Introdução às Mônadas

Programação Funcional

Prof. Rodrigo Ribeiro

Setup inicial

```haskell
module Main where

import Control.Monad
import Control.Monad.Fail
import Data.List (sortBy)
import Data.Function (on)
import Numeric.Natural

main :: IO ()
main = return ()
```

Definindo expressões

```haskell
data Exp
  = Const Int
  | Exp :+: Exp
  | Exp :*: Exp
  deriving (Eq, Ord, Show)
```

Interpretador

```
eval :: Exp -> Int
eval (Const n)
   = n
eval (e :+: e')
   = eval e + eval e'
eval (e :*: e')
   = eval e * eval e'
```

Interpretador

- Tanto o tipo de expressões, quanto seu interpretador são definidos sem dificuldade.
- Problema: incluir uma operação de divisão.

Definindo expressões - Versão 1.0

```haskell
data Exp1
  = Const1 Int
  | Exp1 :++: Exp1
  | Exp1 :**: Exp1
  | Exp1 :/: Exp1
  deriving (Eq, Ord, Show)
```

Problema da divisão

- Possibilidade de divisão por zero.
- Temos que propagar esse erro por todo o código do intepretador.
- Representaremos erros usando o tipo Maybe

Interpretador — 1

```
eval1 :: Exp1 -> Maybe Int
eval1 (Const1 n)
   = Just n
eval1 (e :++: e')
   = case eval1 e of
       Just n ->
         case eval1 e' of
           Just m -> Just (n + m)
           Nothing -> Nothing
       Nothing -> Nothing
```

Interpretador — 2

```
eval1 (e :**: e')
   = case eval1 e of
       Just n ->
         case eval1 e' of
           Just m -> Just (n * m)
           Nothing -> Nothing
       Nothing -> Nothing
```

Interpretador — 3

```
eval1 (e :/: e')
   = case eval1 e of
       Just n ->
         case eval1 e' of
           Just m -> if m == 0 then Nothing
                               else Just (n `div` m)
           Nothing -> Nothing
       Nothing -> Nothing
```

Interpretador

- Excesso de repetição de casamento de padrão sobre o tipo Maybe.
- Solução: Criar uma função para abstrair esse casamento.

Interpretador

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
(Just v) >>? f = f v
Nothing  >>? _ = Nothing
```

Interpretador — 1

- Usando a função anterior, temos a seguinte nova versão do interpretador.

```
eval2 :: Exp1 -> Maybe Int
eval2 (Const1 n) = Just n
eval2 (e :++: e') =
   eval2 e >>? (\ n ->
       eval2 e' >>? \m -> Just (n + m))
```

Interpretador — 2

```
eval2 (e :**: e') =
    eval2 e >>? (\ n ->
        eval2 e' >>? \m -> Just (n * m))
eval2 (e :/: e') =
    eval2 e >>? (\ n ->
        eval2 e' >>? \m -> if m == 0
                           then Nothing
                           else Just (n `div` m))
```

Interpretador

- Note que a função »? impõe uma ordem de execução sobre as ações a serem feitas por uma função.
- Isso não é verdade, em geral, em código Haskell.
- A função »? é uma das que define, em Haskell, o conceito de mônada

Monadas

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

Maybe, the Monad

```haskell
instance Monad Maybe where
    return = Just
    Nothing >>= _ = Nothing
    (Just x) >>= f = f x
```

`do` Notation

- `do` Notation pode ser usada sobre qualquer tipo que implementa a classe Monad.
- Exemplos: IO, Maybe, [a], etc. . .

Maybe Monad

- Usada para computações parciais: situações de erro.
- Uso de monadas permite que tais “erros” fiquem confinados em funções que envolvem valores deste tipo (Maybe).

Interpretador, Monad version — 1

```
eval3 :: Monad m => Exp1 -> m Int
eval3 (Const1 n) = return n
eval3 (e :++: e')
  = eval3 e >>= \ n ->
    eval3 e' >>= \ m ->
    return (n + m)
```

Interpretador, Monad version — 2

```
eval3 (e :**: e')
  = eval3 e >>= \ n ->
    eval3 e' >>= \ m ->
    return (n * m)
eval3 (e :/: e')
  = eval3 e >>= \ n ->
    eval3 e' >>= \ m ->
    if m == 0 then fail "Division by zero"
      else return (n `div` m)
```

Interpretador, do notation — 1

```
eval4 :: Monad m => Exp1 -> m Int
eval4 (Const1 n) = return n
eval4 (e :++: e')
  = do
       n <- eval4 e
       m <- eval4 e'
       return (n + m)
```

Interpretador, do notation — 2

```
eval4 (e :**: e')
  = do
       n <- eval4 e
       m <- eval4 e'
       return (n * m)
eval4 (e :/: e')
  = do
       n <- eval4 e
       m <- eval4 e'
       if m == 0 then fail "Division by zero"
         else return (n `div` m)
```

List Monad

- Permite a definição de computações que retornam uma lista de resultados (não-determinismo).

List Monad

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

En guard!

- Útil para parar a busca exaustiva.

