

Transformatory monad

Czyli jak łatwo pisać modularne programy

Łukasz Czapliński

October 17, 2013

Szybkie przypomnienie

- Monady to „design pattern” programowania funkcyjnego.
- Pozwalają operować na jednym z elementów pudełka (funktora) bez znajomości jego struktury ($>>=$).
- Pozwalają na modelowanie efektów w czystym języku funkcyjnym (np. stanu).

- `return >=> f = f`
- `f >=> return = f`
- `a >=> (b >=> c) = (a >=> b) >=> c`

- `return >=> f = f`
- `f >=> return = f`
- `a >=> (b >=> c) = (a >=> b) >=> c`
- Monoid! Operacja łączna z elementem neutralnym.
- `(g :: Functor f => a -> f b` nazywamy endofunktorem)

Napisać interpreter prostego języka o następujących cechach:

- posiada zmienne,
- występują funkcje,
- obsługuje operacje logiczne.

Dlaczego interpreter? Bo da się przenieść na dowolny inny program.

Jak interpretować?

```
data Term = And Term Term | Or Term Term
          | Var Variable | Not Term
          | Let Variable Term Term | Const Value
          | Appl Function Term deriving (Read, Show, Ord, Eq)

interp :: InterpMonad m => Term -> m Value
```

Czym jest InterpMonad?

```
class (MonadState m, MonadEnv m) => InterpMonad m where  
  start :: (Show a, InterpMonad m) => m a -> String
```

Monada stanowa

```
class Monad m => MonadState m where
  modState :: (State -> State) -> m State
  putVar  :: Variable -> Term -> m ()
  getVar  :: Variable -> m (Maybe Term)
```


Monada środowiskowa

```
class Monad m => MonadEnv m where
  parseEnv :: String -> m (Maybe Env)
  inEnv    :: Env -> m a -> m a
  lookupEnv :: Function -> m (Maybe Func)
```

Implementacja - podejście 1: wprost.

```
newtype StateC a = StateC { runStateC :: State -> (State,
    a) }

instance Monad StateC where
    return x = StateC $ \s -> (s,x)
    m >>= f = StateC $ (\s -> let (s',a) = runStateC m $ s
        in runStateC (f a) $ s')
```



```
instance MonadState StateC where
    modState f = StateC $ \s -> (f s, s)
    putVar v t = modState (putState v t) >> return ()
    getVar v = StateC $ \s -> (s, lookup v s)
```

Podobnie dla MonadEnv.

Dla ciekawskich: MonadEnv

```
newtype EnvC a = EnvC { runEnvC :: Env -> a }

instance Monad EnvC where
    return x = EnvC $ \_ -> x
    m >=> f = EnvC $ \r -> runEnvC (f (runEnvC m $ r)) $
        r

instance MonadEnv EnvC where
    parseEnv = return.strToEnv
    inEnv e f = return $ runEnvC f e
    lookupEnv f = EnvC $ \r -> lookup f r
```

Dygresja: podział pracy na monady

- Modularyzacja
- Trudniej o błędy
- Prostsze struktury danych
- Możliwość kontroli zachowania

Składanie w całość

```
newtype OurMonad a = OurMonad { run0:: EnvC (StateC a) }  
-- EnvC (Env -> StateC (State -> (State, a)))  
instance Monad OurMonad where  
    return x = OurMonad $ return $ return x  
    m >>= f = OurMonad $ EnvC ( \r -> runEnvC (run0 m) r  
        >>= (\a -> runEnvC (run0 (f a)) r))
```

A może by tak dodać obsługę błędów?

```
class Monad m => MonadError m where
  err :: Error -> m a
newtype OurMonad a = OurMonad { run0 :: EnvC (StateC (
  ErrorC a)) }
-- EnvC (Env -> StateC (State -> (State, ErrorC (Either
  Err a))))
```

Nowy bind - ile trzeba zmienić? Za dużo.

Co można zmienić?

Co chcielibyśmy otrzymać?

→ Możliwość składania monad. Możemy to uzyskać np: używając w definicji bind bind monady bazowej.

Co jeśli w wyniku nie będzie monady?

→ Stwórzmy sztuczną: Monad Id

```
newtype Id a = Id { runId :: a }  
instance Monad Id where  
    return = Id  
    m >>= f = f (runId m)
```

Teraz możemy być pewni, że nasza monada jako typ wynikowy będzie miała pewną monadę. Wobec tego możemy w definicji `bind` użyć `bind` monady wynikowej. Otrzymujemy typ:

```
Monad m => Monad (t m)
```

gdzie `t` jest tworzonym przez nas typem. Stąd nazwa transformatory monad.

Transformatory monad - formalnie

- mają typ $(* \rightarrow *) \rightarrow * \rightarrow *$
- jeśli argumentem jest monada, to wynikiem też.

