Podpůrné nástroje (HLint, Haddock), funktory

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2018

GHC warnings

- GHC při kompilaci může vypisovat kromě chyb i varování
- varování se zapíná argumentem při volaní GHC,
  resp. `:set <flag>` v GHCi
  - `-W` zapíná větší sadu varování
  - `-Wall` zapíná většinu varování
  - `-w` vypíná všechna varování
  - `-Werror` vyhodnocuje varování jako chyby
- úplný přehled varování v sekci dokumentace 4.8 Warnings and sanity checking

GHC warnings

- GHC při kompilaci může vypisovat kromě chyb i varování
- varování se zapíná argumentem při volaní GHC,
  resp. `:set <flag>` v GHCi
  - `-W` zapíná větší sadu varování
  - `-Wall` zapíná většinu varování
  - `-w` vypíná všechna varování
  - `-Werror` vyhodnocuje varování jako chyby
- úplný přehled varování v sekci dokumentace 4.8 Warnings and sanity checking

Standard pro domácí úkoly IB016 je čistý překlad s -Wall!

- možno přidat `:set -Wall` do `~/.ghc/ghci.conf`

Hlint

Nástroj hlásící návrhy na zlepšení kódu („linter“)

- samostatný balík z Hackage, nutno doinstalovat
  - často dostupný přímo v repozitářích distribuce
  - FI PC: nainstalovaný z repozitářů Ubuntu
  - více podrobností v samostatném návodu v ISu
- závisí na generátoru parsrů Happy
- `cabal install happy hlint`
- `hlint [--hint <extra-definice>] <zdrojový-kód>`
- možno integrovat přímo do GHCi, viz návod v ISu
- soubor s extra definicemi v ISu

Dokumentace v Haskellu: 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'
```

- syntax komentářů pro automatické zpracování
- generování HTML dokumentace programem `haddock`
  - `mkdir -p doc && haddock file.hs --html -o doc`
- více info viz oficiální dokumentace

Testování pomocí balíku HUnit

Základní v Haskellu obecně:

- mnoho příspupů i balíků
- dnes `HUnit`, ideově vycházející z `JUnit`
- v průběhu semestru ještě `QuickCheck`

Testování pomocí balíku HUnit

Základní v Haskellu obecně:

- mnoho příspupů i balíků
- dnes `HUnit`, ideově vycházející z `JUnit`
- v průběhu semestru ještě `QuickCheck`

Balík `HUnit` ve zkratce:

- jednoduché unit testování (hierarchie testů)
- v balíku `HUnit`
  instalace: `cabal update && cabal install hunit`
- kompletní dokumentace na Hackage

HUnit: základní syntax

Operátory pro konstrukci testů:

- `(~?) :: (...) => t -> String -> Test`
- `(~=?) :: (...) => a -> a -> Test`
- `(~?=) :: (...) => a -> a -> Test`

HUnit: základní syntax

Operátory pro konstrukci testů:

- `(~?) :: (...) => t -> String -> Test`
- `(~=?) :: (...) => a -> a -> Test`
- `(~?=) :: (...) => a -> a -> Test`

Hierarchie a pojmenovávání testů:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

HUnit: základní syntax

Operátory pro konstrukci testů:

- `(~?) :: (...) => t -> String -> Test`
- `(~=?) :: (...) => a -> a -> Test`
- `(~?=) :: (...) => a -> a -> Test`

Hierarchie a pojmenovávání testů:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

Spuštění testů:

- `runTestTT :: Test -> IO Counts`

HUnit: ukázka

```
module HUnitExample (fact, runTests) where
import Test.HUnit

fact :: Integer -> Integer
fact n = product [1..n]

runTests :: IO Counts
runTests = runTestTT $ TestList [testSet1, testSet2]

testSet1 :: Test
testSet1 = TestLabel "Factorials" $
  TestList [ fact 4 ~?= 25, fact 0 ~?= 1 ]

testSet2 :: Test
testSet2 = TestLabel "Numerical functions" $
  TestList [ even 4 ~? "4 even?", odd 4 ~? "4 odd?" ]
```

Typový konstruktor Either

```
data Either a b = Left a
                | Right b
                deriving (Eq, Ord, Read, Show)
```

- používá se, když může mít výpočet dva typy výsledků
- často se používá jako rozšíření `Maybe`
  - `Left a` označuje chybný výpočet, hodnota specifikuje chybu
  - `Right b` označuje korektní výpočet, hodnota je výsledkem

Typový konstruktor Either

```
data Either a b = Left a
                | Right b
                deriving (Eq, Ord, Read, Show)
```

- používá se, když může mít výpočet dva typy výsledků
- často se používá jako rozšíření `Maybe`
  - `Left a` označuje chybný výpočet, hodnota specifikuje chybu
  - `Right b` označuje korektní výpočet, hodnota je výsledkem

Jaké případy pro použití `Either` vám napadají?

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu $*$

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
Maybe :: * -> *
Either :: * -> * -> *
[] :: * -> *
(,) :: * -> * -> *
```

- opět platí princip částečné aplikace

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
Maybe :: * -> *
Either :: * -> * -> *
[] :: * -> *
(,) :: * -> * -> *
```

- opět platí princip částečné aplikace

```
Either String :: * -> *
```

- GHCi definuje povel `:k` na určení druhu

Motivace: map

- Funkce `map` na seznamech:

```
data [a] = [] | a : [a]

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = fx : map f xs
```

Motivace: map

- Funkce `map` na seznamech:

```
data [a] = [] | a : [a]

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = fx : map f xs
```

- Funkce `map` na binárních stromech:

```
data BinTree a = BNode a (BinTree a) (BinTree a)
               | BEmpty

treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ BEmpty = BEmpty
treeMap f (BNode v l r) =
    BNode (f v) (treeMap f l) (treeMap f r)
```

Motivace: map

- Funkce `map` na seznamech:

```
data [a] = [] | a : [a]

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = fx : map f xs
```

- Funkce `map` na binárních stromech:

```
data BinTree a = BNode a (BinTree a) (BinTree a)
               | BEmpty

treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ BEmpty = BEmpty
treeMap f (BNode v l r) =
    BNode (f v) (treeMap f l) (treeMap f r)
```

- Nedalo by se to zobecnit? Jaký je obecný typ funkce `map`?

Typová třída Functor

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- definováno v modulu Data.Functor
- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
  - instance pro `[]`, `BinTree`, `Maybe`
  - ne pro konkrétní typy (`[String]`, `BinTree a`, Maybe Int)

Typová třída Functor

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- definováno v modulu Data.Functor
- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
  - instance pro `[]`, `BinTree`, `Maybe`
  - ne pro konkrétní typy (`[String]`, `BinTree a`, Maybe Int)
- jiný pohled: funktory tvoří kontext/kontejner pro typy (obalují je další strukturou)

Instance třídy Functor I.

- `instance Functor [] where`

Instance třídy Functor I.

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b]
      fmap = map
  ```

- ```
  instance Functor BinTree where
  ```

Instance třídy Functor I.

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b]
      fmap = map
  ```

- ```
  instance Functor BinTree where
      fmap :: (a -> b) -> BinTree a -> BinTree b
      fmap = treeMap
  ```

- ```
  instance Functor Maybe where
  ```
  ```

Instance třídy Functor I.

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b]
      fmap = map
  ```

- ```
  instance Functor BinTree where
      fmap :: (a -> b) -> BinTree a -> BinTree b
      fmap = treeMap
  ```

- ```
  instance Functor Maybe where
      fmap :: (a -> b) -> Maybe a -> Maybe b
      fmap f (Just x) = Just (f x)
      fmap f Nothing = Nothing
  ```
  ```

Instance třídy Functor II.

- `instance Functor IO where`

Instance třídy Functor II.

```
instance Functor IO where
    fmap :: (a -> b) -> IO a -> IO b
    fmap f action = do
        result <- action
        return (f result)
```

Instance třídy Functor II.

- ```
  instance Functor IO where
      fmap :: (a -> b) -> IO a -> IO b
      fmap f action = do
          result <- action
          return (f result)
  ```

- `Either` je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci na jeden argument.

  ```
  Either e :: * -> *

  instance Functor (Either e) where
  ```
  ```

Instance třídy Functor II.

- ```
  instance Functor IO where
      fmap :: (a -> b) -> IO a -> IO b
      fmap f action = do
          result <- action
          return (f result)
  ```

- `Either` je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci na jeden argument.

  ```
  Either e :: * -> *

  instance Functor (Either e) where
      fmap :: (a -> b) -> Either e a -> Either e b
      fmap f (Right x) = Right (f x)
      fmap f (Left x) = Left x
  ```
  ```

Instance třídy Functor III.

Funkce je vlastně použití binárního typového konstruktoru `(->)`

```
(->) :: * -> * -> *
```

Instance třídy Functor III.

Funkce je vlastně použití binárního typového konstruktoru `(->)`

```
(->) :: * -> * -> *
```

Můžeme tedy zadefinovat instanci pro její částečnou aplikaci na jeden argument.
`(->) r :: * -> *` (tedy „funkce z `r`“)

Instance třídy Functor III.

Funkce je vlastně použití binárního typového konstruktoru `(->)`

```
(->) :: * -> * -> *
```

Můžeme tedy zadefinovat instanci pro její částečnou aplikaci na jeden argument.
`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
```

Instance třídy Functor III.

Funkce je vlastně použití binárního typového konstruktoru `(->)`

```
(->) :: * -> * -> *
```

Můžeme tedy zadefinovat instanci pro její částečnou aplikaci na jeden argument.
`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
    fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap f g = (\x -> f (g x))
```

Pravidla pro třídu Functor

Pro instance třídy `Functor` by mělo platit:

- `fmap id` $\equiv$ `id`
- `fmap (f . g)` $\equiv$ `fmap f . fmap g`

Pravidla pro třídu Functor

Pro instance třídy `Functor` by mělo platit:

- `fmap id` $\equiv$ `id`
- `fmap (f . g)` $\equiv$ `fmap f . fmap g`

Pravidla musí platit!

- kompilátor se spoléhá na výše uvedená pravidla
- jejich platnost musí ověřit programátor (!)
- pro všechny knihovní instance platí

Úkol: instance třídy Functor

Uvažme následující „nové“ datové typy pro mapy a dvojice:

```
newtype MyMap k v = Mp { unMp :: Map k v }
newtype Pair a b = Pr { unPr :: (a, b) }
```

1. Napište instanci pro třídu `Functor` pro tyto struktury. (Jak bude fungovat funkce `map`?)
2. Zamyslete se nad platností pravidel (!) třídy `Functor`.

Úkol: Hlint, Hunit, Haddock

1. Nainstalujte si doplňková pravidla pro HLint a zkontrolujte nimi své řešení z minulé hodiny.
2. Do svého řešení z minulé hodiny doplňte pár testů využívajících knihovnu HUnit.
3. Do svého řešení z minulé hodiny doplňte správně formátované komentáře a vygenerujte k němu dokumentaci pomocí systému Haddock.