# Transformátory monád

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

# Připomenutí: Reader, Writer, State

Monády čtenáře

- `newtype Reader r a = Reader {runReader :: r -> a}`
- `((->) r)` (funkce z `r`)
- read-only přístup ke sdílenému kontextu

Monády písaře

- `newtype Writer w a = Writer {runWriter :: (a, w)}`
- `((,) w)` (dvojice s první složkou `w`)
- zápis do sdíleného výstupu

Stavová monáda

- `newtype State s a = State {runState :: s -> (a, s)}`
- měnitelný stav předávaný mezi výpočty

Pozn.: klíčové slovo `newtype` budeme občas vynechávat, protože se nevleze na snímky.

# Připomenutí: pomocné funkce

- pro pohodlnou práci s monádami existují pomocné funkce...
  - čtenáři: `ask` a `asks`, `local`
  - písaři: `tell`, `listen`, `censor`
  - stav: `get`, `gets`, `put`, `modify`

# Připomenutí: pomocné funkce

- pro pohodlnou práci s monádami existují pomocné funkce...
  - čtenáři: `ask` a `asks`, `local`
  - písaři: `tell`, `listen`, `censor`
  - stav: `get`, `gets`, `put`, `modify`
- ... kterým nezáleží na konkrétním typu monády:
  - `asks :: MonadReader r m => (r -> a) -> m a`

    `m` je libovolná monáda, která se chová jako čtenář z `r`
    (může být `Reader r` nebo `((->) r)`)

# Připomenutí: pomocné funkce

- pro pohodlnou práci s monádami existují pomocné funkce...
  - čtenáři: `ask` a `asks`, `local`
  - písaři: `tell`, `listen`, `censor`
  - stav: `get`, `gets`, `put`, `modify`
- ... kterým nezáleží na konkrétním typu monády:
  - `asks :: MonadReader r m => (r -> a) -> m a`

    `m` je libovolná monáda, která se chová jako čtenář z `r` (může být `Reader r` nebo `((->) r)`)
  - `tell :: MonadWriter w m => w -> m ()`

    `m` je libovolná monáda, která se chová jako písař do `w` (může být `Writer w` nebo `((,) w)`)

# Připomenutí: pomocné funkce

- pro pohodlnou práci s monádami existují pomocné funkce...
  - čtenáři: `ask` a `asks`, `local`
  - písaři: `tell`, `listen`, `censor`
  - stav: `get`, `gets`, `put`, `modify`
- ... kterým nezáleží na konkrétním typu monády:
  - `asks :: MonadReader r m => (r -> a) -> m a`

    `m` je libovolná monáda, která se chová jako čtenář z `r`
    (může být `Reader r` nebo `((->) r)`)
  - `tell :: MonadWriter w m => w -> m ()`

    `m` je libovolná monáda, která se chová jako písař do `w`
    (může být `Writer w` nebo `((,) w)`)
  - `modify :: MonadState s m => (s -> s) -> m ()`

    `m` je libovolná monáda, která má měnitelný stav typu `s`
    (může být `State s` nebo...?)

Všimněte si, že jsou třídy parametrisované více typy (vyžaduje rozšíření `MultiParamTypeClasses`).

# Příklad: vyhodnocování výrokových formulí

```haskell
data Formula = Var String | And Formula Formula
             | Not Formula | Or Formula Formula
             | Let String Formula Formula
             deriving (Eq, Ord, Show)

type Valuation = Map String Bool
eval :: Formula -> Reader Valuation Bool
eval (Var v)   = asks $ Map.findWithDefault False v
eval (And x y) = liftM2 (&&) (eval x) (eval y)
eval (Or  x y) = liftM2 (||) (eval x) (eval y)
eval (Not x)   = not <$> eval x
eval (Let v x y) = eval x >>= \xv -> local (Map.insert v xv) (eval y)
```

- pracujeme v monádě čtenáře (čteme valuaci)

# Příklad: vyhodnocování výrokových formulí

```haskell
data Formula = Var String | And Formula Formula
             | Not Formula | Or Formula Formula
             | Let String Formula Formula
             deriving (Eq, Ord, Show)

type Valuation = Map String Bool
eval :: Formula -> Reader Valuation Bool
eval (Var v)   = asks $ Map.findWithDefault False v
eval (And x y) = liftM2 (&&) (eval x) (eval y)
eval (Or  x y) = liftM2 (||) (eval x) (eval y)
eval (Not x)   = not <$> eval x
eval (Let v x y) = eval x >>= \xv -> local (Map.insert v xv) (eval y)
```

- pracujeme v monádě čtenáře (čteme valuaci)
- chceme navíc sbírat nějakou „písařskou“ informaci
  - množina nedefinovaných proměnných (**`Set String`**)
  - zda dochází k překrytí proměnné (**`Any`**)
  - velikost formule (**`Sum Integer`**)
  - ...

