Programación Declarativa 2020-2
`Functor`, `Applicative` y `Monad`:
Todo lo que siempre quisiste saber pero tenías miedo de preguntar

Javier Enríquez Mendoza

Facultad de Ciencias UNAM

21 de mayo de 2020

Motivación

La función `map` es una función bien conocida

Se trata de una función de orden superior que aplica una función a todos los elementos de una lista

```
map :: (a -> b) -> [a] -> [b]
```

Después de todo este tiempo programando en Haskell es obvia la importancia y lo útil que es `map`

¿No sería interesante tener una función map para todas las estructuras de datos?

De eso se tratan los funtores

Directo de Wiki Haskell

Funtor: acción uniforme sobre un tipo parametrizado, generalizando la función del `map` *de listas.*

Funtor

Una definición mas terrenal es

Un funtor es un tipo paramentado sobre el cual existe una función de mapeo

En Haskell los funtores están definidos por la clase `Functor`

Y la función de mapeo es el método `fmap`

La bella clase `Functor`

La clase `Functor` está definida como sigue:

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

`fmap` es la función de mapeo

`(<$>)` es la versión infija de `fmap`

`(<$)` es la función que mapea todos los elementos de funtor `f a` la constante de tipo `a` que recibe como primer argumento

Cuando se instancia la clase basta con dar la definición de `fmap`

Cosas cool que se puede hacer con los funtores

- Mapear sobre cualquier estructura de datos
- Mapear sobre cualquier estructura de datos
- Mapear sobre cualquier estructura de datos

Cuando usar Funtores

Cuando se quiere mapear sobre cualquier estructura de datos

Y sobre todo cuando se quieren modificar los valores que almacena una estructura pero preservar el contexto

Es decir en casos en donde la construcción de la estructura nos da información al igual que los valores que almacena

¿Cómo usar funtores?

Para garantizar que las instancias de la clase `Functor` se van a comportar como deben tienen que cumplir dos propiedades

- `fmap id = id`
- `fmap (f.g) = fmap f . fmap g`

Motivación

Los functores son muy cool y todo pero que tal que queremos aplicar una función que tenemos en un contexto a valores que también se encuentran dentro de un contexto

Por ejemplo tenemos `[succ, pred]` que es un funtor que almacena a las funciones sucesor y predecesor

Y tenemos la lista `[1,2,3]` y nos gustaría mapear ambas funciones a los elementos de la lista

la llamada`map [succ, pred] [1,2,3]` nos regresa un error de tipos

Necesitamos una especie de funtores extendidos que nos permitan hacer esto

Los funtores aplicativos

Funtores Aplicativos

Los funtores aplicativos nos van a permitir aplicar funciones dentro de un contexto a valores que también se encuentran en un contexto

Applicative

Los funtores aplicativos están definidos en Haskell por la clase `Applicative`

la clase se define como sigue:

```
class (Functor f) => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

`pure` es una función que nos permite agregar contexto aun valor (disfrazarlo)

`(<*>)` es un operador muy parecido a `(<$>)` con la única diferencia de que la función a aplicar también se encuentra en un contexto

Es importante observar que `Applicative` es subclase de `Functor`

Ejemplo

Ahora ya podemos resolver el problema que teníamos al principio

```
 > [succ, pred] <*> [1,2,3,4]

[2,3,4,5,0,1,2,3]
```

Ahora si aplico ambas funciones a los elementos de la lista

¿Cómo usar funtores aplicativos?

Como `Applicative` es subclase de `Functor` para generar una instancia de `Applicative` debe existir también una de `Fuctor`

Al igual que los funtores, los aplicativos también deben cumplir algunas propiedades para garantizar su comportamiento

- `pure f <*> x =f <*> x`
- `pude id <*> v = v`
- `pure (.)<*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y =pure ($ y)<*> u`

Mónadas

Las mónadas se pueden ver como una extensión de los aplicativos

Cuando se trata de manejar contextos las mónadas nos brindan la forma mas general de hacerlo

Esto es porque las mónadas manejan el contexto por nosotros automáticamente

La idea intuitiva es que una mónada modele un comportamiento imperativo junto con una operación secuencial

Cada mónada modela un solo comportamiento imperativo, podría modelar mas pero no lo hagan

la clase Monad

En Haskell las mónadas se modelan con la clase `Monad`

