



# Programación Funcional

Clases teóricas

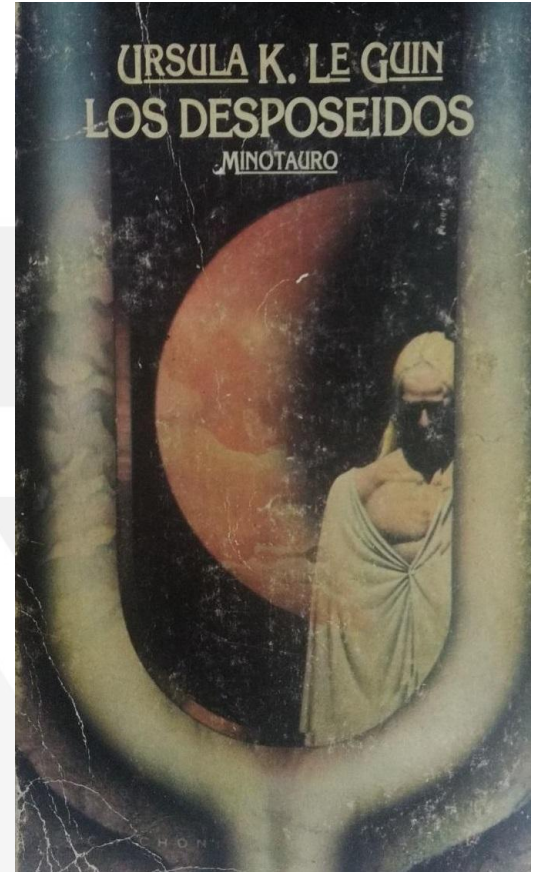
por Pablo E. “Fidel” Martínez López

## 13. Mónadas

*“Había un muro. No parecía importante. (...) Al igual que todos los muros, era ambiguo, bifacético. Lo que había dentro, o fuera de él, dependía del lado en que uno se encontraba.*

*Visto desde uno de los lados el muro encerraba un campo baldío de sesenta acres llamado el Puerto de Anarres. (...) Encerraba al universo, dejando fuera a Anarres, libre.”*

Los Desposeídos  
Úrsula K. Le Guin





# Motivación

# Motivación

- ❑ Las técnicas vistas tienen foco en la transformación de la información
- ❑ ¿Cómo agregar interacción con el medio?
  - ❑ Problemas a resolver
    - ❑ datos inmutables vs. “identidad” y datos mutables
    - ❑ contexto explícito vs. contexto implícito
    - ❑ entorno replicable vs entorno “destrutivo”
  - ❑ ¿Cómo expresar estas ideas sin perder las ventajas?

# Motivación

- ❑ Propuesta: mónadas
  - ❑ ¿Qué son las mónadas? Enfoque de la 1era parte
    - ❑ Philip Wadler, *“Monads for Functional Programming”*, Advanced Functional Programming, LNCS 925, Springer-Verlag, 1995.
    - ❑ Sumado a la “técnica de los recuadros”
  - ❑ ¿Cómo utilizarlas? Enfoque de la 2da parte
    - ❑ Definición e intuición
    - ❑ Ejemplos de uso y detalles

# Philip Wadler



## **Philip Lee Wadler**

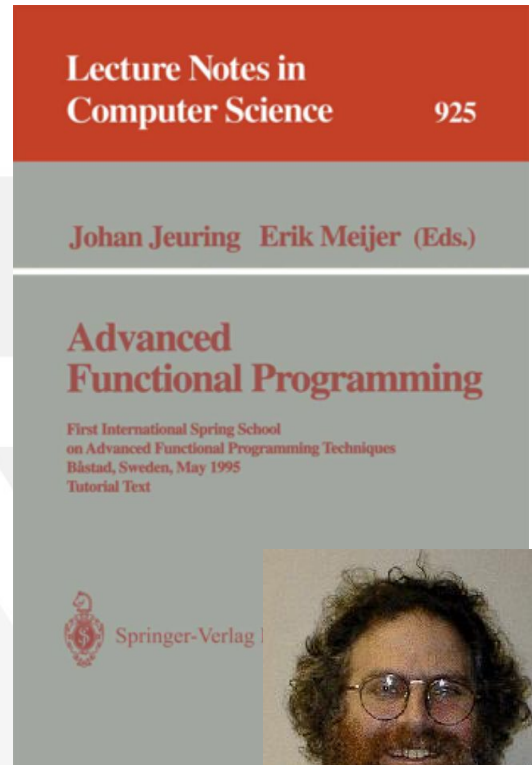
(8 de abril 1956 – ...) es un científico de la computación norteamericano, profesor de la Universidad de Edinburgo, en Inglaterra.

En 1984 recibió su doctorado de la Universidad de Carnegie Mellon y dedicó su carrera al estudio de la programación funcional. Trabajó en numerosas universidades y en Bell Labs, y actualmente es miembro de la empresa IOHK. Entre 1990 y 2004 fue editor del *Journal of Functional Programming*, una prestigiosa revista del área. Fue parte del grupo que desarrolló el lenguaje Haskell y estuvo involucrado en la incorporación de generics en el lenguaje Java. Es un excelente docente, muy histriónico y claro. En 2019 visitó la UNQ, donde dio la charla “Programming Language Foundations in Agda”.

*“Shall I be pure or impure?”*

Monads for Functional Programming  
Philip Wadler

AFP, LNCS 925, Springer Verlag, 1995  
(Previously in “Program Design Calculi”, 1992)



# Mónadas como generalización



# Mónadas como generalización

- ❑ Se usará un ejemplo simple
  - ❑ Una función con muchas variaciones
  - ❑ Se tratará de abstraer el esquema de variación
    - ❑ Técnica de los recuadros
- ❑ El ejemplo será simplificado para focalizar

```
data ExpA = Cte Int
          | Suma ExpA ExpA | Resta ExpA ExpA
          | Mult ExpA ExpA | Div   ExpA ExpA

evalExpA :: ExpA -> Int
```

# Mónadas como generalización

- ❑ Se usará un ejemplo simple
  - ❑ Una función con muchas variaciones
  - ❑ Se tratará de abstraer el esquema de variación
    - ❑ Técnica de los recuadros
- ❑ El ejemplo será simplificado para focalizar

```
data E      = Cte Int
```

```
| Div  E  E
```

```
eval      :: E  -> Int
```

# Mónadas como generalización

- ❑ Se usará un ejemplo simple
  - ❑ Una función con muchas variaciones
  - ❑ Se tratará de abstraer el esquema de variación
    - ❑ Técnica de los recuadros
- ❑ El ejemplo será simplificado para focalizar

```
data E = Cte Int | Div E E
```

```
eval :: E -> Int
```

```
eval ...
```

# Mónadas como generalización

- ❑ Se usará un ejemplo simple
  - ❑ Una función con muchas variaciones
  - ❑ Se tratará de abstraer el esquema de variación
    - ❑ Técnica de los recuadros
- ❑ El ejemplo será simplificado para focalizar

```
data E = Cte Int | Div E E
```

```
eval :: E -> Int
```

```
eval (Cte n) = n
```

```
eval (Div e1 e2) = eval e1 `div` eval e2
```

# Mónadas como generalización

- Ejemplo simplificado para focalizar
  - Variaciones: **original**, error, imprimir y estado

```
eval :: E -> Int
eval (Cte n)      = n
eval (Div e1 e2) =  $\underbrace{\text{eval } e1}_{v1} \text{ `div` } \underbrace{\text{eval } e2}_{v2}$ 
```

- Para mejor referencia, es interesante nombrar las aplicaciones recursivas

# Mónadas como generalización

- Ejemplo simplificado para focalizar
  - Variaciones: **original**, error, imprimir y estado

```
eval :: E -> Int
eval (Cte n)      = n
eval (Div e1 e2) = let v1 = eval e1
                   in let v2 = eval e2
                   in v1 `div` v2
```

- Para nombrar, existe **let**

# Mónadas como generalización

- Ejemplo simplificado para focalizar
  - Variaciones: **original**, error, imprimir y estado

```
eval :: E -> Int
eval (Cte n)      = n
eval (Div e1 e2) = let v1 = eval e1
                   in let v2 = eval e2
                   in v1 `div` v2
```

- ¿Es total o parcial?

```
data E = Cte Int | Div E E
eval   :: E -> Int
```

# Mónadas como generalización

- Ejemplo simplificado para focalizar
  - Variaciones: original, **error**, imprimir y estado

```
evalM :: E -> ... Int
evalM (Cte n)      = ...
evalM (Div e1 e2) =
    ... evalM e1 ...
    ...
    ... evalM e2 ...
    ...
```

- ¿Cómo volverla total?



# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```

- Ejemplo simplificado para focalizar
  - Variaciones: original, **error**, imprimir y estado

```
evalM :: E -> Maybe Int
evalM (Cte n)      = ...
evalM (Div e1 e2) =
    ... evalM e1 ...
    ...
    ... evalM e2 ...
    ...
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```

- Ejemplo simplificado para focalizar
  - Variaciones: original, **error**, imprimir y estado

```
evalM :: E -> Maybe Int
evalM (Cte n)      = ...
evalM (Div e1 e2) =
  case evalM e1 of
    Nothing -> Nothing
    Just v1  -> ... evalM e2 ...
    ...
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, **error**, imprimir y estado

```
evalM :: E -> Maybe Int
evalM (Cte n)      = ...
evalM (Div e1 e2) =
  case evalM e1 of
    Nothing -> Nothing
    Just v1  -> case evalM e2 of
      Nothing -> Nothing
      Just v2  ->
        ...
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```

- ❑ Ejemplo simplificado para focalizar
- ❑ Variaciones: original, **error**, imprimir y estado

```
evalM :: E -> Maybe Int
evalM (Cte n)      = ...
evalM (Div e1 e2) =
  case evalM e1 of
    Nothing -> Nothing
    Just v1  -> case evalM e2 of
      Nothing -> Nothing
      Just v2  ->
        ...
        Just (v1 `div` v2)
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```

- ❑ Ejemplo simplificado para focalizar
- ❑ Variaciones: original, **error**, imprimir y estado

```
evalM :: E -> Maybe Int
evalM (Cte n)      = ...
evalM (Div e1 e2) =
  case evalM e1 of
    Nothing -> Nothing
    Just v1  -> case evalM e2 of
      Nothing -> Nothing
      Just v2  -> if v2 == 0
                    then raiseError
                    else Just (v1 `div` v2)
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

```
data E = Cte Int | Div E E
eval   :: E -> Int
```

# Mónadas como generalización

- ❑ Ejemplo simplificado para focalizar
- ❑ Variaciones: original, **error**, imprimir y estado

```
evalM :: E -> Maybe Int
evalM (Cte n)      = Just n
evalM (Div e1 e2) =
  case evalM e1 of
    Nothing -> Nothing
    Just v1  -> case evalM e2 of
      Nothing -> Nothing
      Just v2  -> if v2 == 0
                    then raiseError
                    else Just (v1 `div` v2)
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
  - Variaciones: original, error, **imprimir** y estado

```
evalP :: E -> ... Int
evalP (Cte n) = ...
evalP (Div e1 e2) =
    ... evalP e1
    ... evalP e2
    ...
```

- ¿Cómo mostrar una traza (*trace*)?

# Mónadas como generalización

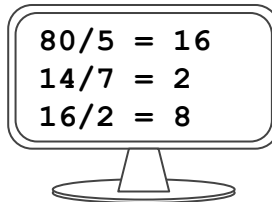
```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
  - Variaciones: original, error, **imprimir** y estado

```
evalP :: E -> ... Int
evalP (Cte n) = ...
evalP (Div e1 e2) =
    ... evalP e1
    ... evalP e2
    ...
```

- ¿Cómo mostrar una traza (*trace*)?

```
evalP (Div (Div (Cte 80) (Cte 5))
           (Div (Cte 14) (Cte 7)))
= 8
```





# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
  - Variaciones: original, error, **imprimir** y estado

```
evalP :: E -> ... Int
evalP (Cte n) = ...
evalP (Div e1 e2) =
    ... evalP e1
    ... evalP e2
    ...
```

```
evalP (Div (Div (Cte 80) (Cte 5))
           (Div (Cte 14) (Cte 7)))
= (8, "80/5 = 16
      14/7 = 2
      16/2 = 8")
```

- ¿Cómo mostrar una traza (*trace*)?

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, error, imprimir y estado

```
evalP :: E -> Output Int
evalP (Cte n)      = ...
evalP (Div e1 e2) =
    ... evalP e1
    ... evalP e2
    ...
```

```
type Output a = (a, Screen)
type Screen = String
evalP (Div (Div (Cte 80) (Cte 5))
         (Div (Cte 14) (Cte 7)))
= (8, "80/5 = 16
      14/7 = 2
      16/2 = 8")
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, error, imprimir y estado

```
evalP :: E -> Output Int
evalP (Cte n)      = ...
evalP (Div e1 e2) =
    ... evalP e1
    ... evalP e2
    ...
```

```
type Output a = (a, Screen)
type Screen = String
printf :: String -> Screen
printf msg = msg
formatDiv x y d =
    show x ++ "/" ++ show y ++ "="
    ++ show d ++ "\n"
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, error, imprimir y estado

```
evalP :: E -> Output Int
evalP (Cte n)      = ...
evalP (Div e1 e2) =
  let (v1,p1) = evalP e1
  in          ...      evalP e2
  ...
```

```
type Output a = (a, Screen)
type Screen = String
printf :: String -> Screen
printf msg = msg
formatDiv x y d =
  show x ++ "/" ++ show y ++ "="
  ++ show d ++ "\n"
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, error, imprimir y estado

```
evalP :: E -> Output Int
evalP (Cte n)      = ...
evalP (Div e1 e2) =
  let (v1,p1) = evalP e1
  in let (v2,p2) = evalP e2
     in
       ...
       ... v1 `div` v2 ...
```

```
type Output a = (a, Screen)
type Screen = String
printf :: String -> Screen
printf msg = msg
formatDiv x y d =
  show x ++ "/" ++ show y ++ "="
  ++ show d ++ "\n"
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, error, imprimir y estado

```
evalP :: E -> Output Int
evalP (Cte n)      = ...
evalP (Div e1 e2) =
  let (v1,p1) = evalP e1
  in let (v2,p2) = evalP e2
    in let p3 =
        ...
    in (v1 `div` v2, p1++p2++p3)
```

```
type Output a = (a, Screen)
type Screen = String
printf :: String -> Screen
printf msg = msg
formatDiv x y d =
  show x ++ "/" ++ show y ++ "="
  ++ show d ++ "\n"
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- ❑ Ejemplo simplificado para focalizar
- ❑ Variaciones: original, error, imprimir y estado

```
evalP :: E -> Output Int
evalP (Cte n)      = ...
evalP (Div e1 e2) =
  let (v1,p1) = evalP e1
  in let (v2,p2) = evalP e2
     in let p3 = printf
        (formatDiv v1 v2 (v1 `div` v2))
     in (v1 `div` v2, p1++p2++p3)
```

```
type Output a = (a, Screen)
type Screen = String
printf :: String -> Screen
printf msg = msg
formatDiv x y d =
  show x ++ "/" ++ show y ++ "="
  ++ show d ++ "\n"
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
```

- ❑ Ejemplo simplificado para focalizar
- ❑ Variaciones: original, error, imprimir y estado

```
evalP :: E -> Output Int
evalP (Cte n)      = (n, "")
evalP (Div e1 e2) =
  let (v1,p1) = evalP e1
  in let (v2,p2) = evalP e2
    in let p3 = printf
      (formatDiv v1 v2 (v1 `div` v2))
    in (v1 `div` v2, p1++p2++p3)
```

```
type Output a = (a, Screen)
type Screen = String
printf :: String -> Screen
printf msg = msg
formatDiv x y d =
  show x ++ "/" ++ show y ++ "="
  ++ show d ++ "\n"
```



# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

- Ejemplo simplificado para focalizar
  - Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> ... Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
    ... evalST e1 ...
    ... evalST e2 ...
    ...
```

- ¿Cómo calcular la cantidad de divisiones?

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

- Ejemplo simplificado para focalizar
  - Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> ... Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
    ... evalST e1 ...
    ... evalST e2 ...
    ...
```

```
evalP (Div (Div (Cte 80) (Cte 5))
          (Div (Cte 14) (Cte 7)))
= (8, ("nDiv", 3))
```

- ¿Cómo calcular la cantidad de divisiones?
  - Una variable “estática” que guarde ese número

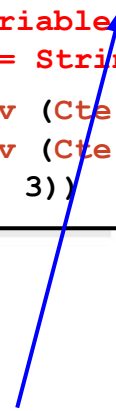
# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
    ... evalST e1 ...
    ... evalST e2 ...
    ...
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable Int)
type Variable = String
evalP (Div (Div (Cte 80) (Cte 5))
          (Div (Cte 14) (Cte 7)))
    = (8, ("nDiv", 3))
```



- ¿Cómo calcular la cantidad de divisiones?
  - Una variable “estática” que guarde ese número
  - Es necesario transformar una estado de memoria

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

■ Ejemplo simplificado para focalizar

■ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
    ... evalST e1 ...
    ... evalST e2 ...
    ...
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
    ("nDiv",d+1)
```

■ ¿Pero cómo usar este transformador de estados?

