Programación Declarativa, 2021-1
Nota de clase 8: Conceptos Avanzados sobre Funciones*

Manuel Soto Romero

5 de noviembre de 2020
Facultad de Ciencias UNAM

Cuando revisamos los Fundamentos Teóricos de la Programación Funcional, veíamos que en el Cálculo λ, las funciones podían tomar a otras funciones como argumento e incluso devolver otras funciones como resultado. En Haskell, las funciones se comportan de la misma manera que en el Cálculo λ. Llamamos a este tipo de funciones *funciones de orden superior*. En esta nota revisaremos cómo se aplican todos estos conceptos directamente en Haskell y hablaremos brevemente del concepto de composición que tiene sus fundamentos en la *Teoría de Categorías*.

8.1. Currificación

Al igual que en el Cálculo λ, las funciones sólo pueden tomar un argumento aunque no lo vemos explícitamente. Esto se debe al azúcar sintáctica que aplica Haskell de manera que nos sea más sencillo entender la sintaxis. Todas las funciones que hemos revisado hasta este momento tienen una currificación implícita, de forma tal que podemos aplicar las funciones parcialmente. Por ejemplo, definamos la función que obtiene el máximo entre dos números enteros: [2]

```
maximo a b | a > b = a | otherwise = b
```

Ahora, realicemos algunas evaluaciones:

```
Prelude> maximo 4 5
5
Prelude> (maximo 4) 5
5
Prelude> let parcial = (maximo 4)
Prelude> parcial 5
5
Prelude> parcial 3
4
```

*Basada en *Learn You a Haskell for Great Good! A Beginner's Guide*

Podemos apreciar a partir de la segunda expresión que ejecutamos, que colocar paréntesis al rededor del nombre de la función y el primer parámetro no altera el resultado. Esto ocurre porque primero se crea una función que toma un sólo argumento que regresará 4 o dicho argumento, dependiendo de cuál sea mayor. Luego es aplicado a esa función y ésta produce el resultado deseado. Lo mismo puede apreciarse en las últimas 3 expresiones, donde almacenamos en la variable `parcial` la función antes mencionada. Vemos que por ejemplo si la llamamos con el argumento 3, funciona correctamente.

Si definiéramos esta función, luciría de esta forma:

```
maximo4 b
   | 4 > b = 4
   | otherwise = b
```

La curificación explica por qué en las funciones, únicamente separamos éstos por flechas, donde los paréntesis son implícitos, justamente porque la aplicación de funciones tiene la mayor precedencia y asocia a la izquierda (como en el Cálculo λ). Por ejemplo, la firma de `maximo` es:

```
maximo :: Int -> Int -> Int
```

Haciendo los paréntesis explícitos, la firma de `maximo` luciría de esta forma:

```
maximo :: Int -> (Int -> Int)
```

En este sentido, dependiendo del contexto, podemos omitir argumentos en algunos casos. Por ejemplo, analicemos la siguiente función:

```
compara :: Int -> Int -> String
compara a b
   | a == b = "Igual"
   | b > a  = "Mayor"
   | b < a  = "Menor"
```

Usando esta definición, podríamos dar una función que indique la relación que hay entre cualquier número y el 1729. Es decir:

```
compara1729 :: Int -> String
compara1729 x = compara 1729 x
```

```
Prelude> compara 2 3
‘‘Mayor’’
Prelude> compara1729 3
‘‘Menor’’
```

Si aplicamos lo que sabemos de currificación en nuestra función `compara1729`, podemos omitir el argumento `x`, pues sabemos que al omitir el segundo argumento de `compara`, se crea una función que espera el segundo valor, que en este caso, se comparará con 1729.

```
compara1729 :: String
compara1729 = compara 1729
```

```
Prelude> compara1729 3
‘‘Menor’’
```

Otro uso que le podemos dar a la currificación, es la definición de funciones infijas. Para seccionar una función, simplemente colocamos ésta entre paréntesis y proporcionamos uno de sus parámetros de un lado. Esto crea una función currificada. Veamos algunos ejemplos:

- Sección usando la división: `divide10 = (/10)`.
- Sección usando la función `elem`: `esMayuscula = (‘elem‘ ['A'..'Z'])`.
- Sección usando la función `compara`: `(2 ‘compara‘)`.

8.2. Funciones de Orden Superior

Las funciones de orden superior más populares y usadas dentro de HASKELL son `map`, `filter` y las funciones de plegado `foldr` y `foldl`. Veamos algunos ejemplos de uso y la implementación de las funciones `map` y `filter`. Más adelante, en otra nota, hablaremos de la llamada Programación Origami relacionada con `foldr` y `foldl`.

8.2.1. La función `map`

Esta función recibe una función `(a → b)` y una lista (`[a]`) y regresa una nueva lista con el resultado de aplicar la función a cada elemento de la lista (`[b]`). La implementación de esta lista se muestra a continuación. Se aprecia cómo se aplica la función `f` a cada elemento de la lista y se aplica recursivamente a la cola. Se aprecia que el primer parámetro en la definición de `map` es una función.

