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 . 0 = 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:xs) = sum . concat (x:xs)
      ```

    ```
        sum . map sum (x:xs) = sum . concat (x:xs)
        -- Defincion de aplicar map sum a la cabeza de x:xs. Definimos sum' x como
        el resultado de sum x.
        sum [sum' x] . map sum xs = sum . concat (x:xs)
        -- Definicion de aplicar sum a una lista con un elemento (sum' x).
        (sum' x) + sum . map xs = sum . concat (x:xs)
        -- Hipotesis.
        (sum' x) + sum . concat (xs) = sum . concat (x:xs)
        -- Metemos la suma a la funci n sum.
        sum [sum' x] . concat (xs) = sum . concat (x:xs)
        -- 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:xs) = sum . concat (x:xs)
    ```

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

  - Caso base:

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

  - Hipótesis

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

  - Paso inductivo

    - Por demostrar

      ```
          sum . sort (x:xs) = sum (x:xs)
      ```

```haskell
sum . sort (x:xs) = sum (x:xs)

```

Donde, `double` se define de la siguiente manera:

```haskell
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. Demuesrta o da un contraejemplo:

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

3 Función map

Consideremos la siguiente afirmación

```haskell
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?

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