FLP – druhé cvičení
Haskell: vyšší funkce, datové typy, monády

S. Židek, P. Matula, M. Kidoň, L. Škarvada

ÚIFS FIT VUT

Funkcionální a logické programování, 2019/2020

Částečná aplikace (opak.)

```
(+)       :: Num a =>  a -> a -> a
(+) 1     :: Num a =>       a -> a
(+) 1 2   :: Num a =>            a
```

Funkce v Haskellu mají jen jeden parametr

```
a -> b -> c -> d   ≡   a -> (b -> (c -> d))

    f x y z        ≡        ((f x) y) z
```

Funkce vyššího řádu

Jejich parametrem je funkce.

Výběr těch prvků ze seznamu, které splňují daný predikát:

```
filter :: (a -> Bool) -> [a] -> [a]
-- filter (<10) [1, 9, 10, 11, 20]
-- filter isUpper "AaaBbbbC12"
```

Aplikování funkce na všechny prvky seznamu:

```
map :: (a -> b) -> [a] -> [b]
-- map (*2) [1, 2, 4, 8, 16]
```

Kompozice funkcí (`g` po `f`, kde `f :: a -> b`, `g :: b -> c`):

```
(.) :: (b -> c) -> (a -> b) -> a -> c
g . f = \x -> g (f x)
```

Fold

Funkce **`foldr`**, **`foldl`** aplikují binární operaci na všechny prvky seznamu. Nazývají se též akumulační funkce.

- **`foldr`**: zprava doleva
- **`foldl`**: zleva doprava

Příklad:

```
foldr (+) 0 [2, 4, 8] = (2 + (4 + (8 + 0)))
foldl (*) 1 [3, 5, 7] = ((1 * 3) * 5) * 7)
```

Schéma: **`fold`**`x op unit list`

`unit` je hodnota funkce pro prázdný seznam, typicky to bývá neutrální prvek operace `op`.

Fold

Definice (specializované na seznamy):

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op z []     = z
foldr op z (x:xs) = x `op` foldr op z xs

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op z []     = z
foldl op z (x:xs) = foldl op (z `op` x) xs
```

Příklady knihovních funkcí:

```
sum = foldl (+) 0
product = foldl (*) 1
or = foldr (||) False
concat = foldr (++) []
```

Fold pro neprázdné seznamy

$$\texttt{foldr1}\ (\otimes)\ [x_1, \ldots, x_n] \quad = \quad x_1 \otimes (x_2 \otimes (\ldots (x_{n-1} \otimes x_n) \ldots))$$
$$\texttt{foldl1}\ (\otimes)\ [x_1, \ldots, x_n] \quad = \quad ((\ldots (x_1 \otimes x_2) \ldots) \otimes x_{n-1}) \otimes x_n$$

Definice (specializované na seznamy):

```
foldr1           ::  (a -> a -> a) -> [a] -> a
foldr1 _   [x]    =  x
foldr1 op (x:xs)  =  x `op` foldr1 op xs

foldl1           ::  (a -> a -> a) -> [a] -> a
foldl1 op (x:xs)  =  foldl op x xs
```

Například

```
minimum = foldl1 min
maximum = foldl1 max
```

Fold

Jaké jsou dostatečné podmínky pro `op`, `u`, aby platily ekvivalence

1. `foldl1 op` $\sim$ `foldr1 op`
2. `foldl op u` $\sim$ `foldr op u`

Poznáte, co dělají funkce `f1`, `f2`?

```
f1 :: (a -> b) -> [a] -> [b]
f1 g = foldr (\x y -> g x : y) []
     -- neboli foldr ((:) . g) []

f2 :: [a] -> Int
f2 = foldl (\c _ -> c+1) 0
     -- neboli foldl (const . (1+)) 0
```

Datové typy

Datové typy lze definovat dvěma způsoby:

- typové synonymum – nové jméno pro existující typ

```
type SeznamCelychCisel = [Int]
```

- nový datový typ

```
data TypeConstructor <typevars>
  = DataConstructor1 <types>
  | DataConstructor2 <types>
  ...
  | DataConstructorN <types>
```

- nový datový typ s jediným datovým konstruktorem

```
newtype TypeConstructor <typevars>
  = DataConstructor <types>
```

Datové typy – knihovní i vlastní

- Logické hodnoty

```
data Bool = False | True
```

- Výčtové typy

```
data Den = Po | Ut | St | Ct | Pa | So | Ne
```

- Typ parametrizovaný třemi čísly

```
data Color = RGB Int Int Int
```

- Typ s přidanou speciální hodnotou

```
data Maybe a = Nothing | Just a
```

- Disjunktní sjednocení typů

```
data Either a b = Left a | Right b
```

- Rekurzivní typ přirozených čísel (uměle definovaný)

```
data Nat = Zero | Succ Nat
```

- Znovu definovaný seznam (s prvky obecného typu `a`)

