Principes de Programmation
TD2

11 février 2019

Exercice 1 (Fold).
Les fonctions `foldl` et `foldr` sont définies sur les listes comme suit :

```
foldr   :: (a -> b -> b) -> b -> [a] -> b
foldr _ x []     = x
foldr f x (y:l) = f y (foldr f x l)

foldl   :: (b -> a -> b) -> b -> [a] -> b
foldl _ x []     = x
foldl f x (y:l) = foldl f (f x y) l
```

1. Que calculent les programmes suivant : [1]

```
sum :: [Int] -> Int
sum = foldl (+) 0
```

Reponse:
La somme des éléments de la liste (existe vraimement en haskell)

```
calculatrice :: Int
calculatrice = foldr (\f x -> f x) 6 [(+3),(*10),(+1)]
```

Reponse:
73

```
calculatrice :: Int
calculatrice = foldl (\x f -> f x) 6 [(+3),(*10),(+1)]
```

Reponse:
91

2. Quelle est la différence entre `foldl` et `foldr` ?
Reponse:
`foldl` applique `f` de droite à gauche et `foldr` de gauche à droite :

```
foldl f 0 [1,2,3] ~= f (f (f 0 1) 2) 3
foldr f 0 [1,2,3) ~= f 1 (f 2 (f 3 0))
```

1. L'opérateur (++) : : [a]->[a]->[a] est la concaténation de listes et l'opérateur ($) :: (a->b)->a->b est l'application.

3. Prouvez que pour toute liste finie on a l'égalité :

```
foldl (+) x l ~= foldr (+) x l
```

Reponse:
Par récurtion sur la taille de la liste

```
-- cas de base []
foldl (+) x [] ~= x                          -- def foldl
               ~=  foldr (+) x []            -- def foldr
-- cas (y:l)
foldl (+) x (y:l) ~=  foldl (+) ((+) x y) l  -- def foldl
                  ~=  foldl (+) (x + y) l    -- def (+)
                  ~=  y + (foldl (+) x l)    -- Lemme 1 (voir plus bas)
                  ~=  y + (foldr (+) x l)    -- hypothese de recursion
                  ~=  (+) y (foldr (+) x l)  -- def (+)
                  ~=  foldr (+) x (y:l)      -- def foldr
```

Lemme 1 : `foldl (+)` (x + y) l = y + (foldl (+) x l) + :

```
-- cas de base []
foldl (+) (x+y) [] ~= x+y                                  -- def foldl
                   ~= y+x                                  -- commutativite
                   ~= y + (foldl (+) x l)                  -- def foldl
-- cas (z:l)
foldl (+) (x + y) (z:l) ~= foldl (+) ((+) (x + y) z) l     -- def foldl
                        ~= foldl (+) ((x + y) + z) l       -- def (+)
                        ~= foldl (+) (x + (y + z)) l       -- associativite
                        ~= foldl (+) (x + (z + y)) l       -- commutativite
                        ~= foldl (+) ((x + z) + y) l       -- associativite
                        ~= y + (foldl (+) (x + z) l)       -- hypothese de recur
                        ~= y + (foldl (+) ((+) x z) l)     -- def (+)
                        ~= y + (foldl (+) x (z:l))         -- def foldl
```

4. (Bonus) Prouvez que la traduction suivante est correcte.

```
foldr'' f x l = foldl (\g z -> g . (f z)) id l x
```

Reponse:
Dans le cas général c'est impossible. En effet, sur une liste infinie, comme `[1..]`, foldr termine (au sens où on trouve le premier élément) alors que foldl diverge, donc les deux traductions diverges. On se contente donc de prouver l'équivalence pour le cas des listes finies.
La preuve pour `foldr'` est dificile. celle pour `foldr''` est plus simple. On va donc fair celle-ci.
On prouve, plus généralement que pour toute liste `l::[a]`, toute fonction `f::b->a->b`, tout terme `x::b` et toute fonction `h::b->b`, on a :

```
foldl (\g z -> g . (f z)) h l x  ~=  h (foldr f x l)
```

Pour ca, on procède par induction sur la taille de la liste `l` (suposée finie). Soit elle est vide et alors :

```
foldl (\g z -> g . (f z)) h [] x  ~=  h x
                                  ~=  h (foldr f x [])
```

Soit elle est non vide et alors :

```
foldl (\g z -> g . (f z)) h (y:l) x
  ~= foldl (\g z -> g . (f z)) ((\g z -> g . (f z)) h y) l x
  ~= foldl (\g z -> g . (f z)) (h.(f y)) l x
  ~= (h.(f y)) (foldr f x l)
  ~= h (f y (foldr f x l))
  ~= h (foldr f x (y:l))
```

5. (Bonus) Prouvez que la traduction suivante de `foldl` à partire de `foldr` est correcte.

```
foldl' f x l = foldr (\x g y -> g (f y x)) id l z
```

Reponse:

On prouve, que pour toute liste `l::[a]`, toute fonction `f::b->a->b`, tout terme `x::b`, on a :

```
foldr (\x g y -> g (f y x)) id l z  ~=  foldl f z l
```

Pour ca, on procède par induction sur la taille de la liste `l` (suposée finie).
Soit elle est vide et alors :

```
foldr (\x g y -> g (f y x)) id [] z
  ~=  id z
  ~=  z
  ~=  foldl f z []
```

Soit elle est non vide et alors :

```
foldr (\x g z' -> g (f z' x)) id (y:l) z
  ~= (\x g z' -> g (f z' x)) y (foldr (\x g z' -> g (f z' x)) id l) z
  ~= foldr (\x g y' -> g (f y' x)) id l (f z y)
  ~= foldl f (f z y) l
  ~= foldl f z (y:l)
```