# Příklad: vyhodnocování výrokových formulí

```haskell
data Formula = Var String | And Formula Formula
             | Not Formula | Or Formula Formula
             | Let String Formula Formula
             deriving (Eq, Ord, Show)

type Valuation = Map String Bool
eval :: Formula -> Valuation -> Writer (Set String) Bool
eval (Var v)   val = case ... val of ... tell ...
eval (And x y) val = liftM2 (&&) (eval x val) (eval y val)
eval (Or  x y) val = liftM2 (||) (eval x val) (eval y val)
eval (Not x)   val = not <$> eval x val
eval (Let v x y) val = eval x val >>= \xv -> eval y (insert v xv val)
```

- řešení 1: nahradíme čtenáře písařem
- read-only kontext teď musíme posílat ručně, nemáme `local`
- struktura se příliš nemění, liftování je skoro stejné
- otravné; nemohl by čtenář a písař fungovat zároveň?

# Kombinace monád

- **do**-blok, liftování apod. neumí pracovat ve více monádách zároveň
- na použití `ask` a `tell` zaráz ale nepotřebujeme více monád
- stačí instance pro **`MonadReader`** i pro **`MonadWriter`** zároveň

# Kombinace monád

- **do**-blok, liftování apod. neumí pracovat ve více monádách zároveň
- na použití `ask` a `tell` zaráz ale nepotřebujeme více monád
- stačí instance pro **MonadReader** i pro **MonadWriter** zároveň
- řešení 2: napíšeme vlastní monádu, která se chová jako čtenář `r` i jako písař `w`:

```
newtype RW r w a = RW { runRW :: r -> (w, a) }
```

★ zkuste si jako procvičení napsat instance pro **Functor**, **Applicative** a **Monad**

# Řešení pomocí RW

```
newtype RW r w a = RW { runRW :: r -> (w, a) }
    ... -- instance vynechány
eval :: Formula -> RW Valuation (Set String) Bool
eval (Var v)   = asks (Map.lookup v) >>= \case
                   Nothing -> tell (Set.singleton v) $> False
                   Just vx -> pure vx
eval (And x y) = liftM2 (&&) (eval x) (eval y)
eval (Or  x y) = liftM2 (||) (eval x) (eval y)
eval (Not x)   = not <$> eval x
eval (Let v x y) = eval x >>= \xv -> local (Map.insert v xv) (eval y)
```

- to už vypadá velmi použitelně
- seznam nedefinovaných proměnných použitých ve formuli:
  `toList . fst $ runRW (eval formula) valuation`

# Třídy MonadReader a MonadWriter

- Naši monádu stačí už jen naučit číst a psát...

```haskell
class Monad m => MonadReader r m | m -> r[1] where
    {-# MINIMAL (ask | reader), local #-}
    ask    :: m r
    local  :: (r -> r) -> m a -> m a
    reader :: (r -> a) -> m a

class (Monoid w, Monad m) => MonadWriter w m | m -> w[1] where
    {-# MINIMAL (writer | tell), listen, pass #-}
    writer :: (a,w) -> m a
    tell   :: w -> m ()
    listen :: m a -> m (a, w)
    pass   :: m (a, w -> w) -> m a
```

[1] zápis „... | m -> r“ znamená, že typ `r` musí být lze jednoznačně odvodit z typu `m` (tzv. funkční závislost). Vyžaduje rozšíření `FunctionalDependencies`.
Psaní instancí pak vyžaduje `FlexibleInstances`.

# Třídy MonadReader a MonadWriter

- Naši monádu stačí už jen naučit číst a psát...

```
class Monad m => MonadReader r m | m -> r[1] where
    {-# MINIMAL (ask | reader), local #-}
    ask    :: m r
    local  :: (r -> r) -> m a -> m a
    reader :: (r -> a) -> m a

class (Monoid w, Monad m) => MonadWriter w m | m -> w[1] where
    {-# MINIMAL (writer | tell), listen, pass #-}
    writer :: (a,w) -> m a
    tell   :: w -> m ()
    listen :: m a -> m (a, w)
    pass   :: m (a, w -> w) -> m a
```

- ... což je ale hrozně moc práce!