Przykład 1: StateT

```
newtype StateT m a = StateT { runStateT :: State -> m (
    State,a) }

instance Monad m => Monad (StateT m) where
    return x = StateT $ \s -> return (s,x)
    m >>= f = StateT $ \s -> do
        (s',a) <- runStateT m s
        runStateT (f a) s'

instance Monad m => MonadState (StateT m) where
    modState f = StateT $ \s -> return (f s, s)
    putVar v t = modState (putState v t) >> return ()
    getVar v = StateT $ \s -> return (s, lookup v s)
```

Przykład 2: Env

```
newtype EnvT m a = EnvT { runEnvT :: Env -> m a }
instance Monad m => Monad (EnvT m) where
    return x = EnvT $ \_ -> return x
    m >>= f = EnvT $ \e -> do
        a <- runEnvT m e
        runEnvT (f a) e

instance Monad m => MonadEnv (EnvT m) where
    parseEnv = return.strToEnv
    inEnv e f = EnvT $ \_ -> runEnvT f e
    lookupEnv f = EnvT $ \e -> return $ lookup f e
```

Niespodziewany problem

Jak teraz podnieść obliczenia w monadzie bazowej?

Przedtem wystarczyło zrobić `return`.

Teraz: `return :: a -> (t m) a`, a potrzeba `lift :: m a -> (t m) a`

```
class MonadT t where
  lift :: Monad m => m a -> t m a
```

```
instance MonadT StateT where
  lift m = StateT $ \s -> do
    a <- m
    return (s,a)
```

```
instance MonadT EnvT where
  lift m = EnvT $ \_ -> m
```

To po prostu `return` na resorach.

Podejście 2: z transformatorami

```
newtype OurMonad a = OurMonad { run0 :: EnvT (StateT (
    Id)) a }
liftFromEnv :: EnvT (StateT (Id)) a -> OurMonad a
liftFromEnv = OurMonad
liftFromState :: StateT (Id) a -> OurMonad a
liftFromState = liftFromEnv.lift

instance Monad OurMonad where
    return = OurMonad . return
    m >>= f = OurMonad $ do
        a <- run0 m
        run0 $ f a
```

Dodajmy error

```
newtype OurMonad a = OurMonad { run0 :: ErrorT (EnvT (
    StateT (Id))) a }

liftFromErr :: ErrorT (EnvT (StateT (Id))) a -> OurMonad
    a

liftFromErr = OurMonad

liftFromEnv :: EnvT (StateT (Id)) a -> OurMonad a

liftFromEnv = liftFromErr.lift

liftFromState :: StateT (Id) a -> OurMonad a

liftFromState = liftFromEnv.lift

instance Monad OurMonad where
    return = OurMonad . return
    m >>= f = OurMonad $ do
        a <- run0 m
        run0 $ f a
```

Właściwie bez zmian!

Dokończmy implementację

```
instance MonadError OurMonad where
    err str = liftFromErr $ err str
```

```
instance MonadState OurMonad where
    modState f = liftFromState $ modState f
    putVar v t = liftFromState $ putVar v t
    getVar v = liftFromState $ getVar v
```

```
instance MonadEnv OurMonad where
    inEnv e f = liftFromErr $ ErrorT $ inEnv e $ runErrorT
        $ run0 f
    parseEnv s = liftFromEnv $ parseEnv s
    lookupEnv f = liftFromEnv $ lookupEnv f
```

Pozostaje formalność

```
instance InterpMonad OurMonad where
  start o = case runId $ runStateT (runEnvT (runErrorT $
    run0 o) basicEnv) basicState of
    (s, Left err) -> "Error: " ++ err ++ " in state:[ " ++
      show s ++ " ]"
    (_, Right a) -> show a
```

Teraz mamy gotowy interpreter.

- kolejność składania \rightarrow $\text{StateT } (\text{ErrorT } \text{Id})$ a czy $\text{ErrorT } (\text{StateT } \text{Id})$ a?
- monada listowa \rightarrow odpowiedni transformator
- lift \rightarrow co jeśli chcemy podnieść funkcję?

Kolejność składania monad

```
StateT (ErrorT Id) a  
State -> Either Err (State, a)
```

```
ErrorT (StateT Id) a  
State -> (State, Either Err a)
```

Otrzymujemy inne właściwości monady wynikowej.

Monada listowa

```
class Monad m => MonadList m where
    merge :: m a -> m a -> m a

instance Monad m => MonadList (t m) where
    merge :: (t m) a -> (t m) a -> (t m) a
```

LiftT z Control.Monad.Trans.List

```
newtype ListT = ListT { runListT :: m [a] }
```

```
test1 :: ListT IO Int
```

```
test1 = do
```

```
  r <- liftIO (newIORef 0)
```

```
  (next r `mplus` next r >> next r `mplus` next r) >>  
    next r `mplus` next r
```

```
test2 :: ListT IO Int
```

```
test2 = do
```

```
  r <- liftIO (newIORef 0)
```

```
  next r `mplus` next r >> (next r `mplus` next r >> next  
    r `mplus` next r)
```

```
next :: IORef Int -> ListT IO Int
```

```
next r = liftIO $ do  x <- readIORef r  
                      writeIORef r (x+1)  
                      return x
```

```
main = do
  arg <- getArgs
  case head arg of
    "1" -> runListT test1
    _    -> runListT test2
```

```
$ runghc example.hs 1
[6,7,8,9,10,11,12,13]
$ runghc example.hs 2
[4,5,6,7,10,11,12,13]
```

