All told, monad is just a monoid
in the category of endofunctors

–Saunders Mac Lane

Julián Tricanico

February 27, 2023

Contenidos

Decimos que `m` es un monoide si tiene una instancia de

```
(<>)    :: m -> m -> m
mempty  :: m
```

y satisface

```
x <> (y <> z) == (x <> y) <> z
x <> mempty   == x
mempty <> x   == x
```

Reescribiendo, `m` es un monoide si tiene una instancia de

```
mu  :: (m , m) -> m
eta :: () -> m
```

y satisface

```
mu (x , (mu (y , z))) == mu ((mu (x , y)) , z)
mu (x , eta ())       == x
mu (eta () , x)       == x
```

Los nombres vienen de *multiplication* y *identity*. (El fonema de η original se encuentra entre medio de los fonemas de *i* y *e*.)

Nos gustaría abstraer estas propiedades para poder definir monoides en categorías además de `Hask`.
Para eso comenzamos tratando de conseguir una versión 'point-free' de las propiedades de arriba.

Recuerdo las definiciones de `Functor` y `Bifunctor`.

```
class Functor f where
  fmap  :: (a -> b)                  -> f a    -> f b

class Bifunctor f where
  bimap :: (a -> b) -> (a' -> b') -> f a a' -> f b b'
```

más las leyes asociadas.

Notar que las leyes de monoide se pueden reescribir como

```
    mu . (bimap id mu) $ (x , (y , z))
== mu . (bimap mu id) $ ((x , y) , z)

mu . (bimap id eta) $ (x , ()) == x
mu . (bimap eta id) $ (() , x) == x
```

Consideremos los siguientes polimorfismos:

```
alpha :: ((a , b) , c) -> (a , (b , c))
alpha    ((x , y) , z)  = (x , (y , z))

rho    :: (a , ()) -> a
rho       (x , ())  = x
lambda :: (() , a) -> a
lambda    (() , x)  = x
```

Los nombres vienen de *associator*, *right* y *left*.

Luego las leyes de monoide se pueden escribir como

```
mu . (bimap id mu) . alpha == mu . (bimap mu id)

mu . (bimap id eta)         == rho
mu . (bimap eta id)         == lambda
```

La nota más importante a hacer para generalizar esta definición de monoide a otras categorías, es que no usamos ninguna propiedad particular de `( , )` ni de `()`.

A una categoría equipada con un producto (llamado tensorial y anotado $\otimes$) y un objeto distinguido (anotado i) se le llama *Categoría Monoidal*. Luego decimos que un *monoide* en una categoría monoidal es una elección de un `mu` y un `eta` que se porten bien con el producto tensorial $\otimes$ y unidad i. Es decir

```
mu  :: m ⊗ m -> m
eta :: i -> m
```

con las leyes de arriba y `m` un elemento fijo en nuestra categoría.

Veamos qué sucede en el caso particular en que los objetos de nuestra categoría son endofuntores.

Recuerdo que un endofuntor es un funtor con dominio igual a su codominio. Es decir podríamos redefinir

```
class Endofunctor f where
  fmap :: (a -> a) -> f a -> f a
```

más las leyes asociadas.

Queremos definir al producto $\otimes$ como la composición de funtores; con i la identidad. Por eso precisamos que tengan igual dominio y codominio.

Para esto hay que ver cómo se ven en esta categoría

- flechas
- `bimap`
- leyes de monoide

Definimos una transformación natural entre dos funtores `F` y `G` como un polimorfismo

```
alpha :: F x -> G x
```

que además satisfaga

```
alpha . fmap f == fmap f . alpha :: F a -> G b
  where f :: a -> b
```

Estas son las flechas en nuestra categoría.
En haskell se suele anotar `alpha :: F ~> G` .
(Cuidado con la sutil diferencia entre - y ~.)

Notar que por un lado podemos definir

```
alpha' . alpha :: F  x -> F'' x
  where alpha  :: F  x -> F'  x
        alpha' :: F' x -> F'' x
```

llamada composición vertical.

Y por otro lado podemos definir

```
beta . alpha  :: G (F x) -> G' (F' x)
  where alpha :: F x -> F' x
        beta  :: G x -> G' x
```

llamada composición horizontal.

El lector involucrado podrá demostrar la existencia de estas composiciones a partir de la definición de transformación natural.

Recordar

```
class  Bifunctor f where
  bimap :: (a -> b) -> (a' -> b') -> f a a' -> f b b'
```

En nuestro caso particular en que nuestros objetos son endofuntores nos queda

```
bimap :: (F ~> F') -> (G ~> G') -> G . F ~> G' . F'
```

que es precisamente el tipo de la composición horizontal. Luego las leyes de `Bifunctor` nos dicen que es precisamente la composición horizontal. (Y usamos la precomposición como bifuntor.)

Por último un monoide en esta categoría es elegir un endofuntor `m` junto con un `mu :: m^2 ~> m` y un `eta :: I ~> m` siendo `I` la identidad.

Desarrollando la definición de `~>`

```
mu  :: m (m a) -> m a
eta :: a -> m a
```

que se pueden reconocer como los mapas `join` y `return` en haskell respectivamente; que son una definición minimal para una mónada.

Ejercicio para el lector: verificar que las leyes de monoide en este caso coinciden con las de una mónada.