Monády pro čtení a/nebo zapisování stavu (`Reader`, `Writer`, `State`)

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Připomenutí: monády

Monáda ≈ abstrakce výpočtu.

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

- `return x` vyrábí „konstantní výpočet“ hodnoty `x`
- `mx >>= f` komponuje výpočty: pomocí výsledku výpočtu `mx` je vytvořen nový výpočet `(f x)`.

Připomenutí: monády

IO: výpočet se vstupně-výstupními efekty

- `echo = getLine >>= putStrLn >> echo`

Maybe: výpočet, který může selhat

- ```
  lookup "xmatous3" login2uid
      >>= \uid -> lookup uid2uco uid
      >>= \uco -> lookup uco2name uco ⇝* Just "Adam"
  ```
- zobecnění: `Either e` – „selhání s odůvodněním“

[]: výpočet s více možnými výsledky

- `[1,4] >>= \x -> [pred x, x, succ x] ⇝* [0..5]`
  ```

Monáda čtenáře (Reader)

Motivace

```haskell
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 -> Valuation -> Int
eval (Var v) val = M.findWithDefault 0 v val
eval (Const n) _ = n
eval (Add l r) val = eval l val + eval r val
eval (Sub l r) val = eval l val - eval r val
eval (Mul l r) val = eval l val * eval r val
```

Motivace: mírně refaktorovaná

```haskell
eval :: Expr -> Valuation -> Int
eval (Var v) val = M.findWithDefault 0 v val
eval (Const n) _ = n
eval (Add l r) val = evalOp (+) l r val
eval (Sub l r) val = evalOp (-) l r val
eval (Mul l r) val = evalOp (*) l r val

evalOp :: (Int -> Int -> Int) -> Expr -> Expr -> Valuation -> Int
evalOp op l r val = op (eval l val) (eval r val)
```

Monáda čtenáře

- existuje monáda `Reader r`:
  ```
  newtype Reader r a = Reader {runReader :: r -> a}
  ```
- automaticky předává hodnotu typu `r` mezi výpočty s výsledky typu `a`
- hodnota se nedá měnit; dá se jen pustit podvýpočet se zadanou hodnotou
- hodnota se dá přečíst pomocí funkce `ask :: Reader r r`
- typické použití: konfigurace
- hledejte v modulu `Control.Monad.Reader` balíku `mtl`

Monáda čtenáře

```haskell
eval' :: Expr -> Reader Valuation Int
eval' (Var v) = do { val <- ask; return (M.findWithDefault 0 v val) }
-- nebo eval' (Var v) = M.findWithDefault 0 v <$> ask
eval' (Const n) = return n
eval' (Add l r) = evalOp' (+) l r
eval' (Sub l r) = evalOp' (-) l r
eval' (Mul l r) = evalOp' (*) l r

evalOp' :: (Int -> Int -> Int) -> Expr -> Expr -> Reader Valuation Int
evalOp' op l r = liftM2 op (eval' l) (eval' r)

eval e val = runReader (eval' e) val
-- nebo eval = runReader . eval'
```

Modul `Control.Monad.Reader`

Funkce pro práci s **`Reader`** monádou:

- `ask :: Reader r r`
  přečte kontext (např. `do {ctx <- ask; ...}`)
- `asks :: (r -> a) -> Reader r a`
  čtení přes „getter“ (např. `do {x <- asks fst; ...}`)
- `local :: (r -> r) -> Reader r a -> Reader r a`
  spustí výpočet s lokálně změněným kontextem

Monáda čtenáře

Funkce jako aplikativní funktor

- `((->) r)` je typový konstruktor pro funkce z `r`
- `((->) r)` je funktor:

```
  -- :: (a -> b) ->  f    a  ->  f    b
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Funkce jako aplikativní funktor

- `((->) r)` je typový konstruktor pro funkce z `r`
- `((->) r)` je funktor:

```
  -- :: (a -> b) ->  f    a  ->  f    b
fmap :: (a -> b) -> (r -> a) -> (r -> b)
fmap f ra = \k -> f (ra k)    -- === f . ra
```

