Universidad Nacional Autónoma de México
Facultad de Ciencias
Programación Declarativa

Tarea 2 Parte Teórica

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

24 de Febrero 2020

1. Demuestra las siguientes propiedades

```
sum . map double = double . sum
sum . map sum = sum . concat
sum . sort = sum
```

en donde `double` se define de la siguiente manera:

```
double :: Integer -> Integer
double x = 2 * x
```

y `sum`, `map`, `sort` y `concat` son las definidas en el `Prelude` de Haskell.

Solución: Se escribirán las definiciones de las funciones `sum`[1], `map`, `sort`[2] y `concat` para ser usadas en la demostración.

```
sum :: [Int] -> Int
sum []     = 0              -- (sum.1)
sum (x:xs) = x + sum xs     -- (sum.2)
```

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

---

[1] Se escribirá la definición de la función para facilitar la demostración porque la definición de `sum` en Haskell usa `foldr` aquí para mayor referencia `https://hackage.haskell.org/package/base-4.12.0.0/docs/src/Data.Foldable.html#sum`.

[2] No escribí la definición completa del sort porque hay muchas implementaciones, solo asumiré que funciona.

```
sort :: Ord a => [a] -> [a]
sort []     = []      -- (sort.1)
sort (x:xs) = ...     -- (sort.2)
```

```
concat :: [[a]] -> [a]
concat []        = []                   -- (concat.1)
concat (xs:xss) = xs ++ concat xss     -- (concat.2)
```

Demostración. Por inducción sobre la lista xs.

```
sum . map double = double . sum
```

- xs = []

```
(sum . map double) xs
        = (sum . map double) []      -- Tomando xs = []
        = sum []                     -- Por (map.1)
        = 0                          -- Por (sum.1)
        = double 0                   -- Porque double 0 = 0
        = (double . sum) []          -- Por (sum.1)
        = (double . sum) xs          -- Tomando [] = xs
```

- xs = y:ys

```
(sum . map double) xs
        = (sum . map double) y:ys                 -- Tomando xs = y:ys
        = sum ((double y) : (map double ys))      -- Por (map.2)
        = (double y) + ((sum . map double) ys)    -- Por (sum.2)
        = (double y) + ((double . sum) ys)        -- Por hipótesis de inducción
        = double (y + (sum ys))                   -- Factorizando
        = double (sum (y:ys))                     -- Por (sum.2)
        = (double . sum) (y:ys)                   -- Por composición de función
        = (double . sum) xs                       -- Tomando y:ys = xs
```

□

Demostración. Por inducción sobre la lista xs.

```
sum . map sum = sum . concat
```

- xs = []

```
(sum . map sum) xs
        = (sum . map sum) []      -- Tomando xs = []
        = sum (map sum [])        -- Asociatividad de funciones
        = sum []                  -- Por (map.1)
        = sum (concat [])         -- Por (concat.1)
        = (sum . concat) []       -- Composición de funciones
        = (sum . concat) xs       -- Tomando [] = xs
```

- xs = ys:yss

```
(sum . concat) xs
        = (sum . map) ys:yss                  -- Tomando xs = y:ys
        = sum (concat ys:yss)                 -- Asociatividad de funciones
        = sum (ys ++ (concat yss))            -- Por (concat.2)
        = (sum ys) + sum (concat yss)         -- `sum` se distribuye sobre (++)
        = (sum ys) + ((sum . concat) yss)     -- Composición de funciones
        = (sum ys) + ((sum . map sum) yss)    -- Por hipótesis de inducción
        = (sum ys) + (sum (map sum yss))      -- Rescribiendo la composición
        = sum ((sum ys) : (map sum yss))      -- Por (sum.2)
        = sum (map sum (ys : yss))            -- Por (map.2)
        = (sum . map sum) (ys : yss)          -- Composición de funciones
        = (sum . map sum) xs                  -- Tomando ys:yss = xs
```

□

Demostración. Por inducción sobre la lista xs.

```
sum . sort = sum
```

- xs = []

```
(sum . sort) xs
        = (sum . sort) []    -- Tomando xs = []
        = sum (sort [])      -- Composición de funciones
        = sum []             -- Por (sort.1)
        = sum xs             -- Tomando [] = xs
```

- xs = y:ys

```
(sum . sort) xs
        = (sum . sort) y:ys    -- Tomando xs = y:ys
        = sum (sort y:ys)      -- Composición de funciones
        = sum y':ys'           -- Por (sort.2) Es la misma lista salvo orden
        = sum xs               -- Tomando y:ys = xs
```

□

2. En Haskell la función `take` n toma los primeros n elementos de una lista, mientras que `drop` n regresa la lista sin los primeros n elementos de ésta. Demuestra o da un contraejemplo de cada una de las siguientes propiedades.

```
take n xs ++ drop n xs = xs
take m . take n = take (min m n)
map f . take n = take n . map f
filter p . concat = concat . map (filter p)
```

<u>Solución</u>: Todas la propiedades enunciadas arriba son verdaderas, por lo tanto se deben demostrar.

