

# Programación declarativa. Tarea 2

## The Imperative is Dark and Full of Terrors

Juan Alfonso Garduño Solís  
Emiliano Galeana Araujo  
Facultad de ciencias, UNAM

8.5

Fecha de entrega: Lunes 24 de febrero de 2020

### 1 Demostraciones de propiedades

- `sum . map double = double . sum`

– Caso base:

```
1 sum . map double [] = double . sum []
2 sum . [] = double . sum []
3 0 = double . sum []
4 sum [] = double sum []
5 double . sum [] = double . sum []
```


– Hipótesis

```
1 sum . map double xs = double . sum xs
```

– Paso inductivo

– Por demostrar

```
1 sum . map double (x:xs) = double . sum (x:xs)
2 -- Definicion de aplicar double a la cabeza de x:xs.
3 sum . [double x] ++ map double xs = double . sum (x:xs)
4 -- Definicion de aplicar sum ([double x] es igual a [2*x])
5
6 2*x + sum . map double xs = double . sum (x:xs)
7 -- Hipotesis.
8 2*x + double . sum xs = double . sum (x:xs)
9 -- Defincion de double inversa.
10 double x + sum xs = double . sum (x:xs)
11 -- Metemos x a la funcion sum.
12 double . sum (x:xs) = double . sum (x:xs)
```



- `sum . map sum = sum . concat`

– Caso base:

```

1      sum . map sum [] = sum . concat []
2      sum [] = sum . concat []
3      0 = sum . concat []
4      sum [] = sum . concat []
5      sum . concat [] = sum . concat []

```

– Hipótesis

```

1      sum . map sum xs = sum . concat xs

```

– Paso inductivo

– Por demostrar

```

1      sum . map sum (x:xss) = sum . concat (x:xss)

```

```

1      sum . map sum (x:xss) = sum . concat (x:xss)
2      -- Definicion de aplicar map sum a la cabeza de x:xss.
   Definimos
3      -- sum' x como el resultado de sum x.
4      sum [sum' x] . map sum xss = sum . concat (x:xss)
5      -- Definicion de aplicar sum a una lista con un
6      -- elemento (sum' x).
7      (sum' x) + sum . map xss = sum . concat (x:xss)
8      -- Hipotesis.
9      (sum' x) + sum . concat (xss) = sum . concat (x:xss)
10     -- Metemos la suma a la funcion sum.
11     sum [sum' x] . concat (xss) = sum . concat (x:xss)
12     -- Como [sum' x] es el resultado de la aplicar sum a la
13     -- lista x. Siendo x una lista, podemos hacer lo siguiente.
14     sum . concat (x:xss) = sum . concat (x:xss)

```



• `sum . sort = sum`

– Definicion sort:

```

1      sort [] = []
2      sort [x] = [x]
3      sort xs = mezcla (sort f) (sort s)
4      where (f,s) = parte xs

```

Suponemos que parte está bien definido y parte de (x:xs) regresa una tupla (f,s) donde  $f ++ s = (x : xs)$

– Caso base:

```

1      sum . sort [] = sum []
2      sum [] = sum []

```

– Hipótesis

```

1      sum . sort xs = sum xs

```

– Paso inductivo

– Por demostrar

```

1      sum . sort (x:xs) = sum (x:xs)
2      sum . mezcla (sort f) (sort s) = sum (x:xs)
3      -- Por la propiedad demostrada en clase sabemos que
4      -- sum . mezcla (xs ys) = sum (xs ys), entonces
5      sum . (sort f) ++ (sort s) = sum (x:xs)
6      -- Por la otra propiedad demostrada en clase sabemos

```

```

7      -- que: sum (xs ++ ys) = sum xs + sum ys, entonces:
8      sum (sort f) + sum (sort s) = sum (x:xs)
9      -- aplicamos la hipotesis de induccion:
10     sum f + sum s = sum (x:xs)
11     -- Aplicamos la segunda propiedad demostrada en
12     -- clase hacia el otro lado
13     sum (f ++ s) = sum (x:xs)
14     -- Sabemos que f ++ s = (x:xs) porque como se
15     -- establecio al principio, parte esta bien definida
16
17     y
18
19     -- terminamos
20     sum (x:xs) = sum (x:xs)

```



Donde, double se define de la siguiente manera:

```

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

```

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

## 2 Función take

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 esta.

