Monády pro čtení kontextu, transformery monád (`Reader`, `ReaderT`)

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Motivace: přístup ke kontextu/konfiguraci

uvažme aritmetické výrazy z osmého domácího úkolu IB015 (2017):

```
type Varname = String

data AExpr = Con Integer
           | Var Varname
           | Add AExpr AExpr
           | Mul AExpr AExpr
           deriving ( Eq, Show, Read )
```

Motivace: přístup ke kontextu/konfiguraci

uvažme aritmetické výrazy z osmého domácího úkolu IB015 (2017):

```
type Varname = String

data AExpr = Con Integer
           | Var Varname
           | Add AExpr AExpr
           | Mul AExpr AExpr
           deriving ( Eq, Show, Read )
```

- vyhodnocení vyžaduje přiřazení hodnot proměnným

```
type Assignment = Map Varname Integer
val' = Map.findWithDefault 0

evala' :: AExpr -> Assignment -> Integer
evala' (Con x) _    = x
evala' (Var x) a    = val' x a
evala' (Add x y) a = evala' x a + evala' y a
```

Reader

- funkce `evala'` má nevýhodu, že musí pořád explicitně přeposílat přiřazení
- zavedeme si typ „výpočtu se čtením z kontextu“

```
newtype Reader r a = Reader { runReader :: r -> a }
```

- budeme chtít:

```
evala :: AExpr -> Reader Assignment Integer

runReader (evala (Mul (Con 2) (Var "x")))
          (Map.singleton "x" 3) ~>* 6
```

Reader

- funkce `evala'` má nevýhodu, že musí pořád explicitně přeposílat přiřazení
- zavedeme si typ „výpočtu se čtením z kontextu“

```
newtype Reader r a = Reader { runReader :: r -> a }
```

- budeme chtít:

```
evala :: AExpr -> Reader Assignment Integer

runReader (evala (Mul (Con 2) (Var "x")))
          (Map.singleton "x" 3) ~>* 6
```

- budeme chtít operace s `Reader` komponovat pomocí functor/applicative/monad

Reader jako monáda I

```haskell
newtype Reader r a = Reader { runReader :: r -> a }
```

Reader jako monáda I

```haskell
newtype Reader r a = Reader { runReader :: r -> a }


instance Functor (Reader r) where
  fmap :: (a -> b) -> Reader r a -> Reader r b
  fmap f x = Reader $ f . runReader x
```

Reader jako monáda I

```haskell
newtype Reader r a = Reader { runReader :: r -> a }


instance Functor (Reader r) where
  fmap :: (a -> b) -> Reader r a -> Reader r b
  fmap f x = Reader $ f . runReader x


instance Applicative (Reader r) where
  (<*>) :: Reader r (a -> b) -> Reader r a
                             -> Reader r b
  -- runReader f :: r -> (a -> b)
  -- runReader x :: r -> a
  f <*> x = Reader $ \r -> runReader f r (runReader x r)
  pure x  = Reader $ \_ -> x
```

Reader jako monáda II

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
  --        m          a     (a -> m          b) -> m          b
  (>>=) :: Reader r a -> (a -> Reader r b) -> Reader r b
  -- runReader x :: r -> a
  -- runReader . f :: a -> r -> b
  x >>= f = Reader $
            \r -> (runReader . f) (runReader x r) r
```

Evaluátor pomocí Reader

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

ask :: Reader r r
ask = Reader id

reader :: (r -> a) -> Reader r a
reader f = Reader f

val :: String -> Reader Assignment Integer
val v = reader (Map.findWithDefault 0 v)

evala :: AExpr -> Reader Assignment Integer
evala (Con x)   = pure x
evala (Var x)   = val x
evala (Add x y) = liftA2 (+) (evala x) (evala y)
evala (Mul x y) = liftA2 (*) (evala x) (evala y)
```

