Typové třídy Functor, Applicative, Monad

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2015

Typový konstruktor Either

```
data Either a b = Left a
                | Right b
                deriving (Eq, Ord, Read, Show)
```

- používá se, když může mít výpočet dva typy výsledků
- často se používá jako rozšíření `Maybe`
  - `Left a` označuje chybný výpočet, hodnota specifikuje chybu
  - `Right b` označuje korektní výpočet, hodnota je výsledkem

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
Maybe :: * -> *
Either :: * -> * -> *
[] :: * -> *
(,) :: * -> * -> *
```

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
Maybe :: * -> *
Either :: * -> * -> *
[] :: * -> *
(,) :: * -> * -> *
```

- opět platí princip částečné aplikace

```
Either String :: * -> *
```

- GHCi definuje povel `:k` na určení druhu

Motivace: map

- Funkce `map` na seznamech:

```
data [a] = [] | a : [a]

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = fx : map f xs
```

Motivace: map

- Funkce `map` na seznamech:

```
data [a] = [] | a : [a]

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = fx : map f xs
```

- Funkce `map` na binárních stromech:

```
data BinTree a = BNode a (BinTree a) (BinTree a)
               | BEmpty

treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ BEmpty = BEmpty
treeMap f (BNode v l r) =
    BNode (f v) (treeMap f l) (treeMap f r)
```

Motivace: map

- Funkce `map` na seznamech:

```
data [a] = [] | a : [a]

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = fx : map f xs
```

- Funkce `map` na binárních stromech:

```
data BinTree a = BNode a (BinTree a) (BinTree a)
               | BEmpty

treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ BEmpty = BEmpty
treeMap f (BNode v l r) =
    BNode (f v) (treeMap f l) (treeMap f r)
```

- Nedalo by se to nějak zobecnit?

Typová třída Functor

Zaveďme typovou třídu `Functor`:

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

- definováno v modulu Data.Functor
- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
  - instance pro `[]`, `BinTree`, `Maybe`
  - ne pro konkrétní typy (`[String]`, `BinTree a`, Maybe Int)

Typová třída Functor

Zaveďme typovou třídu `Functor`:

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

- definováno v modulu Data.Functor
- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
  - instance pro `[]`, `BinTree`, `Maybe`
  - ne pro konkrétní typy (`[String]`, `BinTree a`, Maybe Int)
- jiný pohled: funktory tvoří kontext/kontejner pro typy (obalují je další strukturou)

Instance třídy Functor I.

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b]
      fmap = map
  ```

- ```
  instance Functor BinTree where
      fmap :: (a -> b) -> BinTree a -> BinTree b
      fmap = treeMap
  ```

Instance třídy Functor I.

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b]
      fmap = map
  ```

- ```
  instance Functor BinTree where
      fmap :: (a -> b) -> BinTree a -> BinTree b
      fmap = treeMap
  ```

- ```
  instance Functor Maybe where
      fmap :: (a -> b) -> Maybe a -> Maybe b
      fmap f (Just x) = Just (f x)
      fmap f Nothing = Nothing
  ```
  ```

Instance třídy Functor II.

```
instance Functor IO where
    fmap :: (a -> b) -> IO a -> IO b
    fmap f action = do
        result <- action
        return (f result)
```

Instance třídy Functor II.

- ```
  instance Functor IO where
      fmap :: (a -> b) -> IO a -> IO b
      fmap f action = do
          result <- action
          return (f result)
  ```

- `Either` je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci na jeden argument.

