Programación funcional avanzada

Use of Monads

Federico Sawady

Universidad Nacional de Quilmes

11 de junio de 2018

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

Monads

Existen cuatro formas de entender mónadas:

- Para qué sirven y cómo se usan en lenguajes de programación funcionales (el álgebra de las mónadas)
- Cómo se implementan en un lenguaje de programación
- Como una estructura algebraica con dos operaciones que cumplen una serie de propiedades
- A través de su definición en teoría de categorías

Monads

Cita (chiste)

A monad is just a monoid in the category of endofunctors, what's the problem?

Monads

Monads originally come from a branch of mathematics called Category Theory. Fortunately, it is entirely unnecessary to understand category theory in order to understand and use monads in Haskell.

Historia

Eugenio Moggi first described the general use of monads to structure programs in 1991. Several people built on his work, including programming language researchers Philip Wadler and Simon Peyton Jones (both of whom were involved in the specification of Haskell). Early versions of Haskell used a problematic "lazy list"model for I/O, and Haskell 1.3 introduced monads as a more flexible way to combine I/O with lazy evaluation.

¿Por qué nadie las entiende?

Cita

I think the problem with explaining monads is simply that they're more abstract than most people are used to dealing with. There is no simple one-paragraph description that fully explains what they are and how they're relevant.

Motivación sobre Monads

Comparación entre Mónadas y Monoides: intentar explicar monoides antes de haber utilizado fuertemente `Int`, `Bool`, `Maybe` y `(a -> a)`, y entender qué eran.

Motivación sobre Monads

Cuidado con los *tutoriales* y no vean dibujos.

Monads

Las mónadas son:

- Una estructura algebraica con dos operaciones que cumplen una serie de propiedades
- Una forma de estructurar transformaciones para que se ejecuten en un mismo contexto computacional, pasando el resultado de una función a la siguiente.
- Una forma de encapsular efectos laterales, representados como tipos, dentro de un lenguaje lazy y con transparencia referencial.

Monads

Cita

Purity was a necessary side effect of having lazy evaluation, because it's next to impossible to just have ambient effects in the language semantics with lazy evaluation as it's very hard to tell when or if they will happen. Monads are all the plumbing of state and effect sequencing that is implicit in the procedural style.

Monads as Computations

If *m* is a monad value, then an object of type *m a* represents a computation that is expected to produce a result of type *a*. The choice of monad reflects the (possible) use of particular programming language features as part of the computation.

Monads as Computations

Cita

I love how monads allow you to interpret the same bit of code with different semantics. Tricky to achieve in most programming languages

Dan Piponi.

Monad Class

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

Todas las mónadas son functors, pero no todos los functors son mónadas.

Effects

- La operación >>= (se denomina bind), cumple el rol prinicipal.
- Recordemos su tipo `(>>=) :: m a -> (a -> m b) -> m b`
- Sólo mirando su tipo nos damos cuenta de lo siguiente:
  1. Toma un tipo (functor) que contiene un valor
  2. Toma una función que toma dicho valor y devuelve un nuevo contenedor (functor)
  3. Devuelve el resultado de dicha función.

Effects

- ¿Por qué secuencia efectos? ¿Cómo lo hace? ¿Cuándo lo hace?
- Volvamos a mirar su tipo: `(>>=) :: m a -> (a -> m b) -> m b`.
- No hay forma de ejecutar la función que recibe por parámetro si no extrae el valor del primer parámetro, ahí es cuando el efecto se ejecuta. Luego, devuelve un nuevo contenedor, que si otra operación de bind se ejecuta, abre dicho contenedor, y ejecuta el siguiente efecto.

