Monády

IB016 Seminář z funkcionálního programování

Vypráví Adam Matoušek

Fakulta informatiky, Masarykova univerzita

Jaro 2022

Připomenutí: Maybe, Either, druhy

`Maybe a`: datový typ rozšířený o jednu hodnotu

- `data Maybe a = Nothing | Just a`
- často: `Nothing` je selhání

`Either a b`: sjednocení dvou datových typů

- `data Either a b = Left a | Right b`
- často: specifikace chyby, nebo korektní výsledek

Druhy: „typování typů“

- `Maybe Int :: *`
- `Maybe :: * -> *`
- `Either :: * -> * -> *`

Připomenutí: Functor

Motivace: Zobecňování funkce `map`

```
    map :: (a -> b) -> [a]        -> [b]
treeMap :: (a -> b) -> BinTree a -> BinTree b
        ....
```

Typová třída `Functor`:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
- instance pro `[]`, `Maybe`, `IO`, `Either e`, `(r ->)`, `(w, )`

Monády

Monády

Co je to monáda?

- ~~zastaralý výraz pro prvoka~~
- ~~termín z teorie kategorií~~
- funktor, ke kterému přidáme ještě nějaké operace kromě `fmap`
- abstrakce výpočtů a jejich řetězení

Navrhovaná strategie pochopení monád:

- dívat se na různé funktory „výpočtovým pohledem“
- zkoumat u každého **zvlášť**, co znamená „řetězit výpočet“
- abstrakce vyplyne časem sama

Výpočty a jejich výsledky

Výpočet ≈ hodnota ≈ výsledek vyhodnocení ⇒ výsledek výpočtu

```
x :: Int
x = 19
```
, ale i
```
x = succ $ succ (2 * 4) * 2
```

```
y :: Maybe String
y = Just "adamat"
```
nebo `y = Nothing`, ale i
```
y = lookup 445763 (getStudents db)
```

(Výsledkem výpočtu je `"adamat"`, nebo žádný neexistuje.)

Maybe jako výpočet

`Maybe` a je výpočet s možností selhání (parciální funkce):

- $\rightsquigarrow^*$ `Just 42` výsledkem je 42
- $\rightsquigarrow^*$ `Nothing` výsledek není, výpočet selhal

Jak kombinovat výpočty? Máme `(!!)` a chceme...

- ...vybrat `2`. prvek z `[10, 20, 30]`.
- ...vybrat `2`. prvek z `Just [10, 20, 30]`.
- ...vybrat `Just 2`. prvek z `[10, 20, 30]`.
- ...vybrat `Just 2`. prvek z `Just [10, 20, 30]`.
- ...vybrat `Nothing`tý prvek z `Just [10, 20, 30]`.
- ...vybrat `Just 2`. prvek z `Nothing`.
- ★ ...vybrat `Just 2`. prvek z `Just [1]`.

Kombinace výpočtů v Maybe

```
liftM2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

Kombinace výpočtů pomocí (čisté[1]) funkce:

- provedeme výpočet „operandů“ (podvýpočty),
- pokud některý selže, i kombinace selže (`Nothing`),
- jinak se výsledkem (v `Just`) kombinace podvýsledků.

Kombinací dostaneme zase výpočet (tj. něco v `Maybe`).

Příklad:

```
liftA2 (!!) (Just [10, 20, 30]) (Just 2) ⇝* Just 30
```

[1] ve smyslu „není sama monadickým výpočtem“[2]

[2] konkrétně zde „nebalí výsledek do Maybe“

Řetězení výpočtů v Maybe

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Řetězení výpočtů:

- provedeme výpočet mezivýsledku,
- pokud se povedl (`Just x`), pokračujeme dál s hodnotou `x`
- pokud selhal (`Nothing`), nepokračujeme ($\rightsquigarrow^*$ `Nothing`)

Příklad:

