Programación Declarativa 2020-2
Mónadas 101 State

Javier Enríquez Mendoza Favio E. Miranda Perea

Facultad de Ciencias UNAM

28 de mayo de 2020

Introducción

Las mónadas son un concepto abstracto descubierto por los programadores después de varios intentos de refactorizar y generalizar programas.

Se puede introducir el concepto de mónada partiendo de la definición y abstrayendo información a partir de ella, como se hizo en la nota de clase.

En esta presentación se tratará abstraer la definición partiendo de un ejemplo práctico.

Árboles Binarios

Consideremos la siguiente definición de árboles binarios que solo almacenan información en las hojas como un tipo de dato algebraico en Haskell.

```
data Tree a = Leaf a|Node (Tree a)(Tree a)
```

Esta definición tiene dos constructores, el correspondiente a una hoja en donde se almacena información y el correspondiente a un nodo interno que es recursivo pues tiene 2 subárboles.

leaves

Definamos ahora una función que regresa el número de hojas de un árbol

```
leaves :: Tree a -> Int
leaves (Leaf _)    = 1
leaves (Node l r) = (leaves l)+(leaves r)
```

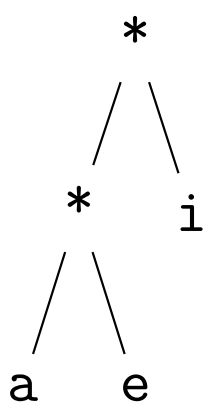
Esta es una clásica definición de una función recursiva, siendo el caso correspondiente al constructor `Leaf` el caso base y el caso de `Node` el caso recursivo.

Easy Peasy!!

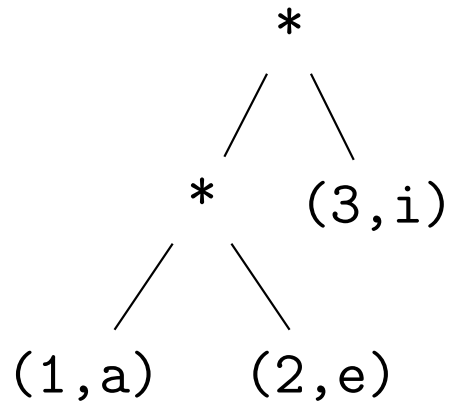
too easy?

Vamos a complicar un poco las cosas.

Queremos definir ahora un función que indexe las hojas de un árbol de izquierda a derecha.



se transforma en



relabel

Como programadores de un lenguaje de programación funcional, la primera idea claramente es hacer una función recursiva de forma similar a la función anterior.

```
relabel :: Tree a -> Tree (Int,a)
relabel (Leaf x) = (??,x)
```

Pero ¿Que indice le corresponde a cada hoja?

relabel round 2

No hay problema, esto se puede corregir con un contador que vaya aumentando en cada llamada recursiva.

```
relabel :: Tree a -> Int -> Tree (Int,a)
relabel (Leaf x) i   = Leaf (i,x)
relabel (Node l r) i = Node (relabel l i)
                            (relabel r ??)
```

¿Que indice pasamos a la llamada recursiva del subárbol derecho?

Se podría pensar en pasarle como argumento `i+1` pero ¿y si hay mas de una hoja en el subárbol izquierdo?

relabel round 3

Esto no nos va a detener. Sigamos pensando

(=_=)

¿Qué tal si regresamos el contador junto con el resultado de la función?

```
relabel :: Tree a -> Int -> (Tree (Int,a), Int)
relabel (Leaf x) i    = (Leaf (i,x),i+1)
relabel (Node l r) i = (Node l' r', i2)
 where (l',i1) = relabel l i
       (r',i2) = relabel r i1
```

¡¡Excelente idea!!

Es claro que funciona, pero es una función muy fea.


Esta horrible, me encanta.

Counter

Vamos a intentar refactorizarla para hacerla mas legible.

El problema es que se esta combinando la lógica del programa con aspectos técnicos como lo es el contador.

Para deshacernos de esto vamos a definir un sinónimo para funciones que requieren un contador.

```
type Counter b = Int -> (b,Int)
```

`b` es el tipo que originalmente espera recibir la función mientras que el parámetro entero representa el contador.

Así el tipo de `relable ::Tree a ->Counter (Tree a)`

next

Ahora intentemos encapsular el comportamiento del caso de `Node` en una función.

```
relabel (Node l r) i = (Node l' r', i2)
 where (l',i1) = relabel l i
       (r',i2) = relabel r i1
```

Se puede definir la siguiente función

```
next::Counter a -> (a -> Counter b) -> Counter b
f `next` g = \i -> g r i' where (r,i') = f i
```

single

Ahora definamos una función que recibe un valor y le asigna un contador.

```
single :: a -> Counter a
single x = \i -> (x,i)
```

Esta función nos sirve para transformar un valor en algo de tipo `Counter`.

Nueva definición de relabel

De esta forma podemos reescribir `relabel` de una forma mucho mas limpia.

```
relabel :: Tree a -> Counter (Tree a)
relabel (Leaf x)   = \i -> (Leaf (i,x), i+1)
relabel (Node r l) = relabel l `next` \l' ->
                     relabel r `next` \r' ->
                     single (Node l' r')
```

Sin tener que preocuparnos explícitamente por el contador

Así escondimos el tecnicismo del contador en un sinónimo para no tener que preocuparnos por él.


QUE ELEGANCIA..
LA DE FRANCIA!
memegenerator.es

Resumen

En resumen, definimos el tipo `Counter` con las funciones `next` y `single` de la siguiente forma:

```
type Counter b = Int -> (b,Int)

next::Counter a -> (a -> Counter b) -> Counter b
f `next` g = \i -> g r i' where (r,i') = f i

single :: a -> Counter a
single x = \i -> (x,i)
```

Espera un momento ...

