IB015 Neimperativní programování

Akumulační funkce, Definice typů, Vstup/výstup

Jiří Barnat
Libor Škarvada

Akumulační funkce

Akumulační funkce `foldl, foldr`

Pozorování

- Seznam je posloupnost oddělených prvků.
- Motivací akumulačních funkcí je "spojit" jednotlivé prvky seznamu dohromady, tj. akumulovat informaci uloženou v těchto jednotlivých prvcích do jedné hodnoty.
- Počet prvků seznamu je variabilní, proto se tato akumulace realizuje pomocí binárního operátoru postupně.

Spojení hodnot v seznamu pomocí binární operace

$$\texttt{foldl1}\ \oplus\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* ((\ldots(x_1\oplus x_2)\ldots)\oplus x_{n-1})\oplus x_n$$

$$\texttt{foldr1}\ \oplus\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* x_1\oplus(x_2\oplus(\ldots(x_{n-1}\oplus\ x_n)\ldots))$$

Akumulační funkce na neprázdných seznamech

Příklady použití

```
foldl1 (*) [1,2,3,4,5] ⇝ ... ⇝ 120
foldl1 (&&) [True, True, True, False, True] ⇝ ... ⇝ False
foldl1 (-) [2,3,2] ⇝ ... ⇝ -3
foldr1 (-) [2,3,2] ⇝ ... ⇝ 1
foldr1 (min) [18,12,23] ⇝ ... ⇝ 12
```

Funkce `foldl1, foldr1` nejsou definovány pro `[]`

```
foldl1 (*) [] ⇝ ERROR
foldr1 (&&) [] ⇝ ERROR
```

Na jednoprvkových seznamech je to identita s kontrolou typu

```
foldr1 (*) [0] ⇝ 0
foldr1 (*) [1] ⇝ 1
foldr1 (*) [True] ⇝ ERROR
```

Akumulační funkce `foldl, foldr`

Princip

- Akumulační funkce, které mají fungovat i na prázdných seznamech, vyžadují navíc **iniciální hodnotu** pro proces akumulace.
- Směr závorkování určuje i místo použití iniciální hodnoty.

Akumulace hodnot s využitím iniciální hodnoty

$$\texttt{foldl}\ \oplus\ \mathbf{v}\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* ((\ldots((\mathbf{v}\oplus x_1)\oplus x_2)\ldots)\oplus x_{n-1})\oplus x_n$$

$$\texttt{foldr}\ \oplus\ \mathbf{v}\ [x_1,x_2,\ldots,x_n] \rightsquigarrow^* x_1\oplus(x_2\oplus(\ldots(x_{n-1}\oplus(x_n\oplus\mathbf{v}))\ldots))$$

Akumulační funkce na libovolných seznamech

Příklady

```
foldl (*) 0 [1,2,3,4,5] ⇝ ... ⇝ 0
foldl (&&) False [True, True, True, True] ⇝ ... ⇝ False
foldl (-) 0 [2,3,2] ⇝ ... ⇝ -7
foldr (-) 0 [2,3,2] ⇝ ... ⇝ 1
```

Aplikace na prázdné seznamy

```
foldl max 100 [] ⇝ ... ⇝ 100
foldr (++) "Nic"[] ⇝ ... ⇝ "Nic"
```

Výsledek může být opět seznam!

```
foldr (:) [] "Coze?"⇝ ... ⇝ "Coze?"
foldr (\x y->(x+1):y) [100] [1,2,3] ⇝ ... ⇝ [2,3,4,100]
```

Definice akumulačních funkcí

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ v [] = v
foldl op v (x:s) = foldl op (v `op` x) s


foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr op v (x:s) = x `op` foldr op v s


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


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

Uživatelem definované typy

Riziko nepoužití uživatelem definovaných typů

Pozorování

- Počítač veškerá data reprezentuje čísly.
- Programátoři jej dobrovolně, či nedobrovolně napodobují.

