Řešení chyb, transformátory monád (`MaybeT`, `ErrorT`, `WriterT`, `StateT`)

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Motivace pro `Writer`: logování

- někdy chceme v průběhu výpočtu produkovat dodatečné informace – log, statistiky

Motivace pro `Writer`: logování

- někdy chceme v průběhu výpočtu produkovat dodatečné informace – log, statistiky
- myšlenka: produkujeme dvojice

```
foo        :: Integer -> Integer ->  Integer
fooWithLog :: Integer -> Integer -> (Integer, String)
```

Motivace pro `Writer`: logování

- někdy chceme v průběhu výpočtu produkovat dodatečné informace – log, statistiky
- myšlenka: produkujeme dvojice

```
foo        :: Integer -> Integer ->  Integer
fooWithLog :: Integer -> Integer -> (Integer, String)
```

- problém: kompozice funkcí komplikovaná
  - je třeba extrahovat výsledek
  - je třeba spojovat logy

Problém kompozice

- při výpočtu typicky chceme pracovat s výstupem a logy jen skládat
- prozatím logy zanedbáme

Problém kompozice

- při výpočtu typicky chceme pracovat s výstupem a logy jen skládat
- prozatím logy zanedbáme

- potřebujeme tedy vždy extrahovat jednu část dvojice a s ní dále pracovat

Problém kompozice

- při výpočtu typicky chceme pracovat s výstupem a logy jen skládat
- prozatím logy zanedbáme

- potřebujeme tedy vždy extrahovat jednu část dvojice a s ní dále pracovat

- to přesně odpovídá monádě: první složka dvojice je výsledek, druhá (monadický) kontext

Problém skládání logů

- logy třeba posbírat z celého programu

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String`?

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String`? `(++)`

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String`? `(++)`
- `[a]`?

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String? (++)`
- `[a]? (++)`

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String? (++)`
- `[a]? (++)`
- `IO ()?`

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String? (++)`
- `[a]? (++)`
- `IO ()? (>>)`

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String? (++)`
- `[a]? (++)`
- `IO ()? (>>)`
- `Text?`

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String? (++)`
- `[a]? (++)`
- `IO ()? (>>)`
- `Text? append`

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String? (++)`
- `[a]? (++)`
- `IO ()? (>>)`
- `Text? append`
- potřebujeme nějak popsat typy, jejichž hodnoty lze spojovat

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

Problém skládání logů

- logy třeba posbírat z celého programu
- spojení logu závisí na typu:
- `String? (++)`
- `[a]? (++)`
- `IO ()? (>>)`
- `Text? append`
- potřebujeme nějak popsat typy, jejichž hodnoty lze spojovat

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

  - pro `[a]` je `mappend = (++)`
  - pro `IO a` je `mappend = liftA2 mappend`
  - tedy pro `IO ()` je `mappend` ekvivalentní `(>>)`

Použití `Writer`

```
plus    ::  Num a           => a -> a ->                 a
plusLog :: (Num a, Show a) => a -> a -> Writer String a
plusLog x y = tell (show x ++ " + " ++ show y ++ "; ")
              >> pure (x + y)
```

například:

```
> runWriter $ foldrM plusLog 0 [1..10]
> runWriter $ foldlM plusLog 0 [1..10]
```

Zobecňujeme: `WriterT`

- `Writer` se špatně kombinuje s jinými monádami
- `Writer` můžeme zobecnit na `WriterT`

Zobecňujeme: `WriterT`

- `Writer` se špatně kombinuje s jinými monádami
- `Writer` můžeme zobecnit na `WriterT`

```
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
```

Zobecňujeme: `WriterT`

- `Writer` se špatně kombinuje s jinými monádami
- `Writer` můžeme zobecnit na `WriterT`

```
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }


instance Functor m => Functor (WriterT w m) where
  fmap :: (a -> b) -> WriterT w m a -> WriterT w m b
  -- fmap (first f) :: m (a, w) -> m (b, w)
  fmap f x = WriterT . fmap (first f) $ runWriterT x

first :: (a -> b) -> (a, c) -> (b, c)
first f (x, y) = (f x, y)
```

WriterT

```haskell
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
writer :: Applicative m => (a, w) -> WriterT w m a
writer = WriterT . pure
```

