Řešení chyb v čistém a monadickém kódu, monadické transformátory

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2017

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

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
```

`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)`
- `readMaybe x >>= \nx -> readMaybe y >>= pure . (nx +)`

Maybe/Either v monádě

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

- `mapM readMaybe`
- `fmap (+ 1) (readMaybe x)`
- `readMaybe x >>= \nx -> readMaybe y >>= pure . (nx +)`
- `(+) <$> readMaybe x <*> 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 =>`

Kombinace IO/Maybe: problém

Problém je, Maybe se špatně kombinuje s jinými monádami, například IO:

```
import Text.Read ( readMaybe )
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 -> return ()
        Just x  -> doAverage (sum + x) (cnt + 1)

main = doAverage 0 0
```

Kombinace IO/Maybe: problém

Problém je, Maybe se špatně kombinuje s jinými monádami, například IO:

```
import Text.Read ( readMaybe )
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 -> return ()
        Just x  -> doAverage (sum + x) (cnt + 1)

main = doAverage 0 0
```

- ideálně bychom chtěli, aby se na řádku (#) rozbalilo IO i Maybe

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 vrací `Maybe` hodnoty
- pokud narazí na `Nothing`, další akce ignoruje

Je třeba zabalit do `newtype`

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

instance Functor IOMaybe where
```

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

Je třeba zabalit do `newtype`

```
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
        pure $ fmap f ma  -- fmap in Maybe monad
```

Pokud chcete zadat signatury funkcí v instanci, musíte zapnout rozšíření GHC `InstanceSigs` (viz soubor v ISu, nebo `ghci -XInstanceSigs`).

Kombinace IO/Maybe: řešení

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

instance Applicative IOMaybe where
```

Kombinace IO/Maybe: řešení

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

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

Kombinace IO/Maybe: řešení

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

instance Monad IOMaybe where
```

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 -> pure Nothing
            Just px -> runIOMaybe (f px) -- px :: a

    return = pure
    fail _ = IOMaybe (pure 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`

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

Zobecňujeme: MaybeT

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

```
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 . pure
    (<*>) :: 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 = pure
    fail _ = MaybeT (return Nothing)

liftMaybeT :: Monad m => m a -> MaybeT m a
liftMaybeT x = MaybeT (Just <$> x)
```

Monadické transformátory

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

Monadické transformátory

- 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
```

Musí platit:

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

ExceptT

- `Control.Monad.Trans.Except`
- přidává práci s chybami do monády
- `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

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

- `IOMaybe` (stejně jako `MaybeT/ExceptT`) nemá žádnou podporu pro zachytávání výjimek
- 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 zabalený `Nothing`) a `x` pokud je `act` neúspěšná.
- 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).[1]

[1] Nápověda: Zaveďte si pomocnou funkci `catchIOE` jako typovou specializaci `Control.Exception.catch`, vyřešíte tím problém „The type variable ‘e0’ is ambiguous“:

```
catchIOE :: IO a -> (IOException -> IO a) -> IO a
catchIOE = catch
```