Riziko

- Mnohdy číselná reprezentace různých hodnot není přímočará a tedy umožňuje nechtěné zadání neplatných hodnot.
- Neplatné hodnoty mohou vzniknout i neopatrnou aplikací číselných operací.
- **Použití neplatných hodnot může být nebezpečné.**

Riziko nepoužití uživatelem definovaných typů – příklad

Příklad

- Chceme reprezentovat den v týdnu a definovat funkce pracující s touto reprezentací.
- Možné číselné kódování, je následující:

  pondělí $= 1$, úterý $= 2$, ..., neděle $= 7$

- Funkce `zitra` (s chybou) a funkce `je_pondeli` :

```
zitra :: Int -> Int
zitra x = x+1                          - nesprávně i (x+1) `mod` 7

je_pondeli :: Int -> Bool
je_pondeli x = if (x==1) then True else False
```

Chyba ve výpočtu

`je_pondeli 8` $\rightsquigarrow$ `False`

`je_pondeli (zitra 7)` $\rightsquigarrow$ `...` $\rightsquigarrow$ `False`

Uživatelem definované typy

Definice typů

- V Haskellu pomocí klíčového slova `data` .
- Obecná šáblona:

  `data` **Název_typu** `=` **Datové_konstruktory**
- Jednotlivé datové konstruktory se oddělují znakem `|`
- Syntaktické omezení Haskellu: nově definovaný typ i datové konstruktory musí začínat velkým písmenem.

Příklad

- Dny v týdnu lze definovat jako nový typ, který má 7 hodnot.

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

- Hodnoty jsou definovány výčtem.
- Jsou použity nulární datové konstruktory – konstanty.

Definice n-árních datových konstruktorů

Uživatelem definované

- Obecná šablona pro n-ární datový konstruktor:

  Jméno Typ_1 ... Typ_n

- Příklad typu s ternárním datovým konstruktorem:

  `data Barva =` **`RGB Int Int Int`**

- Hodnoty typu Barva:

```
RGB 42 42 42
RGB 12 (-23) 45
```

Částečná aplikace datového konstruktoru

```
RGB          :: Int -> Int -> Int -> Barva
RGB 23       ::        Int -> Int -> Barva
RGB 23 23    ::               Int -> Barva
RGB 23 23 23 ::                      Barva
```

Typové konstruktory

Typové konstanty

- Definicí dle šablony:

  `data` **Název_typu** `= Datové_konstruktory`

  zavádíme nový typ s označením `Název_typu`.
- `Název_typu` je nulární typový konstruktor, typová konstanta.

N-ární typové konstruktory

- Typové konstruktory jako například `->` nebo `[]` nedefinují typ, pouze předpis jak nový typ vyrobit.

Tvorba typu

- Každá typová konstanta definuje typ.
- Typ získám také úplnou aplikací n-árních typových konstruktorů na již definované typy.

  `(->) Dny Bool =` **Dny -> Bool**

  `[] Dny =` **[Dny]**

  `(->) (Dny -> Bool) [Dny] =` **(Dny -> Bool) -> [Dny]**

Hodnoty a typy

Tvorba nových hodnot

- Aplikace datových konstruktorů vytváří nové hodnoty.

Tvorba nových typů

- Aplikace typových konstruktorů vytváří nové typy.

Uspořádané n-tice a seznamy

- Používá se stejné označení pro typové i datové konstruktory!

```
'a' :: Char
[('a','a'),('a','a')] :: [(Char,Char)]          - datové
[('a','a'), ('a','a')] :: [(Char,Char)]         - typové
```

Příklad použití uživatelem definovaných typů

```
data Policie =  Hlidka (String,String) | Oddeleni [Policie]
                deriving Show

h1 = Hlidka ("Pepa", "Emil")
h2 = Hlidka ("Jason", "Drson")
o1 = Oddeleni [h1, h2]

jmena :: Policie -> [String]
jmena (Hlidka (a,b)) = a:b:[]
jmena (Oddeleni []) = []
jmena (Oddeleni (x:s)) = jmena x ++ jmena (Oddeleni s)
```

Definice polymorfních typových konstruktorů

Polymorfní typové konstruktory

- Seznam prvků typu `a` , strom hodnot typu `a` , ...