  ```
  Either e :: * -> *
  ```
  ```

Instance třídy Functor II.

- ```
  instance Functor IO where
      fmap :: (a -> b) -> IO a -> IO b
      fmap f action = do
          result <- action
          return (f result)
  ```

- `Either` je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci na jeden argument.

  ```
  Either e :: * -> *

  instance Functor (Either e) where
      fmap :: (a -> b) -> Either e a -> Either e b
      fmap f (Right x) = Right (f x)
      fmap f (Left x) = Left x
  ```
  ```

Instance třídy Functor III.

Funkce je vlastně použití binárního typového konstruktoru `(->)`

```
(->) :: * -> * -> *
```

Instance třídy Functor III.

Funkce je vlastně použití binárního typového konstruktoru `(->)`

```
(->) :: * -> * -> *
```

Můžeme tedy zadefinovat instanci pro její částečnou aplikaci na jeden argument.
`(->) r :: * -> *` (tedy „funkce z `r`“)

Instance třídy Functor III.

Funkce je vlastně použití binárního typového konstruktoru `(->)`

```
(->) :: * -> * -> *
```

Můžeme tedy zadefinovat instanci pro její částečnou aplikaci na jeden argument.
`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
    fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap f g = (\x -> f (g x))
```

Pravidla pro třídu Functor

Pro instance třídy `Functor` by mělo platit:

- `fmap id` $\equiv$ `id`
- `fmap (f . g)` $\equiv$ `fmap f . fmap g`

Pravidla pro třídu Functor

Pro instance třídy `Functor` by mělo platit:

- `fmap id` $\equiv$ `id`
- `fmap (f . g)` $\equiv$ `fmap f . fmap g`

Pravidla musí platit!

- kompilátor se spoléhá na výše uvedená pravidla
- jejich platnost musí ověřit programátor (!)
- pro všechny knihovní instance platí

Motivace: Co s funkcemi?

Jak aplikovat funkci na hodnotu s kontextem?

```
(+5) $ Just 2 ⇝ Just 7
(+5) $ Nothing ⇝ Nothing
```

Motivace: Co s funkcemi?

Jak aplikovat funkci na hodnotu s kontextem?

```
(+5) $ Just 2 ⇝ Just 7
(+5) $ Nothing ⇝ Nothing
```

- `fmap :: Functor f => (a -> b) -> f a -> f b`

Motivace: Co s funkcemi?

Jak aplikovat funkci na hodnotu s kontextem?

```
(+5) $ Just 2 ⇝ Just 7
(+5) $ Nothing ⇝ Nothing
```

- `fmap :: Functor f => (a -> b) -> f a -> f b`

Jak aplikovat funkci v kontextu na jinou hodnotu v kontextu?

```
Just (+5) `apply` Just 2 ⇝ Just 7
Just (+5) `apply` Nothing ⇝ Nothing
Nothing `apply` Just 2 ⇝ Nothing
```

Motivace: Co s funkcemi?

Jak aplikovat funkci na hodnotu s kontextem?

```
(+5) $ Just 2 ⇝ Just 7
(+5) $ Nothing ⇝ Nothing
```

- `fmap :: Functor f => (a -> b) -> f a -> f b`

Jak aplikovat funkci v kontextu na jinou hodnotu v kontextu?

```
Just (+5) `apply` Just 2 ⇝ Just 7
Just (+5) `apply` Nothing ⇝ Nothing
Nothing `apply` Just 2 ⇝ Nothing

fmap (+) (Just 5) `apply` Just 2 ⇝ Just 7
```

Motivace: Co s funkcemi?

Jak aplikovat funkci na hodnotu s kontextem?

```
(+5) $ Just 2 ⇝ Just 7
(+5) $ Nothing ⇝ Nothing
```

- `fmap :: Functor f => (a -> b) -> f a -> f b`

Jak aplikovat funkci v kontextu na jinou hodnotu v kontextu?

```
Just (+5) `apply` Just 2 ⇝ Just 7
Just (+5) `apply` Nothing ⇝ Nothing
Nothing `apply` Just 2 ⇝ Nothing

fmap (+) (Just 5) `apply` Just 2 ⇝ Just 7
```

- `apply :: Functor f => f (a -> b) -> f a -> f b`

Typová třída Applicative

Zaveďme typovou třídu `Applicative`:

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

- definováno v modulu Control.Applicative
- rozšíření třídy `Functor` pro pohodlnou práci s funkcemi v kontextu
- funkce `pure` „obalí“ hodnotu minimální strukturou („přidá nejjednodušší kontext“)

Typová třída Applicative

Zaveďme typovou třídu `Applicative`:

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

- definováno v modulu Control.Applicative
- rozšíření třídy `Functor` pro pohodlnou práci s funkcemi v kontextu
- funkce `pure` „obalí“ hodnotu minimální strukturou („přidá nejjednodušší kontext“)
- `Applicative` definuje infixové synonymum pro `fmap`

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

Instance třídy Applicative I.

```
instance Applicative Maybe where
    pure :: a -> Maybe a
    pure = Just

    (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

Instance třídy Applicative I.

- ```
  instance Applicative Maybe where
      pure :: a -> Maybe a
      pure = Just

      (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
      Nothing <*> _ = Nothing
      (Just f) <*> something = fmap f something
  ```

- Co pro seznamy? Aplikujme každou funkci na každý seznam.

  ```
  instance Applicative [] where
      pure :: a -> [a]
      pure x = [x]

      (<*>) :: [a -> b] -> [a] -> [b]
      fs <*> xs = [f x | f <- fs, x <- xs]
  ```
  ```

Instance třídy Applicative II.

```
instance Applicative IO where
    pure :: a -> IO a
    pure = return

    (<*>) :: IO (a -> b) -> IO a -> IO b
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Instance třídy Applicative III.

```
instance Applicative (Either e) where
    pure :: a -> Either e a
    pure = Right

    (<*>) :: Either e (a -> b)
          -> Either e a -> Either e b
    Left e <*> _ = Left e
    Right f <*> r = fmap f r
```

Instance třídy Applicative III.

- ```
  instance Applicative (Either e) where
      pure :: a -> Either e a
      pure = Right

      (<*>) :: Either e (a -> b)
            -> Either e a -> Either e b
      Left e <*> _ = Left e
      Right f <*> r = fmap f r
  ```

- ```
  instance Applicative ((->) r) where
      pure :: a -> (r -> a)
      pure x = (\_ -> x)

      (<*>) :: (r -> (a -> b)) ->
                  (r -> a) -> (r -> b)
      f <*> g = \x -> f x (g x)
  ```

ZipList I.

Seznamy můžou mít i jinou instanci v `Applicative`:

- aplikuje se prví funkce jenom na první hodnotu, druhá na druhou, ...

ZipList I.

Seznamy můžou mít i jinou instanci v `Applicative`:

- aplikuje se prví funkce jenom na první hodnotu, druhá na druhou, ...

Jelikož však nemůžou existovat 2 různé instance pro tentýž typ, musíme si zavíst nový typ pro seznamy, tzv. `ZipList`:

```
newtype ZipList a = ZipList { getZipList :: [a] }
    deriving (Show, Eq, Ord, Read)
```

ZipList II.

```
instance Functor ZipList where
    fmap :: (a -> b) -> ZipList a -> ZipList b
    fmap f (ZipList xs) = ZipList (map f xs)
```

ZipList II.

```
instance Functor ZipList where
    fmap :: (a -> b) -> ZipList a -> ZipList b
    fmap f (ZipList xs) = ZipList (map f xs)

instance Applicative ZipList where
    pure :: a -> ZipList a
    pure x = ZipList (repeat x)

