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

IB016 Seminář z funkcionálního programování

Mnoho autorů napříč věky

Fakulta informatiky, Masarykova univerzita

Jaro 2021

Připomenutí: monády

Monáda $\approx$ 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`
  `>>= flip lookup uid2uco`
  `>>= flip lookup uco2name` $\rightsquigarrow^*$ `Just "Adam"`
- zobecnění: `Either e` – „selhání s odůvodněním“

`[]`: výpočet s více možnými výsledky

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

Dvojice jako aplikativní funktor

- `((,) w)` je typový k-tor 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

```
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)
```

- výpočet, který navíc produkuje výstup typu `w`
- „boční“ výstupy všech výpočtů se kumulují operátorem `<>`
- samotný výsledek je ve druhé složce dvojice
- typické použití: logy

Příklad: monáda písaře

```haskell
loudNeg :: Int -> ([String], Int)
loudNeg n = (["neg " ++ show n], negate n)

loudPlus, loudMinus :: Int -> Int -> ([String], Int)
loudPlus n m = ([show n ++ " + " ++ show m], n + m)
loudMinus n m = loudNeg m >>= loudPlus n

loudTimes :: Int -> Int -> ([String], Int)
loudTimes 0 _ = pure 0
loudTimes n m
    | n > 0 = do n' <- loudMinus n 1
                 m' <- loudTimes n' m
                 loudPlus m' m
    | n < 0 = loudNeg n >>= flip loudTimes m >>= loudNeg

loudTimes (-2) 3 ⇝* (["neg -2","neg 1","2 + -1","neg 1",
↪  "1 + -1","0 + 3","3 + 3","neg 6"],-6)
```

Modul Control.Monad.Writer

- existuje také „explicitně pojmenovaná“ monáda `Writer w`:

  ```
  newtype Writer w a = Writer {runWriter :: (a, w)}
  ```

  (pozor, prvky dvojice jsou naopak)
- uvnitř obou písařů lze použít pomocné funkce:[1]

  ```
  tell :: w -> m ()     -- tedy w -> (w, ())
  ```

  pouze zapíše „do logu“

  ```
  listen :: m a -> m (a, w)
  ```

  zachytí (i zapíše) „log“ zadaného výpočtu

  ```
  censor :: (w -> w) -> m a -> m a
  ```

  před zápisem změní „log“ zadaného výpočtu.
- vše hledejte v modulu `Control.Monad.Writer` balíku `mtl`
- striktní (nelíná) verse: `Control.Monad.Writer.Strict`

[1] je to zajištěno typovou třídou `MonadWriter` sdružující písaře všeho druhu

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

```
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
```

- výpočet, který čte kontext typu `r`
- všechny komponované výpočty dostanou týž kontext
- typické použití: konfigurace

Příklad: monáda čtenáře

```haskell
import Data.Map (Map)
import qualified Data.Map as Map
import Control.Applicative (liftA2)

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

type Valuation = Map String Bool
eval :: Formula -> (Valuation -> Bool)
eval (Var v)   = Map.findWithDefault False v
eval (And x y) = do lhs <- eval x
                    if not lhs then pure False
                               else eval y
eval (Or  x y) = liftA2 (||) (eval x) (eval y)
eval (Not x)   = not <$> eval x
```

Modul Control.Monad.Reader

- existuje také „explicitně pojmenovaná“ monáda `Reader r`:

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

- uvnitř obou čtenářů lze použít pomocné funkce:[2]

  ```
  ask :: m r      -- r -> r
  ```
  přečte kontext (např. `do {ctx <- ask; ...}`)

  ```
  asks :: (r -> a) -> m a
  ```
  čtení přes „getter“ (např. `do {x <- asks fst; ...}`)

  ```
  local :: (r -> r) ->  m    a  ->  m    a
  --"-- :: (r -> r) -> (r -> a) -> (r -> a)
  ```
  spustí výpočet s lokálně změněným kontextem

- vše hledejte v modulu `Control.Monad.Reader` balíku `mtl`

[2] Opět je to zajištěno typovou třídou `MonadReader`, jíž jsou `Reader r` i `((->) r)` instancemi.

Omezení čtenářů a písařů

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

```
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`

Příklad: průchod grafem

```haskell
type Edge = (Vertex, Vertex)
type Stamp = (Vertex, Int, Int)

dfs :: Vertex -> [Edge] -> State (Int, Set Vertex) [Stamp]
dfs v es = do
    _2 %= Set.insert v
    tDisc <- gets fst
    _1 += 1
    let succs = snd <$> filter ((== v).fst) es
    succStamps <- forM succs $ \s -> do
        visited <- gets snd
        if s `elem` visited then return []
                            else dfs s es
    tFin <- gets fst
    _1 += 1
    return $ (v, tDisc, tFin) : concat succStamps

dfs' :: Vertex -> [Edge] -> [Stamp]
dfs' v es = evalState (dfs v es) (1, Set.empty)
```