

# IB015 Neimperativní programování

## Funkcionální programování v reálném světě

Jiří Barnat  
Libor Škarvada  
Vladimír Štil

## Užitečné konstrukce Haskellu

## 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`  
    `pattern1 -> expression1`  
    `...`  
    `patternn -> expressionn`
- Vzory fungují stejně jako v definici funkce
  - pomocí datových konstruktorů
  - u čísel pomocí (`==`)
- Všechny výrazy na pravých stranách musí být stejného typu.

## Úkol 1

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

## 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 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`.

## 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
```

## 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`
  - | `guard1 = expression1`
  - ...
  - | `guardn = expressionn`
  - | `otherwise = default expression`

## Pozorování

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



## Příklad 1

- `f x | (x>3) = "vetsi nez 3"`  
    `| (x>2) = "vetsi nez 2"`  
    `| (x>1) = "vetsi nez 1"`
- `f 2 ~>* "vetsi nez 1"`  
  `f 1 ~>* ERROR`

## Příklad 2

- `g (a:x)`  
    `| x==[]           = "Almost empty."`  
    `| x/=[]           = "At least 2 members."`  
    `| otherwise       = "Unreachable code."`  
  `g _                = "Nothingness."`
- `g [] ~>* "Nothingness."`  
  `g "Ahoj" ~>* "At least 2 members."`
- **Pozor: vynucuje porovnatelnost typů v seznamu!**

## Moduly a modulární návrh programů

## Motivace

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

## Zapouzdření

- Při definici modulu je 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.

## Obecná definice

- `[ module Jméno [ (export1, ..., exportn ) ] where ]`  
`[ import M1 [spec1] ]`  
`[       ⋮ ]`  
`[ import Mm [specm] ]`  
`[ 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 M1, ..., Mm modul Prelude, doplní se`  
`import Prelude`

## Hlavní funkce

- 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 ()
```

## Datový typ Fifo

- Datový kontejner (struktura, která uchovává prvky) přístupovaný 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) [])
```

# 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.

## Standardní použití

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

## 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

## 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")
```

```
...
```

- QuickCheck definuje typovou třídu `Arbitrary`, typy hodnot, na kterých se testuje, musí být v této třídě
- 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 ()`

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

Modifikátory generování (Test.QuickCheck.Modifiers)

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

```
>>= 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 je postavené na QuickCheck

- původní návrh Martin Jonáš (bc. práce)
- rozšíření, propojení s ISem, údržba Vladimír Štill
- <https://github.com/vlstill/hsExprTest>, asi 1300 řádků kódu v Haskellu

Testování je postavené na QuickCheck

- původní návrh Martin Jonáš (bc. práce)
- rozšíření, propojení s ISem, údržba Vladimír Štill
- <https://github.com/vlstill/hsExprTest>, asi 1300 řádků kódu v Haskellu

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

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 interpretrem vyhodnotí
- 6 výsledek testu se odešle

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

Je toto řešením libovolného úkolu?

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

Je toto řešením libovolného úkolu?

Není, modul `Solution` nejde importovat.

```
import qualified Solution  
eval = Solution.eval
```

Zkontrolovat syntax

Test failed: **CompileError:**

**Student.hs:1:1:**

**Solution: Can't be safely imported! The module itself isn't safe.**

**check\_id=961026087**

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

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

... 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 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, ...)

## Lambda funkce/anonymní funkce

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

## 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

## 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)

## **Parametrický polymorfismus** (typové proměnné)

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

## **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, ...)





## 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.