Abstraktní datové typy

doc. Dr. Ing. Miroslav Beneš
katedra informatiky, A-1007
☎ 59 732 4213

Obsah přednášky

- Motivace, vlastnosti ADT
- Moduly v Haskellu
- Obecné principy návrhu ADT
- ADT Fronta
- ADT Vyhledávací strom
- ADT Množina
- ADT Relace
- Příklady na procvičení

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

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`

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"
```


ADT Vyhledávací strom

```
module Tree
  ( Tree,
    nil,        -- Tree a
    isNil,      -- Tree a -> Bool
    isNode,     -- Tree a -> Bool
    leftSub,    -- Tree a -> Tree a
    rightSub,   -- Tree a -> Tree a
    treeVal,    -- Tree a -> a
    insTree,    -- Ord a => a -> Tree a -> Tree a
    delete,     -- Ord a => a -> Tree a -> Tree a
    minTree     -- Ord a => Tree a -> Maybe a
  ) where
```


ADT Vyhledávací strom

```
-- definice typu
data Tree a = Nil | Node a (Tree a) (Tree a)
-- konstrukce hodnoty
nil :: Tree a
nil = nil
-- dotaz na druh objektu
isNil :: Tree a -> Bool
isNil Nil = True
isNil _   = False
-- výběr složky objektu
leftSub :: Tree a -> Tree a
leftSub Nil              = error "leftSub"
leftSub (Node _ t1 _) = t1
```

ADT Vyhledávací strom

```
-- vložení prvku do stromu
insTree :: Ord a => a -> Tree a -> Tree a

insTree val Nil = (Node val Nil Nil)
insTree val (Node v t1 t2)
  | v == val  = Node v t1 t2
  | val > v   = Node v t1 (insTree val t2)
  | val < v   = Node v (insTree val t1) t2
```


ADT Vyhledávací strom

```
-- vyhledání nejmenšího prvku ve stromu
minTree :: Ord a => Tree a -> Maybe a

minTree t
  | isNil t   = Nothing
  | isNil t1  = Just v
  | otherwise = minTree t1
  where t1 = leftSub t
        v  = treeVal t

data Maybe a = Nothing | Just a
```


ADT Vyhledávací strom

```
-- odstranění prvku ze stromu
delete :: Ord a => a -> Tree a -> Tree a

delete val (Node v t1 t2)
  | val < v   = Node v (delete val t1) t2
  | val > v   = Node v t1 (delete val t2)
  | isNil t2  = t1
  | isNil t1  = t2
  | otherwise = join t1 t2

join :: Ord a => Tree a -> Tree a -> Tree a
join t1 t2 = Node mini t1 newt
             where (Just mini) = minTree t2
                   newt = delete mini t2
```


ADT Množina

- Seznam – uspořádaná posloupnost
- Množina – bez uspořádání, bez opakování

```
module Set
  ( Set,
    empty,          -- Set a
    incl,           -- Eq a => a -> Set a -> Set a
    makeSet,        -- Eq a => [a] -> Set a
    isEmpty,        -- Set a -> Bool
    inSet,          -- Eq a => Set a -> a -> Bool
    union, inter, diff,
                    -- Eq a => Set a -> Set a -> Set a
    eqSet, subSet,-- Eq a => Set a -> Set a -> Bool
    card            -- Set a -> Int
  ) where
```

ADT Množina

- Implementace
  - seznam neopakujících se prvků
    - **`newtype Set a = St [a]`**
  - seznam uspořádaných prvků
    - lineární složitost vyhledávání
  - vyhledávací strom
    - logaritmická složitost vyhledávání
  - ...

ADT Množina

```
-- zakrytí funkce union
import List hiding (union)

-- deklarace instance třídy Eq
instance Eq a => Eq (Set a) where
   (==) = eqSet

-- definice typu
newtype Set a = St [a]
```


ADT Množina

```
-- konstruktory
empty :: Set a
empty = St []

incl :: Eq a => a -> Set a -> Set a
incl x (St st) = St (incl1 x st)
  where incl1 x [] = [x]
        incl1 x s@(y:ys)
          | x == y    = s
          | otherwise = y : (incl1 x ys)

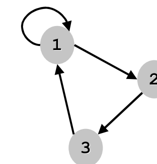
makeSet :: Eq a => [a] -> Set a
makeSet [] = empty
makeSet (x:xs) = incl x (makeSet xs)
```


ADT Relace

- *Binární relace na množině A: podmnožina kartézského součinu (A x A)*
- *Množina uspořádaných dvojic (x,y)∈(AxA)*

  `type Relation a = Set (a,a)`

	1	2	3
1	X	X	
2			X
3	X		



ADT Relace

- Kompozice relací
  - $(x1, x2) \in R1, (x2, x3) \in R2 \to (x1,x3) \in R2 \bullet R1$
  - ```
    compose r2 r1 =
      [(x1,x3) | (x1, x2) <- r1,
                 (x2',x3) <- r2,
                x2 == x2']
    ```
- Mocnina relace
  - $r^1 = r, r^{n+1} = r \bullet r^n$
  - ```
    pow 1     r = r
    pow (n+1) r = compose r (pow n r)
    ```


ADT Relace

- Vlastnosti relace $R \subseteq U \times U$
  - reflexivita:
    $\forall x \in U . (x,x) \in R$
  - symetrie
    $\forall x,y . (x,y) \in R \Rightarrow (y,x) \in R$
  - tranzitivita
    $\forall x,y,z . (x,y) \in R \wedge (y,z) \in R \Rightarrow (x,z) \in R$
- Pozor!
  - ```
    isReflexive :: Eq a => Rel a => Set a => Bool
    isReflexive r u = and [inSet (x,x) r | x <- u]
    ```


Úkoly na procvičení

- Definujte abstraktní datový typ Stack reprezentující zásobník s operacemi
  - `push :: a -> Stack a -> Stack a`
  - `pop :: Stack a -> Stack a`
  - `top :: Stack a -> a`
  - `isEmpty :: Stack a ->Bool`
- Definujte abstraktní datový typ Deque reprezentující oboustranně ukončenou frontu s operacemi
  - addFirst, addLast
  - removeFirst, removeLast
  - …
    ```