Monads

Ejemplos de instancias (observar que todos son functors):

```
data Maybe a = Nothing | Just a

data Either b a = Left b | Right a

data Id a = Id a

data [a] = [] | a : [a]

data IO a = ...
```

Monads

```haskell
x :: Maybe a -- es un valor
x :: [a] -- es un valor

x :: IO a -- es un valor

x :: Monad m => m a -- es un valor

-- ¡las acciones son valores!
```

Effects

```haskell
return :: Monad m => a -> m a

-- devuelve un valor de tipo m, que no tiene efectos

-- return es el nuetro de bind
```

Effects

Recordemos lo que pasa con monoides

```
mempty :: Monoid o => o

mempty :: Sum

mempty :: Maybe Sum

mempty :: String

mempty :: All

mempty :: Any
```

Effects

```haskell
-- return no aplica ningún efecto, sólo encapsula un valor

return :: Monad m => a -> m a

return 5 :: Maybe Int

return 5 :: Id Int

return 5 :: Either String Int

return 5 :: [Int]

return 5 :: IO Int
```

Effects

```haskell
m :: Maybe Int
m = return 5
```

Effects: Unit

```haskell
m :: Maybe ()
m = return ()

-- () es el valor unitario
-- el valor utilizado cuando queremos llenar un hueco
-- y nos interesa un valor particular
```

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

Effects

`IO` encapsula a un efecto de entrada-salida.

Effects

```haskell
print :: Show a => a -> IO ()

main :: IO ()
main = print "hola mundo"
```

Effects

```haskell
print :: Show a => a -> IO ()

main :: IO ()
main = print [1,2,3]
```

Effects

```haskell
main :: IO ()
main = do
    print "hola"
    print "mundo"
```

Effects

`do` junta valores de **un mismo tipo de mónada**. Si la expresión es de tipo `m a` se puede juntar con otra de tipo `m b`, pero la `m` queda fija. Sólo el resultado de cada acción puede variar.

Effects

Necesitamos el valor de la acción. Y esto no nos lo provee:

```
getLine :: IO String

main :: IO ()
main = do
    getLine
    print ???
```

Effects

Entonces:

```
getLine :: IO String

main :: IO ()
main = do
    x <- getLine
    print x
```

Effects

Puedo extraer el resultado de la acción usamos <-

Effects

¿Y si hago lo siguiente?

```
print :: IO ()

main :: IO ()
main = do
    y <- print "10"
    print y
```

Effects

En el medio puedo realizar transformaciones

```
import Data.Char

main :: IO ()
main = do
    x <- getLine
    print (map toUpper x)
    main
```

Effects

Puedo hacer subtareas:

```
print3 :: String -> IO ()
print3 x = do
    print x
    print x
    print x

main :: IO ()
main = do
    print3 "10"
    print3 "10"
    print3 "10"
```

Effects

¿Cómo sería printN?

```
printN :: Int -> String -> IO ()
printN 0 x = return ()
printN n x = do
    print x
    printN (n-1) x

main :: IO ()
main = do
    printN 10 "nananana"
    print "batman"
```

Effects

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            print (reverseWords line)
            main
reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

Effects

No me jodas Haskell, ¿de verdad siempre tengo que poner siempre ambas ramas?

Effects

Si sólo me importa la rama then

```
when :: Monad m => Bool -> m () -> m ()
when b m = if b then m else return ()
```

Effects

```
import Control.Monad

main = do
     line <- getLine
     when (not (null line))
          (do print line; main)
```

Effects

```
import Control.Monad

main = do
     line <- getLine
     when (not (null line)) $
          do
            print line
            main
```

Effects

```haskell
-- return no aplica ningún efecto
-- sólo encapsula un valor
-- en cualquier tipo de mónada
return 5 :: [Int]

return 5 :: Maybe Int

return 5 :: IO Int

return :: Monad m => a -> m a
```

Effects

```haskell
main :: IO ()
main = do
    return ()
    return "Fidel"
    line <- getLine
    return "PF PF PF"
    return 4
    print line

-- Todos estos return
-- tienen tipo return :: IO a
```

Effects

No corta el flujo de los programas:

```
main :: IO ()
main = do
    x <- getLine
    when (null x) (return ())
    print x
    main
```

Effects

```haskell
main :: IO ()
main = do
    a <- return "hell"
    b <- return "yeah!"
    print (a ++ " " ++ b)
```

