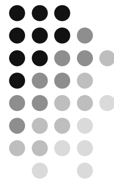
Uživatelem definované typy

Ing. Lumír Návrat
✉ katedra informatiky, A 1018
☎ 59 732 3252



Definice uživatelského typu

- data Color = Red | Green | Blue
  - Color – typový *konstruktor*
  - Red / Green / Blue – datové konstruktory
- data Point = Point Float Float
  - `dist (Point x1 y1) (Point x2 y2) = sqrt ((x2-x1)**2 + (y2-y1)**2)`
  - *dist (Point 1.0 2.0) (Point 4.0 5.0) = 5.0*
- data Point a = Point a a
  - polymorfismus
  - konstruktor `Point :: a -> a -> Point a`

Součinové datové typy

- data Point = Point Int Int
  - existuje jen jediná varianta
  - typová kontrola v době překladu
  - žádná kontrola v době běhu
- data Dvojice a b = Dvojice a b
  type Dvojice a b = (a,b)
  - izomorfní s uspořádanými n-ticemi

Součtové datové typy

- data Color = Red | Green | Blue
  - existuje více variant
  - každá varianta je součinovým typem
  - nutnost kontroly v době běhu

```
isRed :: Color -> Bool
isRed Red = True
```

- může nastat chyba: isRed Blue = ???

Rekurzivní datové typy

Seznam

```
data List a = Null
            | Cons a (List a)

lst :: List Int
  lst = Cons 1 (Cons 2 (Cons 3 Null))

append Null        ys = ys
  append (Cons x xs) ys =
                 Cons x (append xs ys)
```


Rekurzivní datové typy

Strom

```
data Tree1 a = Leaf a
             | Branch (Tree1 a) (Tree1 a)
data Tree2 a = Leaf a
             | Branch a (Tree2 a) (Tree2 a)
data Tree3 a = Null
             | Branch a (Tree3 a) (Tree3 a)

t2l (Leaf x) = [x]
  t2l (Branch lt rt) = (t2l lt) ++ (t2l rt)
```


Synonyma datových typů

- `type String = [Char]`

```
type Name = String
data Address = None | Addr String
type Person = (Name, Address)

type Table a = [(String, a)]
```

- jsou ekvivalentní původním typům
- představují pouze zkratky

Zavedení nového typu

- newtype Natural = MakeNatural Int
  - není ekvivalentní původnímu typu Int
  - vyžaduje explicitní konverzní funkce

```
toNatural :: Int -> Natural
toNatural x | x < 0 = error "Chyba"
            | otherwise = MakeNatural x

fromNatural :: Natural -> Int
fromNatural (MakeNatural x) = x
```

Pojmenované složky typů

- Selektor – vrací složku hodnoty
  - ```
    data Point = Pt Float Float
    px :: Point -> Float
    px (Pt x y) = x
    ```
- Pojmenované složky = selektory
  - ```
    data Point = Pt { px, py :: Float }

    abs (Pt px=x, py=y) = sqrt (x*x+y*y)
    abs p = sqrt ((px p)**2 + (py p)**2)
    ```
- Vytvoření modifikované kopie
  - `p { px = x }`

Motivace

- Funkce elemBool
  - elemBool :: Bool -> [Bool] ->Bool
  - elemBool x [] = false
  - elemBool x (y:ys) = (x $==_{Bool}$ y) || elemBool x ys
- Funkce elemInt
  - elemInt :: Int -> [Int] ->Bool
- Funkce elemGen
  - elemGen :: (a -> a ->Bool) -> a -> [a] -> Bool
  - elemGen ($==_{Bool}$)

Řešení ?

