Moduly, balíčky, užitečné nástroje, spustitelné programy

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Úvod

Organizace předmětu, domácí úkoly, podmínky zápočtu

- vizte interaktivní osnovu v ISu
- úkoly jsou již zveřejněné (klidně si je otevřete)
  - technické i jiné problémy hlaste do fóra nebo na konci semináře
  - můžete s námi svá řešení konzultovat emailem nebo na konci semináře (i před odevzdáním)

Softwarové vybavení: překladač GHC + interpret GHCi

- ve verzi alespoň 8.4[1]
- na fakultních počítačích: `$ module add ghc`
- na vlastních počítačích: to zvládnete :-) (ale zkusíme vám pomoci)

[1] oproti v. 9.2 na FI může být potřeba zapínat některá rozšíření explicitně

Moduly

Moduly (`Data.Char`, `Data.Map.Lazy`, ...)

- skupina funkcí ve stejném „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí **`import`**
- Selektivní import

```
import Data.Set (Set, empty, insert)

courses :: Set String
courses = insert "IB016" empty
```

Moduly

Moduly (`Data.Char`, `Data.Map.Lazy`, …)

- skupina funkcí ve stejném „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí **`import`**
- Selektivní import

```
import Data.Set (Set, empty, insert)

courses :: Set String
courses = insert "IB016" empty
```

- Kvalifikovaný import

```
import qualified Data.Set as S

courses :: S.Set String
courses = S.insert "IB016" S.empty
```

Moduly

Balíky (`containers-0.5.10.1`, `brainfuck-0.1.0.3`, ...)

- skupina modulů, která se instaluje spolu
- název většinou malými písmeny, nemá hierarchii, má verzi
- balík base (základní balík modulů)
- spravuje typicky balíčkovací systém `cabal`

Moduly v praxi

Informace o modulech/balících:

- databáze balíků Hackage
  - resp. Stackage („Stable Hackage“)
- vyhledávač Hoogle (hledání podle funkcí, typů, balíků, . . .)
  - fakultní instance: `https://hoogle.fi.muni.cz`

Moduly v praxi

Informace o modulech/balících:

- databáze balíků Hackage
  - resp. Stackage („Stable Hackage“)
- vyhledávač Hoogle (hledání podle funkcí, typů, balíků, ...)
  - fakultní instance: `https://hoogle.fi.muni.cz`

Použití při programování

- nainstalovat balík přes cabal:
  `cabal update && cabal install --lib balík`
- import v kódu: `import Modul`
- přidání modulu v GHCi: přes `import` nebo `:m + Modul`

Cabal

Nástroj pro správu balíků

- aktualizace databáze balíků: `cabal update`
- instalace z Hackage: `cabal install [--lib] balík`
  - Pozor, `cabal uninstall` neexistuje!
- skoro všechno ukládá do `$HOME/.cabal/`
  - do `$PATH` je třeba přidat `$HOME/.cabal/bin`
  - konfiguraci a instalace je možno snadno smazat
  - při mazání je také nutno pročistit adresář `$HOME/.ghc`

Sandboxing

- je nutno založit projekt: `cabal init`
- v souboru `projekt.cabal` nastavit požadované balíky
- `cabal build` stáhne a sestaví jen to, co je potřeba
  - různé projekty sdílí stejné balíky
  - je možno mít více verzí jednoho balíku
- GHCi je možné spustit běžně nebo přes `cabal repl`

Více v dokumentaci Cabalu

Cabal v síti FI

- součástí modulu ghc
- na nymfách lokálně nainstalována zastaralá verze, nutno přebít modulem
- na aise je zapotřebí novější překladač gcc (`module add gcc`)
- kvůli rozdílné verzi systémových knihoven na aise a nymfách:
  - balíky sestavené na nymfách nebudou fungovat na aise
  - balíky sestavené na aise *by měly* fungovat na nymfách
  - **doporučení: instalaci všech balíků provádět na aise**

HLint

Nástroj hlásící návrhy na zlepšení kódu („linter“)

- samostatný balík z Hackage, nutno doinstalovat
  - často dostupný přímo v repozitářích distribuce
  - FI: na nymfách nainstalovaný z repozitářů Ubuntu
  - více podrobností v samostatném návodu v ISu
- `aisa$ cabal install hlint`
- `hlint [--hint soubor_definic_navíc] zdroják`
- možno integrovat přímo do GHCi, vizte dokumentaci

Haddock

```haskell
-- | The 'square' function squares an integer.
square :: Int -> Int
square x = x * x

data T a b
    = C1 a b  -- ^ info about constructor 'C1'
    | C2 a b  -- ^ info about constructor 'C2'
```

- syntaxe komentářů pro automatické zpracování
- generování HTML dokumentace programem haddock
  - `haddock file.hs --html -o doc`
- na FI: součástí modulu ghc
- více informací v oficiální dokumentaci