[1] zápis „... | m -> r“ znamená, že typ `r` musí být lze jednoznačně odvodit z typu `m` (tzv. funkční závislost). Vyžaduje rozšíření `FunctionalDependencies`.
Psaní instancí pak vyžaduje `FlexibleInstances`.

# Kombinace monád podruhé

- předchozí řešení je funkční, ale má několik vad:
  - velká vývojářská režie – museli jsme napsat pět instancí
  - kromě otravnosti to zvyšuje možnost zanesení chyby
  - jiné kombinace musíme napsat zase od znovu
- chtěli bychom nějaký příjemnější a obecnější způsob jak tvořit monády s více funkcionalitami

# Kombinace monád podruhé

- předchozí řešení je funkční, ale má několik vad:
  - velká vývojářská režie – museli jsme napsat pět instancí
  - kromě otravnosti to zvyšuje možnost zanesení chyby
  - jiné kombinace musíme napsat zase od znovu
- chtěli bychom nějaký příjemnější a obecnější způsob jak tvořit monády s více funkcionalitami
- myšlenka: navrstvit „poskytovatele monadické funkcionality“ na sebe
- mohli bychom pak zadefinovat něco jako

  ```
  type RW r w = AddWriting w (Reader r)
  ```

  a o žádné instance se nestarat

# Transformátory monád

- transformátor monád = z monády udělá jinou monádu
- poskytované balíkem `mtl` (Monad Transformer Library)
- přidává monadickou funkcionalitu k existující monádě
- příklad: `WriterT :: * -> (* -> *) -> (* -> *)`
  - `WriterT w m a = WriterT {runWriterT :: m (a, w)}`

★ Poznámka k výrazivu: ~~monadické~~ ~~transformery~~

# Transformátory monád

- transformátor monád = z monády udělá jinou monádu
- poskytované balíkem `mtl` (Monad Transformer Library)
- přidává monadickou funkcionalitu k existující monádě
- příklad: `WriterT :: * -> (* -> *) -> (* -> *)`
  - `WriterT w m a = WriterT {runWriterT :: m (a, w)}`
  - přidává možnost logování, neboli existují instance

    ```
    (Monoid w, Monad m) => Monad (WriterT w m)
    (Monoid w, Monad m) => MonadWriter w (WriterT w m)
    ```

★ Poznámka k výrazivu: ~~monadické~~ ~~transformery~~

# Transformátory monád

- transformátor monád = z monády udělá jinou monádu
- poskytované balíkem `mtl` (Monad Transformer Library)
- přidává monadickou funkcionalitu k existující monádě
- příklad: `WriterT :: * -> (* -> *) -> (* -> *)`
  - `WriterT w m a = WriterT {runWriterT :: m (a, w)}`
  - přidává možnost logování, neboli existují instance

    ```
    (Monoid w, Monad m) => Monad (WriterT w m)
    (Monoid w, Monad m) => MonadWriter w (WriterT w m)
    ```
  - ```
    promptLog :: String -> WriterT [String] IO String
    promptLog prompt = do putStr prompt
                          line <- getLine
                          tell [prompt ++ line]
                          pure line
    ```

★ Poznámka k výrazivu: ~~monadické~~ ~~transformery~~
    ```

# Transformátory monád

- transformátor monád = z monády udělá jinou monádu
- poskytované balíkem `mtl` (Monad Transformer Library)
- přidává monadickou funkcionalitu k existující monádě
- příklad: `WriterT :: * -> (* -> *) -> (* -> *)`
  - `WriterT w m a = WriterT {runWriterT :: m (a, w)}`
  - přidává možnost logování, neboli existují instance

    ```
    (Monoid w, Monad m) => Monad (WriterT w m)
    (Monoid w, Monad m) => MonadWriter w (WriterT w m)
    ```
  - 
    ```
    promptLog :: String -> WriterT [String] IO String
    promptLog prompt = do putStr prompt   -- error!
                          line <- getLine -- error!
                          tell [prompt ++ line]
                          pure line
    ```

★ Poznámka k výrazivu: ~~monadické~~ ~~transformery~~

# Liftování

```
promptLog :: String -> WriterT [String] IO String
promptLog prompt = do putStr prompt   -- error!
                      line <- getLine -- error!
                      tell [prompt ++ line]
                      pure line
```