WriterT

```haskell
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
writer :: Applicative m => (a, w) -> WriterT w m a
writer = WriterT . pure

instance (Monoid w, Applicative m)
         => Applicative (WriterT w m) where
  pure x = writer (x, mempty)

  (<*>) :: WriterT w m (a -> b) -> WriterT w m a
        -> WriterT w m b
  -- runWriterT x :: m (a, w)
  -- runWriterT f :: m (a -> b, w)
  f <*> x = WriterT $
              liftA2 app (runWriterT f) (runWriterT x)
    where
      -- app :: (a -> b, w) -> (a, w) -> (b, w)
      app (f, wf) (x, wx) = (f x, wf <> wx)
```

WriterT

```haskell
instance (Monoid w, Monad m) => Monad (WriterT w m) where
    (>>=) :: WriterT w m a -> (a -> WriterT w m b)
          -> WriterT w m b
    x >>= f = WriterT $ do
        (x', wx) <- runWriterT x
        (r, rfx) <- runWriterT (f x')
        pure (r, wx <> rfx)

instance Monoid w => MonadTrans (WriterT w) where
    lift act = WriterT $ fmap (\x -> (x, mempty)) act
```

Dvojice jako writer

- standardně je v knihovně instance `Monad` pro `(,) w`, která se chová jako writer
- oproti `Writer`/`WriterT` je pořadí prvků ve dvojici přehozené

Dvojice jako writer

- standardně je v knihovně instance `Monad` pro `(,) w`, která se chová jako writer
- oproti `Writer`/`WriterT` je pořadí prvků ve dvojici přehozené

```
foo, bar :: Int -> (String, Int)
foo x = ("foo: " ++ show x ++ "; ", x + 42)
bar x = ("bar; ", -x)

> foo 0 >>= foo >>= bar
("foo: 0; foo: 42; bar; ",-84)

> foo 0 >>= foo >>= bar >>= \x -> ("msg", ()) >> pure x
("foo: 0; foo: 42; bar; msg",-84)
```

`Writer` a `WriterT` v knihovně

- balík `transformers`: `Control.Monad.Trans.Writer.{Lazy,Strict}`
- definuje `WriterT`, `writer`, `runWriterT`, `tell`...
- `type Writer w = WriterT w Identity`
- instance pro `MonadTrans`
- balík `mtl`: `Control.Monad.Writer.{Lazy,Strict}`
- re-exportuje `writer`, `runWriterT`, ... z příslušného `transformers` modulu
- typová třída `MonadWriter w m`
- uniformní přístup k věcem, které se chovají jako `WriterT`
- včetně `((,) w)`

State

- `Reader` ani `Writer` neumí zároveň zapisovat a číst
- `State` umožňuje zápis i čtení – drží si modifikovatelný stav
- ```
  newtype StateT s m a =
          StateT { runStateT :: s -> m (a, s) }
  ```
- stav `s`, výsledek `a`
  ```

State

- `Reader` ani `Writer` neumí zároveň zapisovat a číst
- `State` umožňuje zápis i čtení – drží si modifikovatelný stav
- ```
  newtype StateT s m a =
          StateT { runStateT :: s -> m (a, s) }
  ```
- stav `s`, výsledek `a`
- `type State s a = StateT s Identity a`
  ```

State

- `Reader` ani `Writer` neumí zároveň zapisovat a číst
- `State` umožňuje zápis i čtení – drží si modifikovatelný stav
- ```
  newtype StateT s m a =
          StateT { runStateT :: s -> m (a, s) }
  ```
- stav `s`, výsledek `a`
- `type State s a = StateT s Identity a`
- `runState :: State s a -> s -> (a, s)`
  ```

State

- `Reader` ani `Writer` neumí zároveň zapisovat a číst
- `State` umožňuje zápis i čtení – drží si modifikovatelný stav
- ```
  newtype StateT s m a =
          StateT { runStateT :: s -> m (a, s) }
  ```
- stav `s`, výsledek `a`
- `type State s a = StateT s Identity a`
- `runState :: State s a -> s -> (a, s)`
- `get :: Monad m => StateT s m s`
- `put :: Monad m => s -> StateT s m ()`
- `modify :: Monad m => (s -> s) -> StateT s m ()`
- `gets :: Monad m => (s -> a) -> StateT s m a`
  ```

State

- `Reader` ani `Writer` neumí zároveň zapisovat a číst
- `State` umožňuje zápis i čtení – drží si modifikovatelný stav
- ```
  newtype StateT s m a =
          StateT { runStateT :: s -> m (a, s) }
  ```
- stav `s`, výsledek `a`
- `type State s a = StateT s Identity a`
- `runState :: State s a -> s -> (a, s)`
- `get :: Monad m => StateT s m s`
- `put :: Monad m => s -> StateT s m ()`
- `modify :: Monad m => (s -> s) -> StateT s m ()`
- `gets :: Monad m => (s -> a) -> StateT s m a`

