arbre binaire de recherche. Cette fonction prendra en argument *la fonction qui permettra de comparer les éléments de l'arbre*. Pour obtenir le bon type pour `insere`, il faudra réfléchir au type de cette fonction de comparaison par rapport au type des autres arguments de `insere`.
7. Donner une version alternative de `assoc` avec les hypothèses suivantes :
 - l'arbre est un arbre de recherche ordonné selon les clés (de type `'b`) ;
 - on a *au plus* une occurrence de chaque clé dans l'arbre ;
 - le résultat n'est plus une liste mais une `'c option`.

Corrections

Solution de l'exercice 1

- ```
(* Type des AB d'int *)
type abr_int = Vide | Noeud of int * abr_int * abr_int;;

(* Arbre vide *)
Vide;;

(* Arbre avec 1 int *)
Noeud (42,Vide,Vide);;

(* Arbres avec 2 int *)
Noeud (37, Vide, Noeud (42, Vide, Vide));;
Noeud (37, Noeud (42, Vide, Vide), Vide);;

(* Arbres avec 3 int *)
Noeud(7, Noeud(37,Vide,Vide), Noeud(42, Vide, Vide));;
Noeud(7, Noeud (37, Vide, Noeud (42, Vide, Vide)), Vide);;
Noeud(7, Vide, Noeud (37, Vide, Noeud (42, Vide, Vide)));;
Noeud(7, Vide, Noeud (37, Noeud (42, Vide, Vide), Vide));;
Noeud(7, Noeud (37, Noeud (42, Vide, Vide), Vide), Vide);;
```
- Ce n'est pas possible car la construction type ne permet pas de spécifier des contraintes sur les valeurs. On peut contourner le problème en imposant que la construction des noeuds se fasse *systématiquement* par l'intermédiaire d'une fonction de création de noeud qui fait la vérification avant d'appeler le constructeur Noeud. On peut même aller un peu plus loin en remplaçant la fonction de construction par une fonction récursive d'insertion dans l'arbre binaire.
- Une fonction effectue un calcul alors qu'un constructeur crée une structure en mémoire. Une fonction peut utiliser une autre fonction ou un constructeur dans son calcul alors qu'un constructeur n'effectuant pas de calcul, il ne peut utiliser ni un autre constructeur, ni une fonction. Enfin, contrairement à une fonction, un constructeur peut être utilisé dans un pattern matching pour déconstruire une valeur.

### Solution de l'exercice 2

Code (les réponses aux questions qui ne sont pas du code sont à la suite) :

```
(* 1. *)
type 'a abr = Vide | Noeud of 'a * 'a abr * 'a abr

(* 2. *)
let rec renverse (arbre : 'a abr) : 'a abr =
 match arbre with
 | Vide -> Vide
 | Noeud (e, fg, fd) -> Noeud (e, renverse fd, renverse fg)

(* 4. *)
let rec assoc (e : 'b) (arbre : ('b * 'c) abr) : 'c list =
 match arbre with
```

```

| Vide -> []
| Noeud ((k, v), fg, fd) ->
 let dans_fils = assoc e fg @ assoc e fd in
 if e = k then v :: dans_fils else dans_fils

(* Une version avec accumulateur et sans concatenation *)
let assoc' : 'b -> ('b * 'c) abr -> 'c list =
 let rec aux (e : 'b) (arbre : ('b * 'c) abr) (acc : 'c list) :
 'c list =
 match arbre with
 | Vide -> acc
 | Noeud ((k, v), fg, fd) ->
 let dans_fils = aux e fg (aux e fd acc) in
 if e = k then v :: dans_fils else dans_fils
 in
 fun e arbre -> aux e arbre []

(* 6. *)
let rec insere (cmp : 'a -> 'a -> bool) (e : 'a) (arbre : 'a abr) :
 'a abr =
 match arbre with
 | Vide -> Noeud (e, Vide, Vide)
 | Noeud (e', fg, fd) ->
 if cmp e e' then Noeud (e', insere cmp e fg, fd)
 else Noeud (e', fg, insere cmp e fd)

(* 7. *)
let rec assoc_abr (cmp : 'a -> 'a -> bool) (e : 'a)
 (arbre : ('a * 'b) abr) : 'b option =
 match arbre with
 | Vide -> None
 | Noeud ((k, v), fg, fd) ->
 if k = e then Some v
 else if cmp e k then assoc_abr cmp e fg
 else assoc_abr cmp e fd

```

3. Si on avait un paramètre de type différent entre l'argument de la fonction et le résultat de celle-ci, cela signifierait que l'on change le type des éléments de l'arbre. Or cette fonction change juste *la structure* de l'arbre et pas les éléments qui y sont stockés. On garde donc le même paramètre en résultat.
5. La seule utilisation des valeurs se fait avec = et cet opérateur est lui-même générique : son type est 'a -> 'a -> bool. Il ne va donc pas imposer d'instancier 'b sur un type concret. Par contre il impose que les types de e et k soient les mêmes. C'est pour cela que le type de assoc est 'b -> ('b \* 'c) abr -> 'c list et pas 'd -> ('b \* 'c) abr -> 'c list.

**Remarque** Dans la question 7, la fonction de comparaison est utilisée différemment que dans `insere`. Elle a d'ailleurs un type différent. Donc si on utilise `insere` pour créer des arbres binaire de recherche d'associations, on a deux fonctions de comparaison différentes. On pourrait générer la fonction de comparaison à utiliser avec `insere` à partir de celle à utiliser avec `assoc_abr` :

```

let cmp_insere_of_cmp_assoc (cmp : 'a -> 'a -> bool) :

```

```
'a * 'b -> 'a * 'b -> bool =
fun (p : 'a * 'b) (q : 'a * 'b) : bool ->
 match (p, q) with (kp, _), (kq, _) -> cmp kp kq
```