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

```
Prelude> map sqrt [1,4,9,16,25,36,49,64,81,100]
[1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0]
```

8.2.2. La función `filter`

Esta función recibe una función que verifica una propiedad, es decir, que regrese un valor booleano, `(a → Bool)`, una lista (`[a]`) y regresa una nueva lista con aquellos elementos que cumplen la propiedad, es decir, que se reducen a `True` después de aplicar la función (`[a]`). La implementación de esta función se muestra a continuación. Se aprecia cómo la función verifica si se cumple la propiedad (`p x`) en cuyo caso mantiene el elemento en la lista y en caso contrario, lo elimina haciendo la recursión sin pegar dicho elemento.

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

```
Prelude> filter even [1,2,3,4,5,6,7,8,9,10]
[2,4,6,8,10]
```

8.2.3. Definición de funciones de orden superior

Al igual que `map` y `filter`, podemos definir nuestras propias funciones de orden superior. En el preludio de HASKELL se encuentra definida la función `zipWith` que toma una función y dos listas y las concatena aplicando la función entre los correspondientes elementos.

```
Prelude> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
```

Daremos nuestra propia definición de esta función. Hasta este punto hemos definido funciones proporcionando primero su firma (tipo) y después dando la especificación de la misma. Sin embargo, si no estamos seguros de cómo se escribe este tipo, podemos hacerlo en orden inverso y usar el comando `:t` para conocer el tipo de lo que estamos definiendo.

Esta función se comporta muy similar a la concatenación. Haremos entonces nuestra recursión sobre uno de los parámetros, el primero por conveniencia.

```
zipWith' _ [] _ = []
zipWith' f (x:xs) (y:ys) = (f x y):(zipWith' f xs ys)
```

Para saber el tipo hacemos:

```
Prelude> :t zipWith'
zipWith' :: (t1 → t2 → a) → [t1] → [t2] → [a]
```

Como podemos apreciar, la función recibe una función como parámetro, esta función recibe dos parámetros (en realidad uno, pero eso ya lo sabes :D) y el tipo de las dos listas corresponde con éstos, generando una lista con un tipo completamente nuevo.

```
Prelude> zipWith' (++) [[1,2,3],[4,5,6]] [[7,8,9],[10,11,12]]
[[1,2,3,7,8,9],[4,5,6,10,11,12]]
```

8.3. Lambdas

Como vimos en el Cálculo λ, las funciones son anónimas, es decir, no tienen un nombre asociado. Ahora que hemos estudiado a las funciones de orden superior, contar con funciones anónimas tiene todo el sentido del mundo, debido a que es bastante común que se necesite definir una función que no esté incluida en el preludio de Haskell y que únicamente se use para aplicar con la función de orden superior.

Para definir una lambda, se usa la siguiente sintaxis:

```
\<parámetro>* → <cuerpo>
```

Veamos algunos ejemplos:

```
Prelude> zipWith (\a b → (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
Prelude> map (\(a,b) → (a + b)) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

Analicemos un ejemplo con todos los conceptos que hemos revisado hasta ahora.

Ejemplo 8.1. La secuencia de Collatz se define para cualquier número natural de la siguiente forma:

- Si el número es par, lo dividimos entre dos.
- Si es impar, lo multiplicamos por tres y le sumamos uno.
- Continuamos recursivamente con el número resultante.
- La secuencia termina cuando llegamos al número 1.

Por ejemplo, para el número 13, tenemos la siguiente secuencia:

$13, 13 \times 3 + 1$
$13, 40, 40/2$

$13, 40, 20, 20/2$
$13, 40, 20, 10, 10/2$
$13, 40, 20, 10, 5, 5 \times 3 + 1$
$13, 40, 20, 10, 5, 16, 16/2$
$13, 40, 20, 10, 5, 16, 8, 8/2$
$13, 40, 20, 10, 5, 16, 8, 4, 4/2$
$13, 40, 20, 10, 5, 16, 8, 4, 2, 2/2$
$13, 40, 20, 10, 5, 16, 8, 4, 2, 1$

Pregunta: Para cada número entre 1 y 100, ¿Cuántas secuencias tienen una longitud mayor a 15?

Para responder a esta pregunta, primero debemos definir la función que genere la secuencia para un número.

```
collatz :: Int -> [Int]
collatz 1 = [1]
collatz n
   | even n = n:collatz (div n 2)
   | odd n  = n:collatz (n*3+1)
```

Ahora, podemos aplicar un mapeo sobre una rango entre 1 y 100, después sólo filtramos aquellas secuencias que tengan una longitud mayor a 15.

