Examen du cours “Introduction à la Programmation Fonctionnelle” 2019/2020 – première session – Durée: 2h00

Exercice 1. (Des expressions et des types, 10 minutes)

1. Quel est le type de l'expression `fun x -> x + 1` ?
2. Quel est le type de l'expression `fun x -> (x, x)` ?
3. Quel est le type de l'expression `fun x y -> Some (x, y)` ?
4. Quel est le type de l'expression `(fun x -> (x, x)) 0` ?
5. Quel est le type de l'expression `(fun x y -> (x, y)) 0 1` ?
6. Quel est le type de l'expression `(fun x y -> (x, y)) 0` ?
7. Quel est le type de l'expression `fun f x y -> f y x` ?
8. Donnez une expression de type `int * int -> int * int`.
9. Donnez une expression de type `(int -> int) -> (int -> int)`.
10. Donnez une expression de type `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)`.

□

Exercice 2. (Correction de programmes, 30 minutes)

Les définitions suivantes sont incorrectes, corrigez-les ! Elles peuvent être incorrectes parce qu'elles sont rejetées par le compilateur ou bien parce qu'elles implémentent incorrectement leurs spécifications. Indiquez pour chacune d'elles comment modifier le code. Si l'erreur est détectée par le compilateur alors indiquez la forme du message d'erreur qu'il produit selon vous.

1. La fonction suivante attend une valeur `d` de type `'a` et une valeur `o` de type `'a option`. Si `o` est de la forme `Some v` alors elle renvoie `v`, sinon elle renvoie `d`. Ainsi, `default 0 None = 0` et `default 0 (Some 1) = 1`.

```
let default d o =
  match o with
  | Some v -> d
  | None -> v
```

2. La fonction suivante attend une liste d'entiers `l` et renvoie `true` si et seulement si `l` est triée en ordre strictement croissant. Ainsi, nous avons `is_sorted [1; 0] = false` tandis que `is_sorted [73; 99; 131] = true`.

```
let rec is_sorted = function
| [] -> true
| a :: b :: l -> a < b && is_sorted l
```

3. La fonction suivante attend un opérateur binaire `add` de type `'a -> 'a -> 'a`, une valeur `zero` de type `'a` et une liste `[x1; x2; ..; xN]` pour produire la valeur `add x1 (add x2 (... xN))`. Si la liste est vide, on renvoie `zero`. Si la liste est réduite à un élément `x` alors la fonction renvoie `x`. Ainsi, `reduce ( + ) 0 [1;2;3] = 6`, `reduce ( * ) 1 [] = 1` et `reduce ( @ ) [] [[1]] = [1]`.

```
let rec reduce add zero = function
  | [] -> zero
  | a :: b :: bs -> add a (add b bs)
  | [a] -> a
```

4. La fonction suivante `all_addition` attend une liste d'entiers `l` et un entier `n` et produit tous les couples `(x, y)` tels que `x <> y`, `x` et `y` sont dans `l` et `x + y <= n`. Ainsi, `all_addition [1;2;3] 3 = [(1, 2); (2, 1)]`.

```
let combine choices f = List.flatten (List.map f choices)
let all_addition l n =
  combine l (fun x ->
    combine l (fun y ->
      if x + y <= n then [] else [(x, y)]
    ))
```

Rappel : `List.flatten : 'a list list -> 'a list`, `List.map : ('a -> 'b) -> 'a list -> 'b list`.

□

Exercice 3. (Récursion terminale, 10 minutes)
Voici une fonction qui filtre les éléments d'une liste `l` pour ne retenir que les éléments vérifiant le prédicat `p`. Il n'est pas obligatoire de maintenir l'ordre des éléments de la liste `l` dans la liste résultat.

```
let rec filter p l =
  match l with
  | [] -> []
  | x :: xs -> if p x then x :: filter p xs else filter p xs
```

1. Quel est le type de l'argument `p` ?
2. Quel est le type de la fonction `filter` ?
3. Pourquoi `filter` n'est-elle pas récursive terminale ?
4. Proposez une fonction `filter'` qui respecte la même spécification que `filter` mais qui est récursive terminale.

□

Exercice 4. (Problème : Différentes implémentations des séquences (1h10))
Une séquence d'éléments de type `'a` est un type `'a t` équipé des opérations suivantes :

```
(** `destruct l` renvoie `None` si la séquence `l` est vide ou bien `Some (x, xs)` si `l` est une séquence
    qui commence par `x` et se poursuit par la séquence `xs`. *)
val destruct : 'a t -> ('a * 'a t) option

(** `cons x xs` est une séquence débutant par `x` et suivie par `xs`. *)
val cons : 'a -> 'a t -> 'a t

(** `empty` est la séquence vide. *)
val empty : 'a t

(** `concat xs ys` est la séquence qui débute par la séquence `xs` et se poursuit par la séquence `ys`. *)
val concat : 'a t -> 'a t -> 'a t

(** `map f xs` est la séquence formée des éléments de `xs` sur lesquels on a appliqué la fonction `f`. *)
val map : ('a -> 'b) -> 'a t -> 'b t
```

Des séquences implémentées par des listes Supposons que `type 'a t = 'a list`.

1. Proposez une implémentation pour les fonctions `destruct`, `cons`, `empty`, `concat` et `map`.
2. Quelle est la complexité algorithmique en pire cas de la fonction `concat` ? de la fonction `map` ? de la fonction `destruct` ?

Des séquences implémentées par des arbres Supposons maintenant que `'a t` est défini comme suit :

```
type 'a t =
| Empty                 (* La séquence vide. *)
| Single of 'a          (* La séquence formée d'un unique élément. *)
| Concat of 'a t * 'a t (* La séquence formée d'une séquence suivie d'une autre. *)
```

4. Proposez une implémentation pour les fonctions `destruct`, `cons`, `empty`, `concat` et `map`.
5. Quelle est la complexité algorithmique en pire cas de la fonction `concat` ? de la fonction `map` ? de la fonction `destruct` ?

Des séquences paresseuses Supposons maintenant que `'a t` est défini comme suit :

```
type 'a t =
| Empty                       (* La séquence vide. *)
| Cons of 'a * (unit -> 'a t) (* Un élément puis la suite est obtenue en appliquant une fonction. *)
```

6. Proposez une implémentation pour les fonctions `destruct`, `cons`, `empty`, `concat` et `map`.
7. Quelle la complexité algorithmique en pire cas de la fonction `concat` ? de la fonction `map` ? de la fonction `destruct` ?

Déforestation En absence d'effets de bord, il est possible de remplacer toutes les expressions de la forme `map f (map g l)` par une expression de la forme `map (fun x -> f (g x)) l` car elles sont équivalentes. Ce procédé s'appelle une optimisation par déforestation.

8. Pourquoi la déforestation est-elle qualifiée d'optimisation ?
9. Pour quelle(s) implémentation(s) des séquences présentée(s) ci-dessus est-elle pertinente ? Justifiez votre réponse.

□