Una Lambda aventura en Haskell

Francisco José Cháves

Bloom Consulting

28 de septiembre de 2017

«Las matemáticas son el lenguaje con el que Dios ha escrito el universo»

Galileo Galilei

Cálculo Lambda

- Creado por Alonzo Church en la década de 1930
- Sistema formal
- Turing completo (equivalente a la máquina de Turing)
- "Lenguaje de programación" mas pequeño
  (no tiene palabras reservadas)
- Base de los lenguajes funcionales (lisp, miranda, haskell, ml, caml, ocaml,...)

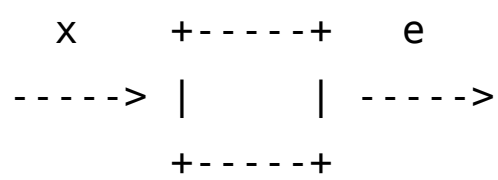
Datos interesantes

- Alonzo Church fue el mentor de doctorado de Alan Turin
- La máquina de turing captura la noción de computación basada en estados.
- El cálculo lambda captura la noción de computación basada en funciones.
- Estas dos nociones de computación son equivalentes (Hipótesis de *Church-Turing*): todo lo que se puede hacer en un computador se puede hacer en el Cálculo Lambda!
- Peter Landin lo utilizó en la especificación, diseño e implementación de lenguajes de programación.
- John McCarty se baso en el cálculo lambda para crear Lisp.

Funciones

Que es una función?

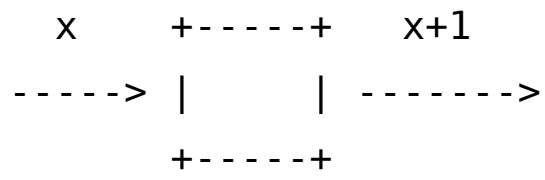
- Para Alonzo Church una caja negra que toma un valor de entrada x y produce una salida (resultado) e:



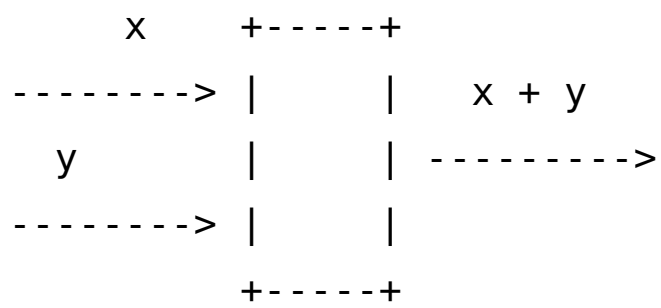
- no se puede mirar que pasa dentro, solo se pueden ver sus entradas y salidas;
- son puras: no contienen estados;
- son anónimas: no tienen nombre.

Ejemplos

- La función sucesor incrementa en 1 el valor de la entrada:



- La función que suma dos valores de entrada x y y:



Expresiones Lambda

Cómo se escriben estas funciones en lambda cálculo?

```
λ x . e
^ ^ ^ ^ expresión de salida (cuerpo de la función)
| | |
| | + Un punto separa la variable de entrada
| |   de la expresión de salida
| |
| +-- variable (parámetro) de entrada
|
+-- El símbolo lambda indica que es una función
```

- La función sucesor:

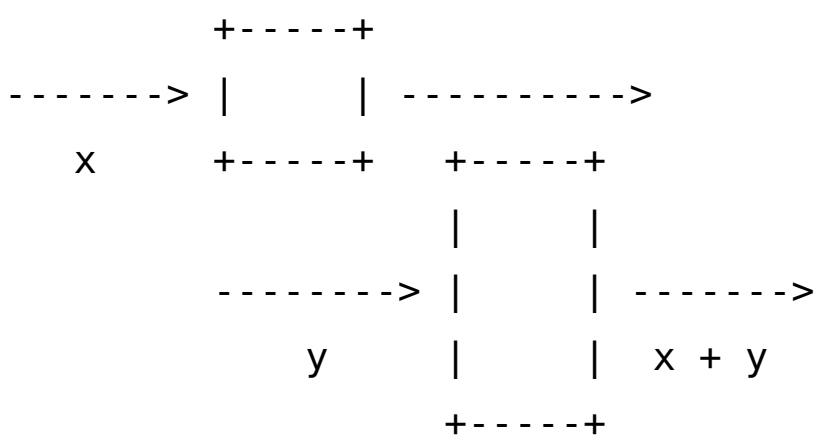
  λ x.x + 1

- Suma de x y y:

  λ x.λ y.x + y

Multiples parámetros

De la notación `λ x.λ y. x + y` se observa que una función de dos parámetros es una función que retorna otra función:



Como funciona?

