IB015 Neimperativní programování

Práce se vstupy a výstupy

Jiří Barnat
Libor Škarvada

Oprávněná stížnost studentů

Úryvek diskuse mezi studenty

- *... myslela jsem, že se budeme učit programovat ...*
- *... zatím si jen hrajeme s ňákým blbým GHCI ...*
- *... akorát obskurně zapisujeme funkce ...*

Oprávněná stížnost studentů

Úryvek diskuse mezi studenty

- *... myslela jsem, že se budeme učit programovat ...*
- *... zatím si jen hrajeme s ňákým blbým GHCI ...*
- *... akorát obskurně zapisujeme funkce ...*

WTF ???

Oprávněná stížnost studentů

Úryvek diskuse mezi studenty

- *... myslela jsem, že se budeme učit programovat ...*
- *... zatím si jen hrajeme s ňákým blbým GHCI ...*
- *... akorát obskurně zapisujeme funkce ...*

WTF ???

Náplň dnešní lekce

- Jak realizovat komunikace mezi programem a jeho okolím.
- Napíšeme, přeložíme a spustíme první program v Haskellu.

Co vlastně dělá takový program?

Samostaně spustitelný program

- **Posloupnost akcí**, při kterých dochází k interakci programu s okolním světem.

Interakce s okolím programu

- Interakce s uživatelem skrze terminál.
- Manipulace se soubory, čtení a zápis do souboru.
- Interakce skrze grafické rozhraní.
- Komunikace s operačním systémem.
- ...

Výpočty obalují vstup-výstupní akce

Práce s interpretem

- Doposud naše programy v Haskellu žádné akce nepřipouštěly.
- Žádný pokyn k výpisu výsledku jsme do programu nevkládali.
- GHCI může za to, že po skončení výpočtu se vypíše výsledná hodnota na terminál.

Samostatný program

- Explicitní pokyny ke komunikaci s uživatelem.
- Při akci čtení vstupu z terminálu, se výpočet zastaví a čeká, až uživatel vloží text.
- Dochází k prokládání výpočtu, tak jak jsme jej znali doposud, a realizace vstup-výstupních akcí.

Vstup/výstup

Typy vstup/výstupních akcí

Typování vstup-výstupních akcí

- Unární typový konstruktor **IO a** .
- Typy vytvořené pomocí typového konstruktoru `IO` mají jednu jedinou hodnotu, a to **vstup-výstupní akci**.
- Tato hodnota nemá textovou reprezentaci (nelze ji vypsat).

Výstupní akce

- Výstupní akce mají typ **IO ()** .

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

Vstupní akce

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

```
getLine :: IO String
```

Jak si představovat IO?

Analogie

- Vstup/výstupní akci se představme jako krabici.
- Hodnota akce je vždycky stejná ... **krabice**.
- Ovšem obsah krabice je vnitřní výsledek akce.
- Obsah je určen typem, avšak může mít různé hodnoty.

7

IO Int IO Int

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.

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.

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 kde je ten 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 kde je ten 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)
```

Krabicová analogie pro operátor >>=

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

   f :: a -> IO b
```

>>= f ⇨

with a-val

with b-val

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. **„Jakmile jste v `IO` , nelze utéct“**.
- Správné použití:

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

Funkce `return`

Funkce `return` jako vstupní akce

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

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

- Možné použití:

```
return ['A','h','o','j'] >>= putStr
```

Funkce `return` jako výstupní akce

- Funkce `return` , může sloužit i jako výstupní akce, která nemá žádný efekt.

```
return :: a -> IO a
return () :: IO ()
```

Funkce `pure`

- Alternativa k funkci `return` .
- Použití funkce `return` je vzhledem ke kontextu známého z imperativních programovacích funkcí částečně matoucí.

Řazení vstup-výstupních akcí

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`

Jak se chovají následující programy?

- `putStr "Jeje" >> putChar '!'`
- `getLine >> putStr "nic"`
- ```
  putStrLn "Napiš mi něco pěkného." >> getLine >>=
      ( \x -> putStr "Napsal jsi:" >> putStrLn x )
  ```
  ```

Funkce `sequence`

Dávkové spuštění akcí

- Máme-li seznam vstup-výstupních akcí **stejného typu**, lze funkce z tohoto seznamu spustit a provést jako dávku.
- ```
  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`

Mapování IO akce na seznam argumentů

- Aplikuje unární vstup-výstupní akci 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) "aa" >>= print`
  po zadání dvou řádků s obsahem `radek1` a `radek2`
  vypíše `["radek1","radek2"]`

Jednoduchá 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`.

