Repaso de Haskell

Lógica Computacional

Edgar Quiroz

1 de noviembre de 2021

Índice

1. Características

1.1. ¿Cómo es Haskell?

- Funcional
  - Priorizar la aplicación de funciones
- Puro
  - Llamar dos veces a la misma función dará el mismo resultado
  - Si cada parte funciona, juntas funcionan
- Sistema de tipos muy agradable
  - Permite trabajar con definiciones matemáticas fácilmente

2. Tipos básicos

2.1. Comentarios

```
-- Dos guiones para comentar
{-
Llave guión para varias líneas
-}
```

2.2. Tipos

- `(::)` es el operador para asignar tipos, pero casi siempre Haskell los puede inferir

```
-- hay enteros
n :: Int
n = 5

-- números de punto flotante
x :: Float
x = 5.6

-- caractéres
c :: Char
c = 'c'
```

2.3. Operades aritméticos: `+`, `-`, `*`, `/`, `div`, `mod`, `succ`, `pred`

```
-- se puede usar haskell como una calculadora
suma = 1 + 3
resta = 2 - 10
producto = 4 * 5
division = 64 / 8
modulo = 18 `mod` 13
sucesor = succ 4
predecesor = pred 10
potencia = 3 ^ 2
```

2.4. Operadores lógicos: `&&`, `||`, `not`

```
-- una calculadora con valores booleanos también
conjuncion = True && False
```

```
disyuncion = False || True
negacion = not False
```

2.5. Tuplas

- Acceder elementos (solo para tuplas de dos)

```
-- como lista de longitud fija
tpl :: (Int, Int)
tpl = (1, 2)

primero = fst tpl -- primer elemento
segundo = snd tpl -- segundo elemento
```

2.6. Clases

- El tipo de `5` es `Num p =>p`, donde `p` es un tipo que implemente la clase `Num`.
- Las clases son como interfaces en Java: `Num, Show, Eq, Ord`

```
class Show a where
  show :: a -> String
```

3. Funciones

3.1. Definición

- Nombre, argumentos y cuerpo

```
-- declara funciones es muy fácil
masUno x = x + 1
```

3.2. Tipo

- `->` parece $\mapsto$, el símbolo para denotar el dominio y contradominio de una función en matemáticas

```
-- es buena práctica ponerle el tipo a las funciones
sumaDos :: Int -> Int
sumaDos x = x + 2
```

3.3. Operadores

- Los operadores son funciones infijas, como `+`, `&&`. Sólo pueden contener símbolos. Se deben declarar con paréntesis.

```
-- los símbolos válidos son !#$%&*+./<=>?@\^|-~:
(#) :: Int -> Int -> Int
n # m = m + n * m
```

3.4. Aplicación parcial

- Solo pueden recibir un parámetro

```
-- esta función se aplica un argumento a la vez
sumaMasTres :: Int -> Int -> Int
sumaMasTres x y = x + y + 3
```

- Llamar `sumaMasTres 1 2` crea la función intermedia

```
-- sumaMasTres 1 2 crea
sumaMasTres' :: Int -> Int
sumaMasTres' y = 1 + y + 3
-- y finalmente evalua 1 + 2 + 3 = 6
```

- Y finalmente se evalua

```
-- se puede tener una aplicación parcial explícita
mod5 :: Int -> Int
mod5 = (`mod` 5)
```

3.5. Subexpresiones

- `sumaDos n` se repite, pero se puede guardar en una expresión

```
-- repetitivo
mulTupla :: Int -> (Int, Int) -> (Int, Int)
mulTupla n (a, b) = (sumaDos n * a, sumaDos n * b)
```

- Usando `let x = e1 in e2`

```
-- más compacto
mulTupla' :: Int -> (Int, Int) -> (Int, Int)
mulTupla' n (a, b) =
  let m = sumaDos n in (m * a, m * b)
```

- O `e2 where x = e1`

```
-- una alternativa
mulTupla'' :: Int -> (Int, Int) -> (Int, Int)
mulTupla'' n (a, b) =
  (m * a, m * b) where m = sumaDos n
```

3.6. Funciones anónimas

- Se llaman lambdas. \ se parece a λ

```
-- crear funciones desechables
cubosTupla :: (Int, Int) -> (Int, Int)
cubosTupla (a, b) =
  (cubo a, cubo b)
  where cubo = \x -> x^3
```

3.7. Condiciones

- Ambas ramas deben tener el mismo tipo

```
-- amabas ramas mismo tipo
absoluto :: Float -> Float
absoluto x =
  if x < 0
     then -x
     else x
```

3.8. Casos

- _ se puede usar para descartar el valor de un parámetro

```
-- para evitar ifs anidados
bordeAlfabeto :: Char -> Bool
bordeAlfabeto c =
  case c of
    'a' -> True
    'z' -> True
    _ -> False
```

3.9. Caza de patrones

- Funciones parciales

```
-- otra sintaxis para casos
bordeAlfabeto' :: Char -> Bool
bordeAlfabeto' 'a' = True
bordeAlfabeto' 'z' = True
bordeAlfabeto' _ = False
```

3.10. Recursión

- Pares

```
-- función recursiva sencilla
par :: Int -> Bool
par 0 = True
par 1 = False
par n = par (n - 2)
```

- Números de Fibonacci

```
-- fibonacci
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

3.11. Guardias

- Aplicar función booleana para decidir
- Opción por omisión debe ser la última y usa la palabra reservada `otherwise`

```
-- otra manera de eveitar ifs anidados
grado :: Int -> String
grado g
  | g < 6 = "NA"
  | g <= 12 = "Primaria"
  | g <= 15 = "Secundaria"
  | g <= 18 = "Preparatoria"
  | otherwise = "Terminaste"
```

3.12. Funciones de orden superior

- Las funciones no son valores especiales