- Funkce elem
  - elem :: a -> [a] -> Bool
  - Výhody
    - Znovupoužitelnost
    - Adaptibilita
  - Nevýhody
    - Jak omezit na porovnatelné typy?
      elem sin [sin,cos,tg] -> co je výsledek?
  - Řešení
    - elem :: Eq a => a -> [a] -> Bool

Typové třídy

- Kontext (C_1 a, C_2 b, …), resp. C a
  - omezení na typové proměnné a, b, ...
  - C_1 , C_2 ... jsou *typové třídy*
- Typ ∀***u*** . *cx* => *t*
  - ***u*** – typové proměnné
  - *cx* – kontext
  - t – typový výraz
  - Př.:
    ```
    Eq a          => a -> [a] -> Bool
    (Eq a, Num b) => (a, b) -> [a]
    ```

Definice typové třídy

- Definice signatury metod a implicitní metody
  - ```
    class Eq a where
        (==), (/=) :: a -> a -> Bool
        x == y = not (x /= y)
        x /= y   = not (x == y)
    ```
- Dědičnost
  - ```
    class (Eq a) => Ord a where
        (<),(<=),(>),(>=) :: a -> a -> Bool
        max, min  :: a -> a -> a
    ```
  - `class (Read a, Show a) => Textual a`

Definice instance typové třídy

```
instance Eq Int where
    x == y  = intEq x y

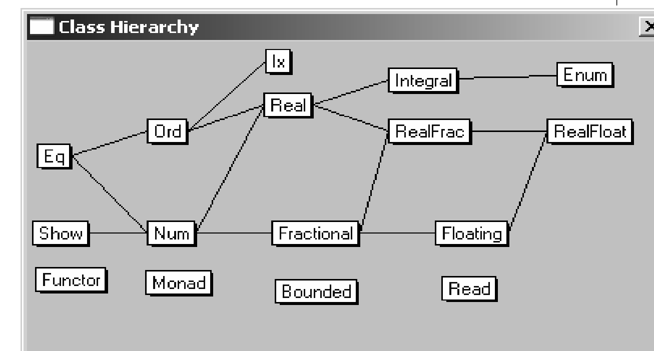
instance Eq a => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _      == _      = False

instance (Eq q, Eq b) => Eq (a,b) where
    (x1,y1) == (x2,y2) = x1==x2 && y1==y2
```


Základní typové třídy

Eq a	(==), (/=)
Eq a => Ord a	(<), (<=), (>), (>=), min, max
Enum a	succ, pred
Read a	readsPrec
Show a	showsPres, show
(Eq a, Show a) => Num a	(+), (-), (*), negate, abs
(Num a) => Fractional a	(/), recip
(Fractional a) => Floating a	pi, exp, log, sqrt, (**), …

Hierarchie tříd



Třída Show

- Hodnoty převoditelné na řetězec
  - ```
    type  ShowS  = String -> String
    class Show a where
      showsPrec  :: Int -> a -> ShowS
      show       :: a -> String
      showList   :: [a] -> ShowS
    ```
  - ```
    showPoint :: Point -> String
    showPoint (Pt x,y) =
      "(" ++ show x ++ ";" ++ show y ++ ")"
    ```
  - ```
    instance Show Point where
       show p = showPoint p
    ```


Třída Read

- Hodnoty převoditelné z řetězce
  - ```
    type ReadS a = String -> [(a,String)]
    class Read a where
        readsPrec :: Int -> ReadS a
        readList  :: ReadS [a]
    ```
  - ```
    readsPoint :: ReadS Point
    readsPoint ('(':s) =
       [ (Pt x y, s'') |
            (x, ';':s') <- reads s,
            (y, ')':s;;) <- reads s']
    ```
  - ```
    instance Read Point where
       readsPrec _ = readsPoint
    ```


Moduly

- Definice modulu
  - ```
    module A where    -- A.hs, A.lhs
    ...
    ```
  - všechny definice jsou viditelné
- Import modulu
  - ```
    module A where
    import B
    ...
    ```
  - dovezou se všechny viditelné definice

Moduly

- Omezení exportu
  - **module Expr (printExpr, Expr(..)) where**
  - **Expr(..)** – exportuje i konstruktory
Expr – pouze export jména typu
- Omezení dovozu
  - **import Expr hiding(printExpr)**
  - **import qualified Expr -- Expr.printExpr**
  - **import Expr as Vyraz -- Vyraz.printExpr**

Motivace

- ***Cíl: Vytvoření kalkulátoru pro číselné výrazy***
  - čtení a zápis hodnot proměnných
- **Modely zapisovatelné paměti**
  - seznam dvojic: `[(String, Int)]`
  - funkce: `(String -> Int)`
- Operace
  - `initial :: Store`
  - `value   :: Store -> String -> Int`
  - `update  :: Store -> String -> Int -> Store`