Effects

```haskell
main :: IO ()
main = do
    a <- "hell"
    b <- "yeah!"
    print (a ++ " " ++ b)

-- Esto no tipa
```

Effects

```haskell
main :: IO ()
main = do
    let a = "hell"
    let b = "yeah!"
    print (a ++ " " ++ b)
```

Effects

Puedo reasignar:

```
main :: IO ()
main = do
    let a = "hell"
    let b = "yeah!"
    let a = "Oh!"
    print (a ++ " " ++ b)
```

Effects

Incluso hacer esto:

```
main :: IO ()
main = do
    let x = 5
    let x = x + 5
    print x
```

Effects

¿Y esto?

```
main :: IO ()
main = do
    let x = x + 1
    print x
```

Effects

Esta expresión no termina porque `x` se usa a sí misma.

```haskell
main :: IO ()
main = do
    let x = x + 1
    print x
```

Effects

```haskell
getContents :: IO String

-- Me permite tomar cualquier
-- string por la linea standard
-- Por ejemplo, contenido
-- de archivos
```

Effects

```haskell
main = do
    input <- getContents
    let separatedLines = lines input
    let shortLines = filterShort separatedLines
    let result = unlines shortLines
    putStr result

  where filterShort lns =
          filter (\line -> length line < 4) lns
```

Effects

```
interact :: (String -> String) -> IO ()

main :: IO ()
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filterShort allLines
        result = unlines shortLines
    in  result

  where filterShort lns =
          filter (\line -> length line < 4) lns
```

Effects

Point-free style

```
main :: IO ()
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly =
    unlines . filter ((<4) . length) . lines
```

Effects

```haskell
interact :: (String -> String) -> IO ()
interact f = ???
```

Effects

```haskell
interact :: (String -> String) -> IO ()
interact f =
    do s <- getContents
       putStr (f s)
```

Effects

¿Genera algún efecto al ejecutarlo en la consola?

```
> [print "1", print "2", print "3"]
```

Effects

¡No! Las acciones son valores. No generan efectos salvo que sean parte de la ejecución de la función `main`.

Effects

```
sequence_ :: ???

main :: IO ()
main = sequence_ [print "1", print "2", print "3"]
```

Effects

¿Como estará definida?

```
sequence_ :: [IO a] -> IO ()
sequence_ = ???
```

Effects

```haskell
sequence_ :: [IO a] -> IO ()
sequence_ []     = return ()
sequence_ (x:xs) =
        do x
           sequence_ xs
```

Effects

```haskell
sequence_ :: [IO a] -> IO ()
sequence_ []     = return ()
sequence_ (x:xs) =
        do x
           sequence_ xs
```

Effects

```
(>>) :: IO a -> IO b -> IO b
x >> y =
    do
      x
      y
```

Effects

¡sequence_ se puede definir con foldr!

```
sequence_ :: [IO a] -> IO ()
sequence_ = foldr (\m1 m2 -> m1 >> m2) (return ())
```

Effects

```haskell
sequence_ :: [IO a] -> IO ()
sequence_ = foldr (>>) (return ())
```

Effects

Pero vimos que `foldr` es más genérico que eso.

```
sequence_ :: (Monad m, Foldable t) => t (m a) -> m ()
sequence_ = foldr (>>) (return ())
```

Effects

```haskell
sequence :: Monad m => [m a] -> m [a]
sequence = ???
```

Effects

```haskell
sequence :: Monad m => [m a] -> m [a]
sequence []     = return []
sequence (m:ms) =
    do
      x  <- m
      xs <- sequence ms
      return (x:xs)
```

Effects

Nos sirve para recolectar resultados

```
main :: IO ()
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

Effects