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

■ Ejemplo simplificado para focalizar

■ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
    ... evalST e1 ...
    ... evalST e2 ...
    ...
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
    ("nDiv",d+1)
```

■ ¿Pero cómo usar este transformador de estados?

```
evalS :: E -> (Int, Mem)
evalS e = ...
```

■ ¿Cuál es el valor inicial de la variable **nDiv**?

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

■ Ejemplo simplificado para focalizar

■ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
    ... evalST e1 ...
    ... evalST e2 ...
    ...
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
    ("nDiv",d+1)
```

■ ¿Pero cómo usar este transformador de estados?

```
evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

■ ¿Cuál es el valor inicial de la variable **nDiv**?

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

❏ Ejemplo simplificado para focalizar

❏ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
  \s0-> ... evalST e1 ...
        ... evalST e2 ...
        ...
```

```
evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
  ("nDiv",d+1)
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

❏ Ejemplo simplificado para focalizar

❏ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
  \s0-> let (v1,s1) = (evalST e1) s0
  in      ...      evalST e2 ...
  ...
```

```
evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
  ("nDiv",d+1)
```



# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

❏ Ejemplo simplificado para focalizar

❏ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
  \s0-> let (v1,s1) = (evalST e1) s0
        in let (v2,s2) = (evalST e2) s1
           in ...
```

```
evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
  ("nDiv",d+1)
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

- Ejemplo simplificado para focalizar
- Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
  \s0-> let (v1,s1) = (evalST e1) s0
        in let (v2,s2) = (evalST e2) s1
            in
              ...
              ... v1 `div` v2 ...

evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
  ("nDiv",d+1)
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

❑ Ejemplo simplificado para focalizar

❑ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
  \s0-> let (v1,s1) = (evalST e1) s0
        in let (v2,s2) = (evalST e2) s1
        in let s3 = ...
        in (v1 `div` v2, s3)

evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
  ("nDiv",d+1)
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

❑ Ejemplo simplificado para focalizar

❑ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = ... n ...
evalST (Div e1 e2) =
  \s0-> let (v1,s1) = (evalST e1) s0
        in let (v2,s2) = (evalST e2) s1
            in let s3 = inc "nDiv" s2
                in (v1 `div` v2, s3)

evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
  ("nDiv",d+1)
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
```

❑ Ejemplo simplificado para focalizar

❑ Variaciones: original, error, imprimir y **estado**

```
evalST :: E -> StateT Int
evalST (Cte n)      = \s0-> (n,s0)
evalST (Div e1 e2) =
  \s0-> let (v1,s1) = (evalST e1) s0
        in let (v2,s2) = (evalST e2) s1
            in let s3 = inc "nDiv" s2
                in (v1 `div` v2, s3)

evalS :: E -> (Int, Mem)
evalS e = (evalST e) ("nDiv",0)
```

```
type StateT a = Mem -> (a, Mem)
type Mem = (Variable, Int)
type Variable = String
inc :: Variable -> Mem -> Mem
inc "nDiv" ("nDiv",d) =
  ("nDiv",d+1)
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
evalST :: E -> StateT Int
```

- ❑ Similitudes y diferencias de todos los ejemplos
  - ❑ Partes en común, en **negro**
  - ❑ Tipos que se desean agregar, en **azul**
  - ❑ Partes que se desean agregar, en **verde**
  - ❑ Partes que se deben agregar obligatoriamente, en **rojo**
- ❑ ¿Cómo minimizar el impacto de los cambios en rojo?
  - ❑ Las partes rojas son “cañerías”
  - ❑ No deberían estar a la vista...
  - ❑ Abstraer las diferencias, con la técnica de los recuadros

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
evalST :: E -> StateT Int
```

- Antes de proceder, analizar los 3 juntos
- Variaciones: original, error, imprimir y estado

```
eval :: E -> Int
eval (Cte n)      = n
eval (Div e1 e2) = let v1 = eval e1
                   in let v2 = eval e2
                   in v1 `div` v2
```

```
evalM :: E -> Maybe Int
evalM (Cte n)      = Just n
evalM (Div e1 e2) =
  case evalM e1 of
    Nothing -> Nothing
    Just v1 -> case evalM e2 of
      Nothing -> Nothing
      Just v2 ->
        if v2 == 0
        then raiseError
        else Just (v1 `div` v2)
```

```
evalP :: E -> Output Int
evalP (Cte n)      = (n, "")
evalP (Div e1 e2) =
  let (v1,p1) = evalP e1
  in let (v2,p2) = evalP e2
  in let p3 =
      printf (formatDiv v1 v2
                    (v1 `div` v2))
  in (v1 `div` v2, p1++p2++p3)
```

```
evalST :: E -> StateT Int
evalST (Cte n)      = \s0-> (n,s0)
evalST (Div e1 e2) =
  \s0-> let (v1,s1) = (evalST e1) s0
  in let (v2,s2) = (evalST e2) s1
  in let s3 = inc "nDiv" s2
  in (v1 `div` v2, s3)
```



Tomar el desvío...



**PRECAUCION**  
**OBRA EN**  
**CONSTRUCCION**





# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```



**PRECAUCION**

**OBRA EN  
CONSTRUCCION**

simplificado para focalizar  
versiones: original, **error**, imprimir y estado

```
: E -> Maybe Int
```

```
Cte n)      = Just n
```

```
Div e1 e2) =
```

```
evalM e1 of
```

```
ng -> Nothing
```

```
v1' -> (\v1-> case evalM e2 of
```

```
Nothing -> Nothing
```

```
Just v2 ->
```

```
if v2 == 0
```

```
then raiseError
```

```
else Just (v1 `div` v2)
```

```
) v1'
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```



**PRECAUCION**

**OBRA EN  
CONSTRUCCION**

simplificado para focalizar  
versiones: original, **error**, imprimir y estado

```
: E -> Maybe Int
```

```
Cte n)      = Just n
```

```
Div e1 e2) =
```

```
evalM e1 of
```

```
ng -> Nothing
```

```
v1' -> (\v1-> case evalM e2 of
```

```
Nothing -> Nothing
```

```
Just v2' -> (\v2-> if v2 == 0
                  then raiseError
                  else Just (v1 `div` v2)
```

```
) v2'
```

```
) v1'
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```

# Mónadas como generalización

```
data E = Cte Int | Div E E
eval   :: E -> Int
```



**PRECAUCION**

**OBRA EN  
CONSTRUCCION**

simplificado para focalizar  
versiones: original, **error**, imprimir y estado

```
: E -> Maybe Int
```

```
Cte n)      = Just n
```

```
Div e1 e2) =
```

```
evalM e1 of
```

```
ng -> Nothing
```

```
v1' -> (\v1-> case evalM e2 of
```

```
Nothing -> Nothing
```

```
Just v2' -> (\v2-> if v2 == 0
```

```
then raiseError
```

```
else Just (v1 `div` v2)
```

```
) v2'
```

```
) v1'
```

```
data Maybe a = Nothing | Just a
raiseError = Nothing
```



Continuar por acá



# Definición de mónadas

# Mónadas: definición

```
data E = Cte Int | Div E E
eval   :: E -> Int
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
evalST :: E -> StateT Int
```

- Una mónada es un tipo paramétrico  $M$   $a$  con operaciones básicas

`return :: a -> M a` -- return

`(>>=) :: M a -> (a -> M b) -> M b` -- bind

- que satisfacen las siguientes propiedades

- para todo  $e$ . para todo  $k$ . `return e >>= k = k e`

- para todo  $m$ . `m >>= \x -> return x = m`

- para todo  $m$ . para todo  $n$ . para todo  $p$ . si  $x$  no aparece en  $p$ , entonces

`m >>= (\x -> n >>= \y -> p) = (m >>= \x -> n) >>= \y -> p`

# Mónadas: intuición

**M** a

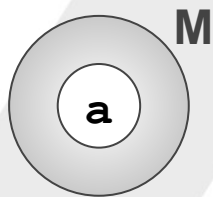
**return** :: a -> **M** a

-- return

(>>=) :: **M** a -> (a -> **M** b) -> **M** b

-- bind

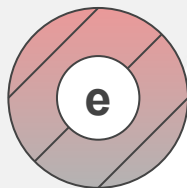
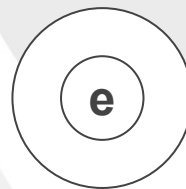
- Una mónada incorpora *efectos* a un valor
- El tipo **M** a incorpora la *información* necesaria
- return** agrega el *efecto nulo*
- (>>=) *combina* efectos con *dependencia* de datos



**return**

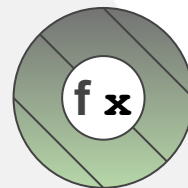


=

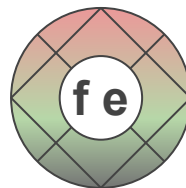


>>=

\x ->



=

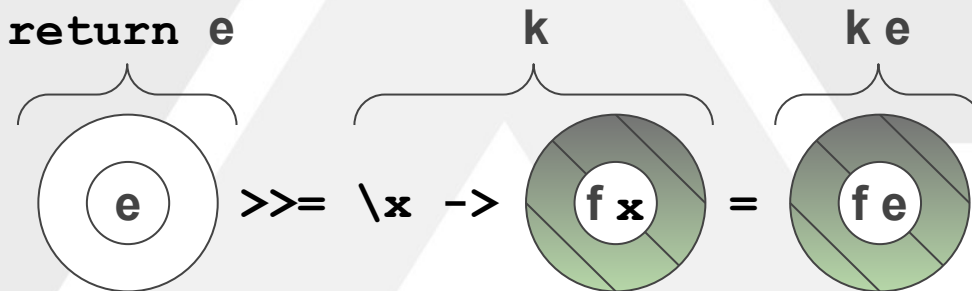


# Mónadas: intuición

■ Sentido de las propiedades

■ para todo  $e$ . para todo  $k$ .