Vamos a averiguarlo ...

Generalización

Comencemos por hacer mas general la definición de `Counter`

```
type State a s = a -> (a,s)
```

El tipo `State` define un estado en el que el primer parámetro es el estado en sí y el segundo es el valor al que se asocia

El tipo `Counter` se define ahora como `State Int`

Estado

Claro que cuando hablamos de estado en programación funcional no es exactamente el mismo concepto que ya conocemos.

Se puede ver como que el estado agrega un contexto a un valor (en nuestro ejemplo el contexto se refiere al contador)

Bajo esta idea `next` hace que el estado funcione de forma secuencial, el resultado de un computo alimenta al siguiente (es muy evidente en `relabel`)

Mónada

Eso justo es una mónada

Algo que simula un comportamiento de los lenguajes imperativos y que tiene una operación de secuencia

Así de sencillo

La instancia de la clase `Monad` de `State` es:

```
instance Monad (State a) where
    return = single
    (>>=) = next
```

Reescribamos la función `relabel` utilizando la notación de mónadas

```
relabel :: Tree a -> State Int (Tree a)
relabel (Leaf x)   = \i -> (Leaf (i,x), i+1)
relabel (Node r l) = relabel l >>= \l' ->
                     relabel r >>= \r' ->
                     return (Node l' r')
```

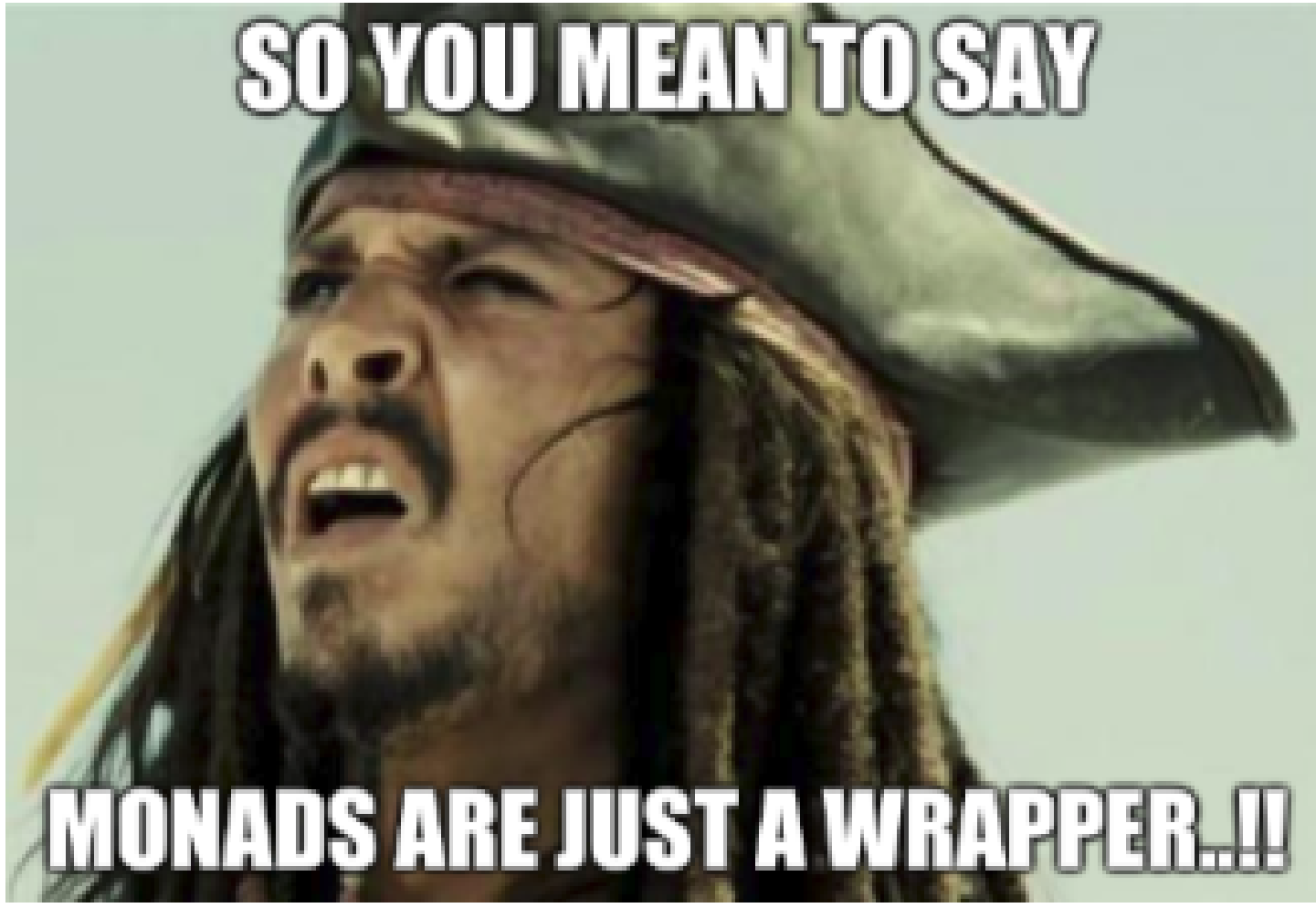
No cambia mucho pero es un poco mas legible

do

Pero como `State` es una instancia de `Monad` podemos usar la notación `do`

```
relabel :: Tree a -> State Int (Tree a)
relabel (Leaf x)   = \i -> (Leaf (i,x), i+1)
relabel (Node r l) = do l' <- relabel l
                        r' <- relabel r
                        return (Node l' r')
```

¿Qué clase de lennguaje imperativo es este?


SO YOU MEAN TO SAY
MONADS ARE JUST A WRAPPER..!!