β-redex

Una expresión que consta de la aplicación de una función a un parámetro se conoce con el nombre de β-redex (expresión reducible por la regla β):

```
(λ x. e) e'
```

Regla β

Reduce un β-redex sustituyendo el parámetro `x` por `e'` en el cuerpo de la función `e`:

```
      +-----+
      V     |
   (λ x. e) e'
= { β }
   (e)[x := e']
```

Ejemplo

```
    (λ x. x + 1) 3
= { β }
    (x + 1)[x := 3]
= { Sustitución }
    (x)[x := 3] (+)[x := 3] (1)[x := 3]
= { Sustitución }
    (3) + 1
= { Aritmética }
    4
```

Resumen

Estos son todos los componentes del lambda cálculo:

$$
\begin{array}{lll}
\Lambda ::= x & x \in V & \text{Variables } x, y, z, \ldots \\
\mid \lambda\, x\, .\, e & x \in V, e \in \Lambda & \text{Funciones o abstracciones} \\
\mid e_1\ e_2 & e_1, e_2 \in \Lambda & \text{Aplicación funcional} \\
\mid (e) & e \in \Lambda & \text{Expresión entre paréntesis}
\end{array}
$$

La aplicación es asociativa a izquierda y tiene la precedencia mas álta:

$$e_1\ e_2\ e_3\ \cdots\ e_n = ((((e_1\ e_2)\ e_3)\ \cdots)\ e_n)$$

La abstracción es asociativa a derecha y tiene la precedencia mas baja:

$$\lambda\, x\, .\, \lambda\, y\, .\, \lambda\, z\, .\, e = \lambda\, x\, .(\, \lambda\, y\, .(\, \lambda\, z\, .\, e))$$

En serio se puede programar con el Cálculo Lambda?

- definiciones?
- booleanos?
- condicionales?
- números?
- recursión?

La idea es codificar en lambda cálculo algo que se comporte igual a estas construcciones!

Lambda cálculo en Haskell

En Haskell se utiliza '\' en lugar de 'λ' y '->' en lugar del '.'

Ejemplos

```
> (\x -> x + 1) 3
4
> (\x -> \y -> x + y) 2 3
5
```

Sintáxis: los parámetros múltiples se pueden escribir en el mismo lambda:

```
> (\x y -> x + y) 2 3
5
```

Definiciones, funciones, valores

Se puede extender el cálculo lambda para dar nombre a las funciones con la instrucción:

```
let name = e in e'
```

en realidad, es azúcar sintáctico, mejora la lectura del lenguaje:

```
   let name = e in e'
= { notación }
   (\name -> e') e
= { β }
   (e')[ name := e ]
```

Definiciones, funciones, valores

Ejemplos:

```
   let id = \x -> x in id id
= { notación }
   (\id -> id id) (\x -> x)
= { β }
   (id id)[ id := \x -> x ]
= { sustitución }
   (\x -> x) (\x -> x)
= { ... }
   ...
```

```
   let one = 1 in one + one
= { notación }
   (\one -> one + one) 1
= { β }
   (one + one)[ one := 1 ]
= { sustitución }
   (1 + 1)
= { ... }
   ...
```

- Note que, en el cálculo lambda, una función es un valor como cualquier otro: (ej. un número, un string, etc.).
- En los scripts de haskell se omiten las palabras reservadas `let` e `in` en las definiciones principales.

Booleanos

A continuación presentamos algunas definiciones de los booleanos en lambda cálculo:

```
true  = \t f -> t
false = \t f -> f
not'  = \b -> b false true
and'  = ?
or'   = ?
if'   = ?
```

- las funciones con '`?`' son *retos* que puede hacer!
- los nombres con apóstrofe (`not'`, `and'`, `or'`, `if'`) son para diferenciarlos de funciones ya definidas en el preludio de Haskell.

