

Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

27 de Marzo de 2018

¿Qué era Haskell?

- Lenguaje funcional puro
- Estáticamente tipado (aunque con polimorfismo)
- Evaluación Lazy
- Llamado así en honor a Haskell Curry.

Utilizaremos la herramienta *ghc* en la materia, junto a su ambiente interactivo, *ghci*.



Repaso: Expresiones y tipos básicos

Tipos elementales

```
1           -- Int
'a'         -- Char
1.2         -- Float
True        -- Bool
[1,2,3]     -- [Int]
(1, True)   -- (Int, Bool)
length      -- [a] -> Int
```

Guardas

```
signo n | n >= 0    = True
        | otherwise = False
```

Pattern matching

```
longitud [] = 0
longitud (x:xs) = 1 + (longitud xs)
```

Repaso: Polimorfismo paramétrico

`todosIguales` es una función que determina si todos los elementos de una lista son iguales entre sí.

Implementar y dar el tipo de la función

```
todosIguales :: ??  
todosIguales = ...
```

- El sistema de tipos de Haskell permite definir funciones para ser usadas con más de un tipo
- Su tipo se expresa con *variables de tipo*



Haskell no necesita que todos los tipos sean especificados a mano ni tampoco requiere anotaciones de tipos en el código. Para eso utiliza un **Inferidor de Tipos** (veremos más en λ -Cálculo).

Funciones de alto orden

Definamos las siguientes funciones

Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

- `mejorSegun :: (a -> a -> Bool) -> [a] -> a`
- Reescribir `maximo` y `listaMasCorta` en base a `mejorSegun`

Funciones sin nombre (*lambdas*)

```
(\x -> x + 1) :: Num a => a -> a  
(\x y -> "hola") :: t1 -> t2 -> [Char]  
(\x y -> x + y) 10 20 ~> 30
```

Currificación y evaluación parcial

Currificación

Correspondencia entre funciones que reciben múltiples argumentos y devuelven un resultado, con funciones que reciben un único argumento y devuelven una función intermedia que completa el trabajo.

- `prod :: (Int, Int) -> Int`
`prod (x, y) = x * y`
- `prod' :: Int -> (Int -> Int)`
`(prod' x) y = x * y`

Currificada o no currificada, esa es la pregunta

¿Cómo descubrir si una función está o no currificada?

Implementar y dar el tipo a las siguientes funciones

- `curry :: ??` que devuelve la versión currificada de una función no currificada
- `uncurry :: ??` que devuelve la versión no currificada de una función currificada

Aplicación parcial

Utilizar las funciones de manera currificada permite aplicación parcial.

No se puede hacer más lento...

Si definimos *succ* de la siguiente manera: `succ x = suma 1 x`

- ¿Cuál es el tipo de *succ*?
- ¿Qué diferencia hay entre `suma 1 3` y `(suma 1) 3`?
- ¿Será posible escribir mejor la definición de *succ*?
- ¿Qué significa `(+)1` en Haskell?

¿Y esto?

`($) :: (a -> b) -> a -> b`

`f $ x = f x`



Implementar las siguientes funciones

- `esMayorDeEdad :: Int -> Bool`
- `inversoMultiplicativo :: Float -> Float`

Funciones de alto orden

- **flip**: invierte los argumentos de una función
ej: `flip (\x y -> x - y) 10 4` reduce a `-6`
- **(.)**: compone dos funciones
ej: `((\x -> x * 4).(\y -> y - 3)) 10` reduce a `28`

Implementar

Las funciones `flip` y `(.)` con sus respectivos tipos.

Ejercicio

ROT13 es un esquema de criptografía muy sencillo que consiste en reemplazar cada caracter con el caracter que aparece 13 lugares después en el alfabeto. Básicamente, $rot13(a) = char((\#a + 13) \bmod 26)$.

- Implementar `rot13 :: Char -> Char` sin escribir variables.

Definición de listas

- Listas por extensión
`[0, 3, 0, 3, 4, 5, 6]`
- Secuencias aritméticas
`[1..4]` `[5, 7..13]`
- Listas por comprensión
`[expresion | selectores, condiciones]`
`[(x, y) | x <- [0..3], y <- [0..3]]`

¿Las listas pueden ser infinitas?

Ejemplo

- `infinitosUnos = 1 : infinitosUnos`
- `naturales = [1..]`
- `multiplosDe3 = [0,3..]`
- `repeat "hola"`
- `primos = [n | n <- [2..], esPrimo n]`

Evaluación Lazy

Modelo de cómputo: Reducción

- Se reemplaza un *redex* (reducible expresion) utilizando las ecuaciones orientadas.
- El redex debe ser una **instancia** del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las correspondientes variables sustituidas.
- El resto de la expresión no cambia.

Para seleccionar el redex: **Orden Normal**, o también llamado **Lazy**



- Se selecciona el redex más externo y más a la izquierda para el que se pueda conocer qué ecuación del programa utilizar.
- En general: primero las funciones más externas y luego los argumentos (sólo si se necesitan).

Ejercicio

Mostrar los pasos necesarios para reducir `nUnos 2`

```
take :: Int -> [a] -> [a]
take 0 l      = []
take n []     = []
take n (x:xs) = x : (take (n-1) xs)

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

Digresión

- ¿Qué sucedería si usáramos otra estrategia de reducción?
- ¿Existe algún término que admita una reducción finita pero para el cual la estrategia lazy no termine?
- Si un término admite otra reducción finita además de la lazy, ¿el resultado de ambas coincide?

Esquemas de recursión sobre listas: Map

`map :: (a -> b) -> [a] -> [b]`

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

O, hablando en francés, la función `map`

- Toma una función que sabe como convertir un tipo `a` en otro `b`,
- Y nos devuelve una función que sabe como convertir listas de `a` en listas de `b`.

```
map f [] = []
```

```
map f (x:xs) = (f x):(map f xs)
```

Definir utilizando map

- `longitudes :: [[a]] -> [Int]`
- `losIesimos :: [Int] -> [[a] -> a]` que devuelve una lista con las funciones que toman los *i*ésimos de una lista.
- `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \dots, i_n]$ y una lista `l`, devuelve la lista $[l_{i_1}, \dots, l_{i_n}]$

Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

O, hablando en francés, la función `filter`

- Toma una función que nos dice si un elemento cumple una condición,
- Y nos devuelve una función que sabe como convertir listas de elementos cualquiera en listas cuyos elementos cumplen la condición deseada.

```
filter p [] = []
```

```
filter p (x:xs) = if p x then x:(filter p xs) else filter p xs
```

Definir utilizando filter

- `deLongitudN :: Int -> [[a]] -> [[a]]`
- `soloPuntosFijos :: [Int -> Int] -> Int -> [Int -> Int]` que toma una lista de funciones y un número n . En el resultado, deja las funciones que al aplicarlas a n dan n .
- `quickSort :: Ord a => [a] -> [a]`

i? i? i? i? i? i? i? i? i? i? i? i? i?