```
data List a = Nil | Cons a (List a)
```

- Binární stromy (s vnitřními uzly ohodnocenými hodnotami typu `a`)

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Příklad

Datový typ pro tělesa

```
data Teleso
     = Kvadr Double Double Double  -- hrany
     | Kuzel Double Double  -- polomer podstavy, vyska
     | Koule Double  -- polomer
```

Příklad použití ve funkci:

```
objem :: Teleso -> Double
objem (Kvadr a b c) = a * b * c
objem (Kuzel r v) = pi * r*r * v / 3
objem (Koule r) = 4 * pi * r*r*r / 3
```

Pattern matching přes datové konstruktory

Vektor

Definujeme vlastní datový typ **`Vector`** pro konečnou posloupnost, která má pevnou délku, a tato délka je součástí hodnoty.

```
data Vector a = Vec Int [a]
```

Definujte funkci `initVector`, která vytvoří vektor ze seznamu a nastaví jeho délku. Dále funkci `dotProd`, která spočte skalární součin číselných vektorů. Využijte knihovních funkcí `zipWith`, `sum`.

```
initVector :: [a] -> Vector a
initVector xs = ...
```

```
dotProd :: Num a => Vector a -> Vector a -> Maybe a
dotProd ... = ...
```

Record syntax

Máme-li datový typ

```
data Predmet = Predmet String [Student] Int
```

v němž je název, seznam studentů a hodinová dotace,
často je užitečné definovat přístupové funkce, tzv. *selektory*:

```
nazev :: Predmet -> String
nazev (Predmet n _ _) = n
zapsani :: Predmet -> [Student]
zapsani (Predmet _ z _) = z
hodin :: Predmet -> Int
hodin (Predmet _ _ h) = h
```

To lze udělat naráz pomocí záznamů (records):

```
data Predmet = Predmet
     { nazev :: String
     , zapsani :: [Student]
     , hodin :: Int
     }
```

Znovu Teleso

Konstrukce:

```
Predmet { nazev="FLP", zapsani=[], hodin=26 }
```

$\sim$

```
Predmet "FLP" [] 26
```

Definujte typ **Teleso** pomocí recordů (v typu budou tři)

```
data Teleso ...
```

Lambda výrazy

Datový typ pro výrazy lambda kalkulu.

```
type VarName = String
data LExp
     = LVar VarName
     | LApp LExp LExp
     | LAbs VarName LExp
deriving Show
```

Funkce `freeVars` vrátí seznam všech volných proměnných v lambda-výrazu.

```
freeVars :: LExp -> [VarName]
freeVars = fv [] where
  fv b (LVar v) = if v `elem` b then [] else [v]
  fv b (LApp e1 e2) = fv b e1 ++ fv b e2
  fv b (LAbs v e) = fv (v:b) e
```

Funkce s lambda výrazy

Zkuste si definovat obdobné operace s lambda výrazy (D.ú.)

- substituce za volnou proměnnou,
  `subst :: VarName -> LExp -> LExp -> LExp`
- jednokroková redukce, `reduce1 :: LExp -> LExp`
- normální forma, `reduceNF :: LExp -> LExp`

Monády: motivace

- Dosud jsme se zabývali čistými funkcemi, tj. funkcemi bez vedlejších efektů.
- Pro praktické programy potřebujeme vedlejší efekty:
  - vstup / výstup
  - jiná interakce s OS
  - generování náhodných čísel
  - databáze
  - síťová komunikace

Akce s vedlejšími efekty jsou monadické hodnoty typu `IO a`
Typový konstruktor `IO` je tzv. monáda.

Pro výpočty s vedlejšími efekty je v haskellovských knihovnách monád celá řada, monáda `IO` patří k nejdůležitějším.

Monády

Monády jsou typové funkce vytvářející monadický typ.
Například `IO` je monáda a `IO String` je typ monadické akce, která umí načíst ze vstupu řetězec. Tento řetězec je „vnitřním“ výsledkem monadické akce a lze ho předat další monadické akci pomocí operátoru `(>>=)`.

- `IO a` akce s výsledkem typu `a`
- `IO String` akce vracející řetězec (např. načtení obsahu souboru)
- `IO ()` akce bez významné návratové hodnoty – důležitý je jen vedlejší efekt

Poznámka: Nejen `IO`, ale i řada běžných typových konstruktorů jsou monády: `Maybe`, `[]`, `Either a`, `Identity`...