- `getLine` je typu **`IO String`**, ale potřebujeme, aby byl typu **`WriterT [String] IO String`**

# Liftování

```
promptLog :: String -> WriterT [String] IO String
promptLog prompt = do putStr prompt   -- error!
                      line <- getLine -- error!
                      tell [prompt ++ line]
                      pure line
```

- `getLine` je typu **`IO String`**, ale potřebujeme, aby byl typu **`WriterT [String] IO String`**
- Transformátory jsou instancemi typové třídy **`MonadTrans`**

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

# Liftování

```
promptLog :: String -> WriterT [String] IO String
promptLog prompt = do putStr prompt   -- error!
                      line <- getLine -- error!
                      tell [prompt ++ line]
                      pure line
```

- `getLine` je typu **`IO String`**, ale potřebujeme, aby byl typu **`WriterT [String] IO String`**
- Transformátory jsou instancemi typové třídy **`MonadTrans`**

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

- `lift` z výpočtu v monádě `m` udělá výpočet v monádě `t m`.

# Liftování

```
promptLog :: String -> WriterT [String] IO String
promptLog prompt = do lift (putStr prompt)
                      line <- lift getLine
                      tell [prompt ++ line]
                      pure line
```

- `getLine` je typu **`IO String`**, ale potřebujeme, aby byl typu **`WriterT [String] IO String`**
- Transformátory jsou instancemi typové třídy **`MonadTrans`**

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

- `lift` z výpočtu v monádě `m` udělá výpočet v monádě `t m`.

# Transformátory z mtl

Všechny tři monády z minula mají svůj přílušný transformátor:

- `WriterT w m a = WriterT {runWriterT :: m (a, w)}`
- `ReaderT r m a = ReaderT {runReaderT :: r -> m a}`
- `StateT s m a = StateT {runStateT :: s -> m (a,s)}`

# Transformátory z `mtl`

Všechny tři monády z minula mají svůj přílušný transformátor:

- `WriterT w m a = WriterT {runWriterT :: m (a, w)}`
- `ReaderT r m a = ReaderT {runReaderT :: r -> m a}`
- `StateT s m a = StateT {runStateT :: s -> m (a,s)}`

- analogicky existují také `evalStateT`, `execWriterT` apod.
- ve skutečnosti jsou monády z minula definovány podle vzoru:
  `type Reader r = ReaderT r Identity`

# Transformátory z mtl

Všechny tři monády z minula mají svůj přílušný transformátor:

- `WriterT w m a = WriterT {runWriterT :: m (a, w)}`
- `ReaderT r m a = ReaderT {runReaderT :: r -> m a}`
- `StateT s m a = StateT {runStateT :: s -> m (a,s)}`

- analogicky existují také `evalStateT`, `execWriterT` apod.
- ve skutečnosti jsou monády z minula definovány podle vzoru:
  `type Reader r = ReaderT r Identity`
- existuje i kombinace předchozích transformátorů:
  `RWST r w s m a = RWST {runRWST :: r -> s -> m (a,s,w)}`
- (a dle očekávání i monáda `RWS r w s = RWST r w s Identity`)

# Řešení úlohy z minula s transformátory

```haskell
eval :: Formula -> ReaderT Valuation (Writer (Set String)) Bool
eval (Var v)   = asks (Map.lookup v) >>= \case
                   Nothing -> lift ( tell (Set.singleton v) ) $> False
                   Just vx -> pure vx
eval (And x y) = liftM2 (&&) (eval x) (eval y)
eval (Or  x y) = liftM2 (||) (eval x) (eval y)
eval (Not x)   = not <$> eval x
eval (Let v x y) = eval x >>= \xv -> local (Map.insert v xv) (eval y)
```

# Řešení úlohy z minula s transformátory

```
eval :: Formula -> ReaderT Valuation (Writer (Set String)) Bool
eval (Var v)   = asks (Map.lookup v) >>= \case
                   Nothing -> lift ( tell (Set.singleton v) ) $> False
                   Just vx -> pure vx
eval (And x y) = liftM2 (&&) (eval x) (eval y)
eval (Or  x y) = liftM2 (||) (eval x) (eval y)
eval (Not x)   = not <$> eval x
eval (Let v x y) = eval x >>= \xv -> local (Map.insert v xv) (eval y)
```