    (<*>) :: ZipList (a -> b)
          -> ZipList a -> ZipList b
    ZipList fs <*> ZipList xs =
        ZipList (zipWith (\f x -> f x) fs xs)
```

Pravidla pro třídu Applicative

- Identita

  `pure id <*> v` $\equiv$ `v`
- Kompozice

  `pure (.) <*> u <*> v <*> w` $\equiv$ `u <*> (v <*> w)`
- Homomorfizmus

  `pure f <*> pure x` $\equiv$ `pure (f x)`
- Výměna

  `u <*> pure y` $\equiv$ `pure ($ y) <*> u`

Pravidla pro třídu Applicative

- Identita
  `pure id <*> v` $\equiv$ `v`
- Kompozice
  `pure (.) <*> u <*> v <*> w` $\equiv$ `u <*> (v <*> w)`
- Homomorfizmus
  `pure f <*> pure x` $\equiv$ `pure (f x)`
- Výměna
  `u <*> pure y` $\equiv$ `pure ($ y) <*> u`

Jako důsledek bude pro instanci platit také

- `pure f <*> x = fmap f x`

Aplikace běžných funkcí na hodnoty v kontextu

Definice funkcí `liftA` pro zjednodušení práce s třídou `Applicative`:

- `fmap (+) (Just 5) <*> (Just 2)` $\rightsquigarrow$ `Just 7`

Aplikace běžných funkcí na hodnoty v kontextu

Definice funkcí `liftA` pro zjednodušení práce s třídou `Applicative`:

- `fmap (+) (Just 5) <*> (Just 2)` $\rightsquigarrow$ `Just 7`
- `(+) <$> (Just 5) <*> (Just 2)` $\rightsquigarrow$ `Just 7`

Aplikace běžných funkcí na hodnoty v kontextu

Definice funkcí `liftA` pro zjednodušení práce s třídou `Applicative`:

- `fmap (+) (Just 5) <*> (Just 2)` ⇝ `Just 7`
- `(+) <$> (Just 5) <*> (Just 2)` ⇝ `Just 7`
- `liftA2 (+) (Just 5) (Just 2)` ⇝ `Just 7`

Aplikace běžných funkcí na hodnoty v kontextu

Definice funkcí `liftA` pro zjednodušení práce s třídou `Applicative`:

- `fmap (+) (Just 5) <*> (Just 2)` $\rightsquigarrow$ `Just 7`
- `(+) <$> (Just 5) <*> (Just 2)` $\rightsquigarrow$ `Just 7`
- `liftA2 (+) (Just 5) (Just 2)` $\rightsquigarrow$ `Just 7`

```
liftA2 :: Applicative f =>
            (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

Existují i varianty pro další arity, viz dokumentace.

Motivace: Co s funkcemi vytvářejícími kontext?

Jak aplikovat funkci na hodnotu v kontextu?

- `fmap :: Functor f => (a -> b) -> f a -> f b`

Motivace: Co s funkcemi vytvářejícími kontext?

Jak aplikovat funkci na hodnotu v kontextu?

- `fmap :: Functor f => (a -> b) -> f a -> f b`

Jak aplikovat funkci v kontextu na hodnotu v kontextu?

- ```
  (<*>) :: Applicative f =>
           f (a -> b) -> f a -> f b
  ```
  ```

Motivace: Co s funkcemi vytvářejícími kontext?

Jak aplikovat funkci na hodnotu v kontextu?

- `fmap :: Functor f => (a -> b) -> f a -> f b`

Jak aplikovat funkci v kontextu na hodnotu v kontextu?

- ```
  (<*>) :: Applicative f =>
           f (a -> b) -> f a -> f b
  ```

Jak aplikovat funkci produkující hodnotu v kontextu na hodnotu v kontextu?

- ```
  apply2 :: Applicative f =>
            f a -> (a -> f b) -> f b
  ```

Typová třída Monad

Zaveďme typovou třídu `Monad`:

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

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
    fail :: String -> m a
    fail msg = error msg
```

- definováno v modulu Control.Monad
- logické rozšíření třídy Applicative pro pohodlnou práci s funkcemi vytvářejícími kontext
- Applicative je přerekvizitou Monad od standardu Haskell 2010
- funkce `fail` se volá například pri selhání vzoru v **do**-notaci

Instance třídy Monad I.

```
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

    fail :: String -> Maybe a
    fail _ = Nothing
```

Instance třídy Monad II.