Funkce jako aplikativní funktor

- `((->) r)` je typový konstruktor pro funkce z `r`
- `((->) r)` je funktor:

```
  -- :: (a -> b) ->  f    a  ->  f    b
fmap :: (a -> b) -> (r -> a) -> (r -> b)
fmap f ra = \k -> f (ra k)    -- === f . ra
```

- je `((->) r)` aplikativní funktor?

```
pure :: a -> (r -> a)
pure x =
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
rf <*> ra =
```

Funkce jako aplikativní funktor

- `((->) r)` je typový konstruktor pro funkce z `r`
- `((->) r)` je funktor:

```
  -- :: (a -> b) ->  f    a  ->  f    b
fmap :: (a -> b) -> (r -> a) -> (r -> b)
fmap f ra = \k -> f (ra k)     -- === f . ra
```

- je `((->) r)` aplikativní funktor?

```
pure :: a -> (r -> a)
pure x =  const x
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
rf <*> ra =
```

Funkce jako aplikativní funktor

- `((->) r)` je typový konstruktor pro funkce z `r`
- `((->) r)` je funktor:

```
  -- :: (a -> b) ->  f    a  ->  f    b
fmap :: (a -> b) -> (r -> a) -> (r -> b)
fmap f ra = \k -> f (ra k)     -- === f . ra
```

- je `((->) r)` aplikativní funktor?

```
pure :: a -> (r -> a)
pure x =  const x
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
rf <*> ra =  \k-> (rf k) (ra k)
```

Funkce jako aplikativní funktor

- `((->) r)` je typový konstruktor pro funkce z `r`
- `((->) r)` je funktor:

```
  -- :: (a -> b) ->  f    a  ->  f    b
fmap :: (a -> b) -> (r -> a) -> (r -> b)
fmap f ra = \k -> f (ra k)    -- === f . ra
```

- je `((->) r)` aplikativní funktor?

```
pure :: a -> (r -> a)
pure x =  const x
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
rf <*> ra =  \k-> (rf k) (ra k)
```

★ rozmyslete si, zda jsou splněna pravidla pro **`Applicative`**

Monáda čtenáře podruhé

```haskell
instance Monad ((->) r) where
       -- ::  m    a  -> (a ->  m    b ) ->  m    b
    (>>=) :: (r -> a) -> (a -> (r -> b)) -> (r -> b)
```

Monáda čtenáře podruhé

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

Monáda čtenáře podruhé

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

- Funkce ask, asks, local jde použít na obě implementace (**Reader** r, (**->**) r)
- Obě implementace jsou instance typové třídy **MonadReader**

Monáda písaře (Writer)

Ukecaný evaluátor

```haskell
evalV :: Expr -> Valuation -> (Int, [String])
evalV (Var v) val = (M.findWithDefault 0 v val, ["var " ++ v])
evalV (Const n) _ = (n, ["const " ++ show n])
evalV (Add l r) val = evalOpV (+) "add" l r val
evalV (Sub l r) val = evalOpV (-) "sub" l r val
evalV (Mul l r) val = evalOpV (*) "mul" l r val

evalOpV :: (Int -> Int -> Int) -> String -> Expr -> Expr -> Valuation ->
↪  (Int, [String])
evalOpV op n l r val = (res, logL ++ logR ++ log)
  where
    (resL, logL) = evalV l val
    (resR, logR) = evalV r val
    res = op resL resR
    log = [n ++ " " ++ show resL ++ " " ++ show resR ++ " -> " ++ show res]
```

Monáda písaře

- existuje monáda `Writer w`:
  ```
  newtype Writer w a = Writer {runWriter :: (a, w)}
  ```
- automaticky kombinuje vedlejší výstup typu `w` výpočtů s výsledky typu `a`
- do vedlejšího výstupu se dá zapsat pomocí funkce `tell :: w -> Writer w ()`
- nemůže číst z výstupu předchozích výpočtů; jen z podvýpočtů
- typické použití: logy
- hledejte v modulu `Control.Monad.Writer` balíku `mtl`

