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

IB015 Neimperativní programování

Kolektiv cvičících IB015

Fakulta informatiky, Masarykova univerzita

podzim 2018

Opakování

Naprogramujte funkci `intersection :: Real a => [(a, a, a)] -> [((a, a, a), (a, a, a))]`, která vezme seznam kružnic v rovině reprezentovaných trojicí (x, y, r), kde x, y jsou souřadnice, r je poloměr, a vrátí seznam dvojic kružnic, které se navzájem protínají.

Naprogramujte funkci `noIntersection :: Real a => [(a, a, a)] -> [(a, a, a)]`, která bere seznam kružnic v rovině a vrací seznam kružnic, které se neprotínají s žádnou jinou.

Vlastní datové typy

Příklad 5.1.1: 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 5.1.1: 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`?

Hodnotové a typové konstruktory I.

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

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

Hodnotové a typové konstruktory II.

Hodnotové konstruktory lze použít v definici podle vzoru:

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

- `[]` a `(:)` jsou hodnotové konstruktory typu `[a]`

Hodnotové konstruktory jsou first-class citizens (stejně jako funkce), tedy můžeme je ve výrazech jako funkce používat.

`(Circle . (*2)) 2.5` $\rightsquigarrow^*$ `Circle 5.0`

Hodnotové a typové konstruktory: příklad

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

Naprogramujte funkci `area :: Shape -> Double`, která spočítá obsah zadaného tvaru.

```
area (Circle 1) ⇝* 3.141592653589793
area (Rectangle 2 2) ⇝* 4
area Point ⇝* 0
```

Datový typ `Maybe a`

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
- za povšimnutí stojí funkce `fromMaybe` z modulu `Data.Maybe` nebo `lookup`

```
fromMaybe :: a -> Maybe a -> a
```

Datový typ `Maybe a`

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
- za povšimnutí stojí funkce `fromMaybe` z modulu `Data.Maybe` nebo `lookup`

```
fromMaybe :: a -> Maybe a -> a

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

Datový typ `Maybe a`: příklad

Příklad 5.2.2: S využitím typového konstruktoru `Maybe` definujte funkci
`divlist :: Integral a => [a] -> [a] -> [Maybe a]`, která celočíselně podělí dva celočíselné seznamy „po složkách“, tj.

`divlist [`x_1`, ..., `x_n`] [`y_1`, ..., `y_n`]`
$\rightsquigarrow^*$ `[div `x_1` `y_1`, ..., div `x_n` `y_n`]`

```
divlist [12, 5, 7] [3, 0, 2] ⇝* [Just 4, Nothing,
    Just 3]
```

a ošetří případy dělení nulou.

Rekurzivní datové typy

Zamyslete se, jak byste definovali vlastní datový typ reprezentující seznamy v Haskellu.

Rekurzivní datové typy

Zamyslete se, jak byste definovali vlastní datový typ reprezentující seznamy v Haskellu.

```
data MyList a = Empty | ListElement a (MyList a)
```

- Uveďte příklady různých hodnot typu `MyList a`

Rekurzivní datové typy: příklad

Příklad 5.1.6: Uvažme následující definici typu `Expr`:

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

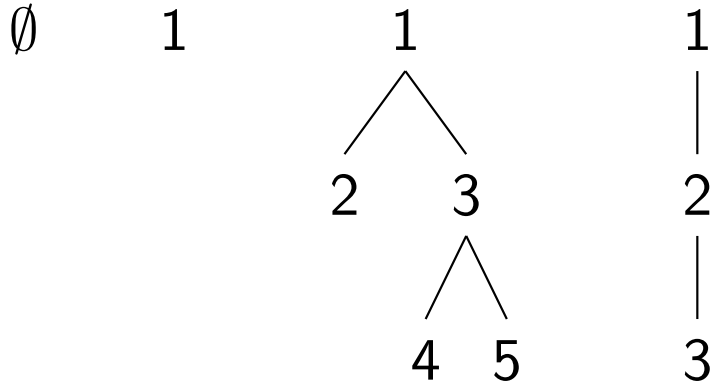
a) Uveďte výraz typu `Expr`, který představuje hodnotu 3.14.
b) Definujte funkci `eval :: Expr -> Float`, která vrátí hodnotu daného výrazu.

Binární stromy

Uvažme následující definici binárních stromů:

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

- Identifikujte v definici hodnotové a typové kostruktory a určete jejich aritu.
- Zkuste pomocí tohoto datového typu zapsat stromy zobrazené níže.



- V nasledujících příkladech by se vám mohl hodit soubor s předdefinovanými stromy na testování.

Binární stromy: příklady

Příklad 5.3.3: 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.

Binární stromy: příklady

Příklad 5.3.4: Pro datový typ `BinTree a` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu.

a) 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`.

b) 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.

Binární stromy: příklady

Příklad 5.3.5: Uvažme datový typ `BinTree a`.

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.

Binární stromy: příklady

Příklad 5.3.8: Uvažme datový typ `BinTree a`.

a) Definujte funkci
`isTreeBST :: (Ord a) => BinTree a -> Bool`, která se vyhodnotí na `True`, jestli bude její první argument validní binární vyhledávací strom.

b) Definujte funkci
`searchBST :: (Ord a) => a -> BinTree a -> Bool`, která projde BST z druhého argumentu v smyslu binárního vyhledávání a vyhodnotí se na `True` v případě, že její první argument najde v uzlech při vyhledávání.