$\text{return } e \gg= k = k e$



$M a$

$\text{return} :: a \rightarrow M a$

-- return

$(\gg=) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b$

-- bind

$\text{return } e =$

$e \gg= \backslash x \rightarrow$

$=$

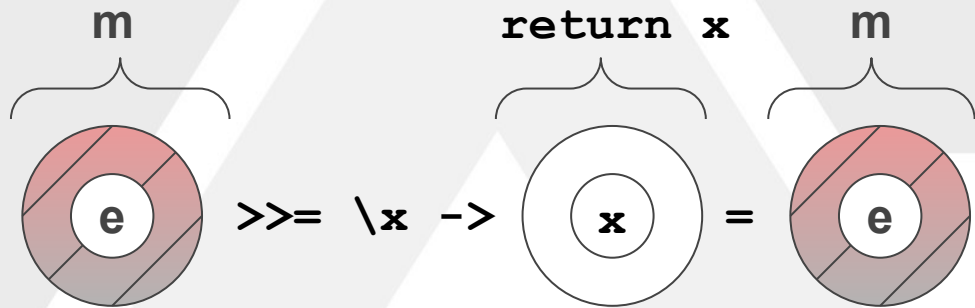


# Mónadas: intuición

■ Sentido de las propiedades

■ para todo  $m$ .

$$m \gg= \backslash x \rightarrow \text{return } x = m$$



$M$   $a$

$\text{return} :: a \rightarrow M a$

-- return

$(\gg=) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b$

-- bind

$$\text{return } e = \text{circle}(e)$$

$$e \gg= \backslash x \rightarrow f x = f e$$

# Mónadas: intuición

**M** a

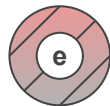
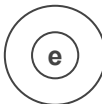
`return :: a -> M a`

-- return

`(>>=) :: M a -> (a -> M b) -> M b`

-- bind

`return e =`



`>>= \x ->`



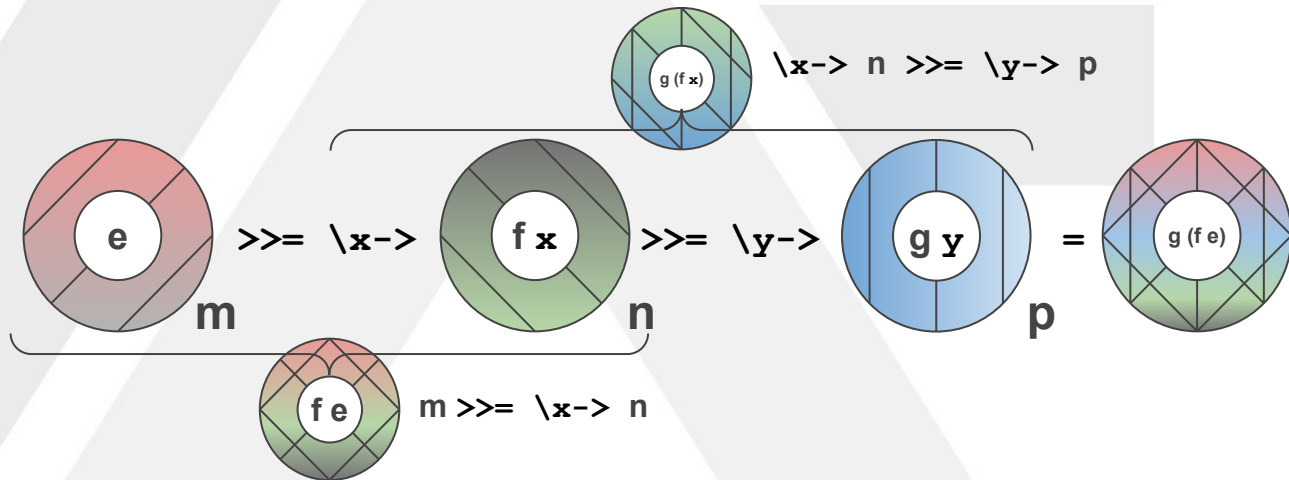
`=`



## ■ Sentido de las propiedades

- para todo **m**. para todo **n**. para todo **p**.  
si **x** no aparece en **p**, entonces

$$m \gg= (\backslash x \rightarrow n \gg= \backslash y \rightarrow p) = (m \gg= \backslash x \rightarrow n) \gg= \backslash y \rightarrow p$$



# Mónadas: intuición

```
M a
return :: a -> M a           -- return
(>>=) :: M a -> (a -> M b) -> M b  -- bind
```

- ❑ Una mónada incorpora *efectos* a un valor
- ❑ Es una forma de *abstraer* comportamientos específicos en un cómputo
- ❑ Es una idea similar al pattern Strategy en OOP
- ❑ Observar el código final de cada ejemplo, y cómo lucen muy similares
- ❑ Sin embargo, pueden ejecutar muy diferente

```
eval (Cte n) = return n
eval (Div e1 e2) =
  eval e1 >>= \v1 ->
  eval e2 >>= \v2 ->
  <alguna modificación>
  return (v1 `div` v2)
```

# Mónadas: definición

```
M a
return :: a -> M a           -- return
(>>=) :: M a -> (a -> M b) -> M b  -- bind
```

- ❑ ¿Pero cuáles efectos combina una mónada?
- ❑ Cada mónada provee una o varias *operaciones adicionales* que la diferencian de otras mónadas
  - ❑ **Maybe** provee **raiseError** (con **returnM** y **bindM**)
  - ❑ **Output** provee **printf** (con **returnO** y **bindO**)
  - ❑ **StateT** provee **inc** (con **returnS** y **bindS**)
- ❑ Estas operaciones adicionales expresan los efectos específicos en cada caso
  - ❑ Fallar, imprimir, modificar el estado, etc.

# Mónadas: notación

- ❑ ¿Hace falta nombrar el **return** y (**>>=**) de cada mónada diferente en forma diferente?
- ❑ Para evitar eso, Haskell provee un *sistema de clases*
  - ❑ **Maybe** provee **raiseError** (con **return** y (**>>=**))
  - ❑ **Output** provee **printf** (con **return** y (**>>=**))
  - ❑ **StateT** provee **inc** (con **return** y (**>>=**))
- ❑ El sistema de clases es una extensión del sistema H-M que permite que el tipo de **return** y (**>>=**) sean genéricos



# **Disgresión: sistema de clases**

# Sistema de clases

- ❑ Definición de clases
  - ❑ Una **clase** en Haskell es un *conjunto de tipos que comparten el nombre* de una o más funciones
  - ❑ Se utiliza la palabra clave **class** para declararla

```
class Eq a where
    (==) :: a -> a -> Bool
```
  - ❑ Es un concepto similar al de *interfaces* en Java, o *prototipos* en otros lenguajes

# Sistema de clases

```
class Eq a where  
  (==) :: a -> a -> Bool
```

## ❑ Instanciación de clases

- ❑ Para pertenecer a la clase, cada tipo tiene que declarar una implementación de las funciones de la clase
- ❑ Se utiliza la palabra clave **instance** para eso

```
instance Eq Bool where  
  True  == True  = True  
  False == False = True  
  _     == _     = False
```

- ❑ La implementación de **(==)** es específica para **Bool**



# Sistema de clases

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  True  == True  = True
  False == False = True
  _      == _     = False
```

- ❑ Instanciación de clases, 2
  - ❑ Para pertenecer a la clase, cada tipo tiene que declarar una implementación de las funciones de la clase
  - ❑ Se utiliza la palabra clave **instance** para eso