Monáda písaře

```
evalV' :: Expr -> Valuation -> Writer [String] Int
evalV' (Var v) val = tell ["var "++v] >> pure (M.findWithDefault 0 v val)
evalV' (Const n) _ = tell ["const " ++ show n] >> return n
evalV' (Add l r) val = evalOpV' (+) "add" l r val
evalV' (Sub l r) val = evalOpV' (-) "sub" l r val
evalV' (Mul l r) val = evalOpV' (*) "mul" l r val

evalOpV' :: (Int -> Int -> Int) -> String -> Expr -> Expr -> Valuation ->
↪  Writer [String] Int
evalOpV' op n l r val = do
    resL <- evalV' l val
    resR <- evalV' r val
    let res = op resL resR
    tell [n ++ " " ++ show resL ++ " " ++ show resR ++ " -> " ++ show res]
    return res
```

Modul Control.Monad.Writer

Funkce pro práci s **Writer** monádou:

- `tell :: w -> Writer w ()`
  pouze zapíše „do logu“
- `listen :: Writer w a -> Writer w (a, w)`
  zachytí (i zapíše) „log“ zadaného výpočtu
- `censor :: (w -> w) -> Writer w a -> Writer w a`
  před zápisem změní „log“ zadaného výpočtu.
- vše hledejte v modulu **Control.Monad.Writer** balíku `mtl`
- striktní (nelíná) verse: **Control.Monad.Writer.Strict**

Monáda čtenáře

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
```

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
fmap f (w, x) = (w, f x)
```

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
fmap f (w, x) = (w, f x)
```

- je `((,) w)` aplikativní funktor?

```
pure :: a -> (w, a)
pure x =


<*> :: (w, a -> b) -> (w, a) -> (w, b)
(w1, f) <*> (w2, x) =
```

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
fmap f (w, x) = (w, f x)
```

- je `((,) w)` aplikativní funktor?

```
pure :: a -> (w, a)
pure x =  ( :??: , x)
```

  - musíme umět zbůhdarma vyrobit hodnotu typu `w`

```
<*> :: (w, a -> b) -> (w, a) -> (w, b)
(w1, f) <*> (w2, x) =
```

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
fmap f (w, x) = (w, f x)
```

- je `((,) w)` aplikativní funktor?

```
pure :: a -> (w, a)
pure x =  ( :??: , x)
```

  - musíme umět zbůhdarma vyrobit hodnotu typu `w`

```
<*> :: (w, a -> b) -> (w, a) -> (w, b)
(w1, f) <*> (w2, x) =  ( :????: , f x)
```

  - musíme umět hodnoty typu `w` kombinovat

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
fmap f (w, x) = (w, f x)
```

- je `((,) w)` aplikativní funktor?

```
pure :: a -> (w, a)
pure x =  ( :??: , x)
```

  - musíme umět zbůhdarma vyrobit hodnotu typu `w`

```
<*> :: (w, a -> b) -> (w, a) -> (w, b)
(w1, f) <*> (w2, x) =  ( :????: , f x)
```

  - musíme umět hodnoty typu `w` kombinovat
  - nechť je typ `w` monoidem!

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
fmap f (w, x) = (w, f x)
```

- je `((,) w)` aplikativní funktor?

```
pure :: Monoid w => a -> (w, a)
pure x =  (mempty, x)
```

  - musíme umět zbůhdarma vyrobit hodnotu typu `w`

```
<*> :: Monoid w => (w, a -> b) -> (w, a) -> (w, b)
(w1, f) <*> (w2, x) =  (w1 <> w2, f x)
```

  - musíme umět hodnoty typu `w` kombinovat
  - nechť je typ `w` monoidem!

Dvojice jako aplikativní funktor

- `((,) w)` je typový konstruktor pro dvojice s první složkou typu `w`
- `((,) w)` je funktor:

```
fmap :: (a -> b) -> (w, a) -> (w, b)
fmap f (w, x) = (w, f x)
```

- je `((,) w)` aplikativní funktor?