-> jako Reader

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
    (>>=) :: Reader r a -> (a -> Reader r b)
                        -> Reader r b
    -- runReader x :: r -> a
    -- runReader . f :: a -> r -> b
    x >>= f = Reader $
              \r -> (runReader . f) (runReader x r) r

instance Monad ((->) r) where
    --             m a  -> (a ->      m b ) ->      m b
    (>>=) :: (r -> a) -> (a -> (r -> b)) -> (r -> b)
    x >>= f = \r -> f (x r) r
```

Evaluátor ještě jednou

využijeme toho, že `((->) r)` je instancí applicative/monad se stejným chováním jako `Reader`

```
evala'' :: AExpr -> Assignment -> Integer
evala'' (Con x)   = pure x
evala'' (Var x)   = val' x
evala'' (Add x y) = liftA2 (+) (evala'' x) (evala'' y)
evala'' (Mul x y) = liftA2 (*) (evala'' x) (evala'' y)
```

Ještě jedna užitečná funkce

```
newtype Reader r a = Reader { runReader :: r -> a }

local :: (r -> r) -> Reader r a -> Reader r a
```

- umožňuje lokálně modifikovat kontext a spustit akci v tomto modifikovaném kontextu
- nejedná se o změnu kontextu – další akce opět uvidí původní

Ještě jedna užitečná funkce

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

local :: (r -> r) -> Reader r a -> Reader r a
```

- umožňuje lokálně modifikovat kontext a spustit akci v tomto modifikovaném kontextu
- nejedná se o změnu kontextu – další akce opět uvidí původní

```haskell
local f act = Reader $ \r -> runReader act (f r)

ex_local :: (Integer, Integer, Integer)
ex_local = runReader (liftA3 (,,)
                (evala (Var "x"))
                (local (Map.insert "x" 42) (evala (Var "x")))
                (evala (Var "x")))
              Map.empty
```

Další použití `Reader`

konfigurace programu

- máme spustitelný program s konfigurací
- většina funkcionality programu potřebuje přístup ke konfiguraci
- myšlenka: funkce vrací `Reader Config a`

Další použití `Reader`

konfigurace programu

- máme spustitelný program s konfigurací
- většina funkcionality programu potřebuje přístup ke konfiguraci
- myšlenka: funkce vrací `Reader Config a`

- problém: nespoulupráce `IO` a `Reader`
- jakmile program potřebuje práci s `IO` je vyžívání `IO` a `Reader` společně nepraktické
- akce typu `Reader r (IO a)` nelze snadno řetězit

Další použití `Reader`

konfigurace programu

- máme spustitelný program s konfigurací
- většina funkcionality programu potřebuje přístup ke konfiguraci
- myšlenka: funkce vrací `Reader Config a`

- problém: nespoulupráce `IO` a `Reader`
- jakmile program potřebuje práci s `IO` je vyžívání `IO` a `Reader` společně nepraktické
- akce typu `Reader r (IO a)` nelze snadno řetězit

- nápad: spojit `IO` a `Reader` do jedné monády

```
newtype ReaderIO r a = ReaderIO
                         { runReaderIO :: r -> IO a }
```

Další použití `Reader`

konfigurace programu

- máme spustitelný program s konfigurací
- většina funkcionality programu potřebuje přístup ke konfiguraci
- myšlenka: funkce vrací `Reader Config a`

- problém: nespoulupráce `IO` a `Reader`
- jakmile program potřebuje práci s `IO` je vyžívání `IO` a `Reader` společně nepraktické
- akce typu `Reader r (IO a)` nelze snadno řetězit

- nápad: spojit `IO` a `Reader` do jedné monády

```
newtype ReaderIO r a = ReaderIO
                         { runReaderIO :: r -> IO a }
```

  - řešení v `Lecture09.hs`

Zobecňujeme

- v definici a instancích k `ReaderIO` jsme nikdy nevyužili toho, že pracujeme konkrétně v `IO` monádě!