ListT done right?

```
-- The monadic list type
data MList' m a = MNil | a `MCons` MList m a
type MList m a  = m (MList' m a)

-- This can be directly used as a monad transformer
newtype ListT m a = ListT { runListT :: MList m a }

-- A "lazy" run function, which only calculates the
  first solution.
runListT' :: Functor m => ListT m a -> m (Maybe (a,
  ListT m a))
runListT' (ListT m) = fmap g m where
  g MNil = Nothing
  g (x `MCons` xs) = Just (x, ListT xs)
```

...i jego problem

```
test = runListT $ do
  x <- liftList [1..3]
  liftIO $ print x
  y <- liftList [6..8]
  liftIO $ print (x,y)
```

```
Using Control.Monad.List:
```

```
Main> test
```

```
1
```

```
(1,6)
```

```
(1,7)
```

```
(1,8)
```

```
2
```

```
(2,6)
```

```
(2,7)
```

```
(2,8)
```

```
3
```

```
(3,6)
```

```
(3,7)
```

```
(3,8)
```

```
Using "ListT done right":
```

```
Main> test
```

```
1
```

```
(1,6)
```


Wobec tego najprościej nie używać transformatora, użyć [] zamiast Id jako monady bazowej.

```
newtype OurListMonad a = OLM { runOLM :: ErrorT (StateT  
    []) a }  
-- State -> [(State, Either Err a)]
```

Wówczas merge to dodanie odpowiednich list. Niestety wówczas rezygnujemy ze skutków ubocznych: nie możemy użyć IO jako monady bazowej. Co oddaje istotę powyższego problemu.

Trochę formalności: lift

Jak powinien działać?

Prawa

- `lift . return = return`
- `lift (m `bind` k) = (lift m) `bind` (lift . k)`

$$\mathcal{L}_\tau \quad :: \quad \tau \rightarrow [\tau]_t$$

$$\mathcal{L}_A = id \quad (1)$$

$$\mathcal{L}_a = id \quad (2)$$

$$\mathcal{L}_{\tau_1 \rightarrow \tau_2} = \backslash f \rightarrow f' \text{ such that } f' \cdot \mathcal{L}_{\tau_1} = \mathcal{L}_{\tau_2} \cdot f \quad (3)$$

$$\mathcal{L}_{(\tau_1, \tau_2)} = \backslash (a, b) \rightarrow (\mathcal{L}_{\tau_1} a, \mathcal{L}_{\tau_2} b) \quad (4)$$

$$\mathcal{L}_{m\ \tau} = lift \cdot (map\ \mathcal{L}_\tau) \quad (5)$$

$$\begin{array}{ccc} [\tau_1]_t & \xrightarrow{f'} & [\tau_2]_t \\ \mathcal{L}_{\tau_1} \uparrow & & \uparrow \mathcal{L}_{\tau_2} \\ \tau_1 & \xrightarrow{f} & \tau_2 \end{array}$$

From Monad Transformers and Modular Interpreters (Sheng Liang Paul Hudak Mark Jones).

Pytania i uwagi?

- Jak połączyć IO z monadami? → piszemy grę
- Jak połączyć IO z różnymi wynikami? → piszemy Prologa

Zadanie

```
data Term = And Term Term | Or Term Term
          | Var Variable | Not Term
          | Let Variable Term Term | Const Value
          | Twofold Variable Term deriving (Read, Show, Ord, Eq)

interp Twofold v t -> do
  mv <- getVar v
  if mv /= Nothing
    then err $ "Variable " ++ v ++ " already taken at
               this point."
    else return ()
  let v1 = branch v True t
  let v2 = branch v False t
  merge v1 v2
```

Zadanie - c.d.

```
class (MonadState m, MonadWriter m, MonadList m,  
      MonadError m) => InterpMonad m where  
  start :: (Show a, InterpMonad m) => m a -> String  
  
class Monad m => MonadWriter m where  
  tell :: String -> m ()
```

Przykład

```
--term4:  
Twofold "x"  
  (Twofold "y"  
    (Or  
      (Or  
        (Let "x" (Const False) (Var"x"))  
        (Var "y"))  
      (Var"z"))))
```

Przykład

```
Testing x as True..
Testing y as True..
Warning: overwriting var
      x .
Done with y.
Done with x.
---- Answer: True
Testing x as True..
Testing y as False..
Warning: overwriting var
      x .
Something went wrong..
  No such variable as z at
    this point.
{State: [("x",Const True)
        ,("y",Const False)]}
```

```
Testing x as False..
Testing y as True..
Warning: overwriting var
      x .
Done with y.
Done with x.
---- Answer: True
Testing x as False..
Testing y as False..
Warning: overwriting var
      x .
Something went wrong..
  No such variable as z at
    this point.
{State: [("x",Const False)
        ,("y",Const False)]}
```


Dziękuję za uwagę