- `Control.Monad.State`/`Control.Monad.Trans.State`
  ```

MonadIO

`Control.Monad.IO.Class`

- definuje třídu `MonadIO m` monád, které umí vstup a výstup
- například transformery na bázi `IO`
- jediná metoda `liftIO :: IO a -> m a`, která umožní spustit `IO` akci v dané monádě

Řešení chyb – opakování

Způsoby řešení chyb v Haskellu?

Řešení chyb – opakování

Způsoby řešení chyb v Haskellu?

Pomocí datových typů `Maybe 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

Řešení chyb – opakování

Způsoby řešení chyb v Haskellu?

Pomocí datových typů `Maybe 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
Text.Read.readEither :: Read a
                     => String -> Either String a
```

Řešení chyb – opakování

Způsoby řešení chyb v Haskellu?

Pomocí datových typů `Maybe 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
Text.Read.readEither :: Read a
                     => String -> Either String a
```

Pomocí výjimek

- ± neprojevuje se v typu funkcí které chyby produkují nebo řeší
- – skutečně použitelné jen v `IO`

`Maybe`/`Either` v monádě

při práci s `Maybe`/`Either` můžeme s výhodou využít toho, že jsou to monády

- `mapM readMaybe`

Maybe/Either v monádě

při práci s `Maybe`/`Either` můžeme s výhodou využít toho, že jsou to monády

- `mapM readMaybe`
- `fmap (+ 1) (readMaybe x)`

Maybe/Either v monádě

při práci s `Maybe`/`Either` můžeme s výhodou využít toho, že jsou to monády

- `mapM readMaybe`
- `fmap (+ 1) (readMaybe x)`
- `liftA2 (+) (readMaybe x) (readMaybe y)`

Maybe/Either v monádě

při práci s `Maybe`/`Either` můžeme s výhodou využít toho, že jsou to monády

- `mapM readMaybe`
- `fmap (+ 1) (readMaybe x)`
- `liftA2 (+) (readMaybe x) (readMaybe y)`
- ```
  do rx <- readMaybe x
     ry <- readMaybe y
     ...
  ```
  ```

Pattern matching v do bloku

V `do` bloku je možné provádět pattern matching:

```
data Foo = Foo String | Bar Int
foo :: Foo -> ...
foo x = do
    Foo str <- x
    ...
```

Pattern matching v do bloku

V `do` bloku je možné provádět pattern matching:

```
data Foo = Foo String | Bar Int
foo :: Foo -> ...
foo x = do
    Foo str <- x
    ...
```

- selhání pattern matchingu způsobí zavolání funkce
  `fail :: Monad m => String -> m a`
- `fail` je defonována v `Monad`

Pattern matching v do bloku

V `do` bloku je možné provádět pattern matching:

```
data Foo = Foo String | Bar Int
foo :: Foo -> ...
foo x = do
    Foo str <- x
    ...
```

- selhání pattern matchingu způsobí zavolání funkce
  `fail :: Monad m => String -> m a`
- `fail` je defonována v `Monad`
- obvykle je `fail` definovaná pomocí `error`

Pattern matching v do bloku

V `do` bloku je možné provádět pattern matching:

```
data Foo = Foo String | Bar Int
foo :: Foo -> ...
foo x = do
    Foo str <- x
    ...
```

- selhání pattern matchingu způsobí zavolání funkce
  `fail :: Monad m => String -> m a`
- `fail` je defonována v `Monad`
- obvykle je `fail` definovaná pomocí `error`
- v `Maybe` je však `fail = Nothing`

Pattern matching v do bloku

V `do` bloku je možné provádět pattern matching:

```
data Foo = Foo String | Bar Int
foo :: Foo -> ...
foo x = do
    Foo str <- x
    ...
```

- selhání pattern matchingu způsobí zavolání funkce
  `fail :: Monad m => String -> m a`
- `fail` je defonována v `Monad`
- obvykle je `fail` definovaná pomocí `error`
- v `Maybe` je však `fail = Nothing` (co `Either`?)

