Vlastní datové typy a aliasy, typové třídy, `Map`, `Set`; podpůrné nástroje Cabal, Haddock a HLint

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2016

Vlastní datové typy: opakování

```
data BinTree a = BNode a (BinTree a) (BinTree a)
               | BEmpty
               deriving (Eq, Read, Show)
```

- definuje parametrizovaný typ binárního stromu
- unární typový konstruktor `BinTree`, ternární dat. konstruktor `BNode :: a -> BinTree a -> BinTree a -> BinTree a`, nulární datový konstruktor `BEmpty :: BinTree a`
- **`deriving`**: odvození instancí (některých typových tříd)
- na získávání hodnot datového typu je možné používat vzory

```
isEmpty BEmpty = True
isEmpty _      = False

btValue (BNode v _ _) = v
btLeft  (BNode _ l _) = l
btRight (BNode _ _ r) = r
```

Vlastní datové typy: záznamy

V definici typu definujeme i funkce pro přístup k hodnotám.

```
data BinTree a = BNode { btValue :: a
                       , btLeft  :: BinTree a
                       , btRight :: BinTree a
                       }
               | BEmpty
               deriving (Eq, Read, Show)
```

- názvy hodnot je možné použít ve vzorech i jako funkce

```
foo1 (BNode { btValue = val, btLeft = l }) = ...
foo2 node = btValue node + ...
```

- je možné modifikovat i podmnožinu atributů záznamu

```
const1tree node = node { btValue = 1 }
```

Typové aliasy

```
type Matrix a = [[a]]
type String = [Char]
```

jen nové pojmenování, lze zaměňovat s původním typem

- až na drobné výjimky: instance typových tříd
- funguje například `map (map (+4)) matrix`, kde `matrix :: Matrix Int`
- bez **`deriving`**

newtype

```
newtype Matrix a = M { unM :: [[a]] } deriving Show
```

v podstatě definice nového typu (jako **data**), ale musí mít právě jeden unární datový konstruktor

- typově rozlišitelné
- `M (map (map (+4)) (unM matrix))`
- rychlejší než **data**, další rozdíly s rozšířeními GHC[1]

[1] nad rámec kurzu: `-XGeneralizedNewtypeDeriving`

Typové třídy: opakování

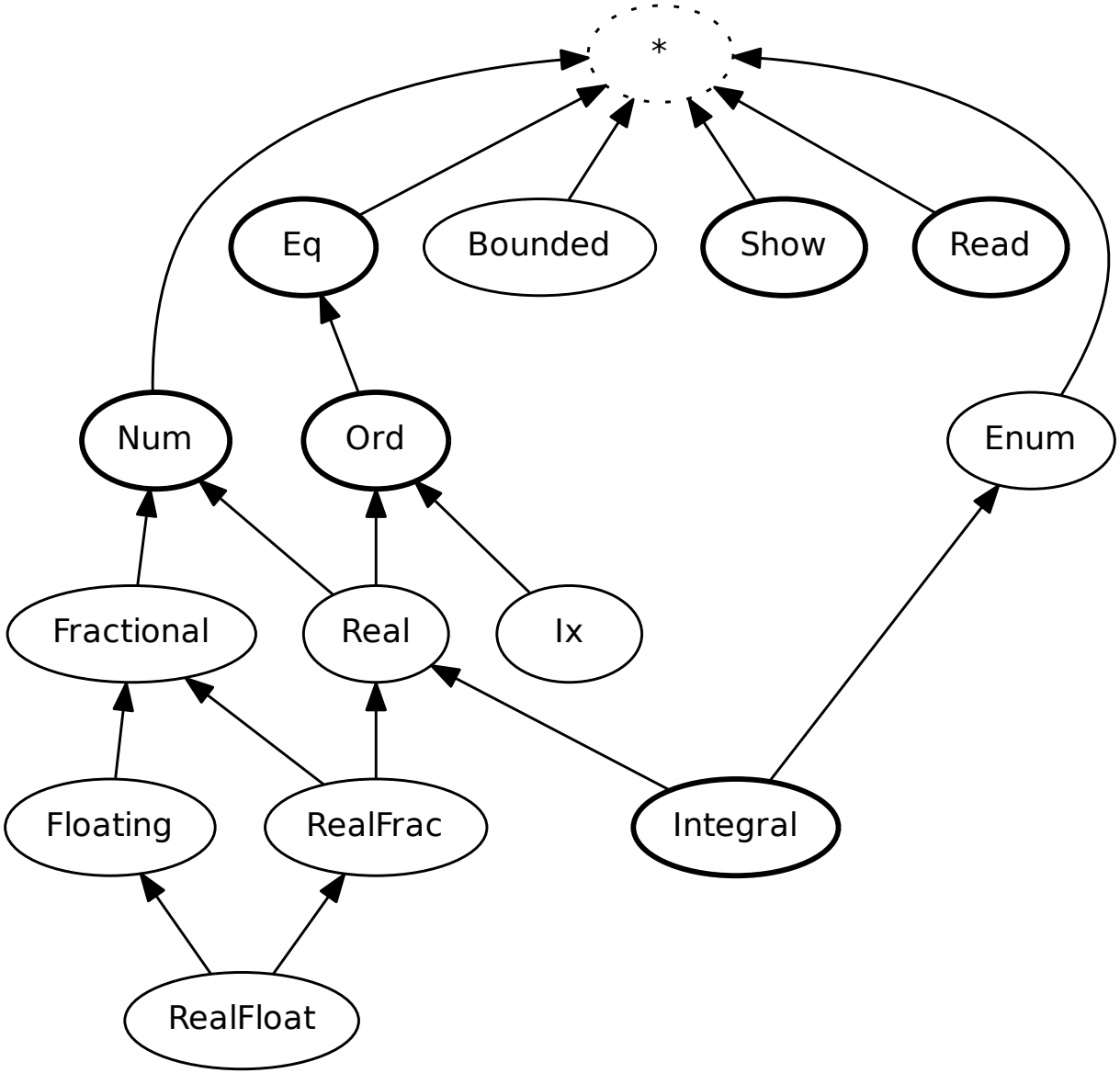
```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    a /= b = not (a == b)
```

