Programación Funcional Avanzada

Transformadores de Monads

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Evaluación pura convencional

Una calculadora, porque lenguajes y eso...

- Disponemos del tipo de datos recursivo

```
data Exp = Const Int
         | Var String
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
         | Empty
         deriving (Eq,Show)
```

- Usaremos `Data.Map` como Tabla de Símbolos

```
env = DM.fromList [ ("foo",42), ("bar",69),
                    ("baz",27), ("qux",0) ]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Evaluación pura convencional

Una calculadora, porque lenguajes y eso...

- Así podemos modelar expresiones de la forma

```
ex1 = (Const 21)
ex2 = (Var "foo")
ex3 = (Mul (Const 2) ex1)
ex4 = (Add (Mul ex2 (Const 5)) (ex3))
ex5 = (Add (Div (Const 42) (Const 2))
           (Mul (Const 3) (Const 7)))
```

- Y queremos evaluarlas con

```
eval :: Expr -> Int
```

UNIVERSIDAD SIMÓN BOLÍVAR

Evaluación pura convencional

Una calculadora, porque lenguajes y eso...

- Una solución pura simple y directa sería

```
eval :: Expr -> Int
eval (Const i) = i
eval (Var n)   = fromJust (DM.lookup n env)
eval (Add l r) = (eval l) + (eval r)
eval (Sub l r) = (eval l) - (eval r)
eval (Mul l r) = (eval l) * (eval r)
eval (Div l r) = (eval l) `div` (eval r)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Evaluación pura convencional

Una calculadora, porque lenguajes y eso...

- Una solución pura simple y directa sería

```
eval :: Expr -> Int
eval (Const i) = i
eval (Var n)   = fromJust (DM.lookup n env)
eval (Add l r) = (eval l) + (eval r)
eval (Sub l r) = (eval l) - (eval r)
eval (Mul l r) = (eval l) * (eval r)
eval (Div l r) = (eval l) `div` (eval r)
```

- Ambiente de referencia "cableado".
- Variable indefinida o división por cero generan excepción.
- Solamente podemos tener el resultado final, cuando exista.

UNIVERSIDAD SIMÓN BOLÍVAR

Abstracción del ambiente de referencia

Una aplicación directa del Monad Reader

- El Monad `Reader` permite efectuar un cómputo el cual puede consultar un ambiente de referencia "sólo para lectura"
- Consultas vía función de orden superior

```
asks :: MonadReader r m => (r -> a) -> m a
```

- Procesamiento indicando el ambiente inicial

```
runReader :: Reader r a -> r -> a
```

UNIVERSIDAD SIMÓN BOLÍVAR

Abstracción del ambiente de referencia

El secreto está en la firma

- Reescribimos

```
evalRD :: Exp -> Reader (DM.Map String Int) Int
evalRD (Const i) = return i
evalRD (Var n)   = do
  asks (\e -> maybe
              (error $ "Var "++ n ++ " not found")
              id (DM.lookup n e))
evalRD (Add l r) = liftM2 (+) (evalRD l) (evalRD r)
evalRD (Sub l r) = liftM2 (-) (evalRD l) (evalRD r)
evalRD (Mul l r) = liftM2 (*) (evalRD l) (evalRD r)
evalRD (Div l r) = liftM2 div (evalRD l) (evalRD r)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Abstracción del ambiente de referencia

El secreto está en la firma

- Reescribimos

```
evalRD :: Exp -> Reader (DM.Map String Int) Int
evalRD (Const i) = return i
evalRD (Var n)   = do
  asks (\e -> maybe
              (error $ "Var "++ n ++ " not found")
              id (DM.lookup n e))
evalRD (Add l r) = liftM2 (+) (evalRD l) (evalRD r)
evalRD (Sub l r) = liftM2 (-) (evalRD l) (evalRD r)
evalRD (Mul l r) = liftM2 (*) (evalRD l) (evalRD r)
evalRD (Div l r) = liftM2 div (evalRD l) (evalRD r)
```

- Y para evaluar

```
evalwithenv :: DM.Map String Int -> Exp -> IO ()
evalwithenv a e = putStrLn $
    "Result: " ++ (show $ runReader (evalRD e) a)
```

Obteniendo valores intermedios

Una aplicación directa del Monad Writer

- El Monad `Writer` permite efectuar un cómputo el cual puede agregar información a una bitácora "sólo para escritura" que podemos recuperar al final
- Registrar usando un monoide – `Data.Sequence` generalmente