Booleanos

Miremos si `not'` funciona correctamente:

```
    not' true
= { definición de not' }
    (\b -> b false true) true
= { β }
    (b false true)[b := true]
= { sustitución }
    true false true
= { definición de true }
    (\t f -> t) false true
= { β }
    (\f -> t)[t := false] true
= { sustitución }
    (\f -> false) true
= { β }
    (false)[ f:= true ]
= { sustitución }
    false
```

```
    not' false
= { definición de not' }
    (\b -> b false true) false
= { β }
    (b false true)[b := false]
= { sustitución }
    false false true
= { definición de false }
    (\t f -> f) false true
= { β }
    (\f -> f)[t := false] true
= { sustitución }
    (\f -> f) true
= { β }
    (f)[ f:= true ]
= { sustitución }
    true
```

Booleanos de Church

Es práctico en Haskell escribir las siguientes funciones:

```
toBool b = b True False

fromBool True = true
fromBool False = false
```

que permiten convertir de un booleano en cálculo lambda a `Bool` y viceversa. i.e. los booleanos de lambda cálculo son isomorfos a `Bool`.

```
> toBool (not' true)
False
> toBool (not' false)
True
```

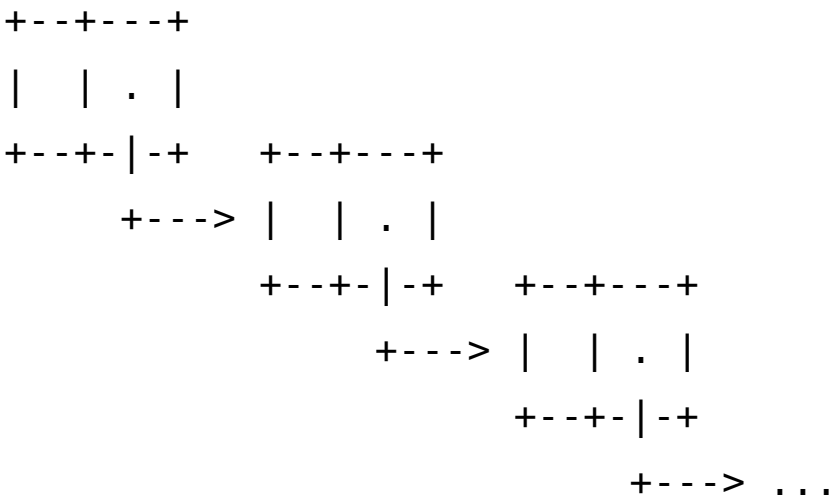
Pares de Church

Los pares son el elemento básico para crear otros tipos de datos:

```
pair = \x y z -> z x y
fst' = \p -> p true
snd' = \p -> p false

toPair p = p (,)
fromPair (x,y) = \z -> z x y
```

Las listas se pueden construir a partir de parejas:



Numerales de Church

Los numerales de Church, corresponden a los números naturales:

```
zero   = \s z -> z
one    = \s z -> s z
two    = \s z -> s (s z)
three  = \s z -> s (s (s z))
...

succ   = \n s z -> s (n s z)
pred   = ?
sum'   = \n m -> \s z -> n s (m s z)
prod   = ?
pow    = ?
minus  = ?

isZero = \n -> n (\x -> false) true
equal  = ?
odd    = ?
even   = ?
```

Isomorfismo con los naturales

Las funciones de conversión entre los naturales de Church y los naturales:

```
toNat n = n (+1) 0

fromNat 0 = zero
fromNat n = succ (fromNat (n - 1))
```

Argumentos

En Haskell los argumentos de las funciones se pueden colocar en la parte izquierda, de donde se tiene la siguiente regla:

```
   f x = e
= { notación }
   f = \x -> e
```

Ejemplo

```
   zero = \s z -> z
= { notación }
   zero s = \z -> z
= { notación }
   zero s z = z
```

Siendo esta última la notación utilizada con mayor frecuencia en Haskell.

Recursión

Consideraciones:

- un loop es algo que se ejecuta de manera cíclica;
- para que se ejecute algo en lambda, debe ser una aplicación funcional;
- para que continue ejecutandose el resultado debe ser equivalente a la misma expresión original.