Samostatně spustitelné programy

S pomocí `IO` lze vytvářet spustitelné programy v Haskellu:

- stačí definovat funkci `main`
- tato funkce se použije jako vstupní bod programu

Spouštění a kompilace:

- kompilace: `ghc File.hs`, vytvoří binárku `File` (nebo `File.exe` na Windows)
- program lze spustit pomocí `./File`

Rychlé spouštění programu:

- `runghc File.hs`, spustí funkci `main` souboru `File.hs`
- `:main` v GHCi,
- v obou případech jde předávat programu argumenty

Vstup a výstup v Haskellu

Běžné funkce v Haskellu nemají vedlejší efekty ani vnitřní stav:

- nesmí modifikovat soubory, vypisovat na obrazovku, ...

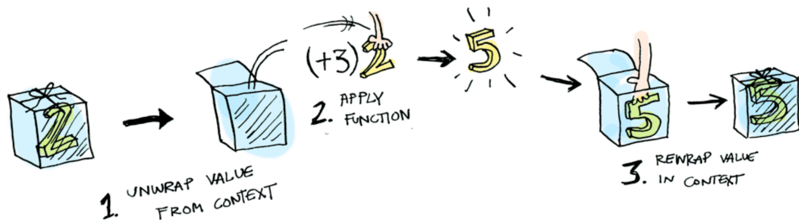
Jak komunikovat se světem (např. načíst vstup)?

Vstup a výstup v Haskellu

- speciální `IO` funkce, mají povoleny vedlejší efekty
  - `putStrLn :: String -> IO ()`
  - `getLine :: IO String`
  - ...
- hodnota typu `IO a`
  - (vstupně-výstupní) **akce**, má **výsledek** typu a
- dva různé způsoby zápisu práce s `IO` funkcemi
  - do-notace
  - bind-operátor (funkce >>= a >>)

IO jako krabice

- k vnitřnímu výsledku IO akcí nelze přistupovat přímo, používáme speciální jazykové konstrukce
- z IO nelze „utéct“
- IO funkce lze jen skládat a lze aplikovat další funkce funkce uvnitř IO



© 2013 Aditya Bhargava, *Functors, Applicatives, And Monads In Pictures*

IO: do-notace

```
echo :: IO ()
echo = do
    putStrLn "Write something."
    line <- getLine
    let out = "You wrote: " ++ line
    putStrLn out
```

- návratovou hodnotu akce `getLine` extrahujeme pomocí `<-`
- blok musí končit akcí, její návratová hodnota je návratovou hodnotou bloku
- `let` uvnitř `do` platí od své definice až po konec bloku
- pozor na zarovnání

Připomenutí: funkce show, read

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

- funkce, která převede svůj argument na řetězec
- možno pouze pro typy ze třídy Show
  - standardní instance pro čísla, seznamy, n-tice, ...

Připomenutí: funkce show, read

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

- funkce, která převede svůj argument na řetězec
- možno pouze pro typy ze třídy Show
  - standardní instance pro čísla, seznamy, n-tice, ...

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

- funkce, která „přečte“ něco z řetězce (parsuje řetězec)
- možno pouze pro typy ze třídy Read
  - standardní instance pro čísla, seznamy, n-tice, ...
- při selhání shodí program s výjimkou `Prelude.read: no parse`
- může být potřeba explicitně uvést typ, který chceme přečíst
  - typ odvozen z kontextu: `read "True" || False`
  - `(read "1")` vs `(read "1" :: Int)`

Přehled užitečných IO funkcí

- `putStr :: String -> IO ()`
  vypíše zadaný řetězec na obrazovku
- `putStrLn :: String -> IO ()`
  vypíše zadaný řetězec na obrazovku a zalomí řádek
- `print :: Show a => a -> IO ()`
  vypíše na výstup hodnotu zobrazitelného typu (typu, který je instancí Show)
- `getLine :: IO String`
  načte ze vstupu řádek (řetězec)
- `pure :: a -> IO a`
  akce, která nemá žádný efekt a vrací zadanou hodnotu
- `sequence :: [IO a] -> IO [a]`
  akce, která postupně spustí všechny zadané akce a vrátí seznam jejich výsledků

Užitečné funkce z modulu Control.Monad

- `when :: Bool -> IO () -> IO ()`
  pokud je podmínka splněná, vrátí zadanou akci, jinak vrátí akci, která nedělá nic
- `unless :: Bool -> IO () -> IO ()`
  opak předchozího
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
  vrátí akci, která každý prvek seznamu předá zadané funkci, spustí výsledné akce a výsledky zahodí
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
  předchozí funkce s přehozenými argumenty
- `forever :: IO a -> IO b`
  akce, která opakuje zadanou akci do nekonečna

A spousta dalších. Dostaneme se k nim v dalších přednáškách.