```
pedirNLineas :: ???
pedirNLineas n = ???
```

Effects

```haskell
pedirNLineas :: IO [String]
pedirNLineas n = sequence (replicate n getLine)

main :: IO ()
main = do
    rs <- pedirNLineas 5
    print rs
```

Effects

```haskell
replicateM :: Monad m => Int -> m a -> m [a]
replicateM n m = sequence (replicate n m)

main :: IO ()
main = do
    rs <- replicateM 5 getLine
    print rs
```

Effects

Con el uso de >>=

```
main :: IO ()
main = replicateM 5 getLine >>= print
```

Effects

Pero en realidad estaría mejor imprimir cada línea:

```
main :: IO ()
main = replicateM 5 getLine >>= mapM_ print
```

Effects

```haskell
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f
```

Effects

Sigamos con funciones genéricas:

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM = ???
```

Effects

```haskell
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = do
  x <- m
  return (f m)
```

Effects

¡Pero eso es `fmap`!

```
liftM :: Monad   m => (a -> b) -> m a -> m b
fmap  :: Functor f => (a -> b) -> f a -> f b
```

Con esto comprobamos que todas las mónadas son functors.

Effects

```haskell
getLineEspejo = fmap reverse getLine

main = do
    line <- getLineEspejo
    print ("You said " ++ line ++ " backwards!")
```

Effects

```haskell
import Control.Monad

forever :: ??

main = forever $
    do
       print "Give me some input: "
       fmap reverse getLine >>= print
```

Effects

```haskell
forever :: Monad m => m a -> m b
forever = ???
```

Effects

```
forever :: Monad m => m a -> m b
forever m = m >> forever m

(>>) :: Monad m => m a -> m b -> m b
```

Effects

¡`IO` no es la única mónada!

Monads

Todas las instancias de mónada más utilizadas:

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

data Reader r a = R (r -> a)

data Writer w a = W (w, a)

data Maybe a = Nothing | Just a

data Id a = Id a

data [a] = [] | a : [a]

data IO a = ...
```

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

Maybe Monad

La mónada `Maybe` representa la posibilidad de fallo del cómputo. Si una subtarea falla, fallará todo el algoritmo (pensar en excepciones en lenguajes de programación, aunque no de forma idéntica).

Maybe Monad

```
buscarValores :: [(String, Int)] -> [String] -> Maybe [Int]
buscarValores m [] = Just []
buscarValores m (k:ks) =
    case buscarValores m of
         Nothing -> Nothing
         Just vs -> Just (v:vs)
```

Maybe Monad

```haskell
buscarValores :: [(String, Int)] -> [String] -> Maybe [Int]
buscarValores m [] = return []
buscarValores m (k:ks) =
    do
        v  <- lookup k m
        vs <- buscarValores m ks
        return (v:vs)
```

Maybe Monad

```haskell
buscarValores' :: [(String, Int)] -> [String] -> Maybe [Int]
buscarValores' m ks = foldr go (return []) ks
  where go k r =
    do v <- lookup k m; fmap (v:) r
```

Maybe Monad

```
routine x =
    case f x of
        Nothing -> Nothing
        Just y -> case g y of
            Nothing -> Nothing
            Just z -> h z of
                Nothing -> Nothing
                Just w -> ...
```

Maybe Monad

```
routine x = do
    y <- f x
    z <- g y
    w <- h z
    ...
```

Maybe Monad

Otro ejemplo:

```
headM :: [a] -> Maybe a
headM [] = Nothing
headM (x:xs) = Just x

collectHeads :: [[a]] -> Maybe [a]
collectHeads = sequence . fmap headM

main = print (collectHeads [[1,2,3], []])
```

Maybe Monad

Otro ejemplo:

```
minimum :: [a] -> Maybe a
minimum [] = Nothing
minimum [x] = Just x
minimum (x:xs) = minM x (minimum xs)

minM x Nothing  = Just x
minM x (Just y) = Just (min x y)
```

Maybe Monad

```
minimum :: [a] -> Maybe a
minimum = foldr minM Nothing
```

Maybe Monad

¿Sirve de algo?

