Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

15 de agosto de 2017

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
```

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
```

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 = ...
```

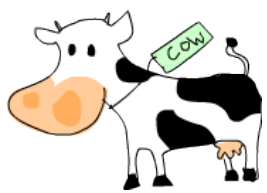
Repaso: Polimorfismo paramétrico

`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*



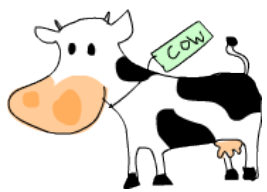
Repaso: Polimorfismo paramétrico

`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).

Listas infinitas

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?

Listas infinitas

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 =`

Listas infinitas

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 = [0..]`
- `multiplosDe3 =`

Listas infinitas

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 = [0..]`
- `multiplosDe3 = [0,3..]`
- `repeat "hola"`
- `primos =`

Listas infinitas

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

Listas infinitas

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 = [0..]`
- `multiplosDe3 = [0,3..]`
- `repeat "hola"`
- `primos = [n | n <- [2..], esPrimo n]`
- ¿Qué sucede al reducir `take 2 infinitosUnos`?
- ¿Qué sucede al reducir `length naturales`?

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.

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

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).

Ejercicios

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
```

Ejercicios

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?

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]`

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 ::`

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`

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
```

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?