```
instance Eq Int where -- Ineficiente
  0 == 0 = True
  0 == _ = False
  n == m = if n>0 then (n-1) == (m-1)
            else (n+1) == (m+1)
```

- ❑ Otra implementación de **(==)**, específica para **Int**

# Sistema de clases

## ■ Tipos de funciones con clase

- Para usar una función de una clase se debe pedir que el tipo pertenezca a ella
- Se usan *contextos* antes de  $\Rightarrow$  para eso

```
elem :: Eq a => a -> [a] -> Bool
elem x []      = False
elem x (y:ys) = x==y || elem x ys
```

- Observar el uso de  $(==)$  entre elementos de tipo **a**
- Si **a** no es de la clase **Eq** no se puede usar **elem**

```
class Eq a where
    (==) :: a -> a -> Bool

instance Eq Bool where
    True == True  = True
    False == False = True
    _     == _    = False

instance Eq Int where
    0 == 0 = True
    0 == _ = False
    n == m =
        if n>0
        then (n-1) == (m-1)
        else (n+1) == (m+1)
```

# Sistema de clases

- Al usar una función, se verifica que sus tipos cumplan con el contexto

- Se puede usar `elem` con `Int` o `Bool`

```
elem True [False] = False
elem 2 [1,2,3] = True
```

- Pero no se puede usar con tipos que no sean instancia

```
elem succ [\x->x+1,id] ::
```

Porque `(Int->Int)`  
NO es instancia de `Eq`

- ¿Qué definición de `(==)` debería usarse?

```
class Eq a where
    (==) :: a -> a -> Bool
instance Eq Bool where
    ...
instance Eq Int where
    ...
elem :: Eq a =>
    a -> [a] -> Bool
elem x [] = False
elem x (y:ys) =
    x==y || elem x ys
```

# Sistema de clases

- ❑ Es una nueva extensión del sistema de tipos H-M
  - ❑ Se conoce como ***polimorfismo ad-hoc*** o ***sobrecarga*** (*overloading*)
  - ❑ La implementación de ciertas funciones se decide recién cuando se utilizan, en base al tipo
  - ❑ Sin embargo, se resuelve todo en forma *estática*
  - ❑ Deben usarse con cuidado porque generan muchas ambigüedades y código con problemas



**Fin de la disgresión**

# Mónadas: declaración

```
M a
return :: a -> M a           -- return
(>>=) :: M a -> (a -> M b) -> M b -- bind
```

- ❑ ¿Hace falta nombrar el `return` y `(>>=)` de cada mónada diferente en forma diferente?
- ❑ Con el *sistema de clases* los nombres son genéricos

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

Definición en **Haskell Standard**.  
GHC lo implementa levemente  
diferente desde v.7.10

- ❑ Haskell agrega `fail` a la clase, para elegir como fallar
- ❑ Acá `m` es un parámetro del sistema de tipos

# Mónadas: declaración

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

- ❑ Para ser una mónada, un tipo debe declararse como instancia de **Monad** (proveyendo **return** y **(>>=)**)

```
instance Monad Maybe where
  return x = Just x
  m >>= k = case m of
    Nothing -> Nothing
    Just v   -> k v
```

- ❑ Hay operaciones no imprescindibles (como **fail**)
- ❑ Las operaciones adicionales de la mónada van aparte

# Mónadas: declaración

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

instance Monad Maybe where ...
```

- ❑ Para ser una mónada, un tipo debe declararse como instancia de **Monad** (proveyendo **return** y **(>>=)**)

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

```
instance Monad StateT where
```

```
  return x      = ST (\s -> (x, s))
```

```
  (ST f) >>= k = ST (\s -> let (v, s') = f s
                               ST g     = k v
                               in g s')
```

- ❑ Las operaciones adicionales de la mónada van aparte



# Mónadas: declaración

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

instance Monad Maybe where ...
instance Monad StateT where ...
```

- ❑ Para ser una mónada, un tipo debe declararse como instancia de **Monad** (proveyendo **return** y **(>>=)**)

```
data Output a = Out (a, String)
```

```
instance Monad Output where
  return x          = Out (x, "")
  Out (v,p) >>= k = Out (let Out (v',p') = k v
                        in (v', p++p'))
```

- ❑ Las operaciones adicionales de la mónada van aparte

# Mónadas: declaración

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...
```

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

```
eval :: ??
eval (Cte n)      = return n
eval (Div e1 e2) = eval e1 >>= \v1 ->
                    eval e2 >>= \v2 ->
                    return (v1 `div` v2)
```

❏ ¿Qué tipo debe tener **eval**?

ANTES

```
data E = Cte Int | Div E E
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
evalST :: E -> StateT Int
```

# Mónadas: declaración

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...
```

❑ Código paramétrico con clases

❑ Al código monádico lo provee el sistema de tipos

```
eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) = eval e1 >>= \v1 ->
                    eval e2 >>= \v2 ->
                    return (v1 `div` v2)
```

ANTES

```
data E = Cte Int | Div E E
evalM  :: E -> Maybe Int
evalP  :: E -> Output Int
evalST :: E -> StateT Int
```

❑ No se decide el tipo exacto de **eval** hasta su utilización

❑ **return** y **(>>=)** admiten diferentes implementaciones

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

```
> eval (Div (Cte 10) (Cte 5))  
??
```

```
class Monad m where  
  return :: a -> m a  
  (>>=)  :: m a -> (a -> m b) -> m b  
  fail   :: String -> m a  
  
instance Monad Maybe where ...  
instance Monad StateT where ...  
instance Monad Output where ...  
  
eval :: Monad m => E -> m Int  
eval (Cte n)      = return n  
eval (Div e1 e2) =  
    eval e1 >>= \v1 ->  
    eval e2 >>= \v2 ->  
    return (v1 `div` v2)
```

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

> eval (Div (Cte 10) (Cte 5))  
error: Unresolved overloading  
>

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...

eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    return (v1 `div` v2)
```

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

```
> eval (Div (Cte 10) (Cte 5))  
error: Unresolved overloading  
> eval (Div (Cte 10) (Cte 5)) :: Maybe Int  
??
```

```
class Monad m where  
  return :: a -> m a  
  (>>=)  :: m a -> (a -> m b) -> m b  
  fail   :: String -> m a  
  
instance Monad Maybe where ...  
instance Monad StateT where ...  
instance Monad Output where ...  
  
eval :: Monad m => E -> m Int  
eval (Cte n)      = return n  
eval (Div e1 e2) =  
    eval e1 >>= \v1 ->  
    eval e2 >>= \v2 ->  
    return (v1 `div` v2)
```

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

```
> eval (Div (Cte 10) (Cte 5))
error: Unresolved overloading
> eval (Div (Cte 10) (Cte 5)) :: Maybe Int
Just 2
>
```

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...

eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    return (v1 `div` v2)
```

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

```
> eval (Div (Cte 10) (Cte 5))
error: Unresolved overloading
> eval (Div (Cte 10) (Cte 5)) :: Maybe Int
Just 2
> eval (Div (Cte 10) (Cte 5)) :: Output Int
??
```

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...

eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    return (v1 `div` v2)
```



# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

```
> eval (Div (Cte 10) (Cte 5))
error: Unresolved overloading
> eval (Div (Cte 10) (Cte 5)) :: Maybe Int
Just 2
> eval (Div (Cte 10) (Cte 5)) :: Output Int
Out (2, "")
>
```

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...

eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    return (v1 `div` v2)
```

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

```
> eval (Div (Cte 10) (Cte 5))
error: Unresolved overloading
> eval (Div (Cte 10) (Cte 5)) :: Maybe Int
Just 2
> eval (Div (Cte 10) (Cte 5)) :: Output Int
Out (2, "")
> fromJust (eval (Div (Cte 10) (Cte 5)))
??
```

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...

eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    return (v1 `div` v2)
```

:: Maybe a -> a

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

```
> eval (Div (Cte 10) (Cte 5))
error: Unresolved overloading
> eval (Div (Cte 10) (Cte 5)) :: Maybe Int
Just 2
> eval (Div (Cte 10) (Cte 5)) :: Output Int
Out (2, "")
> fromJust (eval (Div (Cte 10) (Cte 5)))
2
```

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...

eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    return (v1 `div` v2)
```

:: Maybe a -> a

# Mónadas: declaración

❏ Código paramétrico con clases

❏ Al código monádico lo provee el sistema de tipos

En Hugs. GHC utiliza un sistema de defaults complejo que evita esto.