Zápis pomocí `do` „do notace“

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)
  ```
  ```

Další IO funkce

Další předdefinované funkce

- Existuje řada dalších funkcí, jejich definice však nejsou automaticky zavedeny při startu programu, nebo interpretu.
- Tyto funkce jsou definovány v tzv. **modulech**.

Například

- `System.IO`
- `System.Directory`
- `System.Time`
- ...

Moduly

Moduly v Haskellu

Modul v Haskellu

- Kolekce spolu souvisejících funkcí, typů a typových tříd.
- Program v Haskellu je tvořen hlavním modulem `Main`, který následně importuje a používá funkce z jiných modulů.
- `Prelude` je implicitně zavedený modul, který obsahuje definice funkcí, které jsme doposud běžně používali.
- Moduly umožňují **strukturovat kód projektu**, oddělit a případně později znovupoužívat ucelené části kódu.

Příklady předdefinovaných modulů

- Modul pro práci se seznamy – `Data.List`
- Modul pro práci se znaky – `Data.Char`
- Modul pro práci s komplexními čísly – `Data.Complex`
- ...

Použití modulu

- Před použitím funkcí a typů z modulu je třeba explicitně požádat o zavedení tohoto modulu, tzv. **importovat modul**.
- Klíčové slova `import` .

Možnosti importu

- Importovat pouze vybrané, nebo skrýt některé funkce modulu.
- Vynutit při použití funkce povinnou identifikaci modulu, v podobě `jméno_modulu.jméno_funkce` (tzv. kvalifikace).
- Přejmenovat kvalifikaci.

Příklady

- `import Data.Char`
- `import qualified Data.Char`
- `import qualified Data.Char as C`
- `import qualified Data.Char as C (toUpper)`
- `import qualified Data.Char as C hiding (toUpper)`

Pozorování

- Kvalifikace funkcí z modulů zavede nový význam znaku tečka.

Příklad

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

Definice Modulu v Haskellu

Obecná definice

- `module Jméno` [`(export`$_1$`, ..., export`$_n$ `)`] `where`

  ```
  import M1 [spec1]
       ⋮
  import Mm [specm]

  [ globální_deklarace ]
  ```

Automatické doplnění definice `Main`

- Není-li uvedena hlavička, doplní se

  ```
  module Main (main) where
  ```

- Nevyskytuje-li se mezi importovanými moduly $\texttt{M}_1, \ldots \texttt{M}_m$ modul `Prelude` , doplní se

  ```
  import Prelude
  ```

Příklad modulu – Datový typ Fifo

Datový typ Fifo

- Datový kontejner (struktura, která uchovává prvky) přistupovaný operacemi **vlož prvek** a **vyber prvek**.
- Prvky jsou z datové struktury odebírány v tom pořadí, ve kterém byly vkládány.
- First-In-First-Out = FIFO
- Operace by měly mít konstantní časovou složitost.

Realizace v Haskellu

- Definice modulu `Fifo`
- Použití modulu:

```
import Fifo
```

Příklad Modulu – Datový typ Fifo

```
module Fifo (FifoTyp, emptyq, headq, enqueue, dequeue) where

data FifoTyp a = Q [a] [a]

emptyq :: FifoTyp a
emptyq = Q [] []

enqueue :: a -> FifoTyp a -> FifoTyp a
enqueue x (Q h t) = Q h (x:t)

headq :: FifoTyp a -> a
headq (Q (x:_) _) = x
headq (Q [] []) = error "headq: prázdná fronta"
headq (Q [] t) = headq (Q (reverse t) [])

dequeue :: FifoTyp a -> FifoTyp a
dequeue (Q (_:h) t) = Q h t
dequeue (Q [] []) = error "dequeue: prázdná fronta"
dequeue (Q [] t) = dequeue (Q (reverse t) [])
```

Samostatně stojící programy

Kompilace programu

Připomenutí

- Interpretace vs. kompilace kódu.
- Dosud jsme využívali interpret `ghci` .
- Kód lze přeložit do samostatně spustitelného programu.
- Překladač `ghc` .

Výhody přeloženého kódu

- Přeložený kód je rychlejší než interpretovaný.
- Na cílovou platformu není třeba instalovat interpret, ani vývojové prostředí.

Nazdar světe

Hlavní funkce – `main`

- Spustitelný program musí obsahovat funkci `main :: IO ()` .
- Funkce je automaticky spuštěna po zavedení programu do paměti.

Nazdar světe

- `main = putStrLn "Čaute lidi!"`

Překlad

- Kód umístěn v souboru `hello.hs` .
- Překlad pomocí `ghc hello.hs -o hello`
- Vytvořen spustitelný soubor `hello` .
- Vzniknou pomocné soubory `hello.hi` a `hello.o` , lze smazat.