Zobecňujeme

- v definici a instancích k `ReaderIO` jsme nikdy nevyužili toho, že pracujeme konkrétně v `IO` monádě!
- přístup lze zobecnit na libovolnou monádu:

```
data ReaderT r m a = ReaderT
                       { runReaderT :: r -> m a }
```

Zobecňujeme

- v definici a instancích k `ReaderIO` jsme nikdy nevyužili toho, že pracujeme konkrétně v `IO` monádě!
- přístup lze zobecnit na libovolnou monádu:

```
data ReaderT r m a = ReaderT
                       { runReaderT :: r -> m a }
```

  - jaký je druh `ReaderT`?

Zobecňujeme

- v definici a instancích k `ReaderIO` jsme nikdy nevyužili toho, že pracujeme konkrétně v `IO` monádě!
- přístup lze zobecnit na libovolnou monádu:

```
data ReaderT r m a = ReaderT
                       { runReaderT :: r -> m a }
```

  - jaký je druh `ReaderT`? `* -> (* -> *) -> * -> *`

Zobecňujeme

- v definici a instancích k `ReaderIO` jsme nikdy nevyužili toho, že pracujeme konkrétně v `IO` monádě!
- přístup lze zobecnit na libovolnou monádu:

```
data ReaderT r m a = ReaderT
                       { runReaderT :: r -> m a }
```

  - jaký je druh `ReaderT`? `* -> (* -> *) -> * -> *`
  - v instancích stačí jen nahradit `ReaderIO r` za `ReaderT r m`, `runReaderIO` za `runReaderT` a doplnit kontexty

Zobecňujeme

- v definici a instancích k `ReaderIO` jsme nikdy nevyužili toho, že pracujeme konkrétně v `IO` monádě!
- přístup lze zobecnit na libovolnou monádu:

```
data ReaderT r m a = ReaderT
                       { runReaderT :: r -> m a }
```

  - jaký je druh `ReaderT`? `* -> (* -> *) -> * -> *`
  - v instancích stačí jen nahradit `ReaderIO r` za `ReaderT r m`, `runReaderIO` za `runReaderT` a doplnit kontexty
- `ReaderT` je příkladem *transformeru monád* – transformery přidávají novou funkcionalitu dané monádě

Transformery Monád

transformery monád jsou instancí třídy `MonadTrans`

- definuje jak povýšit akci do transformované monády

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

instance MonadTrans (ReaderT r) where
    lift :: Monad m => m a -> ReaderT r m a
    lift act = ReaderT $ \_ -> act
```

Reader a ReaderT v knihovně

- balík `transformers`: `Control.Monad.Trans.Reader`
- definuje `ReaderT`, `runReaderT`, `ask`, `reader`
- Reader je alias: `type Reader r = ReaderT r Identity`
- `runReader m = runIdentity . runReaderT m`
- `Control.Monad.Trans.Class`: `lift`

Reader a ReaderT v knihovně

- balík `transformers`: `Control.Monad.Trans.Reader`
- definuje `ReaderT`, `runReaderT`, `ask`, `reader`
- `Reader` je alias: `type Reader r = ReaderT r Identity`
- `runReader m = runIdentity . runReaderT m`
- `Control.Monad.Trans.Class`: `lift`

- balík `mtl`: `Control.Monad.Reader`
- re-exportuje `Reader`, `ReaderT` z `transformers`
- typová třída `MonadReader r m` pro monády `m` chovající se jako `Reader` s kontextem `r`
- dvouparametrická třída, typ `r` je jasně dán monádou `m` (rozšíření)
- definuje `ask`, `reader`, `local` tak aby fungoval uniformně v `((->) r)`, `ReaderT`, transformerech nad `ReaderT`, ...
- typicky lepší volba než `transformers`

Používání mtl I

```haskell
{-# LANGUAGE FlexibleContexts #-}
{- imports ... -}

type Varname = String
type Assignment = Map Varname Integer

value :: MonadReader Assignment m => Varname -> m Integer
value v = reader (Map.findWithDefault 0 v)
```

- rozšíření `FlexibleContexts` umožňuje částečně instanciovat typovou třídu v kontextu
- kontext `MonadReader Assignment m`: `m` je nějaká monáda, která se umí chovat jako reader s kontextem `Assignment`

Používání `mtl` II