- přidávání `lift`ů není moc elegantní
- více transformačních vrstev znamená více liftování
- chtěli bychom se obejít bez `lift`ů

# Řešení úlohy z minula s transformátory

```
eval :: Formula -> ReaderT Valuation (Writer (Set String)) Bool
eval (Var v)   = asks (Map.lookup v) >>= \case
                   Nothing -> tell (Set.singleton v) $> False
                   Just vx -> pure vx
eval (And x y) = liftM2 (&&) (eval x) (eval y)
eval (Or  x y) = liftM2 (||) (eval x) (eval y)
eval (Not x)   = not <$> eval x
eval (Let v x y) = eval x >>= \xv -> local (Map.insert v xv) (eval y)
```

- přidávání `lift`ů není moc elegantní
- více transformačních vrstev znamená více liftování
- chtěli bychom se obejít bez `lift`ů
- kód funguje správně i bez nich!

# Spolupráce transformátorů z `mtl`

- transformátory z `mtl` jsou navrženy tak, aby se v jejich kombinacích nemusel používat `lift`
- to je umožněno instancemi jako např:

```
(MonadReader r m) => MonadReader r (WriterT w m)
(MonadReader r m) => MonadReader r (StateT s m)
...
```

- tedy například pokud transformátor obaluje čtenáře, je výsledná monáda opět čtenářem

# Spolupráce transformátorů z `mtl`

- transformátory z `mtl` jsou navrženy tak, aby se v jejich kombinacích nemusel používat `lift`
- to je umožněno instancemi jako např:

```
(MonadReader r m) => MonadReader r (WriterT w m)
(MonadReader r m) => MonadReader r (StateT s m)
...
```

- tedy například pokud transformátor obaluje čtenáře, je výsledná monáda opět čtenářem
- `lift` je třeba použít, obsahuje-li monáda např. více **`ReaderT`**
- bez liftování se neobjdeme při práci s **`IO`**

# IO a monádové transformátory

- neexistuje žádné **IOT**, takže **IO** musí být vždy „na dně“ pod všemi transformátory[1]

[1] To ale neznamená, že bude nejblíže samotnému výsledku výpočtu.

# IO a monádové transformátory

- neexistuje žádné **IOT**, takže **IO** musí být vždy „na dně“ pod všemi transformátory[1]
- vstupně-výstupní akce pracují v **IO**, nikoli v nějakém obecném **MonadIO**, takže transformátory nemohou propagovat samotné V/V akce jako např. ask v případě **MonadReader** r apod.[2]

[1] To ale neznamená, že bude nejblíže samotnému výsledku výpočtu.

[2] Můžete se podívat na balík unliftio, který se přesně o toto snaží.

# IO a monádové transformátory

- neexistuje žádné **IOT**, takže **IO** musí být vždy „na dně“ pod všemi transformátory[1]
- vstupně-výstupní akce pracují v **IO**, nikoli v nějakém obecném **MonadIO**, takže transformátory nemohou propagovat samotné V/V akce jako např. ask v případě **MonadReader** r apod.[2]
- mohou ale propagovat *způsob* jak spustit V/V akce:

```
class Monad m => MonadIO m where
    liftIO :: IO a -> m a
```

[1] To ale neznamená, že bude nejblíže samotnému výsledku výpočtu.

[2] Můžete se podívat na balík unliftio, který se přesně o toto snaží.

# IO a monádové transformátory

- neexistuje žádné **IOT**, takže **IO** musí být vždy „na dně“ pod všemi transformátory[1]
- vstupně-výstupní akce pracují v **IO**, nikoli v nějakém obecném **MonadIO**, takže transformátory nemohou propagovat samotné V/V akce jako např. ask v případě **MonadReader** r apod.[2]
- mohou ale propagovat *způsob* jak spustit V/V akce:

```
class Monad m => MonadIO m where
    liftIO :: IO a -> m a
```

- `liftIO` ≈ „liftovací zkratka“ k **IO** vespod:
  `liftIO getLine` $\rightsquigarrow^*$ `lift . ⋯ . lift $ getLine`

[1] To ale neznamená, že bude nejblíže samotnému výsledku výpočtu.

[2] Můžete se podívat na balík `unliftio`, který se přesně o toto snaží.

