Funktory, aplikativní funktory, monády

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2018

Připomenutí: Maybe, Either, druhy

`Maybe`: datový typ rošířený o jednu hodnotu

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

Připomenutí: Maybe, Either, druhy

`Maybe`: datový typ rozšířený o jednu hodnotu

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

`Either`: sjednocení dvou datových typů

- `data Either a b = Left a | Right b`
- často: chybný a korektní výpočet

Připomenutí: Maybe, Either, druhy

`Maybe`: datový typ rošířený o jednu hodnotu

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

`Either`: sjednocení dvou datových typů

- `data Either a b = Left a | Right b`
- často: chybný a korektní výpočet

Druhy: „typování typů“

- `Maybe Int :: *`
- `Either :: * -> * -> *`
- `(,) :: * -> * -> *`

Typová třída Functor

Motivace: Zobecňování funkce `map`

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

Typová třída 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 `* -> *`
- pravidla ověřuje programátor (!)
- instance pro `[]`, `BinTree`, `Maybe`, `IO`, `Either e`, `(->) r`, ...

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
```

- Jakého typu bude operátor `apply`?

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
```

- Jakého typu bude operátor `apply`?
- `apply :: Functor f => f (a -> b) -> f a -> f b`

Typová třída 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 práci s funkcemi v kontextu
- funkce `pure` „obalí“ hodnotu minimální strukturou („přidá nejjednodušší kontext“)

Typová třída 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 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`

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 Applicative (Either e) where
  ```

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 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 II.

- `instance Applicative IO where`

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 pro seznamy

Co se seznamy?

- `[(+5), (*2), (^2)] <*> [2, 3]`

Instance třídy Applicative pro seznamy

Co se seznamy?

- `[(+5), (*2), (^2)] <*> [2, 3]`

1. Aplikujme každou funkci na každou hodnotu
   - `[2+5, 3+5, 2*2, 3*2, 2^2, 3*2]`
   - $\Rightarrow$ výchozí instance

Instance třídy Applicative pro seznamy

Co se seznamy?

- `[(+5), (*2), (^2)] <*> [2, 3]`

1. Aplikujme každou funkci na každou hodnotu
   - `[2+5, 3+5, 2*2, 3*2, 2^2, 3*2]`
   - $\Rightarrow$ výchozí instance

2. Aplikujme každou funkci na jednu hodnotu
   - `[2+5, 3*2]`
   - 2 různé instance pro tentýž typ nemůžou být $\Rightarrow$ nový typ
   - tzv. `ZipList`

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

Instance třídy Applicative III.

- `instance Applicative [] where`

Instance třídy Applicative III.

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

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

Instance třídy Applicative IV.

- `instance Functor ZipList where`

Instance třídy Applicative IV.

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

instance Applicative ZipList where
```

Instance třídy Applicative IV.

```
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) <*> x` $\equiv$ `x`
- Kompozice
  `(pure (.)) <*> f <*> g <*> x` $\equiv$ `f <*> (g <*> x)`
- Homomorfizmus
  `(pure f) <*> (pure x)` $\equiv$ `pure (f x)`
- Výměna
  `u <*> (pure y)` $\equiv$ `(pure ($ y)) <*> u`

(vícero párů závorek je implicitních)

Pravidla pro třídu Applicative

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

(vícero párů závorek je implicitních)

Jako důsledek bude pro instanci platit také

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

Aplikace běžných funkcí na hodnoty v kontextu

Definice funkcí `liftAX` pro zjednodušení práce s `Applicative`:

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

Aplikace běžných funkcí na hodnoty v kontextu

Definice funkcí `liftAX` pro zjednodušení práce s `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 `liftAX` pro zjednodušení práce s `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 `liftAX` pro zjednodušení práce s `Applicative`:

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

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)

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

Ukázka: vyhodnocování výrazů II.

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

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, která produkuje hodnotu v kontextu, na hodnotu v kontextu?

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

Typová třída Monad

```
class Applicative m => 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
- funkce `fail` se volá například pri selhání vzoru v `do`-notaci

Instance třídy Monad I.

```
instance Monad Maybe where
```

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)`

do-notace

`do`-notace je jenom syntaktický cukr pro `>>=`

- tj. se všemi monády můžeme pracovat přes `do`

do-notace

`do`-notace je jenom syntaktický cukr pro `>>=`

- tj. se všemi monády můžeme pracovat přes `do`

```
maybePlus
  :: Maybe Float -> Maybe Float -> Maybe Float
maybePlus x y = do
  justX <- x
  justY <- y
  return $ justX + justY
```

Úkol: instance Functor a Applicative

Uvažme následující datový typ:

```
newtype Triple a = Triple { unTriple :: (a, a, a) }
                  deriving (Eq, Ord, Show, Read)
```

1. Napište instanci pro třídu `Functor`.
   (Jak bude fungovat funkce `map`?)
2. Zamyslete se nad platností pravidel (!) třídy `Functor`.
3. Napište instanci pro třídu `Applicative`.
   (Jak aplikovat funkce?)
4. Zamyslete se nad platností pravidel (!) třídy `Applicative`.

Úkol: do-notace

1. Napište „nedeterministické“ „skoro-aritmetické“ funkce:
   `2 ~+~ 5` $\rightsquigarrow^*$ `[6, 7, 8]`
   `3 ~*~ 5` $\rightsquigarrow^*$ `[10, 15, 20, 12, 15, 18]`
2. Napište pomocí `do`-notace nedeterministický výpočet:
   `almostPlusTimes x y z` $\equiv$ `(x ~+~ y) ~*~ z`
3. Implementujte funkci `maxMinEval :: [a] -> (a, a)`, která z možných výsledků vybere nejmenší a největší.