```
tell :: w -> m ()
```

- Procesamiento parte con el `mempty` y entrega resultado y bitácora acumulada

```
runWriter :: Writer w a -> (a,w)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Obteniendo valores intermedios

El secreto está en la firma

- Auxiliar para generar los mensajes en `Data.Sequence`

```
logExp :: Exp -> Int -> Seq String
logExp e v = singleton $
   "Exp: " ++ show e ++ " -> Val: " ++ show v
```

UNIVERSIDAD SIMÓN BOLÍVAR

Obteniendo valores intermedios

El secreto está en la firma

- Auxiliar para generar los mensajes en `Data.Sequence`

```
logExp :: Exp -> Int -> Seq String
logExp e v = singleton $
   "Exp: " ++ show e ++ " -> Val: " ++ show v
```

- Y así reescribimos el evaluador

```
evalWR :: Exp -> Writer (Seq String) Int
evalWR t@(Const i) = do
  tell $ logExp t i
  return i
evalWR t@(Var n)   = do
  let v = maybe
            (error $ "Var "++ n ++ " not found")
            id (DM.lookup n env)
  tell $ logExp t v
  return v
```

Obteniendo valores intermedios

El secreto está en la firma

- Las operaciones binarias tienen todas la misma estructura

```
evalWR t@(Add l r) = do
  vl <- evalWR l
  vr <- evalWR r
  let v = vl + vr
  tell $ logExp t v
  return $ v
```

- Y el evaluador principal

```
evalwithlog :: Exp -> IO ()
evalwithlog e = putStr $
  unlines $ [ "Result: " ++ show r ] ++ DF.toList l
    where (r,l) = runWriter (evalWR e)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Manejando las excepciones

Una aplicación directa del Monad Error

- El Monad `Error` permite efectuar un cómputo el cual puede ser exitoso en generar un resultado o fallar indicando un error descriptivo.
- Generalmente se usa el tipo `Either e a` directamente.
- Nos interesa convertir los errores de evaluación en excepciones *puras*.
  - División por cero.
  - Variable indefinida.
  - Número de mala suerte – 13 en cualquier punto intermedio.

UNIVERSIDAD SIMÓN BOLÍVAR

Manejando las excepciones

Ligeramente más complicado

- Modelaremos nuestras excepciones con un tipo de datos algebráicos y lo instanciaremos como tales

```
data ExpError = DivisionPorCero
              | NumeroDeMalaSuerte
              | VariableNoExiste String
              deriving (Show)

instance Error ExpError
```

- Esto nos permite lanzar esas excepciones...

```
throwError :: MonadError e m => e -> m a
```

- ...y que el monad `Either` las atrape automáticamente.

Manejando las excepciones

El secreto está en la firma

- Incorporamos las verificaciones y lanzamos la excepción que corresponda.

```
evalEX :: Exp -> Either ExpError Int
evalEX (Const i) = checkForThirteen i
evalEX (Var n)   =
   maybe (throwError $ VariableNoExiste n)
         checkForThirteen
         (DM.lookup n env)
```

- Donde checkForThirteen es absolutamente pura

```
checkForThirteen 13 = throwError NumeroDeMalaSuerte
checkForThirteen i  = return i
```

UNIVERSIDAD SIMÓN BOLÍVAR

Manejando las excepciones

El secreto está en la firma

- Todos los cómputos intermedios deben ser verificados para detectar el número de mala suerte

```
checkMath op l r =
   liftM2 (op) (evalEX l) (evalEX r) >>=
   checkForThirteen
```

- Y así podemos escribir suma, resta y multiplicación como

```
evalEX (Add l r) = checkMath (+) l r
```

UNIVERSIDAD SIMÓN BOLÍVAR

Manejando las excepciones

El secreto está en la firma

- La división requiere mas verificaciones

```
evalEX (Div l r) = do
  vl <- evalEX l
  vr <- evalEX r
  if vr == 0 then throwError DivisionPorCero
             else checkForThirteen $ vl `div` vr
```

Manejando las excepciones

El secreto está en la firma

- La división requiere mas verificaciones

