Vlastní jednoduché datové typy, konstruktor `Maybe`, rekurzivní datové typy

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

Vlastní datové typy

Příklad 6.1.1: Mějme datový typ `Day` představující dny v týdnu definovaný níže. Definujte funkci `weekend :: Day -> Bool`, která o zadaném dni určí, jestli je to víkendový den.

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
         deriving (Show, Eq, Ord)
```

Vlastní datové typy

Příklad 6.1.1: Mějme datový typ `Day` představující dny v týdnu definovaný níže. Definujte funkci `weekend :: Day -> Bool`, která o zadaném dni určí, jestli je to víkendový den.

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
         deriving (Show, Eq, Ord)
```

- jednotlivé možné hodnoty se oddělují svislítkem
- `deriving` část říká, do kterých typových tříd má typ patřit (instance se odvodí automaticky)

Vlastní datové typy

```
data Shape = Circle Double
           | Rectangle Double Double
           | Point
           deriving (Eq, Show)
```

- Uveďte příklady různých hodnot typu `Shape`.
- Jakého typu jsou výrazy `Circle`, `Rectangle`, `Point`?
- Naprogramujte funkci `area :: Shape -> Double`, která spočítá obsah zadaného tvaru.

Datové a typové konstruktory I.

```
data Shape = Circle Double
           | Rectangle Double Double
           | Point
           deriving (Eq, Show)
```

- `Shape` je (nulární) typový konstruktor
- tři datové konstruktory (funkce vytvářející hodnoty typu `Shape`):
  - unární: `Circle :: Double -> Shape`
  - binární: `Rectangle :: Double -> Double -> Shape`
  - nulární: `Point :: Shape`
- datové i typové kostruktory začínají velkým písmenem (nebo dvojtečkou)

Datové a typové konstruktory II.

Datové konstruktory lze použít v definici podle vzoru:

```
null :: [a] -> Bool
null []    = True
null (_:_) = False

area :: Shape -> Double
area Point           = 0
area (Circle r)      = pi * r * r
area (Rectangle a b) = a * b
```

Vlastní datové typy: příklad

Příklad 6.1.2: Vytvořte nový datový typ `Jar` představující sklenici ve spíži. Každá sklenice je v jednom z následujících stavů:

- je prázdná (`EmptyJar`);
- je v ní ovocná marmeláda (`Jam`), pamatujeme si typ ovoce, ze kterého byla vyrobena (`String`);
- jsou v ní okurky (`Cucumbers`), o nich si nemusíme nic pamatovat, stejně se hned snědí;
- je v ní kompot (`Compote`), pamatujeme si rok výroby (`Int`).

Vaší úlohou je pak nadefinovat funkci `stale :: Jar -> Bool`, která určí, jestli je obsah dané sklenice již zkažený. Prázdné sklenice, okurky ani marmelády se nekazí (možná je to tím, že se příliš rychle snědí), kompoty se pokazí za 10 let od zavaření (zadefinujte si celočíselnou konstantu `today`, ve které budete mít aktuální rok).

Datový typ Maybe

Definice (z `Prelude`):

```
data Maybe a = Nothing | Just a
             deriving (Eq, Ord, Show, Read)
```

- parametrizovaný datový typ `Maybe a` → konkrétní typy: `Maybe Int`, `Maybe [String]`, `Maybe (Int, [Bool])`,...
- reprezentuje hodnotu, jejíž výpočet může selhat
- hodnota může být přítomna (`Just 1`) nebo chybět (`Nothing`)
- hodnoty extrahujeme pomocí vzorů

```
justOrVal :: a -> Maybe a -> a
```

Datový typ Maybe

Definice (z `Prelude`):

```
data Maybe a = Nothing | Just a
             deriving (Eq, Ord, Show, Read)
```

- parametrizovaný datový typ `Maybe a` → konkrétní typy: `Maybe Int`, `Maybe [String]`, `Maybe (Int, [Bool])`,...
- reprezentuje hodnotu, jejíž výpočet může selhat
- hodnota může být přítomna (`Just 1`) nebo chybět (`Nothing`)
- hodnoty extrahujeme pomocí vzorů

```
justOrVal :: a -> Maybe a -> a

justOrVal x Nothing  = x
justOrVal _ (Just y) = y
```

Datový typ `Maybe`: příklad

Příklad 6.2.2: S využitím typového konstruktoru `Maybe` definujte funkci
`divlist :: Integral a => [a] -> [a] -> [Maybe a]`, která celočíselně podělí dva celočíselné seznamy „po složkách“, tj.

$$\texttt{divlist [x}_1\texttt{, ..., x}_n\texttt{] [y}_1\texttt{, ..., y}_n\texttt{]} \\ \quad \rightsquigarrow^* \texttt{[div x}_1\ \texttt{y}_1\texttt{, ..., div x}_n\ \texttt{y}_n\texttt{]}$$

a ošetří případy dělení nulou.