Definice polymorfních typových konstruktorů

- Definice s využitím typových proměnných:

  `data` **Název_typu** $\mathbf{a_1}$ **...** $\mathbf{a_n}$ `= ...`

- Typové proměnné lze použít pro definici datových konstruktorů.

Kompletní obecná šablona

$$\begin{aligned}\texttt{data Tcons } a_1 \texttt{ ... } a_n \texttt{ = } & \texttt{Dcons}_1 \ \texttt{typ}_{(1,1)} \ \texttt{typ}_{(1,2)} \ \texttt{...} \ \texttt{typ}_{(1,arita_1)} \\ & \vdots \\ & \texttt{Dcons}_m \ \texttt{typ}_{(m,1)} \ \texttt{typ}_{(m,2)} \ \texttt{...} \ \texttt{typ}_{(m,arita_m)}\end{aligned}$$

Typový konstruktor `Maybe`

Maybe

- Předdefinovaný unární polymorfní typový konstruktor.

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

- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`

```
deleni :: Fractional a => a -> a -> Maybe a
deleni x y = if (y==0) then Nothing else Just (x/y)
```

- Jaký je výsledek aplikace `deleni` na argumenty `32` a `8` ?

Typový konstruktor `Maybe`

Maybe

- Předdefinovaný unární polymorfní typový konstruktor.

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

- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`

```
deleni :: Fractional a => a -> a -> Maybe a
deleni x y = if (y==0) then Nothing else Just (x/y)
```

- Jaký je výsledek aplikace `deleni` na argumenty `32` a `8` ?

  Just 4.0

- Proč je následující definice špatně?

```
deleni x y = if (y==0) then Nothing else (x/y)
```

Vstup/výstup

Vstup/výstup a referenční transparentnost

Referenční transparentnost

- Daný výraz se vždy vyhodnotí na stejnou hodnotu, bez ohledu na okolí (kontext), ve kterém je použit.
- Programovací jazyk Haskell je referenčně transparentní.

Dopady na vstup-výstupní chování

- **Nelze napsat program, který by zpracoval vstup uživatele a vyhodnotil se podle zadaného vstupu na různé hodnoty.**
- Lze napsat program, který zpracuje vstup a podle vstupu vypíše na výstup různé výsledky.
- Hodnoty předávané skrze vstup-výstupní akce nesouvisí s hodnotou výrazu, který tuto vstup-výstupní akci realizuje.

Typy `IO()` , `IO a`

Vstup-výstupní akce

- Interakce programu s uživatelem nebo operačním systémem.
- Například výpis textu na terminálu, vytvoření nového souboru, načtení hodnoty proměnné prostředí, ...

Myšlenka

- Pro vstup-výstupní akce je zaveden speciální typ – **`IO a`** .
- Tento typ má formálně jednu jedinou textově nereprezentovatelnou hodnotu, a to **vstup-výstupní akci**.
- Výstupní akce mají typ **`IO ()`** .

```
putStrLn "Ahoj!":: IO ()
```

- Vstupní akce mají typ **`IO a`** , kde typová proměnná `a` nabývá hodnoty (typu) podle typu objektu, který vstupuje.

```
getLine :: IO String
```

Vnitřní výsledek a vedlejší efekt

Otázka

- Jestliže `getLine` načte řetězec ze vstupu a přitom má hodnotu vstup-výstupní akce, což je hodnota typu `IO a`, konkrétně zde `IO String`, tak potom by nás zajímalo, kde je onen načtený řetězec?