```
evalEX (Div l r) = do
  vl <- evalEX l
  vr <- evalEX r
  if vr == 0 then throwError DivisionPorCero
             else checkForThirteen $ vl `div` vr
```

- Y el evaluador principal

```
evalwithcare :: Exp -> IO ()
evalwithcare e = either bad good (evalEX e)
  where good  = msg "Result: "
        bad   = msg "Rayos: "
        msg s = putStrLn . (++)s . show
```

UNIVERSIDAD SIMÓN BOLÍVAR

Estadísticas o cualquier cosa que cambie

Una aplicación directa del Monad State

- El Monad `State` permite efectuar un cómputo el cual puede manipular información en un estado mutable que podemos acceder en cualquier punto.
- Obtener o establecer el estado mutable

```
get :: MonadState s m => m s
put :: MonadState s m => s -> m ()
```

- Procesamiento comienza con el estado inicial y produce resultados, estados o ambas cosas según convenga

```
evalState :: State s a -> s -> a
execState :: State s a -> s -> s
runState  :: State s a -> s -> (a,s)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Contando las Operaciones

Ligeramente más complicado

- Nos interesa recopilar estadísticas sobre la evaluación de expresiones.
- Usaremos el tipo de datos

```
data EvalState = EvalState { adds, subs,
                             muls, divs,
                             vars :: Int }
               deriving (Show)
```

para modelar el estado mutable en el cual hacer el conteo.

UNIVERSIDAD SIMÓN BOLÍVAR

Contando las operaciones

El secreto está en la firma

- En cada operación se actualiza el contador relevante

```
evalST :: Exp -> State EvalState Int
evalST (Const i) = return i
evalST (Add l r) = do
  vl <- evalST l
  vr <- evalST r
  s <- get
  put $ s { adds = adds s + 1 }
  return $ vl + vr
```

UNIVERSIDAD SIMÓN BOLÍVAR

Contando las operaciones

El secreto está en la firma

- En cada operación se actualiza el contador relevante

```
evalST :: Exp -> State EvalState Int
evalST (Const i) = return i
evalST (Add l r) = do
  vl <- evalST l
  vr <- evalST r
  s <- get
  put $ s { adds = adds s + 1 }
  return $ vl + vr
```

- Y el evaluador principal

```
evalwithstats :: Exp -> IO ()
evalwithstats e = putStr $
  unlines $ [ "Result: " ++ show r ] ++ [ show s ]
  where (r,s) = runState (evalST e) initialState
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si quiero todos los comportamientos?

A fistful of Monads

- Combinar uno o más comportamientos monádicos requiere construir un *monad stack*.
  - Convertir el cómputo puro al monad `Identity`.
  - Apilar transformadores encima.
  - Reemplazar `Identity` por `IO` de ser necesario.
- Protip: usar `type` para que las firmas sean decentes.

UNIVERSIDAD SIMÓN BOLÍVAR

Cómputo puro a Identity

La "plancha verde" del LEGO

```
type Eval1 a = Identity a

eval1 :: Env -> Exp -> Eval1 Int
eval1 env (Const i) = return i
eval1 env (Var n)   = return $
                        fromJust $ DM.lookup n env
eval1 env (Add l r) = do i1 <- eval1 env l
                         i2 <- eval1 env r
                         return $ i1 + i2
```

- Abstraemos el ambiente de ejecución en la primera etapa – desaparecerá cuando agreguemos comportamiento `Reader`.
- El resto de las operaciones son similares.

UNIVERSIDAD SIMÓN BOLÍVAR

Cómputo puro a Identity

El evaluador es trivial

```
evalM1 : : Eval1 a -> a
evalM1 = runIdentity

ghci> evalM1 (eval1 env ex2)
42
```

UNIVERSIDAD SIMÓN BOLÍVAR

Detección de errores

¿Resultado o error?

- Envolveremos el resultado del cómputo para determinar si fue exitoso o produjo una excepción.
- Utilizaremos el mismo modelo de excepciones previo

```
data ExpError = DivisionPorCero
              | NumeroDeMalaSuerte
              | VariableNoExiste String
              deriving (Show)

instance Error ExpError
```

- . . . pero "encima" del cómputo en `Identity`
- Reutilizaremos `checkForThirteen` y `checkMath`.

UNIVERSIDAD SIMÓN BOLÍVAR

Detección de errores

Transformador `ErrorT`

- Transformación – e es el tipo de los errores