Funkce `show` a `read`

Změna typu

- Vstup/Výstup je obvykle realizován v objektech typu `String` .
- Převést objekty na řetězec a zpět je možné pomocí funkcí:

```
show :: Show a => a -> String
read :: Read a => String -> a
```

Asymetrie použití

- Použití `show` a `read` není symetrické, při použití `read` je třeba uvést do jakého typu chceme řetězec transformovat.

```
show 123 ⇝* "123"
read "123"⇝* ERROR
(read "123")::Int ⇝* 123
```

Příklad s parsováním vstupu

Parsování vstupu

- Převedou se pouze 100% správně strukturované řetězce.

  `(read "[1,2,3]")::[Char]` $\rightsquigarrow^*$ `ERROR`

  `(read "[1,2,3]")::[Int]` $\rightsquigarrow^*$ `[1,2,3]`

  `(read "[1,2,3]")::[Float]` $\rightsquigarrow^*$ `[1.0,2.0,3.0]`

Program `plus1`

- Tento program vyzve uživatele, aby zadal celé číslo, pak toto číslo přečte, převede na objekt typu `Int`, přičte jedničku a zvýšené číslo vytiskne.
- ```
  main = do putStrLn "Napis cele cislo:"
            x <- getLine
            print (1 + read x::Int)
  ```
  ```

Užitečné konstrukce Haskellu

Syntaktická konstrukce case

Pozorování

- Víceřádkové definice funkcí realizují větvení kódu.
- Více řádkovou definici lze ekvivalentně přepsat s využitím klíčového slova `case` .

Syntaktická konstrukce `case`

- ```
  case expression of
      pattern_1 -> expression_1
      ...
      pattern_n -> expression_n
  ```
- Textové zarovnání vzorů je nutné.
- Všechny výrazy na pravých stranách musí být stejného typu.
  ```

Syntaktická konstrukce case – příklady

Úkol 1

- Definujte funkci `take` s využitím konstrukce `case`.
- Řešení:

```
take m ys = case (m,ys) of
   (0,_) -> []
   (_,[]) -> []
   (n,x:xs) -> x : take (n-1) xs
```

Úkol 2

- Zapište pomocí `case` výraz `if e1 then e2 else e3`.
- Řešení:

```
case (e1) of True -> e2
             False -> e3
```

Strážené definice funkcí

Stráž

- Stráž je výraz typu Bool přidružený k definičnímu přiřazení.
- Při výpočtu bude tato definice funkce realizována, pouze pokud bude asociovaný výraz vyhodnocen na `True` .
- ```
  function args
        | guard_1 = expression_1
  ...
        | guard_n = expression_n
        | otherwise = default expression
  ```

Pozorování

- Konstrukce nerozšiřuje výrazové možnosti jazyka, ale je pohodlná (syntaktický cukr).
  ```

Strážené definice funkcí – příklady

Příklad 1

- ```
  f x | (x>3) = "vetsi nez 3"
      | (x>2) = "vetsi nez 2"
      | (x>1) = "vetsi nez 1"
  ```
- `f 2` $\rightsquigarrow^*$ `"vetsi nez 1"`

  `f 1` $\rightsquigarrow^*$ **ERROR**

Příklad 2

- ```
  g (a:x)
    | x==[]         = "Almost empty."
    | x/=[]         = "At least 2 members."
    | otherwise     = "Unreachable code."
  g _               = "Nothingness."
  ```
- `g []` $\rightsquigarrow^*$ `"Nothingness."`

  `g "Ahoj"` $\rightsquigarrow^*$ `"At least 2 members."`

Monáda a typ IO

Pozorování

- Při vypisování typů programů pracující s IO můžete narazit na typovou třídu `Monad`, která je zobecněním typu `IO`.
- Například:

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

- Monadické typy jsou v pokročilém Haskellu běžné, ale jsou za hranicí, která je stanovena pro tuto přednášku.
- Pro účely této přednášky je v pořádku otypovat pomocí `IO`:

```
(>>) :: IO a -> IO b -> IO b
```

Wanna more?

- **IB016 Seminář z funkcionálního programování**

Checkpoint

Vstup/výstup

- Napište program, který vyzve uživatele, aby zadal 16 čísel oddělených mezerou, a poté tato čísla vypíše v matici velikosti 4x4.
- Napište program, který ve vhodně formátovaném výstupu (např. rozbalený souborový strom) vypíše obsah aktuálního adresáře včetně všech jeho podúrovní.
- Vstup-výstupní programy vytvářejte jako samostatně spustitelné programy.

USS Haskell