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

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

    fail :: String -> [a]
    fail _ = []
```

Pravidla pro třídu Monad

- Levá identita

  `return x >>= f` $\equiv$ `f x`
- Pravá identita

  `m >>= return` $\equiv$ `m`
- Asociativita

  `(m >>= f) >>= g` $\equiv$ `m >>= (\x -> f x >>= g)`

Kompozice funkcí vytvářejících kontext

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

- ekvivalent kompozice běžných funkcí `(.)`

Kompozice funkcí vytvářejících kontext

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

- ekvivalent kompozice běžných funkcí `(.)`
- možnost reformulovat pravidla třídy Monad pomocí kompozice
  - Levá identita
    `f <=< return` $\equiv$ `f`
  - Pravá identita
    `return <=< f` $\equiv$ `f`
  - Asociativita
    `f <=< (g <=< h)` $\equiv$ `(f <=< g) <=< h`
- existuje i obrácená varianta `>=>`

Ukázka: vyhodnocování výrazů I.

```
data Expr = Con Float
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
          deriving (Eq, Show)
```

Ukázka: vyhodnocování výrazů I.

```
data Expr = Con Float
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
          deriving (Eq, Show)

eval0 :: Expr -> Float
eval0 (Con x) = x
eval0 (Add x y) = eval0 x + eval0 y
eval0 (Sub x y) = eval0 x - eval0 y
eval0 (Mul x y) = eval0 x * eval0 y
eval0 (Div x y) = eval0 x / eval0 y
```

Ukázka: vyhodnocování výrazů II.

```
eval1 :: Expr -> Maybe Float
eval1 (Con x) = Just x
eval1 (Add x y) = apply (+) (eval1 x) (eval1 y)
eval1 (Sub x y) = apply (-) (eval1 x) (eval1 y)
eval1 (Mul x y) = apply (*) (eval1 x) (eval1 y)
eval1 (Div x y) = apply (/) (eval1 x) yy
  where yy = if eval1 y == Just 0
             then Nothing
             else eval1 y

apply :: (Float -> Float -> Float) -> Maybe Float
          -> Maybe Float -> Maybe Float
apply f (Just x) (Just y) = Just $ f x y
apply _ _ _ = Nothing
```

Ukázka: vyhodnocování výrazů III.

```
eval2 :: Expr -> Maybe Float
eval2 (Con x) = Just x
eval2 (Add x y) = (+) <$> eval2 x <*> eval2 y
eval2 (Sub x y) = (-) <$> eval2 x <*> eval2 y
eval2 (Mul x y) = (*) <$> eval2 x <*> eval2 y
eval2 (Div x y) = (/) <$> eval2 x <*> yy
  where yy = if eval2 y == Just 0
             then Nothing
             else eval2 y
```

Ukázka: vyhodnocování výrazů IV.

```
eval3 :: Expr -> Maybe Float
eval3 (Con x) = Just x
eval3 (Add x y) = liftA2 (+) (eval3 x) (eval3 y)
eval3 (Sub x y) = liftA2 (-) (eval3 x) (eval3 y)
eval3 (Mul x y) = liftA2 (*) (eval3 x) (eval3 y)
eval3 (Div x y) = liftA2 (/) (eval3 x) yy
  where yy = if eval3 y == Just 0
             then Nothing
             else eval3 y
```

Úkol: instance Possibly

Uvažme následující datový typ:

```
data Possibly a = None
                | Once a
                | Twice a a
                deriving (Eq, Ord, Show, Read)
```

Váš úkol je následující:

1. napište instanci pro třídu `Functor`
2. ověřte platnost pravidel (!) třídy `Functor`
3. napište instanci pro třídu `Applicative`
4. ověřte platnost pravidel (!) třídy `Applicative`