Abstraktní datové struktury a rekurzivní datové typy

Michal Vašinek

*VSB-TU Ostrava, 17. listopadu 15/2172, Ostrava, 708 33, Czech Republic
michal.vasinek@vsb.cz

2. prosince 2019

0.1 Algebraické datové typy

Klíčovým slovem **data** říkáme, že vytváříme nový datový typ. Například datový typ **Bool** lze definovat jako:

```
data Bool = False | True
```

Podobně můžeme definovat vlastní datové typy, například **BarvaSemaforu**

```
data BarvaSemaforu =  Cervena | Oranzova | Zelena
```

Část před rovnítkem = určuje název datového typu, za rovnítkem = jsou tzv. konstruktory hodnot. Konstruktory hodnot určují různé hodnoty, které nový datový typ může nabývat. Jednotlivé hodnoty oddělujeme pomocí svislítka |. Svislítko zde má význam "nebo". Datový typ může nabývat hodnot Cervena, Oranzova nebo Zelena.

V příkladu **BarvaSemaforu** jsme definovali tři konstruktory hodnot **Cervena**, **Oranzova** a **Zelena**. Tyto hodnoty můžeme používat při pattern matchingu v definicích funkcí. Předpokládejme funkci **krizovatka**, která vrátí řetězec odpovídající chování automobilu při dané barvě:

```
krizovatka :: BarvaSemaforu -> String
krizovatka Cervena = "stuj"
krizovatka Oranzova = "priprav se"
krizovatka Zelena = "jed"
```

Dosud typové konstruktory obsahovaly pouze pojmenování konstruktoru, nicméně do konstruktoru hodnot můžeme vložit také další datové typy. Uvažujme další příklad, nadefinujme si vlastní datový typ pro reprezentaci dvojic a trojic celých čísel.

```
data Ntice = Dvojice Int Int | Trojice Int Int Int
```

Dvojici čísel typu **Int** jsme si pojmenovali termínem **Dvojice** a trojici čísel typu **Int** potom **Trojice**, jedná se o naše vlastní pojmenování. Ve funkcích s nimi pak pracujeme následovně:

```
prenasob :: Ntice -> Int
prenasob (Dvojice x y) = x*y
prenasob (Trojice x y z) = x*y*z
```

Příklad z knihy Learn You A Haskell[1]:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

[1] `learnyouahaskell.com`

Datový typ **Shape** může nabývat dvou hodnot: kruh **Circle** a obdélník **Rectangle**. Kruh je dán souřadnicemi středu, první dvě hodnoty typu **Float** a poslední hodnota typu **Float** je poloměr. Obdélník je dán souřadnicemi dvou bodů v protějších rozích. Nyní si ukážme, jak lze z těchto předpisů spočítat obsah obou útvarů:

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Pár poznámek k řešení:

- V definici funkce **surface** jsme použili název datového typu **Shape**, nikoliv hodnoty konstruktoru.

  ```
  surface :: Shape -> Float
  ```

- Při pattern matchingu jsme však už použili konkrétní konstruktor hodnot:

  ```
  surface (Circle _ _ r)
  ```

- Povšimněte si zástupného symbolu podtržítka _. Pokud s proměnnými v definici funkce nepracujeme, nemusíme je pojmenovávat a můžeme je označit podtržítkem.

- Funkce **surface** má pouze jeden parametr typu **Shape**. Proto nemůžeme psát přímo:

  ```
  -- špatně
  surface Circle _ _ r
  ```

  správně musíme konkrétní hodnotu **Circle** dát do závorek, pak se bude obsah závorky chovat, jako jeden parametr.

  ```
  -- správně
  surface (Circle _ _ r)
  ```

- Použití speciálního symbolu $ ve výrazu:

  ```
  surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
  ```

  víme, že pokud voláme nějakou funkci, pak její parametry jsou hodnoty, které následují za pojmenováním funkce a jsou oddělené mezerou. Protože speciální symbol $ má nejnižší prioritu a je zprava asociativní, dojde nejdříve k vyhodnocení výrazu x2 - x1 a teprve výsledek je předán jako parametr funkce **abs**. Alternativně můžeme použití $ obejít doplněním dodatečných závorek:

  ```
  surface (Rectangle x1 y1 x2 y2) = (abs (x2 - x1)) * (abs (y2 - y1))
  ```

- Funkci pak můžeme zavolat např. pomocí:

```
surface (Circle 0 0 1)
```

což značí výpočet obsahu kruhu umístěného na souřadnicích [0,0] a poloměru 1.