Definiciones:

– take:

```

1  take _ [] = []
2  take 0 (x:xs) = []
3  take n (x:xs) = x:take (n-1) xs

```

– drop:

```

1  drop _ [] = []
2  drop 0 xs = xs
3  drop n (x:xs) = drop (n-1) xs

```

– ++:

```

1  ++ [] _ = _
2  ++ (x:xs) l = x: xs ++ l

```

– map:

```

1  map _ [] = []
2  map f (x:xs) = f x : map f xs
3

```

– filter:

```

1  filter _ [] = []
2  filter p (x:xs)
3  | p x      = x : filter p xs
4  | otherwise = filter p xs

```

– concat:

```

1  concat [] = []
2  concat (xs:xss) = xs ++ concat xss

```

Demuesrta o da un contraejemplo:

```
1 take n xs ++ drop n xs = xs
```

– Caso base:

```
1 take n [] ++ drop n [] = []
2 [] ++ [] = []
3 [] = []
```

– Hipótesis:

```
1 take n xs ++ drop n xs = xs
```

– Paso inductivo:

– Por demostrar:

```
1 take n (x:xs) ++ drop n (x:xs) = (x:xs)
2 (x : take n-1 xs) ++ (drop n-1 xs) = (x:xs)
3 -- Por definicion de take podemos sacar la x, entonces
4 x:(take n-1 xs ++ drop n-1 xs) = (x:xs)
5 -- Aplicamos la hipotesis de induccion
6 x:xs = x:xs
```

```
1 take m . take n = take (min m n)
```

– Caso base:

```
1 take m . take n [] = take (min m n) []
2 take m . [] = take (min m n) []
3 [] = take (min m n) []
4 -- A partir de la lista vacia construimos el lado derecho.
5 take (min n m) [] = take (min n m) []
6
```

– Hipótesis:

```
1 take m . take n = take (min m n)
2
```

– Paso Inductivo:

– Por demostrar

```
1 take m . take n = take (min m n)
```

–  $n < m$ :

```
1 take m . take n (x:xs) = take (min m n) (x:xs)
2 -- Por la definicion de take
3 take m . x : take n-1 xs = take (min m n) (x:xs)
4 -- Volvemos a aplicar la definicion de take
5 x: take m-1 . take n-1 xs = take (min m n) (x:xs)
6 -- Aplicamos la hipotesis
7 x: take (min n-1 m-1) xs = take (min m n) (x:xs)
8 -- Aplicamos en sentido opuesto la definicion de take
9 take (min n-1 m-1)+1 (x:xs) = take (min m n) (x:xs)
10 --Como suponemos que n < m
11 take n-1+1 (x:xs) = take n (x:xs)
12 take n (x:xs) = take n (x:xs)
```

–  $n > m$ :

```

1  take m . take n (x:xs) = take (min m n) (x:xs)
2  -- Por la definicion de take
3  take m . x : take n-1 xs = take (min m n) (x:xs)
4  -- Volvemos a aplicar la definicion de take
5  x: take m-1 . take n-1 xs = take (min m n) (x:xs)
6  -- Aplicamos la hipotesis
7  x: take (min n-1 m-1) xs = take (min m n) (x:xs)
8  -- Aplicamos en sentido opuesto la definicion de take
9  take (min n-1 m-1)+1 (x:xs) = take (min m n) (x:xs)
10 -- Como suponemos que n < m
11 take m-1+1 (x:xs) = take m (x:xs)
12 take m (x:xs) = take m (x:xs)
13

```

• `map f . take n = take n . map f`

– Caso base:

```

1  map f . take n [] = take n . map f []
2  map f [] = take []
3  [] = []

