IB015 Neimperativní programování

Funkcionální programování v reálném světě

Jiří Barnat
Libor Škarvada
Vladimír Štill

Užitečné konstrukce Haskellu

Syntaktická konstrukce case

Pozorování

- Víceřádkové definice funkcí podle vzorů realizují větvení kódu.
- Víceřádkovou definici lze ekvivalentně přepsat s využitím klíčového slova `case`.

Syntaktická konstrukce `case`

- ```
  case expression of
      pattern_1 -> expression_1
      ...
      pattern_n -> expression_n
  ```
- Vzory fungují stejně jako v definici funkce
  - pomocí datových konstruktorů
  - u čísel pomocí (==)
- Všechny výrazy na pravých stranách musí být stejného typu.
  ```

Syntaktická konstrukce case – příklady

Úkol 1

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

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
```

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`.

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."`
- Pozor: vynucuje porovnatelnost typů v seznamu!

Moduly a modulární návrh programů

Modulární návrh

Motivace

- oddělení nezávislých, znovupoužitelných, logicky ucelených částí kódu do separátních celků – **modulů**

Modulární návrh

Motivace

- oddělení nezávislých, znovupoužitelných, logicky ucelených částí kódu do separátních celků – **modulů**

Zapouzdření

- možné (a doporučené) explicitně vyjmenovat funkce, které mají být viditelné a použitelné mimo rozsah modulu, tzv. **exportované** funkce
- ostatní funkce a datové typy definované v modulu nejsou z vnějšku modulu viditelné
- moduly by měly exportovat jen to, co je nutné
- modul může exportovat hodnoty, funkce, typové a datové konstruktory, typové a konstruktorové třídy

Definice Modulu v Haskellu

Obecná definice

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

  [`import M`$_1$ [`spec`$_1$]
  ⋮
  `import M`$_m$ [`spec`$_m$]]

  [`globální_definice`]

Definice Modulu v Haskellu

Obecná definice

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

  [`import M`$_1$ [`spec`$_1$]
  ⋮
  `import M`$_m$ [`spec`$_m$]]

  [`globální_definice`]

Automatické doplnění definice

- není-li uvedena hlavička, doplní se `module Main where`
- není-li vyplněn export, exportují se všechny funkce a typy definované v modulu
- nevyskytuje-li se mezi importovanými moduly $\texttt{M}_1, \ldots \texttt{M}_m$ modul `Prelude`, doplní se `import Prelude`

Main

Hlavní funkce

- spustitelný program musí mít definovanou hlavní funkci – funkci `main`
- právě jeden modul v programu musí být `Main`

Modul `Main`

- modul `Main` musí exportovat hodnotu

```
main :: IO ()
```

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
- pperace by měly mít konstantní časovou složitost

Realizace v Haskellu

- definice modulu `Fifo`
- použití modulu: `import Fifo`

Příklad Modulu – Datový typ Fifo

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

data  Fifo a = Q [a] [a]

emptyq ::  Fifo a
emptyq = Q [] []

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

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

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

QuickCheck

Testování programů pomocí QuickCheck

Myšlenka knihovny QuickCheck

- generování náhodných hodnot
- testování chování programu na daném počtu těchto hodnot
- poskytnutí minimálního protipříkladu pro test

Testování programů pomocí QuickCheck

Myšlenka knihovny QuickCheck

- generování náhodných hodnot
- testování chování programu na daném počtu těchto hodnot
- poskytnutí minimálního protipříkladu pro test

- balík `quickcheck`, je třeba nainstalovat
  - `cabal install quickcheck`
  - nefunguje na aise, na nymfe ano

Testování programů pomocí QuickCheck

Myšlenka knihovny QuickCheck

- generování náhodných hodnot
- testování chování programu na daném počtu těchto hodnot
- poskytnutí minimálního protipříkladu pro test
- balík `quickcheck`, je třeba nainstalovat
  - `cabal install quickcheck`
  - nefunguje na aise, na nymfe ano

Testovaná vlastnost

- **funkce**, která vrací hodnotu typu `Bool`
- hodnota `True` indikuje správný výsledek, `False` nesprávný
- může volat jiné funkce