```
evala :: MonadReader Assignment m => AExpr -> m Integer
evala (Con x)   = pure x
evala (Var x)   = value x
evala (Add x y) = liftA2 (+) (evala x) (evala y)
evala (Mul x y) = liftA2 (*) (evala x) (evala y)
```

- `evala` lze použít s libovolnou věcí, která je instancí `MonadReader` s kontextem `Assignment`

Používání `mtl` II

```
evala :: MonadReader Assignment m => AExpr -> m Integer
evala (Con x)   = pure x
evala (Var x)   = value x
evala (Add x y) = liftA2 (+) (evala x) (evala y)
evala (Mul x y) = liftA2 (*) (evala x) (evala y)
```

- `evala` lze použít s libovolnou věcí, která je instancí `MonadReader` s kontextem `Assignment`

```
> evala (Var "x" `Add` Con 4) Map.empty
4 -- the ((->) Assignment) monad
```

Používání `mtl` II

```
evala :: MonadReader Assignment m => AExpr -> m Integer
evala (Con x)   = pure x
evala (Var x)   = value x
evala (Add x y) = liftA2 (+) (evala x) (evala y)
evala (Mul x y) = liftA2 (*) (evala x) (evala y)
```

- `evala` lze použít s libovolnou věcí, která je instancí `MonadReader` s kontextem `Assignment`

```
> evala (Var "x" `Add` Con 4) Map.empty
4 -- the ((->) Assignment) monad

> runReader (evala (Var "x" `Add` Con 4)) Map.empty
4 -- the (Reader Assignment) monad
```

Používání mtl II

```
evala :: MonadReader Assignment m => AExpr -> m Integer
evala (Con x)   = pure x
evala (Var x)   = value x
evala (Add x y) = liftA2 (+) (evala x) (evala y)
evala (Mul x y) = liftA2 (*) (evala x) (evala y)
```

- evala lze použít s libovolnou věcí, která je instancí `MonadReader` s kontextem `Assignment`

```
> evala (Var "x" `Add` Con 4) Map.empty
4 -- the ((->) Assignment) monad

> runReader (evala (Var "x" `Add` Con 4)) Map.empty
4 -- the (Reader Assignment) monad

> runReaderT (evala (Var "x" `Add` Con 4)
                >>= liftIO . print) Map.empty
(prints 4) -- the (ReaderT Assignment IO) monad
```

Vnořování `IO`

- pokud je transformer postaven nad `IO`, chceme být schopni ve výsledné monádě spouštět `IO`
- typová třída `MonadIO` monád s `IO` funkcionalitou
- `IO` akce lze vložit pomocí `liftIO`

Samostatná práce I

Uvažte následující typ logických výrazů:

```
data BExpr = Equal AExpr AExpr
           | LEQ AExpr AExpr
           | And BExpr BExpr
           | Not BExpr
          deriving ( Eq, Show, Read )
```

Implementujte funkci `evalb`, která vyhodnocuje logické výrazy pomocí `MonadReader` z `mtl`, využijte při tom naši implementaci `evala`.

Samostatná práce II

Svůj jazyk dále modifikujeme tak, že do aritmetických výrazů přidáme `let`:

```
data LAExpr = LCon Integer
            | LVar Varname
            | LAdd LAExpr LAExpr
            | LMul LAExpr LAExpr
            | Let Varname LAExpr LAExpr
           deriving ( Eq, Show, Read )
```

sémantika `let` je taková, že `Let var exp in` spustí `in` s přiřazením stejným jako je aktuální, ale kde navíc proměnná `var` má hodnotu `exp`

- výraz `exp` může obsahovat `var`, pak se ale odkazuje na `var` v kontextu celého `let`, před modifikací
- implementujte funkci `evalla`, která vyhodnocuje rozšířené aritmetické výrazy pomocí `MonadReader`
- hint: použijte funkci `local`

```
runReaderT (evalla (Let "x" (LCon 40)
```