```
> eval (Div (Cte 10) (Cte 5))
error: Unresolved overloading
> eval (Div (Cte 10) (Cte 5)) :: Maybe Int
Just 2
> eval (Div (Cte 10) (Cte 5)) :: Output Int
Out (2, "")
> fromJust (eval (Div (Cte 10) (Cte 5)))
2
```

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

instance Monad Maybe where ...
instance Monad StateT where ...
instance Monad Output where ...

eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    return (v1 `div` v2)
```

:: Maybe a -> a

❏ Observar que la expresión es siempre *la misma*

# Mónadas: más notación

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

instance Monad Maybe where ...
instance Monad StateT where ...
```

- ❑ La clase Monad es predefinida
- ❑ Permite usar notación especial para mónadas
- ❑ La *do-notation* permite escribir en forma más “imperativa”

```
eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Div e1 e2) = do v1 <- eval e1
                     v2 <- eval e2
                     return (v1 `div` v2)
```

- ❑ Se expresa la función que es 2do argumento de (>>=) como una “asignación” (pero sigue siendo parámetro)

# Mónadas: más notación

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

instance Monad Maybe where ...
instance Monad StateT where ...
```

## Notación do (*do-notation*)

Un cómputo monádico empieza con la palabra clave **do**

La expresión

**m >>= \x -> n**

se escribe

**do x <- m**  
**n**

La expresión

**m >>= \\_ -> n**

se escribe

**do m**  
**n**

Si hay varios **do** seguidos, solo hace falta el primero

El **return** queda sin cambios



# Utilización de mónadas

# Mónadas: utilización

- ❑ ¿Cómo utilizar una mónada?
  - ❑ Los valores monádicos solamente se combinan con otros valores monádicos
  - ❑ Se puede usar una mónada (perdiendo el efecto), pero no volver luego a la misma mónada
    - ❑ Todo lo que requiera efecto se “traslada” hacia dentro de la mónada
    - ❑ El uso de la mónada se puede realizar al final, para obtener el resultado



# Mónadas: utilización

## ❏ Como recordar valores, versión sin mónadas

### ❏ Tipo abstracto de datos ya trabajado

```
data Memoria          -- Tipo abstracto de datos
enBlanco  :: Memoria
  -- Una memoria vacía, que no recuerda nada
cuantoVale :: Variable -> Memoria -> Maybe Int
  -- El valor recordado para la variable, si existe
recordar   :: Variable -> Int -> Memoria -> Memoria
  -- Memoria con el recuerdo del valor para la variable
```

La memoria  
se maneja  
de forma  
explícita

# Mónadas: utilización

- ❑ Como recordar valores, versión CON mónadas
  - ❑ Tipo abstracto *monádico* con operaciones adicionales

```
data MemT ...           -- Tipo abstracto monádico
instance Monad MemT where ...
wipeMem  :: MemT ()
    -- Vacía la memoria para las operaciones que sigan
readMem  :: Variable -> MemT Int
    -- Lee el valor recordado para la variable, si existe
writeMem :: Variable -> Int -> MemT ()
    -- Modifica el recuerdo del valor para la variable
ifFails  :: MemT a -> MemT a -> MemT a
    -- Un try-catch: si la primera falla, usa la segunda
runMem   :: MemT a -> Memoria -> a
    -- Ejecuta la mónada en la memoria dada
```

La memoria  
se maneja  
de forma  
**implícita**

Se usa () cuando  
solamente  
importa el efecto

# Mónadas: utilización

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- Significado de **NExp**, versión sin mónadas
- Esta función fue definida en clases anteriores

```
evalNExp :: NExp -> (Memoria -> Int)
evalNExp (Var x) mem =
  case cuantoVale x mem of
    Nothing -> error ("Variable "++x++" indefinida")
    Just v   -> v
evalNExp (NCte n) mem = n
evalNExp (NBOp bop ne1 ne2) mem =
  evalNBOp bop (evalNExp ne1 mem)
                (evalNExp ne2 mem)
```

La memoria  
se maneja  
de forma  
explícita

# Mónadas: utilización

```
data MemT ...
instance Monad MemT where ...
wipeMem  :: MemT ()
readMem  :: Variable -> MemT Int
writeMem :: Variable -> Int -> MemT ()
ifFails  :: MemT a -> MemT a -> MemT a
runMem   :: MemT a -> Memoria -> a
```

Significado de **NExp**, versión CON monadas

No se menciona más la memoria

```
evalNExp :: NExp -> MemT Int
evalNExp (Var x) =
    readMem x `ifFails` fail ("Variable "
                              ++x++" indefinida")

evalNExp (NCte n) = return n
evalNExp (NBOp bop ne1 ne2) =
    evalNExp ne1 >>= \v1 ->
    evalNExp ne2 >>= \v2 ->
    return (evalNBOp bop v1 v2)
```

La memoria  
se maneja  
de forma  
implícita

Así se accede a los valores  
producidos en forma  
monádica (usando (>>=))

# Mónadas: utilización

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- Significado de un bloque de comandos, sin mónadas
- Otro código que se trabajó en clases anteriores

```
evalBloque :: Bloque -> (Memoria -> Memoria)
evalBloque []      = \mem -> mem
evalBloque (c:cs) =
    \mem -> let mem' = evalComando c mem
             in evalBloque cs mem'
```

La memoria  
se maneja  
de forma  
explícita

# Mónadas: utilización

```
data MemT ...
instance Monad MemT where ...
wipeMem  :: MemT ()
readMem  :: Variable -> MemT Int
writeMem :: Variable -> Int -> MemT ()
ifFails  :: MemT a -> MemT a -> MemT a
runMem   :: MemT a -> Memoria -> a
```

- Significado de un bloque de comandos, CON mónadas

- Nuevamente, no se menciona más la memoria

```
evalBloque :: Bloque -> MemT ()
evalBloque [] = return ()
evalBloque (c:cs) = evalComando c >>= \_ ->
                        evalBloque cs
```

La memoria  
se maneja  
de forma  
implícita

- La mónada **MemT** se encarga de administrar la memoria
  - El **return** deja la memoria *sin* modificar (*efecto nulo*)
  - El **(>>=)** *combina* las modificaciones a la memoria
  - Las operaciones específicas modifican la memoria

# Mónadas: utilización

```
data MemT ...  
instance Monad MemT where ...  
wipeMem  :: MemT ()  
readMem  :: Variable -> MemT Int  
writeMem :: Variable -> Int -> MemT ()  
ifFails  :: MemT a -> MemT a -> MemT a  
runMem   :: MemT a -> Memoria -> a
```

- Significado de un bloque de comandos, CON mónadas

- Nuevamente, no se menciona más la memoria

```
evalBloque :: Bloque -> MemT ()  
evalBloque []      = return ()  
evalBloque (c:cs) = do evalComando c  
                      evalBloque cs
```

La memoria  
se maneja  
de forma  
implícita

- La mónada **MemT** se encarga de administrar la memoria
  - El **return** deja la memoria *sin* modificar (*efecto nulo*)
  - El **(>>=)** *combina* las modificaciones a la memoria
  - Las operaciones específicas modifican la memoria

# Mónadas: utilización

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

■ Significado de un comando, sin mónadas

■ Más código que ya fue presentado antes

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
  \mem -> recordar x
              (evalNExp ne mem) mem
evalComando (If be cs1 cs2) =
  \mem -> if (evalBExp be mem)
              then evalBloque cs1 mem
              else evalBloque cs2 mem
```

...

La memoria  
se maneja  
de forma  
explícita



# Mónadas: utilización

```
data MemT ...  
instance Monad MemT where ...  
wipeMem  :: MemT ()  
readMem  :: Variable -> MemT Int  
writeMem :: Variable -> Int -> MemT ()  
ifFails  :: MemT a -> MemT a -> MemT a  
runMem   :: MemT a -> Memoria -> a
```

■ Significado de un comando, CON mónadas

■ Observar cómo los valores se obtienen con ( $\gg=$ )

```
evalComando :: Comando -> MemT ()  
evalComando (Assign x ne) =  
    evalNExp ne >>= \v ->  
        writeMem x v  
evalComando (If be cs1 cs2) =  
    evalBExp be >>= \b ->  
        if b then evalBloque cs1  
            else evalBloque cs2
```

...

La memoria  
se maneja  
de forma  
implícita

# Mónadas: utilización

```
data MemT ...  
instance Monad MemT where ...  
wipeMem  :: MemT ()  
readMem  :: Variable -> MemT Int  
writeMem :: Variable -> Int -> MemT ()  
ifFails  :: MemT a -> MemT a -> MemT a  
runMem   :: MemT a -> Memoria -> a
```

■ Significado de un comando, CON mónadas

■ Observar cómo los valores se obtienen con ( $\gg=$ )

```
evalComando :: Comando -> MemT ()  
evalComando (Assign x ne) =  
    do v <- evalNExp ne  
       writeMem x v  
evalComando (If be cs1 cs2) =  
    do b <- evalBExp be  
       if b then evalBloque cs1  
              else evalBloque cs2
```

...

La memoria  
se maneja  
de forma  
implícita

# Mónadas: utilización