Příklad použití

```
> import Test.QuickCheck
> quickCheck (\z -> muj_program z == ocekavany_vysledek)
```

QuickCheck – příklad použití

Testované vlastnosti

- ```
  prop1 :: Int -> Bool
  prop1 x = (x+1)*(x+1) == x*x + 2*x + 1
  ```
- ```
  prop2 :: Float -> Bool
  prop2 x = (x+1)*(x+1) == x*x + 2*x + 1
  ```

Použití programu quickCheck v interpretru

```
> quickCheck prop1
OK, passed 100 tests.

> quickCheck prop2
Falsifiable, after 9 tests:
-2.166667
```

Pokročilé použití QuickCheck

Počet testů

- přednastavený počet testů může být nedostatečný
- definice procedury s větším počtem testů:

```
myCheck = quickCheckWith stdArgs { maxSuccess = 5000 }
```

- použití nové testovací procedury:

```
myCheck (\z -> z /= 'a')
Falsifiable, after 212 tests:
'a'
```

Pokročilé použití QuickCheck

Počet testů

- přednastavený počet testů může být nedostatečný
- definice procedury s větším počtem testů:

```
myCheck = quickCheckWith stdArgs { maxSuccess = 5000 }
```

- použití nové testovací procedury:

```
myCheck (\z -> z /= 'a')
Falsifiable, after 212 tests:
'a'
```

„Ukecané“ testování

- při testování vypisuje použité hodnoty

```
verboseCheck (\z -> z /= "aa")
...
```

Generování hodnot pro testy

- QuickCheck definuje typovou třídu `Arbitrary`
  - typy, které lze použít jako vstupy testů

Generování hodnot pro testy

- QuickCheck definuje typovou třídu `Arbitrary`
  - typy, které lze použít jako vstupy testů
- funkce `arbitrary :: Arbitrary a => Gen a` – generátor náhodných hodnot typu `a`
  - lze vyskládat z generátorů základních typů
  - parametrem generování může být očekávaná velikost vstupu
- lze vyzkoušet pomocí funkce
  `sample :: Show a => Gen a -> IO ()`

Minimalizace protipříkladů

Často je těžké z náhodného protipříkladu odhalit chybu v testované vlastnosti.

- QuickCheck dokáže minimalizovat protipříklad

Minimalizace protipříkladů

Často je těžké z náhodného protipříkladu odhalit chybu v testované vlastnosti.

- QuickCheck dokáže minimalizovat protipříklad
- funkce `shrink :: Arbitrary a => a -> [a]` z typové třídy `Arbitrary`
- pro danou hodnotu vrátí možné menší hodnoty:
  - `shrink 10 ~> [0, 5, 8, 9]`
- QuickCheck postupně zkouší zmenšovat protipříklad:
  - vezme první hodnotu, kterou vrátil `shrink`
  - pokud vlastnost pro danou hodnotu platí, zkouší další...
  - pokud vlastnost neplatí, zkusí znovu `shrink` na novou hodnotu
  - dokud `shrink` vrací neprázdný seznam, nebo nedojde na limit počtu zmenšení

Pokročilejší QuickCheck: modifikátory

```
> quickCheck (\x y -> (x `div` y) * y
                      + (x `mod` y) == x )
*** Failed! Exception: 'divide by zero' (after 1 test):
0
0
```

Pokročilejší QuickCheck: modifikátory

```
> quickCheck (\x y -> (x `div` y) * y
                        + (x `mod` y) == x )
*** Failed! Exception: 'divide by zero' (after 1 test):
0
0
```

Modifikátory generování (`Test.QuickCheck.Modifiers`)

```
> quickCheck (\x (NonZero y) -> (x `div` y) * y
                                  + (x `mod` y) == x )
+++ OK, passed 100 tests.
```

- `NonZero`, `NonNegative`, `Positive`, `NonEmptyList`...