```
ErrorT :: m (Either e a) -> ErrorT e m a
```

- Aprovechando las firmas

```
type Eval2 a = ErrorT ExpError Identity a

eval2 :: Env -> Exp -> Eval2 Int
eval2 env (Const i) = checkForThirteen i
eval2 env (Var n)   =
  maybe (throwError $ VariableNoExiste n)
        checkForThirteen
        (DM.lookup n env)
```

Detección de errores

Transformador `ErrorT`

- Los operadores binarios quedan

```
eval2 env (Add l r) = checkMath env (+) l r
eval2 env (Sub l r) = checkMath env (-) l r
eval2 env (Mul l r) = checkMath env (*) l r
eval2 env (Div l r) = do
  vl <- eval2 env l
  vr <- eval2 env r
  if vr == 0 then throwError DivisionPorCero
             else checkForThirteen $ vl `div` vr
```

UNIVERSIDAD SIMÓN BOLÍVAR

Detección de errores

¿Cómo evaluarlo?

- `ErrorT` envuelve a `Identity`.

```
evalM2 : : Eval2 a -> Either ExpError a
evalM2 = runIdentity . runErrorT
```

- La función de evaluación:
  - Desenvuelve de afuera hacia adentro.
  - Produce el valor de cómputo o la excepción en `Either`.

UNIVERSIDAD SIMÓN BOLÍVAR

Detección de errores

¿Cómo evaluarlo?

- `ErrorT` envuelve a `Identity`.

```
evalM2 : : Eval2 a -> Either ExpError a
evalM2 = runIdentity . runErrorT
```

- La función de evaluación:
  - Desenvuelve de afuera hacia adentro.
  - Produce el valor de cómputo o la excepción en `Either`.
- Así podemos evaluar

```
ghci> evalM2 (eval2 env ex2)
Right 42
ghci> evalM2 (eval2 (DM.fromList []) ex2)
Left (VariableNoExiste "foo")
```

UNIVERSIDAD SIMÓN BOLÍVAR

Escondiendo el Ambiente de Referencia

Plomería "sólo-lectura"

- Envolveremos el cómputo anterior para que acarree el ambiente de referencia desde el inicio.
- Transformación – `r` es el tipo del ambiente de referencia.

```
ReaderT :: (r -> m a) -> ReaderT r m a
```

- Aprovechando las firmas

```
type Eval3 a = ReaderT Env
                       (ErrorT ExpError Identity) a
```

- Eliminaremos el argumento `env` – necesitaremos `checkMath'` sin el ambiente como argumento.

UNIVERSIDAD SIMÓN BOLÍVAR

Escondiendo el Ambiente de Referencia

Plomería "sólo-lectura"

- Aprovechando las firmas

```
eval3 :: Exp -> Eval3 Int
eval3 (Const i) = checkForThirteen i
eval3 (Var n)   =
  do env <- ask
     maybe (throwError $ VariableNoExiste n)
           checkForThirteen
           (DM.lookup n env)
eval3 (Add l r) = checkMath' (+) l r
eval3 (Div l r) = do
  vl <- eval3 l
  vr <- eval3 r
  if vr == 0 then throwError DivisionPorCero
             else checkForThirteen $ vl `div` vr
```

- Consultar el ambiente cuando se trata de variables.

Ambiente de Referencia y Errores

¿Cómo evaluarlo?

- `ReaderT` envuelve a `ErrorT` que envuelve a `Identity`.

```
evalM3 :: Env -> Eval3 a -> Either ExpError a
evalM3 env =
  runIdentity . runErrorT . (flip runReaderT) env
```

- La función de evaluación:
  - Desenvuelve de afuera hacia adentro.
  - `runReaderT` recibe el cómputo primero y luego el ambiente.
  - Produce el valor de cómputo o la excepción en `Either`.

UNIVERSIDAD SIMÓN BOLÍVAR

Ambiente de Referencia y Errores

¿Cómo evaluarlo?

- `ReaderT` envuelve a `ErrorT` que envuelve a `Identity`.