```
guard :: Monad m => Bool -> m ()
guard True  = return ()
guard False = fail "failure"
```

Exemplo

- Triplas pitagóricas

```
triples :: Int -> [(Int,Int,Int)]
triples n
  = do
       x <- [1..n]
       y <- [1..n]
       z <- [1..n]
       guard (x^2 == y^2 + z^2)
       return (x,y,z)
```

Representando estado

- Problema: dada uma árvore binária, associar a cada nó um inteiro único.

```
data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)
  deriving (Eq, Ord, Show)
```

Exemplo

- Dada a seguinte árvore

```
tree :: Tree Char
tree =  Node (Node (Leaf 'a') (Leaf 'b'))
             (Leaf 'c')
```

Exemplo

- Produzir a árvore:

```
t2 :: Tree (Char, Int)
t2 = Node (Node (Leaf ('a',0)) (Leaf ('b',1)))
          (Leaf ('c',2))
```

Solução?

```
label :: Tree a -> Tree (a,Int)
label = fst . flip labelAcc 0
   where
     labelAcc (Leaf x) n
        = (Leaf (x,n) , n + 1)
     labelAcc (Node tl tr) n
        = (Node tl' tr' , n2)
          where
            (tl', n1) = labelAcc tl n
            (tr', n2) = labelAcc tr n1
```

Função labelAcc

- A função labelAcc possui tipo

```
labelAcc :: Tree a -> Int -> (Tree a, Int)
```

- Intuitivamente, essa função recebe um "estado" e retorna um valor juntamente com o estado possivelmente atualizado.

Mônada de estado

```
newtype State s a
   = State { runState :: s -> (a, s) }
```

- variável `s`: estado armazenado.
- variável `a`: resultado da computação.

Interface

- Modificar o valor do estado:

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

Interface

- Obter o valor atual do estado.

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

Definição da mônada

```haskell
instance Functor (State s) where
   fmap f (State g)
     = State (\ s ->
          let (v, s') = g s
          in (f v, s'))
```

Definição da mônada

```
instance Applicative (State s) where
   pure v = State (\ s -> (v, s))
   (State f) <*> (State g)
     = State (\s -> let (h, s1) = f s
                        (v, s2) = g s1
                    in (h v, s2))
```

Definição da mônada

```
instance Monad (State s) where
   return = pure
   (State m) >>= f
      = State (\ s -> let (v, s') = m s
                      in runState (f v) s')
```

Definição da mônada

```
instance MonadFail (State s) where
    fail s = error s
```

Reimplementando label

```haskell
fresh :: State Int Int
fresh
    = do
        n <- get
        put (n + 1)
        return n
```

Reimplementando label

```haskell
lbl :: Tree a -> Tree (a, Int)
lbl t = fst (runState (mk t) 0)
  where
    mk (Leaf v)
      = do
          n <- fresh
          return (Leaf (v, n))
    mk (Node tl tr)
      = do
          tl' <- mk tl
          tr' <- mk tr
          return (Node tl' tr')
```

Exemplo

- Interpretador de máquina de pilha.

```
type Var = String

data Instr
  = Push Int | Set Var | Get Var
  | Add | Jump Int | JumpZero Int
  deriving (Eq, Ord, Show)
```

Exemplo

- Estado da máquina de pilha

```
type Stack = [Int]
type PC = Int
type Mem = [(Var,Int)]
type Conf = (PC, Stack, Mem)
type VM a = State Conf a
```

Exemplo

- modificando o PC

```
addPC :: Int -> VM ()
addPC n
   = do
       (pc,st,m) <- get
       put (pc + n, st, m)
```

Exemplo

- Empilhando valores na pilha

```
push :: Int -> VM ()
push n
   = do
       (pc,st,m) <- get
       put (pc, n : st, m)
```

Exemplo

- Alterando a memória.

```
set :: Var -> VM ()
set v
   = do
      (pc,n:st,m) <- get
      put (pc, st, (v,n) : m)
```

Exemplo

- Consultando a memória

```
look :: Var -> VM ()
look v
   = do
       (pc,st,m) <- get
       let st' = maybe 0 id (lookup v m) : st
       put (pc, st', m)
```

Exemplo

- Somando elementos da pilha

```
add :: VM ()
add = do
        (pc,n:p:st,m) <- get
        put (pc, (n + p) : st, m)
```

Exemplo

```
instr :: Instr -> VM ()
instr (Push n)
   = do
        push n
        addPC 1
instr (Set v)
   = do
        set v
        addPC 1
```

Exemplo

```
instr (Get v)
    = do
        look v
        addPC 1
instr Add
    = do
        add
        addPC 1
```

Exemplo

```
instr (Jump n)
    = addPC n
instr (JumpZero n)
    = do
        (pc,m:st,p) <- get
        let pc' = if m == 0 then pc + n else pc
        put (pc',st,p)
```

Exemplo

– Executando um programa

```
execM :: [Instr] -> VM ()
execM = mapM_ instr
```

Exemplo

- Interpretador

```
exec :: [Instr] -> Conf
exec is
  = snd (runState (execM is) initConf)
    where
      initConf = (0,[],[])
```

Exercício

- Apresente as instâncias de Functor, Applicative e Monad para o seguinte tipo de dados.

```
data Rose a
  = RoseNode a [Rose a]
  | RoseLeaf
```