IO pomocí operátoru >>=

do-notace je syntaktickým cukrem pro tzv. operátory *bind*:[2]

- `(>>) :: IO a -> IO b -> IO b`
  pro řetězení akcí (bez použití výsledku první)
- `(>>=) :: IO a -> (a -> IO b) -> IO b`
  umožňuje přistoupit k výsledku akce z funkce, která vrací akci

```
echo =
    putStrLn "Write something: " >>
    getLine >>= putStrLn . ("You wrote: " ++)
```

[2] Uvedené operátory jsou ve skutečnosti obecnější, jsou definované pro všechny *monády*. Prozatím však můžete jakoukoliv `Monad m` považovat za `IO`. Více o *monádách* během příštích týdnů.

Převod mezi notacemi

do-notace	***bind*-notace**
`do f` `   g`	`f >> g`
`do x <- f` `   g`	`f >>= \x -> g`
`do let x = y` `   f`	`let x = y in f`

Příklad převodu:

```
do putStr ">>> "
   s <- getLine
   let t = reverse s
   putStrLn t
```

```
putStr ">>> " >>
getLine >>= \s ->
let t = reverse s in
putStrLn t
```

Proč tak divně?

`IO` je součást typového systému Haskellu nejen z důvodů dokumentace, co je čistá funkce a co není

Bez `IO` by vstup a výstup v *líném programovacím jazyce* byl poměrně nepříčetný

Co kdyby například existovaly funkce

- `myPutStrLn :: String -> ()`
- `myGetLine :: String`

Samostatné programování

do-notace: příklady

Napište program, který postupně od uživatele načítá řádky textu a vypisuje je pozpátku (např. z `"ahoj svete"` udělá `"etevs joha"`), dokud uživatel nezadá prázdný řádek.

Poté napište podobný program, který neobrací celý řádek, ale jen pořadí slov (např. z `"ahoj svete"` udělá `"svete ahoj"`).

Nejdřív programy napište jen pomocí do-notace. Pak zkuste v dokumentaci nastudovat funkci knihovní funkci `interact` a použít ji.

do-notace: příklady

Definujte akci `getInteger :: IO Integer`, která ze standardního vstupu načte celé číslo. Využijte knihovní funkci `read`.

Poté upravte a doplňte následující zdrojový kód tak, aby program vyžadoval a načetl postupně tři celá čísla a o nich určil, zda mohou být délkami hran trojúhelníku.

Nezdráhejte se kód refaktorovat a vytvořit si další pomocné funkce či akce.

```
main :: IO ()
main = do putStrLn "Enter one number:"
  x <- getInteger
  putStrLn (show (1 + x))
```

do-notace: příklady

Napište program, který postupně od uživatele načítá čísla do té doby, než zadá nulu. Poté program vypíše na obrazovku součet všech zadaných čísel.

do-notace: příklady

Napište akci `getIntegers :: IO [Integer]`, která postupně od uživatele načítá čísla do té doby, než zadá nulu, a vrátí seznam zadaných čísel.

Pomocí této akce přepište předchozí příklad.

Další příklady

Napište program, který od uživatele načte dvě čísla (nazvěme je W a H) a poté na standardní výstup ze znaků # vykreslí obdélník o šířce W a výšce H.

Zkuste variantu, kdy obdélník je vyplněný i kdy není vyplněný.

Zkuste dovolit uživateli mezi vyplněným a nevyplněným obdélníkem přepínat pomocí argumentu příkazové řádky (např. `./drawRectangle filled` a `./drawRectangle empty`). Na to by se vám mohla hodit funkce `getArgs :: IO [String]` z modulu `System.Environment`.

Další příklady

Napište program, který postupně načítá od uživatele příkazy „upper“, „reverse“ a „reverseWords“ následované textem, na který má zadaný příkaz aplikovat. Program po každém řádku příkazy vykovává a vypisuje výsledky. Interakci lze ukončit příkazem „quit“. Např.

```
> upper ahoj svete
AHOJ SVETE
> reverse ahoj svete
etevs joha
> reverseWords
svete ahoj
> quit
```

Zkuste společnou funkcionalitu abstrahovat do separátních funkcí.

Další příklady

Napište program, který funguje jako interaktivní kalkulačka v reverzní polské notaci. Tedy má příkazy „input“, která přidá číslo do zásobníku, a „add“, „sub“, „mul“, které vezmou ze zásobníku dvě čísla, aplikují na nich zadanou operaci a výsledek vrátí do zásobníku. Program po každém řádku příkazy vykovává a vypisuje současný stav zásobníku. Např.

```
> input 5
[5]
> input 3
[3, 5]
> sub
[2]
> input 8
[8, 2]
> input 10
[10, 8, 2]
> mul
[80, 2]
> add
[82]
> quit
```

Příště

- vlastní moduly
- pokročilá syntaxe (case, stráže, typované díry, . . .)
- záznamy
- funktory