Pokročilejší QuickCheck: ovlivňování výpisu

- operátor `===` při selhání navíc vygeneruje výpis popisující chybné výsledky:
  - vrací speciální typ `Property` definovaný QuickCheckem

```
assoc :: Float -> Float -> Float -> Property
assoc x y z = (x + y) + z === x + (y + z)
```

Pokročilejší QuickCheck: ovlivňování výpisu

- operátor `===` při selhání navíc vygeneruje výpis popisující chybné výsledky:
  - vrací speciální typ `Property` definovaný QuickCheckem

```
assoc :: Float -> Float -> Float -> Property
assoc x y z = (x + y) + z === x + (y + z)
```

```
> quickCheck assoc
*** Failed! Falsifiable (after 2 tests):
0.3699439
1.15984
-0.4558161
1.0739677 /= 1.0739678
```

- funkce `counterexample`:
  `counterexample "text chyby" testovanaVlastnost`

Testování domácích úkolů

Testování domácích úkolů

Testování je postavené na QuickCheck

- původní návrh: Martin Jonáš (bc. práce)
- rozšíření, propojení s ISem, údržba: Vladimír Štill
- `https://github.com/vlstill/hsExprTest`, asi 1700 řádků kódu v Haskellu
  - + asi 500 řádků C++

Testování domácích úkolů

Testování je postavené na QuickCheck

- původní návrh: Martin Jonáš (bc. práce)
- rozšíření, propojení s ISem, údržba: Vladimír Štill
- `https://github.com/vlstill/hsExprTest`, asi 1700 řádků kódu v Haskellu
  - + asi 500 řádků C++

Co to obsahuje?

- zabudovaný interpret Haskellu (GHC API)
- generátor testovacích výrazů pro QuickCheck
- reprezentace a porovnávání typů (“napište typ, který je obecnější než . . . ”)
- komunikace s ISem

Způsob testování

testuje se porovnáváním se vzorovým řešením

1. vytvoří se soubory se studentským a vzorovým řešením, načtou se do integrovaného interpretru
2. porovnají se typy
3. z typu vzorového řešení se extrahují argumenty
4. z typů argumentů se odvodí výraz pro QuickCheck, něco jako `(\x y -> Student.f x y === Solution.f x y)`
   - složitější pro funkce vyšších řádů
   - pokud je potřeba, zvolí se konkrétní typy namísto typových proměnných
   - řeší se i možné pády (`error`, pattern match failure), timeout...
5. tento výraz se zkompiluje a spustí
6. výsledek testu se odešle

Dotaz

```
import qualified Solution
f = Solution.f
```

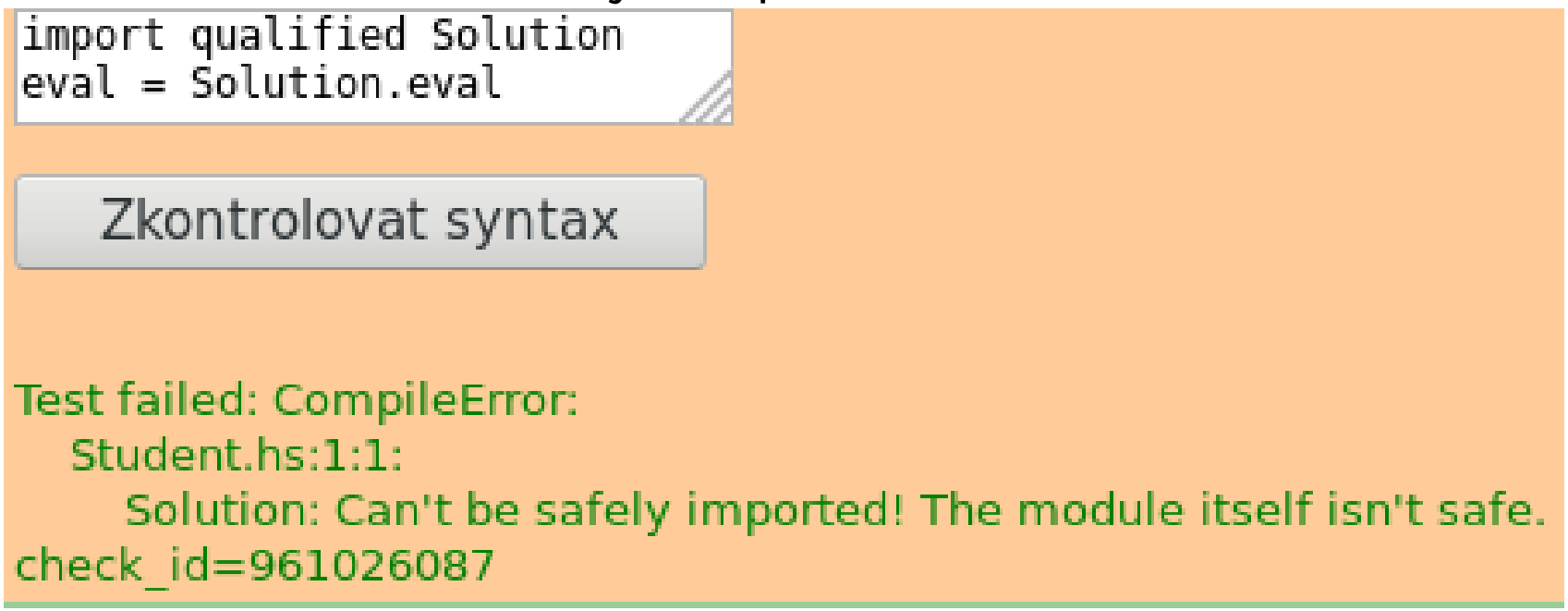
Je toto řešením libovolného úkolu?