# Transformátor pro ošetřování chyb

- výpočty s výjimkami: `Either e`
- příslušný transformátor (z modulu `Control.Monad.Except`):

```
ExceptT e m a = ExceptT (m (Either e a))
```

- 
```
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

# Transformátor pro ošetřování chyb

- výpočty s výjimkami: `Either e`
- příslušný transformátor (z modulu `Control.Monad.Except`):

```
ExceptT e m a = ExceptT (m (Either e a))
```

- 
```
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

- povýšení libovolné `Either`-akce do třídy `MonadError`:

```
liftEither :: MonadError e m => Either e a -> m a
```

# Transformátor pro ošetřování chyb

- výpočty s výjimkami: `Either e`
- příslušný transformátor (z modulu `Control.Monad.Except`):

```
ExceptT e m a = ExceptT (m (Either e a))
```

- 
```
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

- povýšení libovolné `Either`-akce do třídy `MonadError`:

```
liftEither :: MonadError e m => Either e a -> m a
```

- neplést se standardní třídou `MonadFail`, do níž se přesouvá (v GHC 8.8) z `Monad` akce `fail :: String -> m a` volaná při neúspěšném pattern-matchingu v `do`-bloku

# Transformátor pro ošetřování chyb

- výpočty s výjimkami: `Either e`
- příslušný transformátor (z modulu `Control.Monad.Except`):

```
ExceptT e m a = ExceptT (m (Either e a))
```

- 
```
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

- povýšení libovolné `Either`-akce do třídy `MonadError`:

```
liftEither :: MonadError e m => Either e a -> m a
```

- neplést se standardní třídou `MonadFail`, do níž se přesouvá (v GHC 8.8) z `Monad` akce `fail :: String -> m a` volaná při neúspěšném pattern-matchingu v `do`-bloku

★ `MaybeT` existuje, ale samo o sobě neposkytuje funkcionalitu pro ošetřování chyb (tj. nepřináší instanci `MonadError`)

# Pořadí skládání transformátorů I

```haskell
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ReaderT r (Writer w) a
```

vs.

```haskell
WriterT w (Reader r) a
```

# Pořadí skládání transformátorů I

```haskell
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

`ReaderT r (Writer w) a` vs. `WriterT w (Reader r) a`

zapomeneme „zbytečné“ konstruktory

# Pořadí skládání transformátorů I

```haskell
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ReaderT r (Writer w) a
```
vs.
```haskell
WriterT w (Reader r) a
```

zapomeneme „zbytečné“ konstruktory

```haskell
r -> Writer w a
```
vs.
```haskell
Reader r (a, w)
```

# Pořadí skládání transformátorů I

```haskell
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ReaderT r (Writer w) a        vs.        WriterT w (Reader r) a
```

zapomeneme „zbytečné“ konstruktory

```haskell
r -> Writer w a               vs.        Reader r (a, w)
r -> (a, w)                   =          r -> (a, w)
```

# Pořadí skládání transformátorů I

```
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

| | | |
|---|---|---|
| `ReaderT r (Writer w) a` | vs. | `WriterT w (Reader r) a` |
| | zapomeneme „zbytečné“ konstruktory | |
| `r -> Writer w a` | vs. | `Reader r (a, w)` |
| `r -> (a, w)` | = | `r -> (a, w)` |

- **`ReaderT`** a **`WriterT`** můžeme prohodit
- liší se pak jen pořadí jejich vyhodnocení:

```
runWriter.flip runReaderT c        flip runReader c.runWriterT
```

# Pořadí skládání transformátorů II

```haskell
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype StateT  s m a = StateT  {runStateT  :: s -> m (a, s)}
```

Jaký je rozdíl mezi následující dvojicí typů?

`ReaderT r (State s) a` vs. `StateT s (Reader r) a`

# Pořadí skládání transformátorů II

```haskell
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype StateT  s m a = StateT  {runStateT  :: s -> m (a, s)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ReaderT r (State s) a
r -> State s a
```

vs.

```haskell
StateT s (Reader r) a
s -> Reader r (a, s)
```

# Pořadí skládání transformátorů II

```haskell
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype StateT  s m a = StateT  {runStateT  :: s -> m (a, s)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ReaderT r (State s) a                    vs.    StateT s (Reader r) a
r -> State s a                                   s -> Reader r (a, s)
r -> s -> (a, s)                         ≃           s -> r -> (a, s)
```

# Pořadí skládání transformátorů II

```
newtype ReaderT r m a = ReaderT {runReaderT :: r -> m a}
newtype StateT  s m a = StateT  {runStateT  :: s -> m (a, s)}
```