Se escribirán las definiciones de las funciones `take` y `drop` para ser usadas en la demostración:

```haskell
take :: Int -> [a] -> [a]
take 0 _      = []                   -- (take.1)
take _ []     = []                   -- (take.2)
take n (x:xs) = x : take (n-1) xs    -- (take.3)
```

```haskell
drop :: Int -> [a] -> [a]
drop 0 xs     = xs               -- (drop.1)
drop _ []     = []               -- (drop.2)
drop n (_:xs) = drop (n-1) xs    -- (drop.3)
```

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

```haskell
min :: Int -> Int -> Int
min a b = if a < b then a else b    -- (min.1)
```

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []                                            -- (filter.1)
filter p (x:xs) = if p x then x : filter xs else filter p xs    -- (filter.2)
```

Demostración. Por inducción sobre la lista `xs` y `n`.

```haskell
take n xs ++ drop n xs = xs
```

- `n = 0`

```haskell
take n xs ++ drop n xs
    = take 0 xs ++ drop 0 xs    -- Sustituyendo n = 0
    = [] ++ drop 0 xs           -- Por (take.1)
    = [] ++ xs                  -- Por (drop.1)
    = xs                        -- Por (++.1)
```

- Suponer que es válida para n y la lista `xs`.
- `xs = []` y `n = m+1`

```haskell
take n xs ++ drop n xs
    = take (m+1) [] ++ drop (m+1) []    -- Sustituyendo n = 0
    = [] ++ drop (m+1) []               -- Por (take.2)
    = [] ++ []                          -- Por (drop.2)
    = []                                -- Por (++.1)
```

- `xs = y:ys` y `n = m+1`

```
take n xs ++ drop n xs
    = take (m+1) (y:ys) ++ drop (m+1) (y:ys)    -- Sust. n = (m+1) y xs = y:ys
    = (y : (take m ys)) ++ (drop (m+1) (y:ys)   -- Por (take.3)
    = (y : (take m ys)) ++ (drop m ys)          -- Por (drop.3)
    = (y : (take m ys)) ++ (drop m ys)          -- Por (drop.3)
    = y : ((take m ys) ++ (drop m ys))          -- Reescribiendo
    = y : ys                                    -- Hipótesis de inducción
```

□

Figura 1: No acabé todas las demostraciones :(

3. Consideremos la siguiente afirmación

```
map (f . g) xs = map f $ map g xs
```

a) ¿Se cumple para cualquier `xs`? Si es cierta bosqueja la demostración, en caso contrario ¿qué condiciones se deben pedir sobre `xs` para que sea cierta?

Sí se cumple para cualquier `xs`, ya sea la lista con al menos un elemento (o más) o vacía.

Se reescribirá la función de la siguiente manera, la cual es equivalente porque `$` es usada en este caso para evitar los paréntesis quedando así `map f $ map g xs` ≡ `map f . map g xs`. Entonces reescribiendo,

```
map f (map g xs) = map (f . g) xs
```

Demostración.

- Para `xs = []`

```
map f (map g xs)
    = map f (map g [])    -- Tomando xs = []
    = map f []            -- Por (map.1)
    = []                  -- Por (map.1)
    = map (f . g) []      -- Por (map.1)
                          -- Podemos hacer eso porque no le damos
                          -- ningun tratamiento a la función
```

- Para `xs = y:ys`

```
map f (map g xs)
    = map f (map g y:ys)              -- Tomando xs = y:ys
    = map f ((g y) : (map g ys))      -- Por (map.2)
```

```
= (f (g y)) : (map f (map g ys))       -- Por (map.2)
= ((f . g) y) : (map f (map g ys))     -- Composición de funciones
= ((f . g) y) : (map (f . g) ys)       -- Hipótesis de inducción
= map (f . g) y : ys                   -- Por (map.2)
```

□

b) Intuitivamente ¿qué lado de la igualdad resulta mas eficiente? ¿Esto es cierto incluso en lenguajes con evaluación perezosa? justifica ambas respuestas.

Los dos lados de la igualdad son igual de eficientes debido a la evaluación perezosa.

En caso de los lenguajes que no hacen evaluación perezosa, la parte izquierda de la igualdad, ósea `map (f . g) xs`, sería más eficiente porque se recorrería la *lista* una vez, sin embargo en map f $ map g xs se recorrería la *lista* dos veces.