Universidad Nacional Autónoma de México
Facultad de Ciencias
Programación Declarativa

Tarea 3 Parte Teórica

Ángel Iván Gladín García
No. cuenta: 313112470
angelgladin@ciencias.unam.mx

17 de Marzo 2020

Definiciones de funciones

```haskell
map :: (a -> b) -> [a] -> [b]
map f []     = []                -- (map.1)
map f (x:xs) = f x : map f xs    -- (map.2)
```

```haskell
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x    -- (flip.1)
```

```haskell
(++) :: [a] -> [a] -> [a]
(++) [] ys = ys                  -- ((++).1)
(++) (x:xs) ys = x : xs ++ ys    -- ((++).2)
```

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v                          -- (foldr.1)
foldr f v (x:xs) = f x (foldr f v xs)     -- (foldr.2)
```

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ v [] = v                          -- (foldl.1)
foldl f v (x:xs) = foldl f (f v x) xs     -- (foldl.2)
```

```haskell
reverse :: [a] -> [a]
reverse [] = []                         -- (reverse.1)
reverse (x:xs) = reverse xs ++ [x]      -- (reverse.2)
```

```haskell
tails :: [a] -> [[a]]
tails [] = [[]]                     -- (tails.1)
tails (x:xs) = (x:xs):tails xs      -- (tails.2)
```

1. Demuestra las siguientes propiedades de los operadores de plegado:

   a) `foldr f e . map g = foldr (f . g) e`

   Demostración.

```haskell
-- Caso ([])
    (foldr f e . map g) []        -- Caso ([])
        = foldr f e (map g [])    -- Aplicación de función
        = foldr f e []            -- Por (map.1)
        = e                       -- Por (foldr.1)

    foldr (f . g) e []            -- Caso ([])
        = e

-- Caso (x:xs)
    (foldr f e . map g) (x:xs)                    -- Caso (x:xs)
        = foldr f e (map g (x:xs))                -- Aplicación de función
        = foldr f e ((g x) : (map g xs)))         -- Por (map.2)
        = f (g x) (foldr f e (map g xs))          -- Por (foldr.2)
        = f (g x) ((foldr f e . map g) xs))       -- Composición de funciones
        = f (g x) ((foldr (f . g) e) xs))         -- Hipótesis de inducción
        = f (g x) (foldr (f . g) e xs)            -- Aplicación de función
        = (f . g) x (foldr (f . g) e xs)          -- Composición de función
        = foldr (f . g) e (x:xs)                  -- Por (foldr.2)
```

□

   b) `foldl f e xs = foldr (flip f) e (reverse xs)`

   Demostración.

```haskell
-- Caso ([])
    foldl f e []     -- Caso ([])
        = e          -- Por (foldl.1)

    foldr (flip f) e (reverse [])    -- Caso ([])
        = foldr (flip f) e []        -- Por (reverse.1)
```

```
        = e                          -- Por (foldr.1)

-- Caso (x:xs)
    foldl f e (x:xs)                             -- Caso (x:xs)
        = foldl f (f e x) xs                     -- Por (foldl.2)
        = foldr (flip f) (f e x) (reverse xs)    -- Por hipótesis de inducción

    foldr (flip f) e (reverse (x:xs))
        = foldr (flip f) e (reverse xs ++ [x])                 -- Por (reverse.2)
        = foldr (flip f) (foldr (flip g) e [x]) (reverse xs)   -- Por inciso (1.c)
        = foldr (flip f) (f e x) (reverse xs)                  -- Por (*)

    -- Teniendo que:
    foldl f e xs = foldr (flip f) e (reverse xs)    -- Q.E.D.

-- Donde:
    foldr (flip f) e [x] = f e x    -- (*)

    foldr (flip f) e [x]
        = (flip f) x (foldr (flip f) e [])    -- Por (foldr.2)
        = f (foldr (flip f) e []) x           -- Por (flip.1)
        = f e x                               -- Por (foldr.1)
```