```
minimum :: Ord a => [a] -> Maybe a
minimum []  = Nothing
minimum [x] = return x
minimum (x:xs) =
    do
      m <- minimum xs
      return (min x m)

main = print (minimum [1,2,3])
```

Maybe Monad

Hay cosas que es discutible plantearlas monádicas, en este caso no lo haría.

Error Monad

La mónada `Either` representa también cómputos cuyo efecto es la posibilidad de fallo, pero esta vez retornando un valor en caso de fallar.

```
data Either a b = Left a | Right b
```

```
ghci> :t Right 4
Right 4 :: Either a Int
```

```
ghci> :t Left "out of index"
Left "out of index" :: Either String b
```

```
return 4 == Right 4
throwError "error" == Left "error"
```

Error Monad

```haskell
import Control.Monad.Error

get :: Int -> [a] -> Either String a
get n [] = throwError "out of index"
get 0 (x:xs) = return x
get n (x:xs) = get (n-1) xs
```

Error Monad

```haskell
gets :: [Int] -> [a] -> Either String [a]
gets []         xs = return []
gets (i:indxs) xs =
    do
       y  <- get i xs
       ys <- gets indxs xs
       return (y:ys)

main = print (gets [1,3] [1,2,3,4])
```

Error Monad

```haskell
gets :: [Int] -> [a] -> Either String [a]
gets is xs = sequence $ fmap (`get` xs) is
```

Error Monad

```haskell
safeDiv :: Int -> Int -> Either String Int
safeDiv x y = do
    when (y == 0) $ throwError "div by zero"
    return (x `div` y)
```

Error Monad

```
f = do
      n1  <- saveDiv 10 2
      n2  <- saveDiv 30 2
      n3  <- saveDiv 5 5
      return [n1,n2,n3]
```

Error Monad

```
f = do
      n1  <- saveDiv 15 3
      n2  <- saveDiv 40 4
      n3  <- saveDiv 10 0
      return [n1,n2,n3]
```

Error Monad

Tengo dos listas, quiero dividir los números uno a uno:

```
f :: [Int] -> [Int] -> Either String [Int]
f xs ys = sequence $ zipWith safeDiv xs ys
```

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

List monad

- La mónada de listas representa el indeterminismo.
- Si tenemos una lista $[x, y, z]$, significa que el elemento resultante puede ser x, y o z.
- Cuando extraemos un valor a través de <- nuestra función se va a bifucar por cada elemento: dado que no se sabe con cuál continuar, continúa con cada uno por separado.
- `return x = [x]`

List monad

```haskell
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n  <- [1,2,3]
    ch <- ['a','b']
    return (n,ch)
```

List monad

Que sería lo mismo con List Comprehensions

```
ghci> [ (n,ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

List monad

```
signs xs =
    do x <- xs
       [x,-x]

main = print (signs [1,2,3])
```

List monad

```
doubleSigns xs =
    do x <- xs
       y <- [x,-x]
       return (y*2)

main = print (signs [1,2,3])
```

List monad

```haskell
data Lado = Cara | Seca

tiradas :: Int -> [[Lado]]
tiradas 0 = [[]]
tiradas n =
  do
    xs <- tiradas (n-1)
    [Seca:xs, Cara:xs]

main = print (tiradas 3)
```

List monad

```haskell
continueIf :: (a -> Bool) -> a -> [a]
continueIf f x = if f x then [x] else []

keep :: (a -> Bool) -> [a] -> [a]
keep p xs = do
  x <- xs
  continueIf p x
  return x

main = print (keep (>=3) [1,2,3,4,5,6])
```

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

Writer Monad

```
compare9 :: Int -> (Bool, String)
compare9 x = (x > 9, "I compared x to 9")
```

Writer Monad

¿Las tuplas `(a,b)` son functors? ¿Serán mónadas también?