```
data MemT ...
instance Monad MemT where ...
wipeMem  :: MemT ()
readMem  :: Variable -> MemT Int
writeMem :: Variable -> Int -> MemT ()
ifFails  :: MemT a -> MemT a -> MemT a
runMem   :: MemT a -> Memoria -> a
```

■ Utilizando un programa monádico

■ Para ejecutar una mónada debe haber una operación adicional que lo permita

```
evalPrograma :: Programa -> MemT ()
```

```
(evalPrograma pej) :: MemT ()
```

```
runMem (evalPrograma pej) :: Memoria -> ()
```

```
runMem (do evalPrograma pej
            readMem "n") :: Memoria -> Int
```

```
pej =
  Prog
    [ Assign "a" (NCte 21)
      Assign "n"
        (NBOp Mul (Var "a")
                  (NCte 2))
    ]
```

■ Así se usan valores monádicos *fuera* de la mónada



# **Mónadas y el mundo físico**

# Mónada IO: motivación

- ❑ ¿Cómo operar en el mundo físico?
  - ❑ ¿Cómo leer caracteres del teclado?
  - ❑ ¿Cómo escribir caracteres en pantalla?
  - ❑ ¿Cómo leer un archivo?
  - ❑ ¿Cómo escribir un archivo?
    - ❑ Todas involucran interacción con el medio...
    - ❑ ... o sea, no se pueden escribir solamente como transformación de información

# Mónada IO: motivación

- ❑ ¿Cómo operar en el mundo físico?
  - ❑ Una solución que solo transforme información no alcanza
- leerChar :: Char**
- leerDosChar = (leerChar, leerChar)**
- ❑ ¿Qué características tiene **leerDosChar**?
  - ❑ Ambos caracteres son SIEMPRE el mismo
  - ❑ ¿Por qué?
- ❑ Para esto Haskell provee la mónada **IO**  
(por *Input/Output Monad*)

# Mónada IO: operaciones

- ❑ La mónada IO es una mónada *predefinida* en Haskell
- ❑ Tiene *muchísimas* operaciones adicionales

```
instance Monad IO where ...
getChar    :: IO Char
  -- Lee un caracter del buffer de teclado, y lo consume
putChar    :: Char -> IO ()
  -- Escribe el caracter dado en pantalla
readFile   :: FilePath -> IO String
  -- Lee el contenido del disco en forma de string (si puede)
writeFile  :: FilePath -> String -> IO ()
  -- Graba el contenido del string en disco (si puede)
...
  -- Muchísimas otras operaciones para interactuar con el mundo
```

El mundo  
concreto se  
maneja de  
forma  
implícita

# Mónada IO: ejemplos

```
instance Monad IO where ...  
getChar    :: IO Char  
putChar    :: Char -> IO ()  
readFile   :: FilePath -> IO String  
writeFile  :: FilePath -> String -> IO ()  
...
```

- Las operaciones de IO se combinan como las de cualquier otra mónada
- Usando **return** y (**>>=**) , o *do-notation*

```
readTwoChars :: IO (Char, Char)  
readTwoChars = do c1 <- getChar  
                  c2 <- getChar  
                  return (c1, c2)
```

RECORDAR: así  
se obtiene un  
valor de dentro de  
la mónada

Los caracteres  
leídos pueden ser  
DISTINTOS  
¿Por qué?



# Mónada IO: ejemplos

```
instance Monad IO where ...
getChar    :: IO Char
putChar    :: Char -> IO ()
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
...
```

- Las operaciones de IO se combinan como las de cualquier otra mónada
- Pero **no** se puede “salir” de adentro de IO. ¿Por qué?

- Suponer `unsafe :: IO a -> a`

- ¿Qué sucede con este ejemplo?

`leerChar :: Char`

`leerChar = unsafe getChar`

- Pensar si la siguiente propiedad es cierta

`¿ leerChar = leerChar ?`

¡Se *pierde* la  
transparencia  
referencial!

# Mónada IO: ejemplos

```
instance Monad IO where ...
getChar    :: IO Char
putChar    :: Char -> IO ()
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
...
```

- ❑ Las operaciones de IO se combinan como las de cualquier otra mónada
- ❑ Pero **no** se puede “salir” de adentro de IO
- ❑ Una expresión de tipo **IO a** es *pura*
  - ❑ O sea, denota a un *único valor*
  - ❑ Es una **descripción** de los efectos imperativos que sucederán al *ejecutarlo* imperativamente
  - ❑ ¿Cómo se ejecuta una expresión de tipo **IO a**?
    - ❑ Únicamente desde **main**, desde el mundo imperativo

# Mónada IO: ejemplos

```
instance Monad IO where ...
getChar    :: IO Char
putChar    :: Char -> IO ()
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
...
```

- Las operaciones de IO se combinan como las de cualquier otra mónada
- Usando **return** y **(>>=)**, o *do-notation*

```
fileToUpper :: FilePath -> IO ()
fileToUpper fn =
  do putStrLn "Procesando..."
     contents <- readFile fn
     writeFile fn (map toUpper contents)
```

El tipo `()` es como el void de C: indica que no hay valor de retorno

El procesamiento es puramente funcional

# Mónada IO: combinación de mundos

- ❑ ¿Cómo combinar el mundo concreto con el funcional?
- ❑ La lógica de la aplicación, totalmente funcional
- ❑ La entrada/salida se hace separada en “*top-level*”

```
fullApp :: StaticData -> IO ()
fullApp sdata =
    do ddata <- leerDatosDinamicos sdata
       escribirResultados sdata ddata
       (appLogic sdata ddata)
appLogic :: StaticData -> DynamicData -> Resultado
```

# Mónada IO: combinación de mundos

- ❑ ¿Cómo combinar el mundo concreto con el funcional?
- ❑ La lógica de la aplicación, totalmente funcional
- ❑ La entrada/salida se hace separada en “*top-level*”

```
fullApp :: StaticData -> IO ()
fullApp sdata =
    do ddata <- leerDatosDinamicos sdata
       escribirResultados sdata ddata
       (appLogic sdata ddata)
appLogic :: StaticData -> DynamicData -> Resultado
```

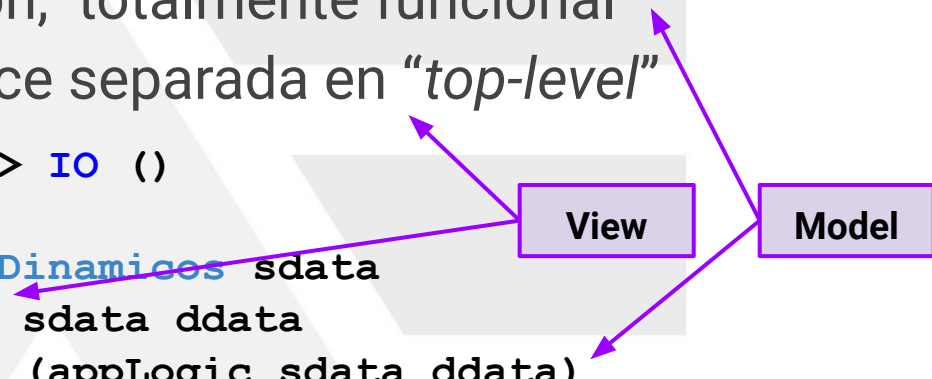
- ❑ Esto es similar al patrón Model-View-Controller (MVC)

# Mónada IO: combinación de mundos

- ❑ ¿Cómo combinar el mundo concreto con el funcional?
- ❑ La lógica de la aplicación, totalmente funcional
- ❑ La entrada/salida se hace separada en “*top-level*”

```
fullApp :: StaticData -> IO ()
fullApp sdata =
  do ddata <- leerDatosDinamicos sdata
     escribirResultados sdata ddata
     (appLogic sdata ddata)
appLogic :: StaticData -> DynamicData -> Resultado
```

- ❑ Esto es similar al patrón Model-View-Controller (MVC)
  - ❑ El **Controller** es el sistema de ejecución de Haskell



# Mónada IO: combinación de mundos

- ❑ ¿Cómo combinar el mundo concreto con el funcional?
- ❑ La lógica de la aplicación, totalmente funcional
- ❑ La entrada/salida se hace separada en “*top-level*”
- ❑ Cita de “*A Gentle Introduction to Haskell, Version 98*”

“The I/O monad constitutes a small imperative sub-language inside Haskell, and thus the I/O component of a program may appear similar to ordinary imperative code. But there is one important difference: There is no special semantics that the user needs to deal with. In particular, equational reasoning in Haskell is not compromised. The imperative feel of the monadic code in a program does not detract from the functional aspect of Haskell. An experienced functional programmer should be able to minimize the imperative component of the program, only using the I/O monad for a minimal amount of top-level sequencing. **The monad cleanly separates the functional and imperative program components.** In contrast, imperative languages with functional subsets do not generally have any well-defined barrier between the purely functional and imperative worlds.”

# Mónada IO: ¿cómo seguir?

- ❑ Deben explorarse las numerosas operaciones de **IO**
- ❑ No debe olvidarse que lógica e interfaz deben separarse
- ❑ Ejemplos de operaciones

- ❑ Más operaciones sobre texto