Náš příklad se zadáním datového typu **Shape** není úplně přehledný, v další ukázce si vytvoříme nový datový typ pro reprezentaci bodu, datový typ **Point** a ten použijeme pro definici datového typu **Shape**.

```
data Point = Point Float Float
data Shape = Circle Point Float | Rectangle Point Point
```

Funkci pro výpočet obsahu **surface** si nyní můžeme přepsat:

```
surface :: Shape -> Float
surface (Circle (Point x1 y1) r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs y2 - y1)
```

a zavoláme ji nyní pomocí:

```
surface (Circle (Point 0 0) 1)
```

- Pokud pro náš datový typ definujeme pouze jeden konstruktor hodnot, jak to vidíme například u datového typu **Point**, pak je dobrým zvykem jej pojmenovat stejně jako je jméno datového typu, proto se **Point** vyskytuje na levé i pravé straně definice datového typu, ale stejně dobře můžeme psát na pravé straně i jiné pojmenování, například:

  ```
  data Point = Bod Float Float
  ```

- Další příklad využití speciálního znaku $:

  ```
  surface $ Circle (Point 0 0) 1
  ```

0.2 Typové parametry

Nyní uvažujme situaci, kdy máme za úkol sečíst dva vektory čísel mezi sebou:

$$[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$$

Tuto operaci můžeme matematicky definovat pro různé množiny čísel, pro množinu přirozených čísel, množinu celých čísel, množinu reálných čísel a další. V Haskellu bychom byli nuceni pro každou takovou funkci definovat vlastní datový typ:

```
data Vec2DInt = Vec2DInt Int Int
data Vec2dFloat = Vec2DFloat Float Float
```

a pro tyto dva vektory zadefinovat dvě různé funkce pro jejich součet.

```haskell
sum2DInt :: Vec2DInt -> Vec2DInt -> Vec2DInt
sum2DInt (Vec2DInt x1 y1) (Vec2DInt x2 y2) = Vec2DInt (x1 + y1) (x2 + y2)

sum2DFloat :: Vec2DFloat -> Vec2DFloat -> Vec2DFloat
sum2DFloat (Vec2DFloat x1 y1) (Vec2DFloat x2 y2) = Vec2DFloat (x1 + y1) (x2 + y2)
```

Je značně neefektivní, pokud bychom museli pro každý další takový typ definovat další funkci a datový typ, namísto toho použijeme typový parametr:

```haskell
data Vec2D a = Vec2D a a
```

Takto definovaný datový typ může nabývat libovolného konkrétního typu **a**, který umožňuje sčítání.

```haskell
sum2D :: (Num a) => Vec2D a -> Vec2D a -> Vec2D a
sum2D (Vec2D x1 y1) (Vec2D x2 y2) = Vec2D (x1 + y1) (x2 + y2)
```

- Povšimněte si, že v definici funkce **sum2D** jsme museli určit typovou třídu **Num**, protože po datovém typu **a** požadujeme, aby jej bylo možné sčítat.
- Takto definovaná funkce **sum** bude fungovat s každým datovým typem implementujícím typovou třídu **Num**.

0.3 Odvození

V této sekci se podíváme na to, jak lze docílit základní funkčnosti nad vlastními datový typy, které si sami vytváříme. Co když bychom chtěli určit, zda dva objekty typu **Vec2D** nebo **Point** jsou si rovny, nebo bychom si je chtěli vypsat, načíst ze stringu a podobně. K tomu slouží tzv. koncept odvozených instancí.

V Haskellu existuje několik typových tříd, typovou třídu **Num** jsme již zmínili, další jsou:

- **Eq** - umožňuje nad datovým typem vykonávat operace == a /=.
- **Ord** - umožňuje porovnat dva stejné datové typy.
- **Enum** - umožňuje definovat předchůdce a následníka prvku datového typu.
- **Bounded** - omezí hodnoty datového typu na nejmenší a největší možnou hodnotu.
- **Show** - umožní převést instanci datového typu na string.
- **Read** - umožní načíst instanci datového typu ze stringu.

Uvažujme opět příklad s vektorem dvou hodnot reprezentovaných námi definovaným datovým typem **Vec2D**. Pokud bychom použili předchozí zápis:

```
data Vec2D a = Vec2D a a
```

a pokusili bychom se porovnat dvě instance tohoto datového typu, pak bychom obdrželi následující chybu:

```
*Main> (Vec2D 2 1) == (Vec2D 2 2)

<interactive>:2:1: error:
    * No instance for (Eq (Vec2D Integer)) arising from a use of `=='
    * In the expression: (Vec2D 2 1) == (Vec2D 2 2)
      In an equation for `it': it = (Vec2D 2 1) == (Vec2D 2 2)
```

My však toto chování můžeme snadno doplnit použitím klíčového slova **deriving** s typovou třídou uvedenou v závorkách:

```
data Vec2D a = Vec2D a a deriving(Eq)
```

potom například dva výrazy:

```
*Main> (Vec2D 2 3) == (Vec2D 2 4)
False
*Main> (Vec2D 2 3) == (Vec2D 2 3)
True
```