Jaký je rozdíl mezi následující dvojicí typů?

| | | |
|---|---|---|
| `ReaderT r (State s) a` | vs. | `StateT s (Reader r) a` |
| `r -> State s a` | | `s -> Reader r (a, s)` |
| `r -> s -> (a, s)` | $\simeq$ | `s -> r -> (a, s)` |

**ReaderT** a **StateT** nekomutují, ale liší se jen pořadím parametrů

# Pořadí skládání transformátorů III

```haskell
newtype ExceptT e m a = ExceptT (m (Either e a))
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ExceptT e (Writer w) a
```

vs.

```haskell
WriterT w (Either e) a
```

# Pořadí skládání transformátorů III

```haskell
newtype ExceptT e m a = ExceptT (m (Either e a))
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ExceptT e (Writer w) a
Writer w (Either e a)
```

vs.

```haskell
WriterT w (Either e) a
Either e (a, w)
```

# Pořadí skládání transformátorů III

```haskell
newtype ExceptT e m a = ExceptT (m (Either e a))
newtype WriterT w m a = WriterT {runWriterT :: m (a, w)}
```

Jaký je rozdíl mezi následující dvojicí typů?

```haskell
ExceptT e (Writer w) a          vs.   WriterT w (Either e) a
Writer w (Either e a)                       Either e (a, w)
(Either e a, w)                 ≠     Either e (a, w)
```

# Pořadí skládání transformátorů III

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

Jaký je rozdíl mezi následující dvojicí typů?

```
ExceptT e (Writer w) a                vs.   WriterT w (Either e) a
Writer w (Either e a)                             Either e (a, w)
(Either e a, w)                       ≠           Either e (a, w)
```

- **ExceptT** a **WriterT** nekomutují
- liší se v tom, jestli při chybě zůstane log platný

# Samostatné programování

# Writer a IO

Mějme následující program, který pro zadanou cestu vypíše počet souborů v daném podstromě.

```
import System.Directory
import System.FilePath

numFiles :: FilePath -> IO Int
numFiles path = do
  contents <- listDirectory path
  let paths = map (path </>) contents
  files <- filterM doesFileExist paths
  dirs <- filterM doesDirectoryExist paths
  fmap (+ length files) (sum <$> forM dirs numFiles)
```

Upravte funkci tak, aby pomocí **Writer** monády produkovala i seznam všech adresářů, které prošla. Tj. napište funkci

```
numFiles' :: FilePath -> WriterT [String] IO Int
```

# Reader a Except

Vzpomeňte si z minula na funkci na vyhodnocení aritmetických výrazů v **Reader** monádě.

```
data Expr = Var String | Const Int
          | Add Expr Expr | Sub Expr Expr | Mul Expr Expr
          deriving (Show, Eq)

type Valuation = Map String Int

eval :: Expr -> Reader Valuation Int
eval (Var v) = asks (M.findWithDefault 0 v)
eval (Const n) = return n
eval (Add l r) = liftM2 (+) (eval l) (eval r)
eval (Sub l r) = liftM2 (-) (eval l) (eval r)
eval (Mul l r) = liftM2 (*) (eval l) (eval r)
```

Upravte funkci tak, aby při použití proměnné bez přiřazené hodnoty nevrátila `0`, ale pomocí transformátoru **ExceptT** vrátila chybu obshahující název nepřiřazené proměnné. Tj. napište funkci `eval' :: Expr -> ExceptT String (Reader Valuation) Int`.

# State a Reader

Vzpomeňte si z minula na implementaci DFS pomocí **State** monády.

```
type Vertex = Char
type Edge = (Vertex, Vertex)

dfs' :: Vertex -> [Edge] -> State (Set Vertex) [Vertex]
dfs' v edges = do
    modify' (Set.insert v)
    let succs = [to | (from, to) <- edges, from == v]
    next <- forM succs $ \s -> do
        visited <- get
        if s `elem` visited then return [] else dfs' s edges
    return $ v : concat next

dfs :: Vertex -> [Edge] -> [Vertex]
dfs v es = evalState (dfs' v es) Set.empty
```

Upravte funkci tak, aby si předávala graf (tj. edges) pomocí **ReaderT**.

# Zbytek z minula

Pokud jste minule nestihli některé příklady na monády **`Reader`**, **`Writer`** a **`State`**, můžete se k nim vrátit.