Programación declarativa. Tarea 2

The Imperative is Dark and Full of Terrors

Juan Alfonso Garduño Solís
Emiliano Galeana Araujo

Facultad de ciencias, UNAM

Fecha de entrega: Lunes 24 de febrero de 2020

1 Demostraciones de propiedades

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

  - Caso base:

    ```
    sum . map double [] = double . sum []
    sum . [] = double . sum []
    0 = double . sum []
    sum [] = double sum []
    double . sum [] = double . sum []
    ```

  - Hipótesis

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

  - Paso inductivo

    - Por demostrar

      ```
      sum . map double (x:xs) = double . sum (x:xs)
      ```

    ```
    sum . map double (x:xs) = double . sum (x:xs)
    -- Definicion de aplicar double a la cabeza de x:xs.
    sum . [double x] ++ map double xs = double . sum (x:xs)
    -- Definicion de aplicar sum ([double x] es igual a [2*x])
    .
    2*x + sum . map double xs = double . sum (x:xs)
    -- Hipotesis.
    2*x + double . sum xs = double . sum (x:xs)
    -- Defincion de double inversa.
    double x + sum xs = double . sum (x:xs)
    -- Metemos x a la funcion sum.
    double . sum (x:xs) = double . sum (x:xs)
    ```

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

  - Caso base:

```
        sum . map sum [] = sum . concat []
        sum [] = sum . concat []
        0 = sum . concat []
        sum [] = sum . concat []
        sum . concat [] = sum . concat []
```

- Hipótesis

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

- Paso inductivo
  - Por demostrar

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

```
        sum . map sum (x:xss) = sum . concat (x:xss)
        -- Defincion de aplicar map sum a la cabeza de x:xss.
  Definimos
        -- sum' x como el resultado de sum x.
        sum [sum' x] . map sum xss = sum . concat (x:xss)
        -- Definicion de aplicar sum a una lista con un
        -- elemento (sum' x).
        (sum' x) + sum . map xss = sum . concat (x:xss)
        -- Hipotesis.
        (sum' x) + sum . concat (xss) = sum . concat (x:xss)
        -- Metemos la suma a la funcion sum.
        sum [sum' x] . concat (xss) = sum . concat (x:xss)
        -- Como [sum' x] es el resultado de la aplicar sum a la
        -- lista x. Siendo x una lista, podemos hacer lo siguiente.
        sum . concat (x:xss) = sum . concat (x:xss)
```

-
```
    sum . sort = sum
```

- Definicion sort:

```
        sort [] = []
        sort [x] = [x]
        sort xs = mezcla (sort f) (sort s)
        where (f,s) = parte xs
```

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

- Caso base:

```
          sum . sort [] = sum []
          sum [] = sum []
```

- Hipótesis

```
          sum . sort xs = sum xs
```

- Paso inductivo
  - Por demostrar

```
            sum . sort (x:xs) = sum (x:xs)
            sum . mezcla (sort f) (sort s)= sum (x:xs)
            -- Por la propiedad demostrada en clase sabemos que
            -- sum . mezcla (xs ys) = sum (xs ys), entonces
            sum . (sort f) ++ (sort s) = sum (x:xs)
            -- Por la otra propiedad demostrada en clase sabemos
```

```
            --  que: sum (xs ++ ys) = sum xs + sum ys, entonces:
            sum (sort f) + sum (sort s) = sum (x:xs)
            -- aplicamos la hipotesis de induccion:
            sum f + sum s = sum (x:xs)
            -- Aplicamos la segunda propiedad demostrada en
            -- clase hacia el otro lado
            sum (f ++ s) = sum (x:xs)
            -- Sabemos que f ++ s = (x:xs) porque como se
            -- establecio al principio, parte esta bien definida
    y
            -- terminamos
            sum (x:xs) = sum (x:xs)
```

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.

