Introducción a funciones de alto orden

Taller de Álgebra I

Primer cuatrimestre 2018

Mapa del curso (Herramientas computacionales)

- Introducción a la computación
- Problema $\rightarrow$ Algoritmo $\rightarrow$ Programa
- Tipado
- Reducción y órdenes de evaluación
- Recursión sobre números
- Recursión sobre listas
- Pattern matching
- Mini introducción a alto orden (usted está aquí)

Componer funciones

Vimos

- Cómo definir funciones simples.
- Cómo definir funciones partidas.
- Cómo definir funciones recursivas.
- Cómo aplicar funciones.

Veamos lo siguiente

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

g :: Integer -> Integer
g x = x + 1
```

Ahora... ¿podemos componer funciones, sin tener que definir una función auxiliar?
$(f \circ g)(x) = f(g(x))$
¡Sí! para eso usamos la función (.)
¿Qué devuelve esto?

```
> (f.g) 2
> (.) f g 2
```

6
¿Qué tipo tendrá (.)?

Funciones como parámetro

Las funciones pueden **tomar** parámetros de cualquier tipo, ¿no?

```
(.) :: <FuncionF> -> <FuncionG> -> DominioG -> CodominioF
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

Entonces las funciones tienen que tener tipo...

Valores

- Al definir un tipo, estamos definiendo un **conjunto de valores** y las **operaciones** asociadas.
- Tipos: `Integer`, `[[Bool]]`, `(Float, Float, Bool)`, `Integer -> Integer`

¿Qué devuelve como valor de salida (.)?

Una función muy útil

¿Qué hace la siguiente función?

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

Magia

- ¿Cuál es el tipo de `magia`?
- ¿Qué devuelven las siguientes expresiones?
  - `magia not [True, True, False, True]`
  - `magia reverse [[1,2,3], [5,6,7]]`
  - `magia id [1,2,3,4,5,6]`

Esta función `magia` viene programada, se llama `map`.
Dada su utilidad, suele estar disponible en varios lenguajes de programación.

Otro problema frecuente

Prototipo de función muy común

```
func []       = []
func (x:xs) | (alguna condición) = x : func xs
            | otherwise          = func xs
```

Podemos pensar la condición como una función auxiliar `cond :: a -> Bool`.
Por lo tanto, definiendo `func :: (a -> Bool)->[a]->[a]`, podemos cubrir este esquema!
Esto también ya viene dado y es la función `filter`

Ejemplos

- ¿Qué devuelven las siguientes expresiones?
  - `filter esPar [2, 3, 5, 10]`
  - `filter esVacia [[1,2,3], [], [2], [7,4], []]`
  - `filter id [False,True,True,False,True]`

Funciones como salida de otras funciones

Valores

- Al definir un tipo, estamos definiendo un **conjunto de valores** y las **operaciones** asociadas.
- Tipos: `Integer`, `[[Bool]]`, `(Float, Float, Bool)`, `Integer -> Integer`

Las funciones también pueden **devolver** valores de cualquier tipo, ¿no?

```
indeciso :: Integer -> Integer -> ([Integer] -> Integer)
indeciso x y | x > y      = product
             | otherwise  = sum
```

¿Qué hace lo siguiente?

```
Prelude> (indeciso 5 10) [3,3,3]
Prelude> (indeciso 15 5) [3,3,3]
```

Para hacer entre todos

- ¿Qué devuelve Haskell si escribimos `:t (indeciso 5 6)`?
- ¿Qué devuelve Haskell si escribimos `:t (indeciso 5)`?

Aplicación parcial: El secreto de las flechas

¿Alguna vez se preguntaron por qué en Haskell se escribe:

```
f1 :: Int -> Int -> Int
```

en vez de

```
f2 :: (Int, Int) -> Int
```

como en la mayoría de los lenguajes? (ya lo notarán en los demás lenguajes)

¿Por qué pasa lo siguiente?:

```
Prelude> max 4 5
5
Prelude> (max 4) 5
5
```

¿Qué devuelve Haskell si escribimos `:t (max 4)`?

La magia está en los paréntesis

- En Haskell, `t1 -> t2 -> t3 -> t4`
  es equivalente a `t1 -> (t2 -> (t3 -> t4))`
- De la misma manera `f a b c`
  es equivalente a `((f a) b) c`
- `max :: Integer -> Integer -> Integer` es equivalente a
  `max :: Integer -> (Integer -> Integer)`

Aplicación parcial: El secreto de las flechas

La magia está en los paréntesis

- En Haskell `t1 -> t2 -> t3 -> t4`
  es equivalente a `t1 -> (t2 -> (t3 -> t4))`
- De la misma manera `f a b c`
  es equivalente a `((f a) b) c`

Para pensar

- ¿Si `(+) :: Integer -> Integer -> Integer`, que devuelve `((+) 10)`?
- ¿Qué tipo tiene la función `doble = (*) 2`?
- ¿Qué devuelve `map ((+) 10) [1,2,3]`?
- ¿Qué devuelve `map ((+) 10)`?
- ¿Qué devuelve `map`?

Notación Lambda (ay chalay)

Nombres, ¿para qué?

- ¿Siempre es necesario poner nombre a todas las funciones? ¿Siempre vale la pena?
- ¿No sería cómodo, a veces, usar funciones "anónimas"?
- La notación lambda nos permite hacer exactamente eso.

¿Qué se obtiene al evaluar las siguientes expresiones?

- `Prelude> (\x -> x + 10) 20`
- `Prelude> (\rad -> (4*pi*rad**3)/3) 10`
- `Prelude> map (\x -> x + 10) [1,2,3,4]`
- `Prelude> (\x y -> x + y) 20 30`

¿Qué diferencia hay entre las siguientes definiciones? ¿Qué tipo tienen?

```
suma x y = x + y
suma x = \y -> x + y
suma = \x -> (\y -> x + y)
suma = \x y -> x + y
```

Parcial

Veamos dos ejercicios del parcial resueltos con Alto Orden:

Ejercicio 3

Definimos:

```
esSumaDeCuadradosCon :: Integer -> Integer -> Bool
esSumaDeCuadradosCon n x = esCuadrado x && esCuadrado (n-x)

esSumaDeCuadrados :: Integer -> Bool
esSumaDeCuadrados x = length (filter (esSumaDeCuadradosCon n)
                                     [1..(n-1)]) > 0
```

(Si quiero saber si aplicar `filter` va a dar vacío, es incluso mejor usar `any`)

Ejercicio 5

Definimos:

```
divisores n = filter (\x -> mod n x == 0) [1.. (n-1)]

partes [] = [[]]
partes (x:xs) = map (x:) pxs ++ pxs
                where pxs = partes xs

esSemiperfecto n = elem n (map sum (partes (divisores n)))
```

Fin

- Para más detalles sobre Haskell, pueden chusmear: `http://learnyouahaskell.com/`
- Ante cada nuevo lenguaje, pregúntense qué propiedades de las vistas en el curso posee.