```
sec15 :: Int
sec15 = length (filter mayor15 (map collatz [1..100]))
   where mayor15 l = length l > 15
```

Nota que la definición de `mayor15` podría darse también con lambdas:

```
sec15 :: Int
sec15 = length (filter (\l → length l > 15) (map collatz [1..100]))
```

8.4. Aplicación de funciones

En secciones anteriores, mencionamos que al igual que en el Cálculo λ, en Haskell la aplicación de funciones tiene el nivel de precedencia más alto y asocia a la izquierda. Por ejemplo, usando la función `doble`:

```
doble :: Int  → Int
doble x = 2*x
```

```
Prelude> doble 2 + 3
...
```

¿Cuál debería ser el resultado de esta llamada? ¿Aplicar `(doble 2) + 3`? ¿Aplicar `doble (2+3)`?

Como dijimos, la aplicación de funciones *siempre tiene la mayor precedencia.* Por lo tanto, primero aplicamos la función y luego sumamos `(doble 2) + 3`:

```
Prelude> doble 2 + 3
7
```

¿Que debería devolver la siguiente ejecución?

```
Prelude> doble doble 2
...
```

¿Aplicar `(doble doble) 2`? ¿Aplicar `doble (doble 2)`?

Como dijimos, la aplicación de función asocia a la izquierda. Es decir, si se tienen las funciones `f`, `g` y `h` a evaluar con `x`, la forma de asociar será: `((f g) h) x`.

De forma que el código anterior, lanzaría un error:

```
Ejemplo> doble doble 2
<interactive>:2:1: error: ...
```

Sin embargo, a pesar de esta precedencia y asociatividad, Haskell provee una función muy útil llamada `$`. La función `$` define una aplicación de función pero usando el nivel de precedencia más bajo y asociando a la derecha. Además se usa infijamente. Por ejemplo:

```
Ejemplo> doble $ 2 + 3
10
Ejemplo> doble $ doble $ 3
12
```

- En el primer caso, se toma la suma de mayor precedencia, evaluandose a 5 y finalmente `doble 5` es 10.
- En el segundo caso, se asocia a la derecha, por lo tanto tenemos primero `doble 3` igual a 6 y finalmente `doble 6` igual a 12.

8.5. Composición de funciones

En matemáticas podemos componer dos funciones de forma que se cree una nueva a partir de esas dos de forma que al aplicar esta nueva función con un argumento, se haga una evaluación compuesta.

Es decir, si tenemos una función, digamos `f` y otra función `g`, las componemos y las evaluamos con un argumento `x`, primero evaluaremos `g x` y ese resultado será el que reciba `f`.

Para componer funciones en Haskell se usa la función `.`.

Por ejemplo, supongamos que tengo una lista de números y quiero hacerlos todos negativos. No podemos simplemente multiplicar por `-1` pues los números negativos se volverán positivos. Así, es más conveniente primero calcular el valor absoluto con la función `abs` y después volverlos negativos con la función `negate`.

Una posible solución sería:

```
Prelude> map negate $ map abs [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Esta solución aunque es útil, hace uso dos veces de `map`. Podríamos usar una lambda que agrupe la aplicación de las dos funciones en una:

```
Prelude> map (\x -> negate $ abs x) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Esto es mucho más corto y de hecho es la definición de la composición de funciones. La lambda que dimos en el ejemplo anterior es lo que devuelve la función `.` con la composición de `negate` y `abs`.

```
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

8.5.1. Estilo libre de puntos

Un concepto muy útil resultante de la composición es la definición de funciones usando el estilo libre de puntos[1]. Por ejemplo, en la función `compara1729` que estudiamos en secciones anteriores, eliminamos el parámetro `x` gracias a la currificación. Esto es a lo que conocemos como estilo libre de puntos, pues nos permite no mencionar explícitamente algunos valores. Sin embargo, ¿Cómo reescribiríamos la siguiente expresión usando este estilo?

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

[1] Del inglés *point-free style* o *pointless style*.

No podemos eliminar simplemente la `x` de ambos lados pues si dejamos `(max 50)`, la función `cos` estaría recibiendo como argumento una función y no es posible hacer esto. Lo que haremos entonces, es expresar `fn` como una composición de funciones:

```
fn = ceiling . negate . tan . cos . max 50
```

Observación 8.1. El términos *estilo libre de puntos*, se originó en Topología, una rama de las matemáticas que trabaja con espacios compuestos de puntos y funciones entre estos espacios. De esta forma, una función en estilo de punto libre, es una función que no menciona explícitamente lo puntos (valores) del espacio sobre los que actúa.

Referencias

[1] Miran Lipovaca, *Learn You a Haskell for Great Good! A Beginner's Guide*, Primera edición, No Starch Press, 2011.

[2] Manuel Soto, *Manual de Prácticas para la Asignatura de Programación Declarativa*, Proyecto de Apoyo a la Docencia, Facultad de Ciencias UNAM, 2019.