```

– Hipótesis:

```

1  map f . take n xs = take n . map f xs

```


– Paso Inductivo:

– Por demostrar

```

1  map f . take n (x:xs) = take n . map f (x:xs)
2  map f . x : take (n-1) xs = take n . map f (x:xs)
3  f x : map f . take (n-1) (xs) = take n . map f (x:
    xs)
4  -- Por hipotesis de induccion
5  f x : take n-1 . map f xs = take n . map f (x:xs)
6  -- Aplicamos take de derecha a izquierda
7  take n (f x : map f xs) = take n . map f (x:xs)
8  -- Aplicamos map de derecha a izquierda
9  take n . map f (x:xs) = take n . map f (x:xs)

```

1 `filter p . concat = concat . map (filter p)` 

Esto es falso verdadero ya que en GCHi indica un error por el operador (.); Sin embargo con \$ se cumple, por lo que vamos a suponer que se usa \$ en lugar de (.) para solucionar el ejercicio.

• – Caso base:

```

1  filter p $ concat [] = concat $ map (filter p) []
2  filter p $ [] = concat $ []
3  [] = []

```

– Hipótesis:

```

1  filter p $ concat xss = concat $ map (filter p) xss

```

– Paso Inductivo:

```

1      filter p $ concat (xs:xss) = concat $ map (filter p) (xs
   :xss)
2      filter p $ xs ++ concat xss = concat $ map (filter p) (
   xs:xss)
3      -- La demostracion de que el filter distribuye se deja
4      -- como ejercicio al lector xD. Suponemos que distribuye
5      filter p xs ++ filter p $ concat xss = concat $ map (
   filter p) (xs:xss)
6      -- Por hipotesis de induccion
7      filter p xs ++ concat $ map (filter p) xss = concat $
   map (filter p) (xs:xss)
8      -- Concat al reves
9      concat $ filter p xs : map (filter p) xss = concat $ map
   (filter p) (xs:xss)
10     -- Map al reves
11     concat $ map (filter p) (xs:xss) = concat $ map (filter
   p) (xs:xss)
12

```

– Por demostrar



### 3 Función map

Consideremos la siguiente afirmación

```

1      map (f . g) xs = map f $ map g xs
2

```

- (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?

Sospechamos fuertemente que si se cumple para cualquier xs, pues hicimos varias pruebas, pero no demostramos para todos los tipos.

```

1      map (f . g) xs = map f $ map g xs
2

```

- Caso base:

```

1      map (f . g) [] = map f $ map g []
2      map f (map g [])
3      map f $ map g []
4

```

- Hipótesis

```

1      map (f . g) xs = map f $ map g xs
2

```

- Paso inductivo

– Por demostrar

```

1      map (f . g) (x:xs) = map f $ map g (x:xs)
2

```

```

1      map (f . g) (x:xs) = map f $ map g (x:xs)
2      map f (map g (x:xs)) = map f $ map g (x:xs)
3      map f (g x : map g xs) = map f $ map g (x:xs)
4      f (g x) : map f (map g xs) = map f $ map g (x:xs)

```

```

5      -- Hipotesis
6      f (g x) : map f $ map g xs = map f $ map g (x:xs)
7      map f (g x) : $ map g xs = map f $ map g (x:xs)
8      map f $ map g (x:xs) = map f $ map g (x:xs)
9

```

- (b) Intuitivamente, ¿Qué lado de la igualdad resulta más eficiente? ¿Esto es cierto incluso en lenguajes con evaluación perezosa? Justifica tu respuesta.

Intuitivamente, el lado derecho resulta más eficiente, pues solo tendríamos que resolver:

```

1      map g xs
2

```

Y poder pasar el resultado, ya que hacer la composición del otro lado, intuitivamente no se ve tan eficiente.

Para el caso de lenguajes con evaluación perezosa no creemos que sea cierto, pues le conviene al lenguaje hacer la composición para ver que pueda proceder a aplicarla y posteriormente aplicarla, ya que un lenguaje con evaluación perezosa no hace nada hasta que tenga que hacerlo.