Una (h)ojeada a Haskell para la programación lógica

Lógica computacional

22 de marzo de 2024

Mutación de los datos

```
x=1
oldID=id(x)
x=x+1
id(x)==oldID
```

Definiendo funciones en Haskell

- Funciones *curriadas* o *no curriadas* (*curried*, *non-curried*)
- Podemos pasar de una definición a la otra
- Nos ahorra teclear y trabajamos en una idea más básica

Haskell con curry y sin curry

```
suma (x,y) = x+y
inc (x) = suma (x,1)
```

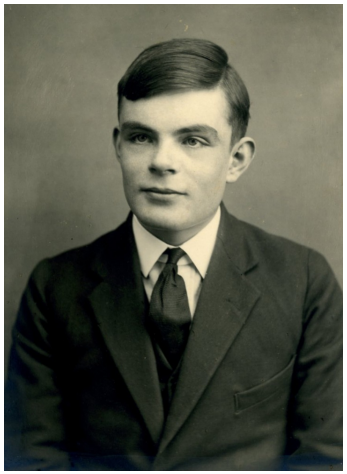
```
suma x y = x+y
inc = suma 1
map inc [1..10]
```

Cálculo λ

(a) Alonzo Church

(b) Emi Leon Post

(c) Haskell Curry



(d) Alan Turing

Figura: Aportaron al cálculo λ, entre muchas personas más. Algunas de estas imágenes cuentan con derechos de autor, son usada con fines educativos.

λ-términos en Haskell

$$suma\ x = \backslash y -> x + y$$
$$suma = \backslash x -> (\backslash y -> x + y)$$
$$suma = \backslash x -> \backslash y -> x + y$$

¿Cuál es el chiste de eso?

$$(+=) = \backslash x -> \backslash y -> x + y$$
$$(+=) = \backslash x\ y -> x + y$$
$$3 +=\ 5$$

Listas

- Conjunto nordenado de objetos no necesariamente del mismo tipo
- Versátiles y base de otro tipo de estructuras
- Los arreglos tienen un nombre y un apuntador
- Listas tienen dos apuntadores

Sumando en una lista

$$sum\ [1.,100]$$

```
sumalista []     = 0
sumalista (n:ns) = n + sumalista ns
```

Prueben $a = [1, 2, 3, 4]$ y $b = ['a', 'b', 'c', 'd']$

Aplicación de funciones en Haskell

Matemática	Haskell
$f(x)$	$f\ x$
$f(x, y)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(y)$	$f\ x\ *\ g\ y$

Comandos para scripts

Comando	Accion
: *load nombre*	Cargar *script*
: *reload*	Recargar script
: *edit nombre*	Cambiar nombre del *script*
: *edit*	Editar el *script*
: *type expr*	Mostrar el tipo de la expresión
:?	Mostrar los comandos
: *q*	Salir de *GHCi*

Cadenas especiales y comentarios

case class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where

- Comentarios ordinarios (máximo una línea), empiezan con $--$.
- Comentarios anidados (multilínea), empiezan con $\{-$ y terminan con $-\}$

Ordenar una lista

```
ordenar [] = []
ordenar (x:xs) = ordenar chico ++ [x] ++ ordenar grande
  where
    chico =[a | a<-xs, a<=x]
    grande = [b | b<- xs, b>x]
```

Prueben $a = [1, 2, 3, 4]$ y $b = ['a', 'b', 'c', 'd']$

Operaciones sobre listas

$$a = [1, 2, 3, 4, 5]$$

$$head\ a$$

$$tail\ a$$

$$init\ a$$

$$last\ a$$

$$a!!2$$

$$take\ 3\ a$$

$$filter\ odd\ a$$

$$drop\ 3\ a$$

$$length\ a$$

$$sum\ a$$

$$product\ a$$

$$[1, 2, 3] + +[4, 5]$$

$$reverse\ a$$

$$null\ a$$

$$minimum\ a$$

$$head\ []\ (¿?)$$

Expresiones condicionales

- Expresiones condicionales

```
absol :: Int -> Int
absol n = if n >= 0 then n else -n
```

- Expresiones condicionales anidadas

```
signi :: Int -> String
signi n = if n < 0 then "negativo" else
            if n == 0 then "cero" else "positivo"
```

Guardas

```
signg :: Int -> String
signg n | n < 0 = "negativo"
        | n > 0 = "positivo"
        |otherwise = "cero"
```

Números primos

```
factores :: Int -> [Int]
factores n = [x | x <- [1..n], n `mod` x == 0]

primo :: Int -> Bool
primo n = factores n == [1,n]

primos :: Int -> [Int]
primos n = [x | x <- [2..n], primo x]
```

Verificador de tautologíoas

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Or Prop Prop
          | Imply Prop Prop
          | Iff Prop Prop
```

Declarando la sustitución

```
type Assoc k v = [(k,v)]
find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k==k']

type Sust = Assoc Char Bool

eval :: Sust -> Prop -> Bool
eval _ (Const b) = b
eval s (Var x) = find x s
eval s (Not p) = not (eval s p)
eval s (And p q) = eval s p && eval s q
eval s (Or p q) = eval s p || eval s q
eval s (Imply p q) = eval s p <= eval s q
eval s (Iff p q) = (eval s p <= eval s q) && (eval s q <= eval s p
```

Las variables en una proposición

```
vars :: Prop -> [Char]
vars (Const _) = []
vars (Var x) = [x]
vars (Not p) = vars p
vars (And p q) = vars p ++ vars q
vars (Or p q) = vars p ++ vars q
vars (Imply p q) = vars p ++ vars q
vars (Iff p q) = vars p ++ vars q
```

Las posibles combinaciones en una sustitución

- Pensando en *veradero* $= 1$ y *falso* $= 0$ es equivalente a contar en binario.
- O se puede hacer de otra forma viendo las regularidades

F	F	F
F	F	T
F	T	F
F	T	T
T	F	F
T	F	T
T	T	F
T	T	T

```
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False:) bss ++ map (True:) bss
          where bss = bools (n-1)
```

Ahora sí a definir todas las posibles sustituciones

- Remover duplicados de la lista de variables
- Generar los posibles valores con *bools*
- Emparejar las lista *zip*

```
rmdumps :: Eq a => [a] -> [a]
rmdumps [] = []
rmdumps (x:xs) = x : filter (/= x) (rmdumps xs)

susts :: Prop -> [Sust]
susts p = map (zip vs) (bools (length vs))
          where vs = rmdumps (vars p)
```

La fase final, el que decide

```
esTaut :: Prop -> Bool
esTaut p = and [eval s p | s <- susts p]

esCont :: Prop -> Bool
esCont p = not (or [eval s p | s <- susts p])
```

Armamos una proposición:

```
p8 :: Prop
p8 = Iff (Var 'P') (Or (Var 'P') (Var 'Q'))
```