```
class (Applicative) => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    fail :: String -> m a
```

`return` tiene la misma función que `pure`

`(>>=)` el operador *bind* es el encargado de ejecutar comando de forma secuencial alimentando un comando con el resultado del anterior

`(>>)` de igual forma es un operador de secuencia pero sin cargar el resultado de la ejecución anterior

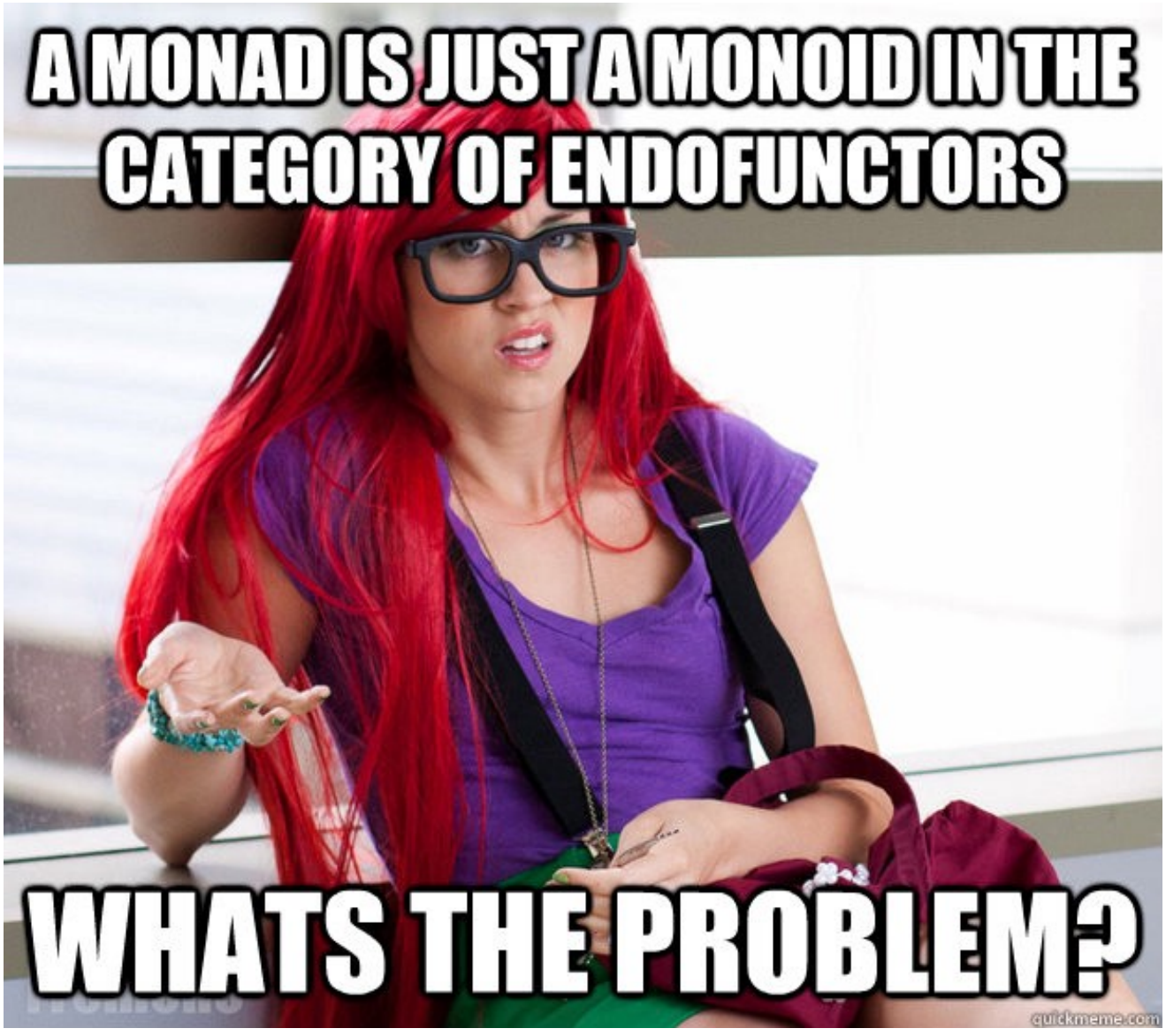
`fail` ya está predefinido y nunca se va a usar explicitamente, es para control interno de Haskell

¿Cómo usar mónadas?

La clase `Monad` es subclase `Applicative` que a su ves es subclase de `Functor`, esto se traduce a que para instanciar `Monad` se necesitan instancias de las otras dos

Las mónadas también tiene que cumplir ciertas propiedades, las famosas *Leyes mónadicas*

- `return x >>=f = f x`
- `m >>= return = m`
- `(m >>= f) >>= g = m >>= (\ x -> f x >>= g)`


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS
WHATS THE PROBLEM?
quickmeme.com