Rekurzivní datové typy: příklad

Příklad 6.1.6: Uvažme následující definici typu `Expr`:

```
data Expr = Con Float
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
```

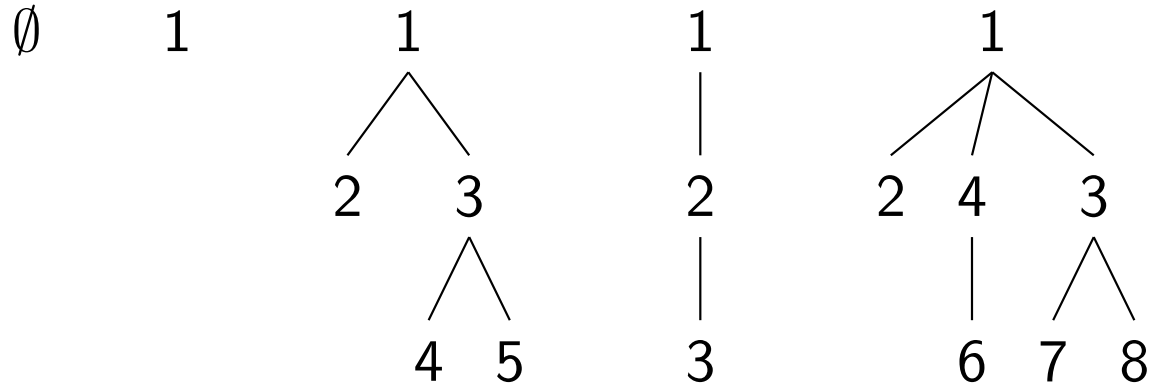
a) Uveďte výraz typu `Expr`, který představuje hodnotu 3.14.
b) Definujte funkci `eval :: Expr -> Float`, která vrátí hodnotu daného výrazu.

Binární a *n*-ární stromy

Uvažme následující definice binárních a *n*-árních stromů:

```
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
data NTree a = NNode a [NTree a]
```

- Identifikujte v definicích datové a typové kostruktory a určete jejich aritu.
- Zkuste pomocí těchto datových typů zapsat stromy zobrazené níže.



Rekurzivní datové typy

Příklad 6.3.3: Uvažte následující rekurzivní datový typ představující binární strom s ohodnocenými uzly:

```
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
```

Definujte následující funkce nad binárními stromy:

a) `treeSize :: BinTree a -> Integer`, která spočítá počet uzlů ve stromě.
b) `listTree :: BinTree a -> [a]`, která převede všechny hodnoty uzlů ve stromu do seznamu.
c) `height :: BinTree a -> Int`, která určí výšku stromu.
d) `longestPath :: BinTree a -> [a]`, která najde nejdelší cestu ve stromě začínající v kořeni a vrátí ohodnocení na ní.

Rekurzivní datové typy

Příklad 6.3.4: Pro datový typ `BinTree` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu.

a) Definujte funkci `treeMax :: Ord a => BinTree a -> a`, která najde maximální hodnotu v uzlech stromu. Pro prázdný strom skončete výpočet s vhodnou chybou.
b) Definujte funkci `fullTree :: Int -> a -> BinTree a`, která pro volání `fullTree n v` vytvoří binární strom výšky `n`, ve kterém jsou všechny větve stejně dlouhé a všechny uzly ohodnocené hodnotou `v`.
c) Definujte funkci `treeZip :: BinTree a -> BinTree b -> BinTree (a,b)` jako analogii seznamové funkce `zip`. Výsledný strom tedy obsahuje pouze ty uzly, které jsou v obou vstupních stromech.

Rekurzivní datové typy

Příklad 6.3.5: Uvažme datový typ `BinTree`.

a) Definujte funkci `treeRepeat :: a -> BinTree a` jako analogii seznamové funkce `repeat`. Funkce tedy vytvoří nekonečný strom, který má zadanou hodnotu v každém uzlu.

b) Pomocí funkce `treeRepeat` vyjádřete nekonečný binární strom `nilTree`, který má v každém uzlu prázdný seznam.

c) Definujte funkci
`treeIterate :: (a->a) -> (a->a) -> a -> BinTree a`
jako analogii seznamové funkce `iterate`. Levý potomek každého uzlu bude mít hodnotu vzniklou aplikací první zadané funkce a pravý aplikací druhé zadané funkce.

Rekurzivní datové typy

Příklad 6.3.8: Uvažte typ *n*-árních stromů definovaný následovně:

```
data NTree a = NNode a [NTree a]
             deriving (Show, Read)
```

Definujte následující:

a) funkci `ntreeSize :: NTree a -> Integer`, která spočítá počet uzlů ve stromě
b) funkci `ntreeSum :: Num a => NTree a -> a`, která sečte ohodnocení všech uzlů stromu
c) funkci `ntreeMap :: (a -> b) -> NTree a -> NTree b`, která bere funkci a strom, a aplikuje danou funkci na ohodnocení v každém uzlu:

```
ntreeMap (+1) (NNode 0 [NNode 1 [], NNode 41 []])
    ⇝* NNode 1 [NNode 2 [], NNode 42 []]
```