```
evalM3 :: Env -> Eval3 a -> Either ExpError a
evalM3 env =
  runIdentity . runErrorT . (flip runReaderT) env
```

- La función de evaluación:
  - Desenvuelve de afuera hacia adentro.
  - `runReaderT` recibe el cómputo primero y luego el ambiente.
  - Produce el valor de cómputo o la excepción en `Either`.
- Así podemos evaluar

```
ghci> evalM3 env (eval3 ex2)
Right 42
ghci> evalM3 (DM.fromList []) (eval3 ex2)
Left (VariableNoExiste "foo")
```

UNIVERSIDAD SIMÓN BOLÍVAR

Calculando estadísticas

Plomería "estado mutable"

- Incorporamos estado mutable al cómputo subyacente para calcular las estadísticas de evaluación sólo cuando haya un resultado concreto.
- Transformación – `s` es el tipo del ambiente de referencia.

```
StateT :: (s -> m (a, s)) -> StateT s m a
```

- Aprovechando las firmas

```
type Eval4 a = ReaderT Env
                 (ErrorT ExpError
                   (StateT EvalState Identity)) a
```

UNIVERSIDAD SIMÓN BOLÍVAR

Calculando estadísticas

Plomería "estado mutable"

- El caso de las variables requiere manipular el estado y consultar el ambiente de referencia

```
eval4 :: Exp -> Eval4 Int
eval4 (Var n)    =
  do s <- get
     env <- ask
     case DM.lookup n env of
       Nothing -> throwError $ VariableNoExiste n
       Just v  -> do put $ s { vars = vars s + 1}
                     checkForThirteen v
```

UNIVERSIDAD SIMÓN BOLÍVAR

Calculando estadísticas

Plomería "estado mutable"

- Para los operadores binarios, sólo hay que modificar el estado.

```
eval4 :: Exp -> Eval4 Int
eval4 (Add l r) = do
  vl <- eval4 l
  vr <- eval4 r
  s  <- get
  put $ s { adds = adds s + 1 }
  return $ vl + vr
```

Ambiente de Referencia, Errores y Estado

¿Cómo evaluarlo?

- `ReaderT` envuelve a `ErrorT` que envuelve a `StateT` sobre `Identity`.

```
evalM4 :: Env -> EvalState -> Eval4 a
       -> (Either ExpError a,EvalState)
evalM4 env init =
  runIdentity . (flip runStateT init) . runErrorT .
                  (flip runReaderT) env
```

- La función de evaluación:
  - Desenvuelve de afuera hacia adentro.
  - `runReaderT` recibe el cómputo primero y luego el ambiente.
  - Produce el valor de cómputo o la excepción en `Either`.
  - m `runStateT` recibe el cómputo primero y luego el estado inicial.

UNIVERSIDAD SIMÓN BOLÍVAR

Ambiente de Referencia, Errores y Estado

¿Cómo evaluarlo?

- Definiremos el estado inicial del cómputo

```
initialState = EvalState {
                           adds = 0,
                           subs = 0,
                           muls = 0,
                           divs = 0,
                           vars = 0,
                           tabs = 0
                         }
```

- Así podemos evaluar

```
ghci> evalM4 env initialState (eval4 ex2)
(Right 42,EvalState {adds = 0, ... vars = 1, ...})
ghci> evalM4 (DM.fromList []) initialState (eval4 ex2
(Left (VariableNoExiste "foo"),EvalState {...})
```

UNIVERSIDAD SIMÓN BOLÍVAR

Registrando valores intermedios

Plomería "bitácora"

- Envolveremos el cómputo anterior con una bitácora para registrar los resultados intermedios en cada paso de evaluación.
- Transformación – `w` es el tipo de la bitácora

```
WriterT :: m (a, w) -> WriterT w m a
```

- Aprovecharemos las firmas tanto para el transformador, como para el tipo de la bitácora

```
type ExpLog = DS.Seq String

type Eval5 a = ReaderT Env
                 (ErrorT ExpError
                   (WriterT ExpLog
                     (StateT EvalState Identity))) a
```

Registrando valores intermedios

Plomería "bitácora"

- El caso de las constantes es ligeramente más complicado

```
eval5 :: Exp -> Eval5 Int
eval5 t@(Const i) = do
  tell $ logExp t i
  checkForThirteen i
```

- El caso de las variables requiere manipular el estado, consultar el ambiente de referencia y registrar

