

Řešení chyb v čistém a monadickém kódu, monadické transformátory

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2015

Řešení chyb – opakování

Pomocí datových typů `Maybe a` a `Either e a`

- + jednoduché, funguje v čistém kódu
- + lze používat `Functor/Applicative/Monad`
- u `Maybe` nemůžeme specifikovat jaká chyba nastala
- špatně se kombinuje s jinými monádami

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

v modulu `Text.Read`:

```
readEither :: Read a => String -> Either String a
```

```
readEither "necislo" :: Either String Int
```

```
  ~>* Left "Prelude.read: no parse"
```

```
readEither "1" :: Either String Integer ~>* Right 1
```

Využití monády Maybe

```
import Text.Read ( readMaybe )
import System.Environment ( getArgs )
import Control.Monad ( mapM )
import Control.Applicative ( (<$>) )

calculateSum :: [String] -> Maybe Double
calculateSum args = sum <$> mapM readMaybe args

calculateAverage :: [String] -> Maybe Double
calculateAverage [] = Nothing
calculateAverage args = do          -- do in Maybe Monad
    sum <- calculateSum args
    let len = fromIntegral $ length args
    return $ sum / len

main = getArgs >>= print . calculateAverage
```

Kombinace IO/Maybe: problém

```
import Text.Read ( readMaybe )
import Control.Monad ( when )
import Control.Applicative ( (<$>) )

doAverage :: Double -> Double -> IO ()
doAverage sum cnt = do                -- do in IO Monad
    when (cnt > 0) . putStrLn $
        "running average: " ++ show (sum / cnt)
    num <- readMaybe <$> getLine
    case num of -- num :: Maybe Double
        Nothing -> return ()
        Just x   -> doAverage (sum + x) (cnt + 1)

main = doAverage 0 0
```

Kombinace IO/Maybe: řešení

IO (Maybe a) by se mohlo chovat jako instance Functor/Applicative/Monad:

- spouští IO akce, které vrací Maybe hodnoty
- pokud narazí na Nothing další akce ignoruje

Kombinace IO/Maybe: řešení

IO (Maybe a) by se mohlo chovat jako instance Functor/Applicative/Monad:

- spouští IO akce, které vrátí Maybe hodnoty
- pokud narazí na Nothing další akce ignoruje

Je třeba zabalit do `newtype`

```
import Control.Applicative
newtype IOMaybe a =
    IOMaybe { runIOMaybe :: IO (Maybe a) }

instance Functor IOMaybe where
    fmap :: (a -> b) -> IOMaybe a -> IOMaybe b
    fmap f a = IOMaybe $ do -- do in IO monad
        ma <- runIOMaybe a -- ma :: Maybe a
        return $ fmap f ma -- fmap in Maybe monad
```

Kombinace IO/Maybe: řešení

```
newtype IOMaybe a =  
    IOMaybe { runIOMaybe :: IO (Maybe a) }  
  
instance Applicative IOMaybe where  
    pure :: a -> IOMaybe a  
    pure = IOMaybe . return . Just  
    (<*>) :: IOMaybe (a -> b) -> IOMaybe a  
        -> IOMaybe b  
    f <*> x = IOMaybe $ do  
        mf <- runIOMaybe f -- mf :: Maybe (a -> b)  
        mx <- runIOMaybe x -- mx :: Maybe a  
        return $ mf <*> mx -- (<*>) in Maybe
```

Kombinace IO/Maybe: řešení

```
newtype IOMaybe a =
  IOMaybe { runIOMaybe :: IO (Maybe a) }

instance Monad IOMaybe where
  (>>=) :: IOMaybe a -> (a -> IOMaybe b)
    -> IOMaybe b
  x >>= f = IOMaybe $ do -- do in IO monad
    mx <- runIOMaybe x -- mx :: Maybe a
    case mx of
      Nothing -> return Nothing
      Just px -> runIOMaybe (f px) -- px :: a

  return = pure
  fail _ = IOMaybe (return Nothing)

liftIOMaybe :: IO a -> IOMaybe a
liftIOMaybe x = IOMaybe (Just <$> x)
```


IOMaybe: užití

```
import Text.Read ( readMaybe )
import Control.Monad ( when, void )
import Control.Applicative ( (<$>) )
import IOMaybe

doAverage :: Double -> Double -> IOMaybe ()
doAverage sum cnt = do          -- do in IOMaybe Monad
    when (cnt > 0) . liftIOMaybe . putStrLn $
        "running average: " ++ show (sum / cnt)
    x <- IOMaybe (readMaybe <$> getLine)
    doAverage (sum + x) (cnt + 1)

main = void . runIOMaybe $ doAverage 0 0
```