Vnitřní výsledek a vedlejší efekt

Otázka

- Jestliže `getLine` načte řetězec ze vstupu a přitom má hodnotu vstup-výstupní akce, což je hodnota typu `IO a`, konkrétně zde `IO String`, tak potom by nás zajímalo, kde je onen načtený řetězec?

Odpověď

- Načtený řetězec se uchová jako tzv. **vnitřní výsledek** provedení této vstupní akce.
- Skutečné načtení řetězce a zapamatování si vnitřního výsledku je realizováno jako **vedlejší efekt** vyhodnocení výrazu `getLine`.

Operátor »=

Přístup k hodnotě vnitřního výsledku

- Pomocí binárního operátoru `»=` .
- Ve výrazu `f »= g` funguje operátor `»=` tak, že vezme vnitřní výsledek vstupní akce `f` a na tento aplikuje unární funkci `g` , **jejímž výsledkem je ovšem vstup-výstupní akce**.
- Výraz `f »= g` tedy znamená, že:

```
f :: IO a
g :: a -> IO b
f »= g :: IO b
```

Operátor »=

- `(»=) :: IO a -> ( a -> IO b ) -> IO b`
- Následující zápis je ekvivalentní:

```
getLine »= putStr
getLine »= (\x -> putStr x)
```

Nesprávné použití operátoru »=

Otázka

- Operátor `(»=)` nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze**:

```
getLine >>= length
getLine >>= (\ x -> length x)
```

Nesprávné použití operátoru »=

Otázka

- Operátor (»=) nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze**:

```
getLine >>= length
getLine >>= (\ x -> length x)
```

Odpověď

- Hodnota výrazu je závislá na zadaném vstupu!
- Porušuje referenční transparentnost.
- Typově nesprávně.
- Správné použití:

```
getLine >>= print . length
getLine >>= (\ x -> print (length x))
```

Řazení vstup-výstupních akcí

Funkce `return`

- Prázdná akce, jejíž provedení má za cíl pouze naplnit hodnotu vnitřního výsledku.

```
return :: a -> IO a
return ['A','h','o','j'] >>= putStr
```

Řazení akcí, operátor `>>`

- Binární operátor, který řadí vstup-výstupní akce.
- Zapomíná/ničí hodnotu vnitřního výsledku.
- Výraz má hodnotu poslední (druhé) vstup-výstupní akce.
- `(>>) :: IO a -> IO b -> IO b`
- Příklady použití:

```
putStr "Jeje" >> putChar '!'
getLine >> putStr "nic"
```

Základní funkce pro výstup

```
putChar :: Char -> IO ()
```

- Zapíše znak na standardní výstup.

```
putStr :: String -> IO ()
```

- Zapíše řetězec na standardní výstup.

```
putStrLn :: String -> IO ()
```

- Zapíše řetězec na standardní výstup a přídá znak konec řádku.

```
print :: Show a => a -> IO ()
```

- Vypíše hodnotu jakéhokoliv tisknutelného typu na standardní výstup, a přidá znak konec řádku.
- Tisknutelné typy jsou instancí třídy `Show` .
- Uživatelem definované typy nutno označit přídomkem `deriving Show` .

Základní funkce pro vstup

```
getChar :: IO Char
```

- Načte znak ze standardního vstupu.

```
getLine :: IO String
```

- Načte řádek ze standardního vstupu.

```
getContents :: IO String
```

- Čte veškerý obsah ze standardního vstupu jako jeden řetězec. Obsah je čten líně, tj. až když je potřeba.

```
interact :: (String -> String) -> IO ()
```

- Argumentem funkce `interact` je funkce, která zpracovává řetězec a vrací řetězec.
- Veškerý obsah ze standardního vstupu je předán této funkci a výsledek vytištěn na standardní výstup.