Obdobně bychom mohli chtít instanci datového typu **Vec2D** vypsat. Pokud bychom pouze zadali výraz:

```
*Main> Vec2D "a" "b"

<interactive>:9:1: error:
    * No instance for (Show (Vec2D [Char]))
        arising from a use of `print'
    * In a stmt of an interactive GHCi command: print it
```

znovu si zde připomeneme, že datový typ **Vec2D** jsme definovali s pomocí typového parametru **a**, proto můžeme do instance **Vec2D** umístit také **string**, jak je vidět na příkladu výše, nebo jiné datové typy. Abychom si mohli instance datového typu **Vec2D** zobrazit v řádce potřebuje ji nejdříve převést na string. Na začátku této kapitoly jsme viděli, že tomu slouží typová třída **Show**, proto si s její pomocí rozšíříme datový typ **Vec2D**:

```
data Vec2D a = Vec2D a a deriving(Eq, Show)
```

nyní si již můžeme konkrétní instanci datového typu vypsat:

```
*Main> Vec2D "a" "b"
Vec2D "a" "b"
```

Pro příklady dalších rozšíření datových typů pomocí typových tříd si zavedeme nový příklad a to seznam dnů v týdnu, začneme však s nejjednodušším příkladem dnů o víkendu:

```
data Vikend = Sobota | Nedele
```

rádi bychom aby náš datový typ **Vikend** dokázal porovnat dny v konstruktoru, tedy kosntruktory **Sobota** a **Nedele**, vyzkoušejem následující výraz, kterým chceme říci, že sobota je před nedělí:

```
*Main> Sobota < Nedele

<interactive>:16:1: error:
    * No instance for (Ord Vikend) arising from a use of `<'
    * In the expression: Sobota < Nedele
      In an equation for `it': it = Sobota < Nedele
```

Opět jsme skončili s chybou. Ta nám však napovídá, že pro definici datového typu **Vikend** nemáme definovánu typovou třídu **Ord**, proto ji bude náš datový typ odvozovat:

```
data Vikend = Sobota | Nedele deriving (Eq, Ord)
```

nyní již můžeme porovnat různé konstruktory hodnot mezi sebou:

```
*Main> Sobota < Nedele
True
*Main> Nedele < Sobota
False
*Main>
```

Menší hodnota je ta, která se v konstruktoru hodnot vyskytuje dříve. Pokud bychom například **Víkend** definovali pomocí:

```
data Vikend = Nedele | Sobota deriving (Eq, Ord)
```

pak by hodnota **Nedele** byla menší, než **Sobota**:

```
*Main> Nedele < Sobota
True
```

- Důležitá poznámka zde je, že typovou třídu **Ord** musíme definovat společně s **Eq**, jinak nám porovnání nebude fungovat.

Další typovou třídu, kterou si vyzkoušíme bude **Enum**, tato třída nám umožní použít operace **succ** a **pred**, které vrátí předchozí a následující hodnotu:

```
data Vikend = Sobota | Nedele deriving (Eq, Ord, Enum, Show)
```

abychom si mohli výsledek operace nechat vypsat používáme i odvození typové třídy **Show**:

```
*Main> succ Sobota
Nedele
```

avšak, co se stane, pokud nás bude zajímat den před sobotou:

```
*Main> pred Sobota
*** Exception: pred{Vikend}: tried to take `pred' of first tag in enumeration
CallStack (from HasCallStack):
  error, called at test.hs:49:50 in main:Main
```

Skončíme chybou. Abychom umožnili zjištění, který den je prvním nebo posledním konstruktorem a případně toto chování ošetřili, použijeme typovou třídu **Bounded**:

```
data Vikend = Sobota | Nedele deriving (Eq, Ord, Enum, Show, Bounded)
```

pak se můžeme na konkrétní nejmenší a největší hodnotu doptat pomocí funkcí **minBound** a **maxBound**:

```
*Main> minBound :: Vikend
Sobota
*Main> maxBound :: Vikend
Nedele
```

0.4 Typová synonyma

Typová synonyma nereprezentují nové datové typy. Typová synonyma jsou pouze zástupnými názvy již existujících datových typů. Tato synonyma používáme pro zpřehlednění kódu. Uvažujme například datový typ reprezentující nějakou firmu (String) a její zisk (Float), bez typových synonym by zápis vypadal následovně:

```
data Firma = Firma String Float
```

z tohoto zápisu však není jasné, co pole **String** a **Float** reprezentují. Zkusme ten samý problém zapsat pomocí typových synonym. Typové synonymum je dáno klíčovým slovem **type** následované synonymním názvem, rovnítkem a typem, který zastupuje:

```
type Nazev = String
type Zisk = Float
data Firma = Firma Nazev Zisk deriving(Show)
```

použití například:

```
*Main> Firma "Google" 1000
Firma "Google" 1000.0
```

0.5 Rekurzivní datové struktury

V naší práci jsme se již s jednou rekurzivní datovou strukturou setkali. Jednalo se o seznam. Rekurzivní datová struktura je taková, která ve své definici používá sebe samu. Zkusme si nyní vytvořit vlastní seznam. Budeme potřebovat hodnotový konstruktor pro reprezentaci prázdného seznamu a konstruktor, který bude brát nějaký prvek a jiný seznam:

```
data Seznam a = Prazdny | Pripoj a (Seznam a) deriving(Show, Read, Eq, Ord)
```