□

c) `foldr f e (xs ++ ys) = foldr f (foldr f e ys) xs`

Demostración.

```
-- Caso ([])
    foldr f e ([] ++ ys)    -- Caso ([])
        = foldr f e ys      -- Por ((++).1)

    foldr f (foldr f e ys) []    -- Caso ([])
        = foldr f e ys           -- Por (foldr.1)

    -- Teniendo que se cumple el caso ([]) por la igualdad anterior.

-- Caso (x:xs)
    foldr f e ((x:xs) ++ ys)                 -- Caso (x:xs)
        = foldr f e (x : (xs ++ ys))         -- Por ((++).2)
        = f x (foldr f e (xs ++ ys))         -- Por (foldr.2)
        = f x (foldr f e (xs ++ ys))         -- Por (foldr.2)
        = f x (foldr f (foldr f e ys) xs)    -- Por hipótesis de inducción
        = foldr f (foldr f e ys) x           -- Por (foldr.2)
```

□

2. Considera el siguiente tipo de dato algebraico en Haskell para definir árboles binarios.

```
data Tree a = Void | Node (Tree a) a (Tree a)
```

Y la función `foldT` que define el operador de plegado para la estructura Tree, definido como sigue:

```haskell
foldT :: (b -> a -> b -> b) -> b -> Tree a -> b
foldT _ v Void = v
foldT f v (Node t1 r t2) = f t1' r t2'
    where t1' = foldT f v t1
          t2' = foldT f v t2
```

a) Da en términos de una función `h` el patrón encapsulado por el operador `foldT`.

```haskell
h :: a -> Tree a -> a
h v Void = v
h v (Node t1 r t2) = ...
```

b) Enuncia y demuestra la propiedad Universal del operador `foldT`, basándote en la Propiedad Universal vista en clase sobre el operador `foldr` de listas.

☹

3. Calcula una definición eficiente para `scanr` partiendo de la siguiente:

```haskell
scan r f e = map (foldr f e) . tails
```

```haskell
scan r f e = map (foldr f e) . tails

-- Caso ([])
    scan r f []     -- Caso ([])
        = [e]

-- Caso (x:xs)
    scan r f e (x:xs)                                  -- Caso (x:xs)
        = map (foldr f e) (tails (x:xs))               -- Por especificación
        = map (foldr f e) ((x:xs):tails xs)            -- Por (tails.2)
        = foldr f e (x:xs):map (foldr f e) (tails xs)  -- Por (map.2)
        = f x (foldr f e xs):scan f e xs               -- Por (foldr.2) y especific.
        = f x (head ys):ys where ys = scanr f e xs     -- Afirmando que:
                                             -- `foldr f e xs = head (scanr f e xs)`
```

4. Considera la siguiente definición de la función `cp` que calcula el producto cartesiano.

```haskell
cp :: [[a]] -> [[a]]
cp = foldr f e
```

a) En la definición anterior ¿Quiénes son `f` y `e`?

```haskell
-- `f` es:
op xs xss = [x:ys | x <- xs, ys <- xss]

-- `e` es:
[[]]

-- Teniendo así que:
cp = foldr op [[]]
    where op xs xss = [x:ys | x <- xs, ys <- xss]
```

b) Dada la siguiente ecuación

```haskell
length . cp = product . map length
```

en donde `length` calcula la longitud de una lista y `product` regresa el resultado de la multiplicación de todos los elementos de una lista. Demuestra que la ecuación es cierta, para esto es necesario reescribir ambos lados de la ecuación como instancias de `foldr` y ver que son idénticas.

```haskell
-- Usando el teorema de fusión podemos expresar a `length . cp` como:
    length . cp = foldr h b

-- Teniendo que:
    length [[]] = b
    length (op xs xss) = h xs (length xss)

-- La primera ecuación da `b = 1` y como
    length (op xs xss) = length xs * length xss

-- la segunda ecuación da
    h = (*) . length

-------------------------------

-- Usando el teorema de fusión podemos expresar a `map length` como:
    map length = foldr f []

-- Donde `f xs ns = length xs:ns`. Una definición más corta es
    f = (:) . length

-------------------------------

-- Tenemos que:
    product . map length = foldr h b

-- Siempre y cuando `product` se estricto y
    product [] = b
    product (length xs:ns) = h xs (product ns)

-- La primera ecuación da `b = 1`, y como
    product (length xs:ns) = length xs * product ns

-- La segunda ecuación da
    h = (*) . length

-------------------------------

-- Las dos definiciones de `h` y `b` son idénticas
```