definuje funkce, jejichž implementace se liší podle typu (přetěžování, v tomto případě podle typu a)

```
data Test a = A | B a
instance Eq a => Eq (Test a) where
    A     == A     = True
    (B x) == (B y) = x == y
    _     == _     = False
```

instance typové třídy pro daný datový typ

Typové třídy v `Prelude` (některé)



Typové třídy: `deriving`

Automatické odvození instance typové třídy

- jen některé zabudované:
  `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`
- podmínky kdy funguje a implementace závisí na typové třídě
- `Ord`: nejprve porovnání konstruktorů (podle pořadí v definici), pro stejné konstruktory lexikografické porovnání hodnot v nich

Datové typy `Map` a `Set`

asociativní mapy a množiny

- moduly `Data.Map` a `Data.Set` součástí běžné distribuce GHC (balík `containers`)
- `Map k v`
  - `k` je typ klíče, musí být `Ord`
  - `v` je typ hodnoty
- `Set a`, `a` musí být `Ord`
- logaritmické vkládání, odstraňování, zjišťování minima a maxima

Datové typy Map a Set – ukázka

- Set, Map definují mnoho funkcí se stejným názvem jako Prelude

```
import Data.Map (Map)
import qualified Data.Map as Map


foo :: Integral i => Map k i -> Map k i
foo = Map.filter even
```

tečka musí být bez mezery (přístup k funkci v modulu)

- prázdná mapa

```
let mapa1 = empty :: Map k a
```

- přidání prvků

```
let mapa2 = insert 5 "Karel" mapa1
```

Cabal

nástroj pro správu balíků v Haskellu

- součást Haskell Platform, nebo samostatný balík
- `cabal install <balík>` instaluje z Hackage
- FI PC: problém s knihovnama, viz návod v ISu
- FI PC: nainstalovaný Cabal není nejaktuálnější, pokud budete instalovat novou verzi, postupujte podle návodu v ISu

Cabal: vytváření balíků

1. vytvořit moduly, včetně všech potřebných importů (kvůli závislostem)
2. `cabal init` v kořenovém adresáři projektu
3. vyplnit dotazované, specielně, zda se jedná o knihovnu nebo o binárku
4. vytvoří soubory `jmeno_projektu.cabal`, `Setup.hs` a `LICENSE`
   - `.cabal` obsahuje definci projektu, možné měnit závislosti, obsažené moduly a další...
   - zbytek většinou nepodstatný
5. případně relaxovat požadované závislosti

více na `https://www.haskell.org/cabal/users-guide/developing-packages.html`

Dokumentace v Haskellu: Haddock

speciální formát komentářů pro dokumentační účely a generátor dokumentace

- program `haddock` je součástí distribuce GHC
- komentáře se speciálním významem:

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

data T a b
    = C1 a b  -- ^ the doc. for the 'C1' constr.
    | C2 a b  -- ^ the doc. for the 'C2' constr.
```

- více info viz oficiální dokumentace
- `mkdir -p haddock`
  `haddock file.hs --html -o haddock`
  vygeneruje dokumentaci do podsložky `haddock`

Hlint

nástroj hlásící návrhy na zlepšení kódu

- samostatný balík z Hackage, nutno doinstalovat
  - často dostupný přímo v repozitářích distribuce
  - FI PC: instalovaný lokálně
  - více 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

Cabal: úkol

1. s pomocí Hayoo zjistěte v jakém balíku se nachází modul `Data.Default`
2. tento balík si nainstalujte cabalem
3. implementujte nějakou instanci typové třídy `Default` pro datový typ

```
data BinTree a = BNode a (BinTree a) (BinTree a)
               | BEmpty
               deriving (Eq, Show, Read)
```

Práce s datovými typy: úkol

1. vytvořte vhodný typ pro telefonní databázi
   pro každého člověka si chcete pamatovat jméno, telefonní číslo a město
2. zadefinujte základní funkce pro práci s databází:
   zobrazení, vyhledávání, vkládání, mazání
   praktické jsou také funkce, které načtou/vypíší celou databázi ze/do seznamu