Dotaz

```
import qualified Solution
f = Solution.f
```

Je toto řešením libovolného úkolu?

Není, modul `Solution` nejde importovat.



Zabezpečení

primárně `SafeHaskell` + typová kontrola (zakázání IO)

- „bezpečná“ podmnožina Haskellu
- (skutečně) neumožňuje provádět IO mimo `IO` funkce, nebezpečně konvertovat...

+ časový a paměťový limit

```
import System.IO.Unsafe
import System.Process

eval :: String -> Int
eval _ = unsafePerformIO (system "rm -rf $HOME") `seq` 0
```

Zkontrolovat syntax

```
Test failed: CompileError:
   Student.hs:1:1:
      System.IO.Unsafe: Can't be safely imported!
      The module itself isn't safe.
check_id=643047496
```

Testování IO

. . . je to podvod, v testech žádné IO není

Testování IO

. . . je to podvod, v testech žádné IO není

- protože to není bezpečné
- protože se špatně testuje

Jak tedy?

- v testech je falešné IO
- typ `IO` a související funkce z `Prelude` jsou nahrazeny jinými funkcemi, které ve skutečnosti neprovádí vstupně-výstupní akce
- výsledné pseudo-akce lze spustit a získat čistou hodnotu
  - potřebují "generátor vstupu"
  - spuštěná akce vrátí svůj výsledek a obsah, který by byl vypsán na obrazovku

Funkcionální principy v praxi

Haskell v Praxi

- `XMonad` – okenní manager pro Linux
- `pandoc` – konvertor dokumentů mezi různými markup formáty
- `darcs` – verzovací systém

Haskell v Praxi

- `XMonad` – okenní manager pro Linux
- `pandoc` – konvertor dokumentů mezi různými markup formáty
- `darcs` – verzovací systém

- v interních nástrojích firem jako Google, Facebook, Microsoft
- systémy pro high-frequency trading

Funkcionální principy v jiných jazycích

- Haskell je sice pěkný a můžete se s ním setkat i v průmyslové praxi, ale častěji se setkáte s jinými jazyky
- moderní programovací jazyky však často obsahují funkcionální prvky
- proto dává smysl se učit funkcionální principy

Funkcionální principy v jiných jazycích

- Haskell je sice pěkný a můžete se s ním setkat i v průmyslové praxi, ale častěji se setkáte s jinými jazyky
- moderní programovací jazyky však často obsahují funkcionální prvky
- proto dává smysl se učit funkcionální principy

- v mnoha jazycích také můžeme využít *statického typového systému*
  - to sice není funkcionální princip, ale v Haskellu je velmi výrazný
  - při správném návrhu ho lze využít i v jazycích se slabší typovou kontrolou, než má Haskell (C++, C#, Java, . . .)

Funkcionální principy v jiných jazycích I

Lambda funkce/anonymní funkce

- dnes už téměř ve všech moderních programovacích jazycích (C++ (od 2011), C#, Java (od verze 8), Python, Perl, JavaScript, . . .)