```
interactive :: (String -> String) -> IO ()  
getLine    :: IO String  
print      :: Show a => a -> IO ()
```

- ❑ Manejo de excepciones

```
catch :: IO a -> (IOError -> IO a) -> IO a  
ioError :: IOError -> IO a
```



# Mónada IO: ¿cómo seguir?

- ❑ Deben explorarse las numerosas operaciones de **IO**
- ❑ No debe olvidarse que lógica e interfaz deben separarse
- ❑ Ejemplos de operaciones
  - ❑ Manejo de carpetas (o directorios)

```
listDirectory :: FilePath -> IO [FilePath]
getCurrentDirectory :: IO FilePath
setCurrentDirectory :: FilePath -> IO ()
withCurrentDirectory :: FilePath -> IO a -> IO a
createDirectory :: FilePath -> IO ()
removeDirectory :: FilePath -> IO ()
renameDirectory :: FilePath -> FilePath -> IO ()
```

# Mónada IO: ¿cómo seguir?

- ❑ Deben explorarse las numerosas operaciones de **IO**
- ❑ No debe olvidarse que lógica e interfaz deben separarse
- ❑ Ejemplos de operaciones
  - ❑ Manejo de horas

```
data Clock = Monotonic | RealTime | ProcessCPUTime | ...
getTime    :: Clock -> IO TimeSpec
getRes     :: Clock -> IO TimeSpec
diffTimeSpec :: TimeSpec -> TimeSpec -> TimeSpec
fromNanoSecs :: Integer -> TimeSpec
toNanoSecs   :: TimeSpec -> Integer
```

# Mónada IO: ¿cómo seguir?

- ❑ Deben explorarse las numerosas operaciones de **IO**
- ❑ No debe olvidarse que lógica e interfaz deben separarse
- ❑ Ejemplos de operaciones
  - ❑ Manejo de fechas

```
getCurrentTime :: IO UTCTime
```

```
getSystemTime  :: IO SystemTime
```

```
currentDate :: IO (Integer, Int, Int) -- (yy, mm, dd)
```

```
currentDate = do t <- getCurrentTime  
               return (toGregorian (utctDay t))
```

# Mónada IO: ¿cómo seguir?

- ❑ Deben explorarse las numerosas operaciones de **IO**
- ❑ No debe olvidarse que lógica e interfaz deben separarse
- ❑ Hay muchas operaciones monádicas más
  - ❑ Manejo de *environment* (variables de entorno, argumentos, etc.)
  - ❑ Concurrencia (vía **forkIO** y **MVars**)
  - ❑ Memoria destructiva (vía **STRefs** y vía **IORefs**)
  - ❑ Manejo de archivos en bajo nivel (vía **hOpen**, **Handlers**, etc.)
  - ❑ Adaptación de números pseudo-aleatorios (vía **IOGenM** y otros)
  - ❑ y muchos más...

# Operaciones monádicas genéricas

# Operaciones monádicas genéricas

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Ejemplo: Combinación de mónadas usando funciones puras

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

```
liftM f mx = do x <- mx  
             return (f x)
```

```
liftM2 :: Monad m => (a->b->c) -> m a -> m b -> m c
```

```
liftM2 f mx my = do x <- mx  
                   y <- my  
                   return (f x y)
```

# Operaciones monádicas genéricas

```
liftM2 :: Monad m => (a->b->c)
      -> m a -> m b -> m c
liftM2 f mx my = do x <- mx
                  y <- my
                  return (f x y)
```

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Usando `liftM2` se puede representar mejor `eval`

```
(<+>) :: Monad m => m Int -> m Int -> m Int
(<+>) = liftM2 (+)
```

```
eval :: Monad m => E -> m Int
eval (Cte n)      = return n
eval (Suma e1 e2) = eval e1 <+> eval e2
...
```

En lugar de:

```
eval (Suma e1 e2) =
  do v1 <- eval e1
     v2 <- eval e2
     return (v1 + v2)
```

# Operaciones monádicas genéricas

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Ejemplo: Combinación de muchas mónadas, con valores

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

```
sequence [Just 1, Just 2, Just 3] = ??
```

```
sequence [Just 1, Nothing, Just 3] = ??
```



# Operaciones monádicas genéricas

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Ejemplo: Combinación de muchas mónadas, con valores

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

```
sequence [Just 1, Just 2, Just 3] = Just [1,2,3]
sequence [Just 1, Nothing, Just 3] = Nothing
```

# Operaciones monádicas g

```
instance Monad IO where ...
getChar    :: IO Char
putChar    :: Char -> IO ()
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
...
```

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Ejemplo: Combinación de muchas mónadas, solo efectos

```
sequence_ :: Monad m => [ m a ] -> m ()
sequence_ []      = return ()
sequence_ (m:ms) = do m
                  sequence_ ms
```

```
sequence_ [ putChar 'H', putChar 'o'
            , putChar 'l', putChar 'a' ]
= ??
```

# Operaciones monádicas g

```
instance Monad IO where ...  
getChar    :: IO Char  
putChar    :: Char -> IO ()  
readFile   :: FilePath -> IO String  
writeFile  :: FilePath -> String -> IO ()  
...
```

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Ejemplo: Combinación de muchas mónadas, solo efectos

```
sequence_ :: Monad m => [ m a ] -> m ()  
sequence_ []      = return ()  
sequence_ (m:ms) = do m  
                  sequence_ ms  
  
sequence_ [ putChar 'H', putChar 'o'  
            , putChar 'l', putChar 'a' ]  
= putStr "Hola"
```

# Operaciones monádicas genéricas

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Estas operaciones se pueden usar en otras

```
putStr, putStrLn :: String -> IO ()  
putStr msg = sequence_ (map putChar msg)  
putStrLn msg = putStr (msg ++ "\n")  
ejemploFinal = ??
```

# Operaciones monádicas genéricas

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Estas operaciones se pueden usar en otras

```
putStr, putStrLn :: String -> IO ()  
putStr msg = sequence_ (map putChar msg)  
putStrLn msg = putStr (msg ++ "\n")  
ejemploFinal = putStrLn "¡Chau, mundo!"
```

Gracias, Guido  
Montorfano  
(cursada virtual  
UNQ, 2020s1)

# Operaciones monádicas genéricas

- ❑ El poder de las mónadas está en la abstracción
- ❑ Se pueden definir infinidad de operaciones genéricas
- ❑ Estas operaciones se pueden usar en otras

```
putStr, putStrLn :: String -> IO ()  
putStr msg = sequence_ (map putChar msg)  
putStrLn msg = putStr (msg ++ "\n")  
ejemploFinal = putStrLn "¡Chau, mundo!"
```

Gracias, Guido  
Montorfano  
(cursada virtual  
UNQ, 2020s1)

- ❑ ¡La esencia del “*Hola, mundo*” NO es sencilla!
  - ❑ ¿Por qué empezar a enseñar un lenguaje por ahí?

# Conclusiones

- ❑ Las mónadas ofrecen un nivel de abstracción mayor, que resulta iluminador
- ❑ Hay un mundo de riquezas monádicas para explorar
  - ❑ La computación secuencial imperativa es solamente UNA de las posibles estrategias de cómputo
  - ❑ Hay *innumerables* otras estrategias para conocer
- ❑ **Pensar en abstracto cumple las promesas**





# Conclusiones

- ❑ Las mónadas ofrecen un nivel de abstracción mayor, que resulta iluminador
- ❑ Hay un mundo de riquezas monádicas para explorar
  - ❑ La computación secuencial imperativa es solamente UNA de las posibles estrategias de cómputo
  - ❑ Hay *innumerables* otras estrategias para conocer
- ❑ **Pensar en abstracto cumple las promesas**
- ❑ ¿Se puede ser todavía más abstracto?



# Resumen

# Resumen

- ❑ Mónadas como abstracción de comportamiento específico en cómputos
  - ❑ Definición
  - ❑ Propiedades
  - ❑ Notación
  - ❑ Ejemplos
  - ❑ Funciones genéricas sobre mónadas
- ❑ Expresión del mundo concreto en forma pura
  - ❑ Mónada IO
  - ❑ Características, ejemplos



¿Fin...?

