Programmation fonctionnelle : feuille 9 2019-2020

Exercice 9.1 1. Définir un type `'a btree` pour les arbres binaires dont les nœuds portent des étiquettes de type `'a`.

2. Modifier le type de la question 1 pour que les nœuds internes portent des valeurs de type `int`.

Exercice 9.2 Écrire une fonction `single x` de type `'a -> 'a btree` qui construit un arbre avec un seul nœud binaire d'étiquette `x`.

Exercice 9.3 Écrire une fonction `btree_size : 'a btree -> int` qui prend en argument un arbre et renvoie son nombre de nœuds.

Exercice 9.4 Écrire une fonction `btree_nb_leaves : 'a btree -> int` qui prend en argument un arbre et renvoie son nombre de feuilles.

Exercice 9.5 Écrire une fonction `btree_nb_internals : 'a btree -> int` qui prend en argument un arbre `t` et renvoie son nombre de nœuds internes.

Exercice 9.6 Écrire une fonction `btree_height : 'a btree -> int` qui calcule la hauteur d'un arbre. Il n'est pour l'instant pas nécessaire de donner une version récursive terminale. Ce sera l'objet d'un autre exercice.

Exercice 9.7 On dit qu'un arbre est ***complet*** si tout nœud interne a exactement deux fils. Écrire une fonction `btree_is_complete : 'a btree -> bool` qui teste si un arbre binaire est complet.

Exercice 9.8 La ***profondeur*** d'un nœud est la longueur de l'unique chemin qui va de la racine à ce nœud. Un arbre est ***parfait***, s'il est complet et toutes ses feuilles ont la même profondeur. Écrire une fonction `btree_is_perfect` de type `'a btree -> bool` qui renvoie `true` si les feuilles de son arbre argument ont toutes la même profondeur, et `false` sinon.

Exercice 9.9 1. Écrire une fonction `btree_every : ('a -> bool) -> 'a btree -> bool` qui prend en paramètres :

- un prédicat `pred` sur des valeurs de type `'a`,
- un arbre `t` dont les étiquettes sont de type `'a`

et qui renvoie `true` si toutes les étiquettes de l'arbre `t` satisfont le prédicat `p`, et `false` sinon.

2. Écrire de même une fonction `btree_exists`, de même type, qui teste s'il existe une étiquette satisfaisant une propriété.

Exercice 9.10 Écrire une fonction `btree_mirror` de type `'a btree -> 'a btree` qui renvoie le symétrique (ou miroir) d'un arbre binaire.

Exercice 9.11 $\star$ (Plus difficile) Écrivez une fonction `btree_height: 'a btree -> int` récursive terminale, qui calcule la hauteur d'un arbre. **Aide** : utilisez un accumulateur contenant des couples `int * 'a btree`.

Exercice 9.12 1. Écrire une fonction `btree_to_list` de type `'a btree -> 'a list` qui renvoie la liste des étiquettes des nœuds internes d'un arbre en ordre infixe. Pour cette question, on pourra utiliser la concaténation de listes.

2. ⋆ Écrire une seconde version qui n'utilise pas la concaténation de listes.

Exercice 9.13 1. Écrire une fonction `btree_sum` qui calcule la somme des étiquettes des nœuds internes d'un arbre, lorsque celles-ci sont des entiers.

2. Écrire une fonction `btree_prod` qui calcule le produit des étiquettes des nœuds internes d'un arbre, lorsque celles-ci sont des entiers.

Exercice 9.14 Écrire une fonction `btree_concat` de type `'a btree -> 'a btree -> 'a btree` qui retourne l'arbre obtenu à partir du premier arbre passé en argument en remplaçant toutes ses feuilles par une copie du deuxième arbre passé en argument.

Exercice 9.15 1. Plusieurs fonctions des exercices précédents peuvent s'écrire comme application d'une fonction `btree_fold` de type

```
'a -> ('a -> 'b -> 'a -> 'a) -> 'b btree -> 'a
```

Par exemple, les fonctions `btree_size` et `btree_height` peuvent se redéfinir en :

```
let btree_size = btree_fold 1 (fun x r y -> x + y + 1)
let btree_height = btree_fold 1 (fun x r y -> 1 + max x  y)
```

Écrivez la fonction `btree_fold`.

