Test 2 : Induction et fonctions d'ordre supérieur

Algorithmes, types de données et preuves, Année 2014/2015

1 Vérification de types

Faites la vérification de type des expressions suivantes, en développant un arbre de dérivation :

1. `fun (g: 'a -> 'b) -> fun (f: 'c -> 'a) -> fun (x: 'c) -> g (f x)`
2. `fun (g: 'a -> 'b) -> fun (f: 'a -> 'a) -> fun (x: 'a) -> (g (f (f x)))`

2 Induction

2.1 Induction sur des nombres naturels

1. Donnez une définition récursive de la fonction de puissance a^n (où n est un nombre naturel)
2. Démontrez, par induction sur les nombres naturels, que
$$a^m \times a^n = a^{m+n}$$
3. Démontrez, par induction sur les nombres naturels, que
$$(a^m)^n = a^{m \times n}$$

Dans les preuves, vous pouvez vous appuyer sur des propriétés bien connues de l'addition et de la multiplication (mais non pas de la puissance ...). Donnez une justification précise de chaque étape de votre preuve.

2.2 Induction sur des listes

Nous avons vu en cours/ TD les fonctions suivantes :
— `listmap f xs`, qui applique la fonction `f` à chaque élément de la liste `xs`.
— `xs @ ys`, qui concatène les listes `xs` et `ys`.
— `length xs`, qui calcule la longueur de la liste `xs`.
Démontrez, par induction sur les listes, les propriétés suivantes :

1. `length (listmap f xs) = length xs`
2. `listmap f (xs @ ys) = (listmap f xs) @ (listmap f ys)`

3 Fonctions d' ordre supérieur

1. Écrire une fonction `exists` qui teste s'il y a un élément dans une liste qui satisfait une propriété.
Exemple d'appliation :
```
# exists (fun x -> x mod 2 = 0) [1;2;3;4] ;;
- : bool = true
# exists (fun x -> x mod 2 = 0) [1;3;5] ;;
- : bool = false
```
2. En utilisant cette fonction, comment est-ce qu'on peut tester
   (a) s'il y a un nombre < 10 dans la liste $[13; 5; 17]$
   (b) s'il y a une paire dans la liste $[(1, 2); (4, 5)]$ dont le premier composant est supérieur au deuxième composant

(c) si un élément de la liste $[false; true; false]$ est *true*

3. Comment est-ce qu'on peut définir la fonction `exists` à l'aide de la fonction `foldr` vue en TD5 ?
4. Définissez la fonction `forall` qui teste si tous les éléments d'une liste satisfont une propriété. Donnez deux définitions : une définition récursive (commme pour `exists`), et une définition non-récursive à l'aide de `exists`.

   Exemple d'application :

   ```
   # forall (fun x -> x mod 2 = 0) [1;2;3;4] ;;
   - : bool = false
   # forall (fun x -> x mod 2 = 0) [2;4] ;;
   - : bool = true
   ```

5. Comment est-ce qu'on peut définir la fonction `selection` à l'aide de la fonction `foldr` ? On rappelle la définition récursive de cette fonction :

   ```
   let rec selection p = function
       [] -> []
     | x :: xs ->
       if (p x) then x::(selection p xs) else (selection p xs)
   ```

4 Arbres binaires

Pour des arbres binaires de type `'a bintree` :

```
type 'a bintree =
    Leaf of 'a
  | Node of 'a  * 'a bintree * 'a bintree
```

définissez une fonction `depth` qui calcule la profondeur d'un arbre, c'est-à-dire, la longueur de la branche la plus longue de l'arbre.

5 Séquences infinies

1. Définissez en Caml une fonction `zip` qui prend deux listes `xs` et `ys` (de type `'a list`) et les combine en alternance (à commencer par `xs`) pour produire une nouvelle liste. Dès que l'une des listes devient vide, on concatène le rest de l'autre liste.

   Exemple : `zip [0; 2; 4; 6; 8; 10] [1; 3; 5]` donne `[0; 1; 2; 3; 4; 5; 6; 8; 10]`

   Note : Le filtrage se fait "en parallèle" sur les deux listes `xs` et `ys`.
2. Définissez une fonction `zipq` qui a un fonctionnement analogue sur des séquences potentiellement infinies (de type `'a seq`).