Con estas consideraciones, se propone la expresión '`Ω = Δ Δ`' de manera que:

```
    Δ Δ    --+
= { ... }    |
    Δ Δ    --+ Un loop infinito!!!
= { ... }    |
    Δ Δ    --+
= { ... }    |
    ...    --+
```

Ups, **`Δ Δ`** es irreducible!

Recursión

Tratemos de encontrar que es Δ:

```
    Δ Δ
= { sustitución  - aplicada de manera inversa }
    (x x)[ x := Δ ]
= { β  - aplicada de manera inversa }
    (λ x.x x) Δ
```

luego **Δ** = λ x.x x
En Haskell, como es fuertemente tipado, **Δ** no se puede definir.

Recursión

El ejemplo de siempre, el factorial!

```
fact n = if n == 0
         then 1
         else n * fact (n - 1)
```

Problema, esta expresión del cálculo lambda, no funciona como se espera:

```
  let fact = \n -> if n == 0
                   then 1
                   else n * fact (n - 1)
  in ...
= { notación }
  (\fact -> ...) (\n -> if n == 0 then 1 else n * fact (n - 1))
```

La variable `fact` no esta definida.

Recursión

Un vistazo desde otro punto de vista:

```
    fact n = if n == 0 then 1 else n * fact (n - 1)
= { notación }
    fact = \n -> if n == 0 then 1 else n * fact (n - 1)
= { sustitución - aplicada en sentido inverso }
    fact = (\n -> if n == 0 then 1 else n * f (n - 1)) [ f := fact ]
= { β - aplicada en sentido inverso }
    fact = (\f n -> if n == 0 then 1 else n * f (n - 1)) fact
```

Bueno, despues de todo este esfuerzo, al menos la expresión

```
(\f n -> if n == 0 then 1 else n * f (n - 1))
```

es una expresión lambda válida!

Recursión

La expresión `fact = (\f n -> if n == 0 then 1 else n * f (n - 1)) fact` es de la forma `fix = f fix`

desenrollandola se tiene:

```
    fix = f fix
= { definición de fix }
    fix = f (f fix)
= { definición de fix }
    fix = f (f (f fix))
= { definición de fix }
    fix = f (f (f (...)))
= { ... }
    ...
```

Faltaria descubrir como expresar `fix` con una expresión del lambda cálculo.
Note que `fix = f fix`, es equivalente a $x = g(x)$, es decir, es un punto fijo de f.

Recursión

Definimos `fix` de manera general, es decir, para cualquier función f:
`fix f = f (fix f)  = f (f (f (...)))`.

- Esto parece similar al loop infinito Ω.
- Definimos Ω' = fix f = Δ' Δ'

```
   fix f = f (fix f)
= { notación }
   fix = \f -> f (fix f)
= { definición de Ω' }
   fix = \f -> f (Δ' Δ')
= { sustitución - aplicada en sentido inverso }
   fix = \f -> (f (x x))[ x: = Δ' ]
= { β }
   fix = \f -> (\x -> (f (x x))) Δ'
= { Δ' = (\x -> (f (x x))) }
   fix = \f -> (\x -> (f (x x))) (\x -> (f (x x)))
```

A la expresión `\f -> (\x -> (f (x x))) (\x -> (f (x x)))` se le conoce con el nombre de combinador Y.

Recursión

Miremos nuevamente:

```
   fix f
= { definición de fix }
   (\f -> (\x -> (f (x x))) (\x -> (f (x x)))) f
= { β, sustitución }
   (\x -> (f (x x))) (\x -> (f (x x)))
= { β }
   (f (x x))[ x := (\x -> (f (x x))) ]
= { sustitución }
   (f ((\x -> (f (x x))) (\x -> (f (x x)))))
= { β, sustitución }
   (f (f ((\x -> (f (x x))) (\x -> (f (x x))))))
= { β, sustitución }
   ...
= { ... }
   (f (f (f (f (...)))))
```

Aunque en Haskell, el combinador Y no se puede definir (no se puede tipar), el operador `fix f = f (fix f)` si!.

Recursión

Finalmente el factorial quedaría:

```
fact = fix (\f n -> if n == 0 then 1 else n * f (n - 1))
```

también se puede con fibonacci:

```
fib = fix (\f n -> if n <= 1 then 1 else f (n - 1) + f (n - 2))
```

Preguntas?