```
-- funciones normales
f1 x = x + 1
f2 x = 2 * x
```

- `(.)` es una función que compone funciones

```
-- función hecha de la composición de otras funciones
h = f1 . f2
```

- `($)` aplica funciones de forma infija con la mayor precedencia

```
-- función hecha de la aplicación de otras funciones
k x = h . f1 . f2 $ x
```

4. Listas

4.1. Definición

- Son recursivas
  - [ ] es la lista vacía
  - Si l es una lista y a un elemento, $a : l$ es una lista
- Se denotan entre corchetes

```
-- usando corchetes, como en python
lista = [1, 5, 8, 0]
```

4.2. Caza de patrones

```
multsDeCinco :: [Int] -> [Int]
-- se define para la lista vacía
multsDeCinco [] = []
-- y para elemento seguido de lista
multsDeCinco (x:xs) =
  if x `mod` 5 == 0
     then x:resto
     else resto
  where resto = multsDeCinco xs
```

4.3. Operaciones básicas

- Concatenar, cabeza, rabo, longitud

```
-- algunas operaciones con listas
concatenada = [1, 2, 3] ++ [4, 5, 6]
cabeza = head [1, 2, 3]
rabo = tail [4, 5, 6]
longitud = length [8, 7, 6]
```

4.4. Rangos

- Inicio, (paso), final

```
-- rango
diez = [1..10]

-- rango con paso
paresVeinte = [0, 2..20]
```

- Crear listas infinitas

```
-- no muere porque haskell es perezoso
quince = take 15 [1..]
```

4.5. Comprensión

- Como conjuntos

```
-- lista de los números menores a n al cuadrado
cuadrados :: Int -> [Int]
cuadrados n = [m^2 | m <- [1..n]]

-- factore de un número usando listas por comprensión
facts :: Int -> [Int]
facts n = [m | m <- [1..n], n `mod` m == 0]
```

- QuickSort

```
-- otro ejemplo de listas por comprensión
quickSort [] = []
quickSort (x:xs) =
  let menores = [y | y <- xs, y <= x]
      mayores = [y | y <- xs, y > x]
  in quickSort menores ++ [x] ++ quickSort mayores
```

5. Funciones sobre listas útiles

5.1. Filtrar listas

```haskell
-- solo mayúsculas usando un filtro
mayusculas :: String -> String
mayusculas = filter (\x -> x `elem` ['A'..'Z'])

-- eliminar repeticiones usando un filtro
unicos :: Eq a => [a] -> [a]
unicos [] = []
unicos (x:xs) = x:unicos (eliminaX xs)
  where eliminaX = filter (/=x)

-- otro ejemplo de filtro
mulsFiltro :: [Int] -> [Int]
mulsFiltro = filter ((==0) . (`mod` 5))
```

5.2. Aplicar una función a cada elemento

```haskell
-- sacar el inverso de los números de una lista
inversoRec :: [Int] -> [Int]
inversoRec [] = []
inversoRec (x:xs) = (-x):xs

-- misma función pero usando un mapeo
inversoMap :: Num a => [a] -> [a]
inversoMap = map negate
```

5.3. Juntar todos los elementos con una operación

```haskell
-- suma recursiva
sumaRec :: Num a => [a] -> a
sumaRec [] = 0
sumaRec (x:xs) = x + sumaRec xs

-- suma usando reducción (fold)
sumaFold :: Num a => [a] -> a
sumaFold = foldr (+) 0

-- otro ejemplo de fold
```

```haskell
mayor10 :: [Int] -> Bool
mayor10 = foldr (\x acc -> acc && x > 10) True
```

6. Errores

6.1. Irrecuperables (no realmente)

- No tienen tipo

```haskell
-- explota
dividir n 0 = error "no se puede"
dividir n m = n `div` m
```

7. Definir tipos

7.1. Alias

- Intercambiables con el tipo original
- Mejorar legibilidad

```haskell
-- otro nombre para String
type Nombre = String
```

7.2. Tipos algebraicos

- Contructores y clases derivadas

```haskell
-- expresiones de sumas
data Expr = Val Int |
            Var Nombre |
            Suma Expr Expr deriving Eq
```

- Implementando clases

```haskell
-- función para representar como cadena
instance Show Expr where
  show (Val val) = show val
  show (Var nombre) = nombre
  show (Suma e1 e2) = (show e1) ++ "+" ++ (show e2)
```

7.3. Caza de patrones

- Se puede hacer sobre tipos definidos

```
-- Como se hacían con listas
subst :: Expr -> Nombre -> Int -> Expr
subst (Val v) _ _ = Val v
subst (Var n) m v =
  if n == m
     then (Var m)
     else Var n
subst (Suma e1 e2) m v =
  Suma (subst e1 m v) (subst e2 m v)
```

- Otro ejemplo

```
-- evaluar expresiones
eval :: Expr -> Int
eval (Val val) = val
eval (Var nombre) = 0
eval (Suma e1 e2) = (eval e1) + (eval e2)
```

7.4. Tipos registro

- Azucar para crear funciones de acceso

```
-- un tipo con varios argumentos
data Color = RGB Int Int Int

-- repetitivo
getR :: Color -> Int
getR (RGB r _ _) = r

-- compacto
data RGBA = RGBA {
  r :: Int,
  g :: Int,
  b :: Int,
  a :: Int
}
```

8. Valores opcionales

8.1. `Maybe`

- No hay `null` (no hay referencias)
- Lidiar explícitamente con valores faltantes

```haskell
-- nunca habrá NullPointerException
mensajeSeguro :: Maybe String -> String
mensajeSeguro (Just s) = "El mensaje es:" ++ s
mensajeSeguro Nothing = "Se perdió"
```