2. Redéfinissez de la même façon les fonctions `btree_sum` et `btree_prod`.

3. Redéfinissez de la même façon la fonction `btree_to_list`.

4. Redéfinissez de la même façon la fonction `btree_twist`.

5. Écrivez une fonctionnelle `btree_map` telle que `btree_map f t` applique la fonction `f` à toutes les étiquettes de l'arbre `t`. Votre solution devra se présenter comme une application de `btree_fold`.

Exercice 9.16 1. Écrire une fonction `btree_forall : ('a -> bool) -> 'a btree -> bool` qui vérifie si toutes les étiquettes d'un arbre vérifient un prédicat donné.

2. Écrire une version de `btree_for_all` utilisant `btree_fold`.

3. Est-ce aussi efficace que votre première version ?

Exercice 9.17 1. Reprendre l'exercice précédent en remplaçant « toutes les étiquettes » par « au moins une étiquette ».

2. Utiliser votre fonction pour tester si une valeur `x` apparaît dans un arbre.

Exercice 9.18 Pour cet exercice, on considère des arbres dont les nœuds internes **et** les feuilles portent des valeurs de type `int`. Définir un type qui permet de représenter ces arbres.

Écrire une fonction qui, étant donné un entier $h \geqslant 0$, renvoie un arbre parfait de hauteur h contenant tous les entiers de 1 à $2^{h+1} - 1$.

Aide : Vous pourrez avoir recours à une fonction récursive auxiliaire appropriée.

Exercice 9.19 1. Définir un type `bool_exp` pour représenter des expressions booléennes sans variables, c'est-à-dire avec uniquement :

- les constantes `Vrai` et `Faux`,
- les opérateurs logiques `Et`, `Ou`, `Non`, `Ou_exclusif`, `Implique`, `Equiv`.

2. Écrire une fonction `evaluer e` qui évalue une telle expression.

Exercice 9.20 On travaille avec des arbres dont seules les feuilles portent des étiquettes. On les représente par le type suivant :

```
type 'a ltree = L of 'a | N of 'a ltree * 'a ltree
```

On considère les fonctions suivantes :

```
let rec fringe t =
match t with
| L a -> [a]
| N(t1,t2) -> fringe t1 @ fringe t2

let same_fringe t1 t2 = fringe t1 = fringe t2
```

1. Que retourne la fonction `fringe` ?
2. Étudier les problèmes d'efficacité de `fringe`, et proposer une amélioration basée sur une fonction auxiliaire appropriée.
3. Même question (en plus difficile) pour `same_fringe`.

Exercice 9.21 Dans cet exercice on travaille avec des arbres dont les nœuds peuvent avoir un nombre arbitraire de fils. Pour les représenter, on utilise le type `'a utree` suivant :

```
type 'a utree = Node of 'a * 'a utree list
```

Ainsi, chaque nœud a une étiquette et une *liste* d'enfants (de longueur arbitraire). En particulier, avec cette représentation, une feuille est un nœud dont la liste d'enfants est *vide*.

Dans les questions qui suivent, il est fortement recommandé d'utiliser les fonctions `List.fold_left` et `List.fold_right` disponibles dans OCaml pour gérer la liste d'enfants d'un nœud donné.

1. Écrire une fonction `utree_height` de type `'a utree -> int` qui retourne la hauteur de son arbre argument.
2. Écrire une fonction `utree_size` de type `'a utree -> int` qui retourne le nombre total de nœuds de son arbre argument.
3. Écrire une fonction `utree_arity` de type `'a utree -> int` qui retourne le nombre maximal de fils qu'un nœud peut avoir dans son arbre argument.
4. Écrire une fonction `is_ranked` de type `'a utree -> bool` qui renvoie `true` si chaque nœud interne dans son arbre argument a le même nombre de fils, et `false` sinon.
5. Écrire une fonction `utree_infix` de type `'a utree -> 'a list` qui renvoie la liste des étiquettes de son arbre argument en ordre infixe. Il est autorisé d'utiliser la concaténation de listes.
6. ♠ Écrire une seconde version de `utree_infix` qui n'utilise pas la concaténation de listes.

Exercice 9.22 ♠ Écrivez une fonction `hauteur: 'a btree -> int` récursive terminale, qui calcule la hauteur d'un arbre. On pourra utiliser une fonction auxiliaire dont l'un des arguments est de type `(int * 'a btree) list`.