```
Just [] >>= \x -> if null x then Nothing else Just (head x)
```

$\rightsquigarrow^*$ `Nothing`

Triviální výpočet

- Na totální funkci můžeme nahlížet jako na parciální.
- S neselhávajícím výpočtem můžeme pracovat, jako by selhání umožňoval.
- Pro hodnotu umíme vyrobit výpočet, který ji prostě vrací.
- ```
  pure :: a -> Maybe a
  pure x = Just x
  ```
  ```

Maybe je monáda

1. Umíme řetězit výpočty:
   ```
   (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
   ```
2. Umíme vyrobit triviální výpočet hodnoty:
   ```
   pure :: a -> Maybe a
   ```

= podstata monády[3]

[3] obě funkce ještě musí splňovat nějaká pravidla, uvidíme později

Seznam = nedeterministické výpočty

Chceme používat funkce, které mohou vracet seznam více hodnot stejného typu. Například:

- chceme všechny komplexní druhé/třetí odmocniny čísla
- máme různé varianty n-té otázky v odpovědníku
- získáváme obsah adresáře
- volíme směr v bludišti

S každým mezivýsledkem má smysl zkusit pokračovat. Řetězením takových funkcí tak můžeme například:

- zjistit všechny komplexní šesté odmocniny čísla
- generovat různé varianty celého odpovědníku
- rekurzivně získat všechny soubory v domovském adresáři
- backtrackingem najít východ z bludiště

Seznam = nedeterministické výpočty

Příklad:

- máme k dispozici funkce

```
-- 2. odmocniny
root2 :: Complex Float -> [Complex Float]
-- 3. odmocniny
root3 :: Complex Float -> [Complex Float]
```

- chceme funkci `root6` vracející všechny 6. odmocniny

```
root6 :: Complex Float -> [Complex Float]
root6  c = concat $ map root3 (root2 c)
```

Seznam = nedeterministické výpočty

Jak vypadá řetězení a `pure` pro nedeterministické výpočty?

```
(>>=) :: [a] -> (a -> [b]) -> [b]
list >>= f = concat $ map f list

pure :: a -> [a]
pure x = [x]
```

Pozorování: `[]` $\approx$ `Nothing` (neexistuje žádný výsledek)

Výpočet s náhodou

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na `seed` → potřebujeme `seed` propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici globální stavové proměnné
- nutné předávat jako argument v celém výpočtu
- náhodné hodnoty můžeme reprezentovat jako funkce:
  - vstup je hodnota `seed`
  - výstup:
    - „náhodná“ hodnota závislá na `seed`
    - nová hodnota `seed'` pro další „náhodnou“ proměnnou
  - **Typ:** `Int -> (a, Int)`

Výpočet s náhodou

Pro přehlednost tento typ pojmenujme

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

Opět chceme řetězit výpočty.
Příklad:

- `rollDie :: Rand Int` vrátí náhodné číslo z intervalu $[1, 6]$
- ```
  roll2Dice :: Rand Int
  roll2Dice = Rand $ \seed ->
      let
          (d1, seed')  = runRand rollDie seed
          (d2, seed'') = runRand rollDie seed'
      in
          (d1 + d2, seed'')
  ```
  ```

Výpočet s náhodou

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

- Princip řetězení náhodných výpočtů zobecníme.
- Levá strana bere `seed` → vytváří nový `seed'` → použije se jako vstup pro náhodné číslo na pravé straně.

```
(>>=) :: Rand a -> (a -> Rand b) -> Rand b
x >>= f = Rand $ \seed ->
    let (val, seed') = runRand x seed
    in runRand (f val) seed'

pure :: a -> Rand a
pure x = Rand $ \seed -> (x, seed)
```

Výpočet s náhodou

Řešení `roll2Dice`:

```
roll2Dice :: Rand Int
roll2Dice =
    rollDie >>= (\d1 ->
    rollDie >>= (\d2 ->
    pure (d1 + d2)))
```

Rand jako monáda s vnitřním stavem

Obecněji: monády s vnitřním stavem.

- `Rand` — stav je sémě generátoru
- `Turtle` — poloha a směr želvy
- `Parser` — stav obsahuje nepřečtenou část textu
- `State s` — vlastní libovolný stav typu `s`
- ★ `IO` — stav je kouzelný[4]

(Některé uvidíme později během semestru.)

[4] Pro zájemce: `https://wiki.haskell.org/IO_inside`

Typová třída Monad

Typová třída Monad

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b   -- čti „bind"
    return :: a -> m a                  -- = pure

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
```

- umožňuje řetězení výpočtů
  - předem definujeme, jak se toto řetězení chová
  - `bind` navenek pracuje s výsledkem = „rozbalenou“ hodnotou
- definice třídy v `Prelude`, více v `Control.Monad`
- prozatím si nevšímejme nadtřídy `Applicative`...

Instance třídy Monad I

```haskell
instance Monad Maybe where
    return :: a -> Maybe a
    return = Just

    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= f = Nothing
    Just x  >>= f = f x
```

- „Parciální“ funkce (výpočet může selhat).
- `>>=`: výsledek se předává, pokud existuje.
- jak vypadá instance pro `Either e`?

Instance třídy Monad II

```
instance Monad [] where
    return :: a -> [a]
    return x = [x]

    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)
```

- „Nedeterministické“ funkce/výpočty
  (`a -> [b]` = jeden vstup, libovolný počet výsledků).
- `>>=`: výpočet probíhá nad každou hodnotou vstupu

Instance třídy Monad III

```
instance Monad ([l], _) where – dvojice se seznamem
    return :: a -> ([l], a)
    return x = ([], x)

    (>>=) :: ([l], a) -> (a -> ([l], b)) -> ([l], b)
    (list, x) >>= f = let (list', y) = f x
                      in (list ++ list', y)
```

- Výpočty s „logováním“.
- `>>=`: udržuje se společný log celého výpočtu.
- Obecněji: monáda písaře. Uvidíme později v semestru.

Instance třídy Monad IV

```
instance Monad (r -> _) where -- fukce z r
    return :: a -> (r -> a)
    return x _ = x

    (>>=) :: (r -> a) -> (a -> (r -> b)) -> (r -> b)
    (>>=) rx f ctx = let x  = rx ctx
                         ry = f x
                     in ry ctx
```

- Výpočty s read-only kontextem (např. konfigurace).
- >>=: všem výpočtům předá týž kontext
- Monáda čtenáře. Uvidíme ještě později v semestru.

Pravidla pro třídu Monad

Korektní instance monády musí splňovat:

- *levá identita*

  ```
  return x >>= f ≡ f x
  ```

- *pravá identita*

  ```
  mx >>= return ≡ mx
  ```

- *asociativita*

  ```
  (mx >>= f) >>= g ≡ mx >>= (\x -> f x >>= g)
  ```

Užitečné funkce třídy Monad

```
(<=<) :: Monad m =>
         (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

- ekvivalent kompozice běžných funkcí `(.)`
- existuje i obrácená varianta `>=>`

```
join :: Monad m => m (m a) -> m a
join mmx = mmx >>= id
```

- slučování vnořených kontextů
- ★ teoretici často místo `bind`u definují `join`

Notace „do“

`do`-notace = syntaktický cukr pro `>>=` u všech monád.

- blok, začíná slovem `do`, řádky zarovnány (jako ve `where`)
- lze spouštět výpočty a (nepovinně) brát jejich výsledky
- lze použít `let`, u někt. monád vzory ve výsledcích
- výsledek `do`-bloku je výsledek posledního výpočtu v něm

```
foo :: NějakáMonáda Int
foo = do x <- gib "x"          --   gib "x" >>= \x ->
         let y = abs x         --   let y = abs x in
         z <- compute x        --   compute x >>= \z ->
         pure (y + z)          --   pure (y + z)
```

Příklad: do-blok a IO