```
pure :: Monoid w => a -> (w, a)
pure x =  (mempty, x)
```

  - musíme umět zbůhdarma vyrobit hodnotu typu `w`

```
<*> :: Monoid w => (w, a -> b) -> (w, a) -> (w, b)
(w1, f) <*> (w2, x) =  (w1 <> w2, f x)
```

  - musíme umět hodnoty typu `w` kombinovat
  - nechť je typ `w` monoidem!
  - ★ proč nestačí libovolný typ s nulární a binární operací?

Monáda písaře podruhé

```haskell
instance Monoid w => Monad ((,) w) where
    (>>=) :: ... => (w, a) -> (a -> (w, b)) -> (w, b)
```

Monáda písaře podruhé

```haskell
instance Monoid w => Monad ((,) w) where
    (>>=) :: ... => (w, a) -> (a -> (w, b)) -> (w, b)
    (w1, x) >>= f = let (w2, y) = f x in (w1 <> w2, y)
```

Monáda písaře podruhé

```
instance Monoid w => Monad ((,) w) where
    (>>=) :: ... => (w, a) -> (a -> (w, b)) -> (w, b)
    (w1, x) >>= f = let (w2, y) = f x in (w1 <> w2, y)
```

- Funkce `tell`, `listen`, `censor` jde použít na obě implementace (`Writer w`, `(,) w`)
- Obě implementace jsou instance typové třídy `MonadWriter`

Monáda stavu (State)

Omezení čtenářů a písařů

- **`Reader`** `r` neumí měnit kontext navazujícím výpočtům
- přes `local` umí měnit kontext vnořených výpočtů

Omezení čtenářů a písařů

- **`Reader`** `r` neumí měnit kontext navazujícím výpočtům
- přes `local` umí měnit kontext vnořených výpočtů
- **`Writer`** `w` neumí číst výstupy předchozích výpočtů
- přes `listen` umí číst výstupy vnořených výpočtů

Omezení čtenářů a písařů

- **`Reader`** `r` neumí měnit kontext navazujícím výpočtům
- přes `local` umí měnit kontext vnořených výpočtů
- **`Writer`** `w` neumí číst výstupy předchozích výpočtů
- přes `listen` umí číst výstupy vnořených výpočtů
- občas bychom chtěli sdílet napříč výpočty měnící se informaci

Omezení čtenářů a písařů

- **`Reader`** `r` neumí měnit kontext navazujícím výpočtům
- přes `local` umí měnit kontext vnořených výpočtů

- **`Writer`** `w` neumí číst výstupy předchozích výpočtů
- přes `listen` umí číst výstupy vnořených výpočtů

- občas bychom chtěli sdílet napříč výpočty měnící se informaci
- příklad: při prohledávání grafu je potřeba udržovat množinu navštívených vrcholů
- příklad: generátor pseudonáhodných čísel si udržuje sémě (*seed*), které se po každém vygenerovaném čísle změní

Monáda State s

```haskell
newtype State s a = State {runState :: s -> (a, s)}
```

- funkce čtoucí stav typu `s` a vracející hodnotu typu `a` a změněný stav
- v modulu `Control.Monad.State` (příp. `.Strict`[3]) z `mtl`
- Co znamenají typy `a -> State s b`, `b -> State s c`
- Jak se skládají takové funkce?

[3] pozor, striktní je pouze sekvencování, nikoli operace se stavem

Monáda State s

```haskell
newtype State s a = State {runState :: s -> (a, s)}
```

- funkce čtoucí stav typu `s` a vracející hodnotu typu `a` a změněný stav
- v modulu `Control.Monad.State` (příp. `.Strict`[3]) z `mtl`
- Co znamenají typy `a -> State s b`, `b -> State s c`
- Jak se skládají takové funkce?
  Výsledný stav první se předá jako iniciální stav druhé.

[3] pozor, striktní je pouze sekvencování, nikoli operace se stavem

Monáda State s