Abstraktní datový typ

- Omezené rozhraní
  - `type Store = [(Int, Var)]`
    `s1, s2:: Store` -> Co je to `s1++s2` ?
- Nezávislost rozhraní na implementaci
  - `type Store = Var -> Int`
    `s1, s2:: Store` -> `s1++s2` nefunguje
- Podmínky pro ADT
  - definice veřejného rozhraní
  - implementace typu není dostupná

ADT a Haskell

- ADT je definován v samostatném modulu
- Explicitní export operací
- Export pouze jména typu bez konstruktorů

```
module Store
  (Store,
   initial, -- Store
   value,         -- Store -> String -> Int
   update         -- Store -> String -> Int -> Store
  ) where
```


Implementace - seznam

- Definice typu

```
data    Store = Sto [(String, Int)]
newtype Store = Sto [(String, Int)]
```

- Definice operací

```
initial :: Store
initial = Sto []

value :: Store -> String -> Int
value (Sto []) _ = 0
value (Sto ((w,n):sto)) v
   | v == w    = n
   | otherwise = value (Sto sto) v

update :: Store -> String -> Int -> Store
update (Sto sto) v n = Sto ((v,n):sto)
```

Implementace - funkce

- Definice typu

```
newtype Store = Sto (String -> Int)
```

- Definice operací

```
initial :: Store
initial = Sto (\v -> 0)

value :: Store -> String -> Int
value (Sto sto) v = sto v

update :: Store -> String -> Int -> Store
update (Sto sto) v n
  = Sto (\w -> if v==w then n else sto w)
```


Obecné principy návrhu ADT

- Identifikace a pojmenování typů
- Neformální popis typů
- Signatury typů
  - Jak vytvořit objekt daného typu?
  - Jak zjistit, o jaký druh objektu jde?
  - Jak se dostaneme ke složkám typu?
  - Můžeme provádět transformace objektů?
  - Jak můžeme objekty kombinovat?
  - Můžeme objekty agregovat?

ADT Fronta

- **inicializace: emptyQ**
- **test prázdné fronty: isEmptyQ**
- **vložení na konec fronty: addQ**
- **výběr prvku ze začátku fronty: remQ**

```
module Queue
  ( Queue,
    emptyQ,-- Queue a
    isEmptyQ,    -- Queue a -> Bool
    addQ,        -- a -> Queue a -> Queue a
    remQ         -- Queue q -> (a, Queue a)
  ) where
```


Implementace fronty 1

```
-- vkládáme na konec, vybíráme ze začátku seznamu
newtype Queue a = Qu [a]

emptyQ = Qu []
isEmptyQ (Qu q) = empty q

addQ x (Qu xs) = Qu (xs++[x])

remQ q@(Qu xs)
  | not (isEmptyQ q) = (head xs, Qu (tail xs))
  | otherwise        = error "remQ"
```

Implementace fronty 2

```
-- vkládáme na začátek, vybíráme z konce seznamu

addQ x (Qu xs) = Qu (x:xs)

remQ q@(Qu xs)
  | not (isEmptyQ q) = (last xs, Qu (init xs))
  | otherwise        = error "remQ"
```

- složitost addQ/remQ
  - konstantní – na začátku seznamu
  - lineární – na konci seznamu

Implementace fronty 3

```
newtype Queue a = Qu [a] [a]

emptyQ = Qu [] []
isEmptyQ (Qu [] []) = True
isEmptyQ _          = False

addQ x (Qu xs ys) = Qu xs (x:ys)

remQ (Qu (x:xs) ys)  = (x, Qu xs ys)
remQ (Qu []     ys)  = remQ (Qu (reverse ys) [])
remQ (Qu []     [])  = error "remQ"
```


Příklady pro cvičení

1) Vyhledání maximální hodnoty ve stromu
  - `maxTree :: Ord a => Tree a -> a`

2) Reprezentace aritmetického výrazu uživatelským datovým typem
  - ```
    data Expr = Plus Expr Expr
              | …
                Num Int
    ```

3) Vyhodnocení výrazu
  - `eval :: Expr -> Int`

4) Převod výrazu na řetězec
  - `instance Show Expr where …`
    ```