```
cat :: String -> IO ()
cat delim = do
    l <- getLine
    if l == delim
        then pure ()
        else do putStrLn l
                cat delim
```

Doporučujeme projít si nepovinné 7. cvičení Sbírky IB015 o `IO`.

Typová třída Applicative

Motivace pro Applicative

Aplikativní funktory jsou na půli cesty mezi `Functor` a `Monad`.

- Umožňují z hodnoty vyrobit triviální výpočet:
  - `pure :: Applicative f => a -> f a`
  - doslova totéž co `return`, ale Historické Důvody™
- Umožňují kombinovat výsledky nezávislých výpočtů:
  - `liftA2 ::...=> (a -> b -> c) -> f a -> f b -> f c`
  - `liftA3` obdobně pro ternární
  - `<*> ::...=> f (a -> b) -> f a -> f b`
  - Nelze měnit *efekt* (`f`) na základě *výsledků* (`a`, `b`). Např. `liftA2 op (Just 1) (Just 2)` neumí vrátit `Nothing`.

Existují aplikativní funktory, které nejsou monádami (`ZipList`).

Typová třída Applicative

```
class Functor f => Applicative f where
    pure   :: a -> f a
    (<*>)  :: f (a -> b) -> f a -> f b
    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

- samotná třída v `Prelude`, více v `Control.Applicative`
- předepsaných funkcí je více, ale dají se odvodit
- stačí jedno z `(<*>)` a `liftA2`
- lze napsat instanci `Monad` a použít `liftA2 = liftM2`
- naopak `return` se automaticky zadefinuje jako `pure`

Applicative vs. Monad

Z Historických Důvodů™ je spousta „aplikativních“ věcí v `Monad`:

- `pure` ≡ `return`
- `(<*>)` ≡ `ap`
- `(*>)` ≡ `(>>)`
- `liftA` ≡ `liftM` (≡ `fmap`)
- `liftA2` ≡ `liftM2`
- `liftA3` ≡ `liftM3`

`Applicative` má navíc `(<*)`, `Monad` zase až `liftM5`.

Libovolně-ární liftování: `f <$> mx <*> my <*> mz <*> ...`

Pravidla pro třídu Applicative

Korektní instance `Applicative` musí splňovat:

- *identita*
  `(pure id) <*> x ≡ x`
- *kompozice*
  `(pure (.)) <*> f <*> g <*> x ≡ f <*> (g <*> x)`
- *homomorfismus*
  `(pure f) <*> (pure x) ≡ pure (f x)`
- *výměna*
  `u <*> (pure y) ≡ (pure ($ y)) <*> u`

Vsuvka: MonadFail

```
class Monad m => MonadFail m where
    fail :: String -> m a
```

- akce `fail` má přerušit probíhající výpočet
- historicky (ještě v GHC 8.6) součástí třídy `Monad`
- automaticky se používá při pattern-matchingu v `do`-bloku
  - `do (x:xs) <- lookup key tableOfLists`
- instance pro `[]`, `Maybe` a `IO`

Funktory a monády – shrnutí

Functor:

- `fmap :: (a -> b) -> f a -> f b`:
- aplikace funkce na výsledek výpočtu

Applicative:

- `pure :: a -> f a`
- triviální výpočet se zadaným výsledkem
- `liftA2 :: (a -> b -> c) -> f a -> f b -> f c`
- kombinace výsledků nezávislých výpočtů

Monad:

- `(>>=) :: m a -> (a -> m b) -> m b`
- výpočet může záviset na výsledku předchozího výpočtu
- `join :: m (m a) -> m a`
- lze spustit výpočet, který je výsledkem výpočtu

Funktory a monády – „krabičkový pohled“

Functor:

- **někdy** odpovídá kontejnerům a krabičkám
- hodnoty uvnitř lze měnit bez změny struktury kontejneru

Applicative:

- umíme vyrobit „singleton“ s „neutrální strukturou“
- umíme zkombinovat dvě nezávislé krabičky:
  - jak kombinovat hodnoty uvnitř, říká dodaná čistá funkce
  - jak sloučit obaly (strukturu), říká instance

Monad:

- ??? . . . ne!
- `join` $\rightarrow$ umíme sloučit vnořené krabičky