Pattern matching v do bloku

V `do` bloku je možné provádět pattern matching:

```
data Foo = Foo String | Bar Int
foo :: Foo -> ...
foo x = do
    Foo str <- x
    ...
```

- selhání pattern matchingu způsobí zavolání funkce
  `fail :: Monad m => String -> m a`
- `fail` je defonována v `Monad`
- obvykle je `fail` definovaná pomocí `error`
- v `Maybe` je však `fail = Nothing` (co `Either`?)
- pro vhodné monády tak můžeme `do` blok použít i k řešení chyb v pattern matchingu

`Monad` vs. `MonadFail`

- protože `fail` nejde rozumně implementovat v mnoha monádách není jeho zařazení do `Monad` moc vhodné

Monad vs. MonadFail

- protože `fail` nejde rozumně implementovat v mnoha monádách není jeho zařazení do `Monad` moc vhodné
- od GHC 8.0 do 8.8 se bude postupně `fail` přesouvat do nové třídy `MonadFail`

Monad vs. MonadFail

- protože `fail` nejde rozumně implementovat v mnoha monádách není jeho zařazení do `Monad` moc vhodné
- od GHC 8.0 do 8.8 se bude postupně `fail` přesouvat do nové třídy `MonadFail`
- `Control.Monad.Fail`, rozšíření `-XMonadFailDesugaring`
- `do` bloky se vzory, které mohou selhat, získají v typu kontext `MonadFail m =>`
- více v MonadFail Proposal

Kombinace IO/Maybe: problém

Problém – Maybe se špatně kombinuje s jinými monádami, například IO

```
import Control.Monad ( when )

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 -> pure ()
        Just x  -> doAverage (sum + x) (cnt + 1)

main = doAverage 0 0
```

Kombinace IO/Maybe: problém

Problém – Maybe se špatně kombinuje s jinými monádami, například IO

```
import Control.Monad ( when )

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 -> pure ()
        Just x  -> doAverage (sum + x) (cnt + 1)

main = doAverage 0 0
```

- chceme, aby se na řádku `num <- readMaybe <$> getLine` rozbalilo IO i Maybe

Řešení: `MaybeT`

Pro libovolnou monádu `m` můžeme zavést: `MaybeT m a`

- přidává schopnost selhání (pomocí `Maybe`)

Řešení: MaybeT

Pro libovolnou monádu `m` můžeme zavést: `MaybeT m a`

- přidává schopnost selhání (pomocí `Maybe`)

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

Řešení: MaybeT

Pro libovolnou monádu `m` můžeme zavést: `MaybeT m a`

- přidává schopnost selhání (pomocí `Maybe`)

```
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 x = MaybeT $ fmap (fmap f) (runMaybeT x)
                -- 1st fmap from Functor m
                -- 2nd fmap from Functor Maybe
```

MaybeT

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

instance Applicative m => Applicative (MaybeT m) where
  pure :: a -> MaybeT m a
  pure = MaybeT . pure . pure

  (<*>) :: MaybeT m (a -> b) -> MaybeT m a -> MaybeT m b
  f <*> x = MaybeT $
            liftA2 (<*>) (runMaybeT f) (runMaybeT x)

  -- (<*>) je typu Maybe (a -> b) -> Maybe a -> Maybe b
  -- liftA2 je uvnitř Applicative m
```

MaybeT

```haskell
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 -> pure Nothing
          Just px -> runMaybeT (f px) -- px :: a

  fail _ = MaybeT (pure Nothing)
```

Transformátory monád

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

Transformátory monád

- přidávání nových vlastností monádám
- například `MaybeT`, `ExceptT` a další
- pro libovolnou monádu `m` jsou `MaybeT m`, `ExceptT 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
```

ExceptT

- `Control.Monad.Trans.Except`
- přidává práci s chybami do monády – narozdíl od `MaybeT` obsahuje i chybovou hodnotu
- ```
  newtype Except e m a =
      ExceptT { runExceptT :: m (Either e a) }
  ```
- instance lze vytvořit obdobně jako pro `MaybeT`

```
throwE :: Monad m => e -> ExceptT e m a
catchE :: Monad m
          => ExceptT e m a          -- computation
          -> (e -> ExceptT e' m a) -- handler
          -> ExceptT e' m a
```
```

Samostatná práce: dokončení evaluátoru

- stáhněte si soubor `Task12.hs` z ISu
- implementujte interpret pro `Command`:
  `eval :: Command -> IO ()`
- interně použijte state pro hodnoty proměnných
- `print` příkaz řešte pomocí writer a zachytávejte všechny hodnoty do seznamu čísel a na konci tento seznam vypište
- na standardní chybový výstup vypište chybu pokud dojde k práci s nedefinovanou proměnnou (a následně se chovejte, jako by v ní byla 0)
- doporučené signatury pomocných funkcí:

```
aeval :: (MonadState Assignment m, MonadIO m)
      => AExpr -> m Integer
beval :: (MonadState Assignment m, MonadIO m)
      => BExpr -> m Bool
ceval :: (MonadState Assignment m
          , MonadWriter [Integer] m, MonadIO m)
      => Command -> m ()
```