```
newtype State s a = State {runState :: s -> (a, s)}
```

- funkce čtoucí stav typu `s` a vracející hodnotu typu `a` a změněný stav
- v modulu `Control.Monad.State` (příp. `.Strict`[3]) z `mtl`
- Co znamenají typy `a -> State s b`, `b -> State s c`
- Jak se skládají takové funkce?
  Výsledný stav první se předá jako iniciální stav druhé.
- Instance `Functor`, `Applicative` a `Monad` jsou ponechány čtenáři jako cvičení.

[3] pozor, striktní je pouze sekvencování, nikoli operace se stavem

Funkce pro práci se stavem

- `get :: State s s` — přečte stav
- `gets :: (s -> a) -> State s a` — čtení přes „getter“
- `put :: s -> State s ()` — zapíše nový stav
- `modify :: (s -> s) -> State s ()` — změní stav
- `modify' :: (s -> s) -> State s ()` — změní stav striktně
- `evalState :: State s a -> s -> a` — zahodí konc. stav

★ balík `lens` nabízí funkce/operátory pro přístup ke stavu (např. vlnovka v operátorech nahrazena rovnítkem)
příklad: hra Pong napsaná pomocí `State` a `lens`

Funkce pro práci se stavem

Funkce `get`, `put`, `modify`, atd. jsou ve skutečnosti obecnější a jsou poskytované typovou třídou **`MonadState`**, jejíž je **`State`** instance.

Důvody a další instance **`MonadState`** se dozvíte v příštích přednáškách.

Příklad: průchod grafem

```haskell
dfs' :: Vertex -> [Edge] -> State (Set Vertex) [Vertex]
dfs' v edges = do
    modify' (Set.insert v)
    let succs = snd <$> filter ((== v).fst) edges
    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
```

Samostatné programování

Monáda čtenáře: funkce `local`

Uvažte rozšíření jazyka aritmetických výrazů o lokální definice, tj. `let x = val in expr` konstrukci. Tedy mějme datový typ

```
data Expr = Var String | Const Int
          | Add Expr Expr | Sub Expr Expr | Mul Expr Expr
          | LetIn String Int Expr -- variable, value, expr
          deriving (Show, Eq)
```

Rozšiřte evaluační funkci `eval`, která používala monádu **`Reader`**, tak, aby nová funkce `eval` podporovala lokální definice. Zkuste použít funkci `local`.

```
> eval (LetIn "y" 42 (Var "x" `Add` Var "y")) $ M.fromList [("x", 2)]
44
> eval (Add (LetIn "y" 42 (Var "x" `Add` Var "y")) (Var "y")) $ M.fromList
    [("x", 2), ("y", 10)]
54
```

Monáda písaře: hledání ve stromě

Mějme klasické definice binárních stromů a stromů libovolné arity:

```
data BinTree a = BinNode a (BinTree a) (BinTree a)
               | Empty deriving (Show, Eq)
data RoseTree a = RoseNode a [RoseTree a] deriving (Show, Eq)
```

Implementujte funkci
`containsBin :: Eq a => a -> BinTree a -> Writer [a] Bool` takovou, že `containsBin v t` rozhodne, jestli strom `t` obsahuje hodnotu `v`, a jako výstup písaře vrátí seznam hodnot všech vrcholů, které při hledání prošla.

Chtěli bychom funkci upravit tak, aby nevracela vrcholy, které prošla, ale jen jejich počet. Co všechno je potřeba upravit? (Hint: Zkuste najít vhodný monoid.)

Implementujte obdobnou funkci
`containsRose :: Eq a => a -> RoseTree a -> Writer [a] Bool` pro stromy libovolné arity.

Monáda State: číslování vrcholů stromu

Pomocí monády **State** napište funkci
`addIndices :: BinTree a -> BinTree (Integer, a)`, která každému uzlu přiřadí unikátní číslo. Na pořadí nezáleží.

Také můžete zkusit napsat obecnější variantu typu
`Enum i => BinTree a -> i -> BinTree (i, a)`, která kořeni přiřadí daný index a dalším uzlům postupně následníky.

Poté napište analogickou funkci
`addIndicesRose :: RoseTree a -> RoseTree (Integer, a)` pro stromy libovolné arity.