Opět si povšimněme, že **Seznam** je definován pomocí typového parametru **a**, protože samozřejmě předpokládáme, že do našeho seznamu budeme chtít vkládat libovolné prvky.

```
*Main> 5 `Pripoj` Prazdny
Pripoj 5 Prazdny
*Main> 4 `Pripoj` (5 `Pripoj` Prazdny)
Pripoj 4 (Pripoj 5 Prazdny)
```

V příkladu výše se konstruktor **Pripoj** chová stejně jako **:** ve výchozím zabudovaném seznamu a konstruktor **Prazdny**, jako [].

V dalším příkladu se podíváme na datovou strukturu typu strom. Konkrétně se podíváme na binární vyhledávací strom [2]. Binární strom má dva potomky, které samy tvoří podstromy. Listový uzel stromu již další stromy neobsahuje, proto si zadefinujeme konstruktor **PrazdnyStrom** Do stromu můžeme vkládat prvky a posléze je vyhledat.

```
data Strom a = PrazdnyStrom | Uzel a (Strom a) (Strom a) deriving (Show, Eq)
```

Nejdříve si vytvoříme strom s jedním uzlem, tedy vložíme první prvek do stromu a tento nový strom vrátíme:

```
vytvorStrom :: a -> Strom a
vytvorStrom x = Uzel x PrazdnyStrom PrazdnyStrom
```

Vyzkoušejme:

```
*Main> strom = vytvorStrom 5
*Main> :t strom
strom :: Num a => Strom a
```

Ve výsledku máme **Strom a**, který obsahuje prvky typu **Num**. Nyní použijeme funkci **vytvorStrom** ve funkci **vlozPrvek**:

```
vlozPrvek :: (Ord a) => a -> Strom a -> Strom a
vlozPrvek x PrazdnyStrom = vytvorStrom x
vlozPrvek x (Uzel a levyPodstrom pravyPodstrom)
  | x == a = Uzel x levyPodstrom pravyPodstrom
  | x < a  = Uzel a (vlozPrvek x levyPodstrom) pravyPodstrom
  | x > a  = Uzel a levyPodstrom (vlozPrvek x pravyPodstrom)
```

[2] https://en.wikipedia.org/wiki/Binary_search_tree

- Prvky stromu potřebujeme porovnávat, proto je funkce **vlozPrvek** definována s typovou třídou **Ord**.
- Haskell nezná koncept ukazatelů, takže pokaždé když něco vkládáme do stromu, vytváříme nový strom seskládaný z podstromů původního stromu doplněného o nově vloženou hodnotu.

Příklad použití:

```
*Main> strom = vlozPrvek 5 PrazdnyStrom
*Main> strom
Uzel 5 PrazdnyStrom PrazdnyStrom
*Main> strom1 = vlozPrvek 2 strom
*Main> strom1
Uzel 5 (Uzel 2 PrazdnyStrom PrazdnyStrom) PrazdnyStrom
*Main> strom2 = vlozPrvek 6 strom1
*Main> strom2
Uzel 5 (Uzel 2 PrazdnyStrom PrazdnyStrom) (Uzel 6 PrazdnyStrom PrazdnyStrom)
```

Z výpisu vidíme, že kořenový uzel obsahuje číslo 5, v levém podstromu je číslo 2 a v pravém podstromu pak číslo 6. Listové uzly obsahují prázdné podstromy. Nyní budeme chtít touto strukturou projít a ověřit, zda se nějaké číslo ve vyhledávcím stromu vyskytuje.

```
prvekStromu :: (Ord a) => a -> Strom a -> Bool
prvekStromu x PrazdnyStrom = False
prvekStromu x (Uzel a pravyPodstrom levyPodstrom)
  | x == a = True
  | x < a  = prvekStromu x pravyPodstrom
  | x > a  = prvekStromu x levyPodstrom
```

a použití:

```
*Main> let cisla = [6,1,2,5,8]
*Main> let strom = foldr vlozPrvek PrazdnyStrom cisla
*Main> prvekStromu 3 strom
False
*Main> prvekStromu 8 strom
True
```