Writer Monad

```haskell
newtype Writer w a = Writer (a,w)
-- w es un monoide

runWriter (Writer (a,b)) = (a,b)
```

Writer Monad

```haskell
instance Monoid w => Monad (Writer w) where
    return x = Writer (x, mempty)
    ...
```

Writer Monad

Alcanza con indicar el tipo del monoide para que los valores se acumulen:

```
ghci> runWriter (return 3 :: Writer String Int)
(3, "")
ghci> runWriter (return 3 :: Writer (Sum Int) Int)
(3, Sum 0)
```

Writer

```haskell
logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Got number: " ++ show x])

multWithLog :: Int -> Int -> Writer [String] Int
multWithLog x y = do
    a <- logNumber x
    b <- logNumber y
    c <- logNumber (a*b)
    return c
```

Writer

```
tell :: Monoid w => w -> Writer w ()
tell w = Writer ((), w)

gotNumber x = tell ["Got number: " ++ show x]

multWithLog :: Int -> Int -> Writer [String] Int
multWithLog x y = do
    gotNumber x
    gotNumber y
    gotNumber (x*y)
    return (x*y)

main = print (runWriter (multWithLog 5 3))
```

Writer

```
ghci> fst $ runWriter (multWithLog 3 5)
15
```

Writer

```
ghci> snd $ runWriter (multWithLog 3 5)
[
  "Got number: 5",
  "Got number: 3",
  "Got number: 15"
]
```

Writer

```
mults :: [(Int,Int)] -> Writer (Sum Int) [Int]
mults [] = return []
mults ((x,y):xs) =
  do
    let z = x*y
    tell (Sum 1)
    rs  <- mults xs
    return (z:rs)

main = print
  (
    snd $
      runWriter (mults [(1,2), (3,5)])
  )

ghci> Sum 2
```

Writer

```haskell
instance Monoid w => Monad (Writer w) where
  return x = Writer (x, mempty)
  (>>=) (Writer (x, w)) f =
      let (Writer (y, w')) = f x
          in (Writer (y, mappend w w'))
```

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

State

La mónada `State` es una de las más generales. Básicamente encierra un cómputo que acarrea un estado como valor en cada transformación (pensar en el tablero de `Gobstones`).

State

```haskell
type Stack a = [a]

push :: a -> Stack a -> Stack a
push x xs = (x:xs)

pop :: Stack a -> Stack a
pop xs  = tail xs

top :: Stack a -> a
top xs = head xs

f :: Stack Int -> Int
f stack = pop (pop (push 0 stack))

-- Ignoremos que puede fallar
```

State

```haskell
f :: Stack Int -> Stack Int
f stack = let
    newStack1 = push 0 stack
    newStack2 = pop newStack1
    in pop newStack2
```

State

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

runState (State f) s = f s
```

State

```haskell
type Stack a = [a]

push :: a -> State (Stack a) ()
push x xs = State ((x:xs), ())

pop :: State (Stack a) ()
pop xs = State ((), tail xs)

f :: Stack (Stack Int) ()
f = do
      push 1
      pop
      pop
```

State

```
>> runState f [1,2,3]
((), [2,3])
>> fst $ runState f [1,2,3]
()
>> snd $ runState f [1,2,3]
[2,3]
```

State

Pero esos son sólo procedimientos, no devuelven nada salvo ()

State

Cambiemos la definición de pop:

```
pop :: State (Strack a) a
pop xs (head xs, tail xs)
```

State

```haskell
f :: Stack (Stack Int) Int
f = do
      push 4
      pop
      x <- pop
      return x
```

State

```
f :: Stack (Stack Int) Int
f = do
      push 4
      pop
      pop
```

State

```
>> runState f [1,2,3]
([1], [2,3])
>> fst $ runState f [1,2,3]
[1]
>> snd $ runState f [1,2,3]
[2,3]
```

State

```haskell
dropN :: Int -> State (Stack a) ()
dropN 0 = return ()
dropN n = do
    pop
    dropN (n-1)
```

State

```haskell
takeN :: Int -> State (Stack a) (Stack a)
takeN 0 = return []
takeN n = do
    x  <- pop
    xs <- takeN (n-1)
    return (x:xs)
```

State Monad

