Introducción a Mónadas

Programación funcional

Motivación

En un modelo de cómputo con transparencia referencial.
¿Cómo describimos la composición de funciones en un pipeline?

- ¿Qué sucede si en un paso del pipeline hay una falla?
- ¿Qué sucede si en un paso del pipeline hay más de un resultado posible?
- ¿Qué sucede si un paso del pipeline modifica el estado del entorno (i.e., alterando el comportamiento futuro de los nodos del pipeline)?

Definición

Mónada

Las mónadas son un tipo abstracto de datos con dos operaciones (llamadas `unit` y `bind`).

En Haskell las definimos usando un typeclass:

```
class Monad m where
  return :: a -> m a   -- unit
  (>>=) :: m a -> (a -> m b) -> m b   -- bind
```

Invariantes

Leyes

Para ser considerada una mónada es necesario que cualquier implementación cumpla las siguientes propiedades:

1. **Idenitidad derecha:**
   `m >>= return` $\equiv$ `m`
2. **Idenitidad izquierda:**
   `return x >>= f` $\equiv$ `f x`
3. **Asociatividad:**
   `(m >>= f) >>= g` $\equiv$ `m >>= (\x -> f x >>= g)`

Ejemplo: Maybe

Ejemplo: Maybe

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

Ejemplo: Maybe

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x


pred3 x =
  pred x >>= \n ->
  pred n >>= \m ->
  pred m >>= \o ->
  return o
  where pred Z     = Nothing
        pred (S x) = Just x
```

Ejemplo: Maybe

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x


pred3 x =
  pred x >>= \n ->
  pred n >>= \m ->
  pred m >>= \o ->
  return o
  where pred Z     = Nothing
        pred (S x) = Just x
```

- Permite modelar secuencias de cómputos que pueden fallar en algún punto.

Ejemplo: List

Ejemplo: List

```
instance Monad [] where
  return x = [x]
  xs >>= f = concatMap f xs
```

Ejemplo: List

```
instance Monad [] where
  return x = [x]
  xs >>= f = concatMap f xs


perms xs =
  if null xs then
    return []
  else
    xs >>= \x ->
    perms (xs // [x]) >>= \xs' ->
    return (x:xs')
```

Ejemplo: List

```
instance Monad [] where
  return x = [x]
  xs >>= f = concatMap f xs


perms xs =
  if null xs then
    return []
  else
    xs >>= \x ->
    perms (xs // [x]) >>= \xs' ->
    return (x:xs')
```

- Permite modelar secuencias de cómputos "no-determinísticos"(i.e., más de un resultado posible).

Ejemplo: State

Ejemplo: State

```
newtype State s a = ST (s -> (s,a))

instance Monad (State s) where
  return x = ST (\s -> (s, x))
  (ST g) >>= f = ST (\s ->
    let (s', x) = g s
        (ST h) = f x
     in h s')
```

Ejemplo: State

```
newtype State s a = ST (s -> (s,a))

instance Monad (State s) where
  return x = ST (\s -> (s, x))
  (ST g) >>= f = ST (\s ->
    let (s', x) = g s
        (ST h) = f x
     in h s')
```

- ¿Qué sucede si el constructor ST es privado?

Ejemplo: State

```
newtype State s a = ST (s -> (s,a))

instance Monad (State s) where
  return x = ST (\s -> (s, x))
  (ST g) >>= f = ST (\s ->
    let (s', x) = g s
        (ST h) = f x
     in h s')
```

- ¿Qué sucede si el constructor ST es privado?
- La mónada de IO no tiene constructor, tenemos una única instancia dada por el SO en la función main.

Ejemplo: State

```
newtype State s a = ST (s -> (s,a))

instance Monad (State s) where
  return x = ST (\s -> (s, x))
  (ST g) >>= f = ST (\s ->
    let (s', x) = g s
        (ST h) = f x
     in h s')
```

- ¿Qué sucede si el constructor `ST` es privado?
- La mónada de `IO` no tiene constructor, tenemos una única instancia dada por el SO en la función main.
- Permite modelar cómputo con estado (`IO` permite modelar cómputo con efecto).

Notación do

Normalmente se evita usar el operador de bind explícitamente.

Do notation

```
do { x }   -->
   x

do {x ; <xs> }   -->
   x >> do { <xs> }

do { a <- x ; <xs> }   -->
   x >>= \a -> do { <xs> }

do { let <declarations> ; xs }   -->
   let <declarations> in do { xs }
```

Input-Output

```
import System.IO

main :: IO ()
main =
  do
    text <- getLine
    if text /= "exit" then do
      appendFile "output.txt" (text ++ "\n")
      main
    else do
      content <- readFile "output.txt"
      putStrLn content
      return ()
```

Composición de mónadas

- La composición de dos mónadas no siempre genera una mónada.

Composición de mónadas

- La composición de dos mónadas no siempre genera una mónada.
- Muchas aplicaciones útiles se pueden expresar como la composición de varias mónadas (e.g., Maybe + State o List + State).

Composición de mónadas

- La composición de dos mónadas no siempre genera una mónada.
- Muchas aplicaciones útiles se pueden expresar como la composición de varias mónadas (e.g., Maybe + State o List + State).
- Como todo tipo abstracto de datos es importante no exponer la representación interna, las operaciones que requieran de la representación interna deben estar en la interfaz, el resto sólo debe utilizar la interfaz.