Zobecňujeme: MaybeT

Můžeme zobecnit pro libovolnou monádu namísto IO: `MaybeT m a`

Zobecňujeme: MaybeT

Můžeme zobecnit pro libovolnou monádu namísto IO: MaybeT m a

```
import Control.Monad
import Control.Applicative

newtype MaybeT m a =
    MaybeT { runMaybeT :: m (Maybe a) }

instance (Functor m) => Functor (MaybeT m) where
    fmap :: (a -> b) -> MaybeT m a -> MaybeT m b
    fmap f = MaybeT . fmap (fmap f) . runMaybeT
        -- 1st fmap from Functor m
        -- 2nd fmap from Functor Maybe
```

Zobecňujeme: MaybeT

```
newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }

instance Applicative m => Applicative (MaybeT m)
  where
    pure :: a -> MaybeT m a
    pure = MaybeT . pure . Just
    (<*>) :: MaybeT m (a -> b) -> MaybeT m a
        -> MaybeT m b
    f <*> x = MaybeT $
        fmap (<*>) (runMaybeT f) <*> runMaybeT x

--      2nd (outer) <*> in Applicative m:
--      (<*>) :: m (Maybe a -> Maybe b)
--            -> m (Maybe a) -> m (Maybe b)
--      fmap (<*>) :: m (Maybe (a -> b))
--                -> m (Maybe a -> Maybe b)
```

Zobecňujeme: MaybeT

```
newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }

instance Monad m => Monad (MaybeT m) where
  (>>=) :: MaybeT m a -> (a -> MaybeT m b)
    -> MaybeT m b
  x >>= f = MaybeT $ do -- do in Monad m
    mx <- runMaybeT x -- mx :: Maybe a
    case mx of
      Nothing -> return Nothing
      Just px -> runMaybeT (f px) -- px :: a

  return = MaybeT . return . Just
  fail _ = MaybeT (return Nothing)

lift :: (Monad m) => m a -> MaybeT m a
lift x = MaybeT (liftM Just x)
```

Monadické transformátory

- přidávání nových vlastností monádám
- například `MaybeT`, `ErrorT` a další
 - pro libovolnou monádu `m` jsou `MaybeT m`, `ErrorT m` monády
- `Control.Monad.Trans.*` (balík `transformers`)

Monadické transformátory

- přidávání nových vlastností monádám
- například `MaybeT`, `ErrorT` a další
 - pro libovolnou monádu `m` jsou `MaybeT m`, `ErrorT m` monády
- `Control.Monad.Trans.*` (balík `transformers`)

Třída `MonadTrans` (`Control.Monad.Trans.Class`)

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

Musí platit:

- `lift . return ≡ return`
- `lift (m >>= f) ≡ lift m >>= (lift . f)`

ErrorT

- `Control.Monad.Trans.Error`
- přidává práci s chybami do monády
- `newtype Error e m a =`
 `ErrorT { runErrorT :: m (Either e a) }`
- kde `e` by mělo být instancí `Error`
 `class Error a where`
 `noMsg :: a`
 `strMsg :: String -> a`
- instance lze vytvořit obdobně jako pro `MaybeT`

```
throwError :: (Monad m, Error e) => e -> ErrorT e m a
catchError :: (Monad m, Error e)
=> ErrorT e m a          -- computation
-> (e -> ErrorT e m a) -- handler
-> ErrorT e m a
```


Samostatná práce

Rozšíření `IOMaybe` (`IOMaybe.hs` je v `ISu`)

- `IOMaybe` (stejně jako `MaybeT/ErrorT`) nemá žádnou podporu pro zachytávání výjimek
- implementujte `liftIOWhandle :: IO a -> IOMaybe a`, která se bude chovat obdobně jako `liftIOMaybe` ale bude navíc zachytávat `IOException` a v případě zachycení výjimky ji vypíše na `stderr` a vrátí `Nothing` pozvednuté do `IOMaybe` (a tedy ukončí výpočet).
- implementujte funkci `withDefault :: a -> IOMaybe a -> IOMaybe a`, která se bude chovat tak, že volání `withDefault x act` vrátí výsledek akce `act` pokud tato je úspěšná (nevrací `Nothing`) a `x` pokud je `act` neúspěšná.
- použijte tyto funkce pro vypsání obsahu adresáře a prvního řádku pro každý soubor v něm (nebo výjimky pokud dojde k chybě při čtení souboru).