```
eval5 t@(Var n)    =
  do s <- get
     env <- ask
     case DM.lookup n env of
       Nothing -> throwError $ VariableNoExiste n
       Just v  -> do put $ s { vars = vars s + 1}
                     tell $ logExp t v
                     checkForThirteen v
```

Calculando estadísticas

Plomería "estado mutable"

- Para los operadores binarios

```
eval5 t@(Add l r) = do
  vl <- eval5 l
  vr <- eval5 r
  s  <- get
  let v = vl + vr
  put $ s { adds = adds s + 1 }
  tell $ logExp t v
  checkForThirteen v
```

Ambiente de Referencia, Errores, Estado y Bitácora

¿Cómo evaluarlo?

- `ReaderT` envuelve a `ErrorT` que envuelve a `WriterT` que envuelve a `StateT` sobre `Identity`.

```
evalM5 :: Env -> EvalState -> Eval5 a
       -> ((Either ExpError a,ExpLog),EvalState)
evalM5 env init =
  runIdentity . (flip runStateT init) .
    runWriterT . runErrorT .
      (flip runReaderT) env
```

- La función de evaluación:
  - Desenvuelve de afuera hacia adentro.
  - `runReaderT` recibe el cómputo primero y luego el ambiente.
  - Produce el valor de cómputo o la excepción en `Either`.
  - m `runStateT` recibe el cómputo primero y luego el estado inicial.

UNIVERSIDAD SIMÓN BOLÍVAR

Ambiente de Referencia, Errores, Estado y Bitácora

¿Cómo evaluarlo?

- Así podemos evaluar

```
ghci> evalM5 env initialState (eval5 ex2)
((Right 42,
  fromList ["Exp: Var \"foo\" -> Val: 42"]),
  EvalState {adds = 0, ..., vars = 1, tabs = 0})
ghci> evalM5 (DM.fromList [])
             initialState (eval5 ex2)
((Left (VariableNoExiste "foo"),
  fromList []),
  EvalState {adds = 0, ..., tabs = 0})
```

¿Y si queremos hacer I/O?

- Si necesitamos interactuar con el mundo exterior, basta reemplazar `Identity` por `IO`.
- En aquellos puntos en los que se necesita I/O usamos `liftIO` para subir la operación al monad combinado.
- En nuestro ejemplo, queremos imprimir la expresión con indentación

```
+
  /
    42
    2
  *
    3
    7
```

y al mismo tiempo calcular todo lo demás.

UNIVERSIDAD SIMÓN BOLÍVAR

Reemplazando `Identity` por `IO`

- Cambiamos la firma

```
type Eval6 a = ReaderT Env
                 (ErrorT ExpError
                   (WriterT ExpLog
                     (StateT EvalState IO))) a
```

- Y el evaluador ya no necesita `Identity` –
  pero debe emplearse en `IO`.

```
evalM6 :: Env -> EvalState -> Eval6 a
       -> IO ((Either ExpError a,ExpLog),EvalState)
evalM6 env init =
  (flip runStateT init) . runWriterT .
            runErrorT .  (flip runReaderT) env
```

UNIVERSIDAD SIMÓN BOLÍVAR

Indentando las expresiones

Para eso sirve el `tabs` del estado mutable.

- Se indenta con espacios

```
indent n s = DL.replicate n ' ' ++ s
```

- Constantes o variables se muestran en la indentación actual.

```
eval6 :: Exp -> Eval6 Int
eval6 t@(Const i) = do
  s <- get
  tell $ logExp t i
  liftIO $ putStrLn $ indent (tabs s) (show i)
  checkForThirteen i
```

UNIVERSIDAD SIMÓN BOLÍVAR

Indentando las expresiones

Para eso sirve el `tabs` del estado mutable.

- Operadores binarios se muestran en la indentación actual, pero las subexpresiones se indentan dos espacios.

```
eval6 t@(Add l r) = do
  p <- get
  liftIO $ putStrLn $ indent (tabs p) "+"
  put $ p { tabs = tabs p + 2 }
  vl <- eval6 l
  vr <- eval6 r
  s  <- get
  let v = vl + vr
  put $ s { adds = adds s + 1, tabs = tabs s - 2 }
  tell $ logExp t v
  checkForThirteen v
```

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- All about monads
- transformers vs. mtl
- Documentación de `Control.Monad.State`
- Documentación de `Control.Monad.Reader`
- Documentación de `Control.Monad.Writer`

UNIVERSIDAD SIMÓN BOLÍVAR