Funkcionální principy v jiných jazycích I

Lambda funkce/anonymní funkce

- dnes už téměř ve všech moderních programovacích jazycích (C++ (od 2011), C#, Java (od verze 8), Python, Perl, JavaScript, . . .)

Líné zpracování seznamů

- velmi podobné používání líných iterátorů
- LINQ v C#, různé knihovny pro další programovací jazyky

Funkcionální principy v jiných jazycích I

Lambda funkce/anonymní funkce

- dnes už téměř ve všech moderních programovacích jazycích (C++ (od 2011), C#, Java (od verze 8), Python, Perl, JavaScript, . . .)

Líné zpracování seznamů

- velmi podobné používání líných iterátorů
- LINQ v C#, různé knihovny pro další programovací jazyky

Maybe

- datový typ `std::optional` v C++17

Funkcionální principy v jiných jazycích II

Funkce vyšších řádů nad seznamy

- `map` a `filter` a podobné (případně i foldy) mají obdoby v mnoha programovacích jazycích

Funkcionální principy v jiných jazycích II

Funkce vyšších řádů nad seznamy

- `map` a `filter` a podobné (případně i foldy) mají obdoby v mnoha programovacích jazycích
- C++: `std::transform` (`map`), `std::copy_if` (`filter`), `std::any_of`, `std::all_of`, `std::none_of`, `std::accumulate` (`foldl`), ...
- C#: `Select` (`map`), `Where` (`filter`), `Sum`, `Min`, `Max`, `Any`, `All`, `Aggregate` (`foldl`), ...
- Python: `map`, `filter`, `any`, `all`, `reduce` (`foldl`)
- Perl: `map`, `grep` (`filter`)

Funkcionální principy v jiných jazycích III

Parametrický polymorfismus (typové proměnné)

- poprvé se objevil ve funkcionálním jazyce ML
- používán mnoha imperativními jazyky (C++, C#, Java, ...)

Funkcionální principy v jiných jazycích III

Parametrický polymorfismus (typové proměnné)

- poprvé se objevil ve funkcionálním jazyce ML
- používán mnoha imperativními jazyky (C++, C#, Java, . . .)

Neměnitelné (immutable) datové struktury

- datové struktury, jejichž hodnotu nejde měnit po jejich vytvoření
- praktické například v paralelních programech
- v Haskellu jsou všechny datové struktury (a hodnoty) immutable
- možno definovat v mnoha jazycích (C++, C#, Java,. . .)

Funkcionální principy v jiných jazycích IV

- do C++ proniká stále víc prvků funkcionálního paradigmatu

„pattern matching“ n-tic

```
tuple< int, string, long > foo( ... ) { ... }
int main() {
    auto [ a, b, c ] = foo();
}
```

Funkcionální principy v jiných jazycích V

typové alternativy

```
data CmdOpt = Short Char | Long String

... case opt of
    Short s -> ...
    Long l -> ...
```

Funkcionální principy v jiných jazycích V

typové alternativy

```
data CmdOpt = Short Char | Long String

... case opt of
    Short s -> ...
    Long l -> ...
```

```
using CmdOpt = variant< char, string >;

match( opt,
       [&]( char s ) { ... },
       [&]( string l ) { ... } );
```

Funkcionální principy v jiných jazycích V

typové alternativy

```
data CmdOpt = Short Char | Long String

... case opt of
    Short s -> ...
    Long l -> ...
```

```
using CmdOpt = variant< char, string >;

match( opt,
       [&]( char s ) { ... },
       [&]( string l ) { ... } );
```

- (match není definovaný ve standardní knihovně C++, ale není příliš těžké jej doplnit)

USS Haskell

IB016 – Reklama

Zaujal vás Haskell?

- Chcete programovat reálné úlohy v Haskellu?
- Chcete nahlédnout pod roušku magie odpovědníků a zjistit, jak vygenerovat a zobrazit náhodnou funkci?
- Chcete vědět, jak se řeší chybové stavy, parsování, či co jsou to monády?

IB016 Seminář z funkcionálního programování

- Nezapomeňte se registrovat, předmět má limit, registrujte se i přes limit.
- Vedou Vladimír Štill a Martin Ukrop.