```haskell
modify :: (s -> s) -> State s ()
modify f = State (\s -> ((), f s))
```

State Monad

```haskell
push :: a -> State (Stack a) ()
push x = modify (x:)

incr :: State Int ()
incr = modify (+1)
```

State Monad

```haskell
get :: State s s
get = State $ \s -> (s,s)

put :: s -> State s ()
put newState = State $ \s -> ((), newState)
```

State Monad

```haskell
vaciar :: State (Stack a) (Stack a)
vaciar = do
    xs <- get
    put []
    return xs
```

State Monad

```
import System.Random

randomSt :: (RandomGen g, Random a) => State g a
randomSt = State random

threeCoins :: State StdGen (Int,Int,Int)
threeCoins = do
    a <- randomSt
    b <- randomSt
    c <- randomSt
    return (a,b,c)
```

State Monad

Existe una variante lazy y una estricta para la mónada state, dado que mantener un estado lazy muy grande puede ser costoso.

State Monad

Definición:

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

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

Reader

La mónada `Reader` representa una cadena de cómputos que necesitan un mismo parámetro, que en ningún momento se puede modificar (es una especie de `State` donde el estado es constante).

Reader

```haskell
addStuff :: Int -> Int
addStuff x = let
    a = (*2) x
    b = (+10) x
    in a+b
```

Reader

```haskell
data Reader r a = Reader (r -> a)

runReader :: Reader r a -> (r -> a)
runReader (Reader f) x = f x
```

Reader

```haskell
data Reader r a = Reader (r -> a)

addStuff :: Reader Int Int
addStuff = do
    a <- (*2)
    b <- (+10)
    return (a+b)

addStuff 5 = (5*2) + (5+10)
```

Reader

```haskell
ask :: Reader a a
ask = Reader (\x -> x)

-- ask = Reader id
```

Reader

```haskell
retrieveAndAdd :: Reader Int [Int, Int]
retrieveAndAdd = do
    x <- ask
    return [x, -x]
```

Reader

Definición:

```
instance Monad (Reader r) where
    return x = Reader (\_ -> x)
    (Reader h) >>= f = Reader $ \w ->
        let (Reader f') = f (h w)
            in f' w
```

Outline

1. Monads
   - Introducción
   - Efectos: entrada y salida
   - Maybe Monad
   - List Monad
   - Writer Monad
   - State Monad
   - Reader Monad
   - Specific Operations

Monad composition

¿Y si quiero más de un tipo de efecto en el mismo algoritmo?

Monad composition

Existen distintas técnicas para lograrlo.

Monad composition

Primero hay que entender que no toda composición de mónadas es una mónada. Entonces, ¿Cómo hacemos para componer estado con cómputos que pueden fallar? Lo dejamos para otro capítulo, es más avanzado.

Monads Effects

Pero... las mónadas se pueden llegar a componer de esta forma (no es la más utilizada)

```
f :: (WriterMonad m, ErrorMonad m, StateMonad m) => m ()
```

Monads Effects

En realidad las funciones específicas están dentro de otras `typeclasses` específicas por cada mónada.

```
class Monad m => StateMonad m where
    modify :: (s -> s) -> m s

incr :: StateMonad m => m Int
incr = modify (1+)
```

Monads Effects

Existe otra técnica llamada que es más simple `Monad Transformers` y más utilizada, aunque tiene pros y contras.

For Further Reading I


Miran LipovačaA.
Learn You a Haskell for Great Good!
April 2011.


Philip Wadler.
Comprehending monads.
ACM, 1990.


Philip Wadler.
Monads for functional programming
August, 1992.


Philip Wadler.
Combining monads.
July, 1992.

For Further Reading II

Philip Wadler.
Imperative functional programming.
January, 1993.

Simon Peyton Jones.
State in Haskell.
December, 1995.

Markr P. Jones.
Functional Programming with Overloading and Higher-Order Polymorphism
Springer-Verlag Lecture Notes in Computer Science 925, May 1995.