Práce se soubory

```
type FilePath = String
```

- Definuje typový alias.

```
readFile :: FilePath -> IO String
```

- Načte obsah souboru jako řetězec. Soubor je čten líně.

```
writeFile :: FilePath -> String -> IO ()
```

- Zapíše řetězec do daného souboru (existující obsah smaže).
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show`.

```
appendFile :: FilePath -> String -> IO ()
```

- Připíše řetězec do daného souboru.
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show`.

Další vstup-výstupní akce

Moduly System a Directory

- Existují další vstup-výstupní funkce pro práci s adresáři či systémovými proměnnými.
- Tyto funkce jsou předdefinovány v modulech `System` a `Directory` .

Použití modulu

- Moduly, jejichž funkce chceme použít, je třeba označit.
- Lze to učinit v souboru s globálními definicemi použitím klíčového slova `import` .
- Příklad:

```
import Char
import Directory
main = getDirectoryContents ".." >>=
       print . map (\x -> (toUpper.head) x : tail x)
```

Zápis pomocí `do`

Pozorování

- Syntaktická konstrukce `do` slouží k alternativnímu zápisu výrazu s operátory `>>=` a `>>`.

Následující zápis je ekvivalentní

- ```
  putStr "vstup?"  >>
  getLine          >>= \ x ->
  putStr "výstup?" >>
  getLine          >>= \ y ->
  readFile x       >>= \ z ->
  writeFile y (map toUpper z)
  ```

  ```
  do putStr "vstup?"
     x <- getLine
     putStr "výstup?"
     y <- getLine
     z <- readFile x
     writeFile y (map toUpper z)
  ```
  ```

Funkce `sequence`

Funkce `sequence`

- Máme-li seznam vstup-výstupních akcí, můžeme je pomocí funkce `sequence` provést dávkově naráz.
- ```
  sequence :: [IO a] -> IO [a]
  sequence [] = return []
  sequence (a:s) = do x<-a
                      t <- sequence s
                      return (x:t)
  ```

Příklady použití

- V případě výstupních akcí je výsledkem vyhodnocení výrazu posloupnost výstupů, viz:

  ```
  sequence [ putStr "Ahoj", putStr " ", putStr "světe!"]
  ```

- V případě vstupních akcí, je výsledkem vyhodnocení výrazu seznam vstupů, který je uložený jako vnitřní výsledek vstup-výstupní akce, viz:

  ```
  sequence [ getLine, getLine, getLine ] >>= print
  ```
  ```

Funkce `mapM`

- Aplikuje unární funkci, jejíž výsledkem je vstup-výstupní akce, na seznam hodnot a vzniklý seznam vstup-výstupních akcí provede.

```
mapM :: (a -> IO b) -> [a] -> IO [b]
mapM f = sequence . map f
```

Příklady použití

- `mapM putStr ["Den","Noc"]`
  vypíše `DenNoc`
- `mapM (\ t -> putStr "Aa") [1,2,3,4,5]`
  vypíše `AaAaAaAaAa`
- `mapM (\ x->getLine) [1,2] >>= print`
  po zadání dvou řádků s obsahem `radek1` a `radek2`
  vypíše `["radek1","radek2"]`

Domácí cvičení

Mentální cvičení

- Zdůvodněte (uvědomte si) proč je typ funkcí `foldr` a `foldl` takový, jaký je.
- Zadefinujte funkci `sequence` bez použití notace `do`.

Programování v Haskellu

- Definujte funkci, která pro 12 měsíčních platů zadaných seznamem vypočítá 15% daň z příjmu s přihlédnutím k tomu, že z celkové výše ročního příjmu se daní pouze část, která převyšuje nezdanitelné minimum ve výši 24 600 Kč.
- Vyhledejte popis funkcí obsažených v modulech `Char`, `Directory` a `IO` a vyzkoušejte je ve svých programech.
- Napište program, který vyzve uživatele, aby zadal 16 čísel oddělených mezerou, a poté tato čísla vypíše v matici velikosti 4x4.