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:

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

- drop:

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

- ++:

```
++ [] _ = _
++ (x:xs) l = x: xs ++ l
```

- map:

```
map _ [] = []
map f (x:xs) = f x : map f xs

```

- filter:

```
filter _ [] = []
filter p (x:xs)
| p x       = x : filter p xs
| otherwise = filter p xs
```

- concat:

```
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

Demuesrta o da un contraejemplo:

- ```
  take n xs ++ drop n xs = xs
  ```

  - Caso base:

    ```
    take n [] ++ drop n [] = []
    [] ++ [] = []
    [] = []
    ```

  - Hipótesis:

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

  - Paso inductivo:

    - Por demostrar:

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

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

  - Caso base:

    ```
    take m . take n [] = take (min m n) []
    take m . [] = take (min m n) []
    [] = take (min m n) []
    -- A partir de la lista vacia construimos el lado derecho.
    take (min n m) [] = take (min n m) []

    ```

  - Hipótesis:

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

    ```

  - Paso Inductivo:

    - Por demostrar

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

    - $n < m$:

      ```
      take m . take n (x:xs) = take (min m n) (x:xs)
      -- Por la definicion de take
      take m . x : take n-1 xs = take (min m n) (x:xs)
      -- Volvemos a aplicar la definicion de take
      x: take m-1 . take n-1 xs = take (min m n) (x:xs)
      -- Aplicamos la hipotesis
      x: take (min n-1 m-1) xs = take (min m n) (x:xs)
      -- Aplicamos en sentido opuesto la definicion de take
      take (min n-1 m-1)+1 (x:xs) = take (min m n) (x:xs)
      --Como suponemos que n < m
      take n-1+1 (x:xs) = take n (x:xs)
      take n (x:xs) = take n (x:xs)
      ```

- $n > m$:

```
take m . take n (x:xs) = take (min m n) (x:xs)
-- Por la definicion de take
take m . x : take n-1 xs = take (min m n) (x:xs)
-- Volvemos a aplicar la definicion de take
x: take m-1 . take n-1 xs = take (min m n) (x:xs)
-- Aplicamos la hipotesis
x: take (min n-1 m-1) xs = take (min m n) (x:xs)
-- Aplicamos en sentido opuesto la definicion de take
take (min n-1 m-1)+1 (x:xs) = take (min m n) (x:xs)
--Como suponemos que n < m
take m-1+1 (x:xs) = take m (x:xs)
take m (x:xs) = take m (x:xs)

```

- ```
  map f . take n = take n . map f
  ```

  - Caso base:

    ```
    map f . take n [] = take n . map f []
    map f [] = take []
    [] = []
    ```

  - Hipótesis:

    ```
    map f . take n xs= take n . map f xs
    ```

  - Paso Inductivo:

    - Por demostrar

      ```
      map f . take n (x:xs)= take n . map f (x:xs)
      map f . x : take (n-1) xs = take n . map f (x:xs)
      f x : map f . take (n-1) (xs) = take n . map f (x:
      xs)
      -- Por hipotesis de induccion
      f x : take n-1 . map f xs = take n . map f (x:xs)
      -- Aplicamos take de derecha a izquierda
      take n (f x : map f xs) = take n . map f (x:xs)
      -- Aplicamos map de derecha a izquierda
      take n . map f (x:xs) = take n . map f (x:xs)
      ```

```
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:

  ```
  filter p $ concat [] = concat $ map (filter p) []
  filter p $ [] = concat $ []
  [] = []
  ```

- Hipótesis:

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

- Paso Inductivo:
  ```

```
            filter p $ concat (xs:xss) = concat $ map (filter p) (xs
    :xss)
            filter p $ xs ++ concat xss = concat $ map (filter p) (
    xs:xss)
            -- La demostracion de que el filter distribuye se deja
            -- como ejercicio al lector xD. Suponemos que distribuye
            filter p xs ++ filter p $ concat xss = concat $ map (
    filter p) (xs:xss)
            -- Por hipotesis de induccion
            filter p xs ++ concat $ map (filter p) xss = concat $
    map (filter p) (xs:xss)
            -- Concat al reves
            concat $ filter p xs : map (filter p) xss = concat $ map
     (filter p) (xs:xss)
            -- Map al reves
            concat $ map (filter p) (xs:xss) = concat $ map (filter
    p) (xs:xss)

```

  - Por demostrar

3 Función map

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?

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

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

```

- Caso base:

```
          map (f . g) [] = map f $ map g []
          map f (map g [])
          map f $ map g []

```

- Hipótesis

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

```

- Paso inductivo
  - Por demostrar

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

```

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

```
            -- Hipotesis
            f (g x) : map f $ map g xs = map f $ map g (x:xs)
            map f (g x) : $ map g xs = map f $ map g (x:xs)
            map f $ map g (x:xs) = map f $ map g (x:xs)

```

(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:

```
    map g xs

```

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 combiene 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.