```
Just 3 >>= return . (2*)
[5,7] >>= return . (2*)
getLine >>= putStrLn . map toUpper
```

Monády a jejich metody

Čistá monadická hodnota: `return` neboli `pure`

```
return :: Monad m => a -> m a
```

- „zabalení“ čisté hodnoty do monadické, „výpočet“ bez vedlejších efektů

Navázání monadické hodnoty na monadickou funkci

`>>=` (čteme „bind“)

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

1. provedení prvního výpočtu
2. „vybalení“ výsledku prvního výpočtu
3. předání tohoto výsledku funkci a její vyhodnocení
4. provedení druhého výpočtu, jenž je funkční hodnotou dané funkce

Výsledek výpočtu zůstává stále „zabalen“ v monadické hodnotě.

Monády a jejich metody

Řazení výpočtů za sebou bez navazování na vnitřní hodnoty:

```
putStr "xs = [" >> print x >> putStrLn "]"
```

Operátor `(>>)` je variantou operátoru `(>>=)`, která ignoruje výsledek prvního výpočtu:

```
(>>) :: Monad m => m a -> m b -> m b
m1 >> m2  =  m1 >>= const m2
```

1. provedení prvního výpočtu
2. provedení druhého výpočtu
3. výsledek druhého výpočtu je výsledkem celého

Co dělá následující akce?

```
getLine >>= \s -> getLine >>=
    \t -> putStrLn t >> putStrLn s
```

Monády a jejich metody

Varianty operátoru *bind*

- `(>>=)` navázání výpočtu na funkci vracející výpočet
- `(=<<)` aplikace funkce vracející výpočet na výsledek výpočtu
- `(>>)` řazení dvou výpočtů za sebou
- `(<=<)` skládání funkcí vracejících výpočty
- `(>=>)` (dopředné) skládání funkcí vracejících výpočty

Zjistěte jejich typy.
(Možná budete potřebovat `import Control.Monad`)

Monáda `IO` a knihovní funkce

Příklady knihovních akcí a funkcí:

```
getLine :: IO String
putStrLn :: String -> IO ()
putChar :: Char -> IO ()
print :: Show a => a -> IO ()
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
doesFileExist :: FilePath -> IO Bool
listDirectory :: FilePath -> IO [FilePath]
getArgs :: IO [String]
getCurrentTime :: IO UTCTime
```

Knihovní funkce Haskellu: `https://hackage.haskell.org/package/base`

Syntax: notace `do`

```
m  >>= (\ a -> f a >>= (\ _ -> g a
   >>= (\ b -> return (h a b))))
```

. . . nepřehledné

```
m >>= \ a ->
    f a >>
    g a >>= \ b ->
    return (h a b)
```

. . . o trochu čitelnější

```
do
   a <- m
   f a
   b <- g a
   return (h a b)
```

. . . ještě přehlednější

V každém případě je však třeba za zápisem `do` vidět výraz s operátory `>>=`.
Notace `do` je jen syntaktický cukr.

Notace do

Definujte funkci `doTwice :: IO a -> IO (a,a)`, která provede danou akci dvakrát a výsledkem akce je dvojice vnitřních výsledků.

Zkuste

```
ghci> import Data.Time
ghci> doTwice getCurrentTime
```

Palindromy

```
module Main where
import Data.Char (isAlpha, toLower)
import System.Environment (getArgs)

main :: IO ()
main = do
    [s] <- getArgs
    putStr ("\"" ++ s ++ "\" ")
    putStr (if pal s then "je" else "neni")
    putStrLn " palindrom."

pal :: String -> Bool
pal s = t == reverse t
    where t = map toLower (filter isAlpha s)
```

Více palindromů

```
module Main where
import Data.Char (isAlpha, toLower)
import System.Environment (getArgs)

main :: IO ()
main = getArgs >>= mapM_ pM

pM :: String -> IO ()
pM s = do
    putStr ("\"" ++ s ++ "\"\t")
    putStr (if pal s then "je" else "neni")
    putStrLn " palindrom"

pal :: String -> Bool
pal s = t == reverse t
    where t = map toLower (filter isAlpha s)
```

Čisté monadické typy

Nejen `IO`, ale i řada běžných typových konstruktorů jsou monády:

- `Identity` identita na typech
- `Maybe`
- `[]` seznamový typový konstruktor
- `Either a` částečně (na jeden typ) aplikovaný typový konstruktor
- `(,) a` částečně aplikovaný typový konstruktor
- `(->) a` částečně aplikovaný typový konstruktor

Pro `Maybe` je

```
instance Monad Maybe  where
    return  =  Just
    Nothing >>= _   =  Nothing
    Just x >>= h   =  h x
```

Při těchto definicích metod `return` a `(>>=)` se skládání `(<=<)` funkcí `g :: b -> Maybe c`, `f :: a -> Maybe b` chová tak, že složená funkce vrací `Nothing`, když výsledkem kterékoli ze skládaných funkcí `g`, `f` je `Nothing`.

```
log2 :: Double -> Maybe Double
log2 x = if x>0 then Just (logBase 2 x) else Nothing
odm :: Double -> Maybe Double
odm x = if x>=0 Just (sqrt x) else Nothing
```

Zkuste: `map (odm <=< log2 <=< log2) [0 .. 4]`