Typové třídy, akumulační funkce na seznamech

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

Typové třídy

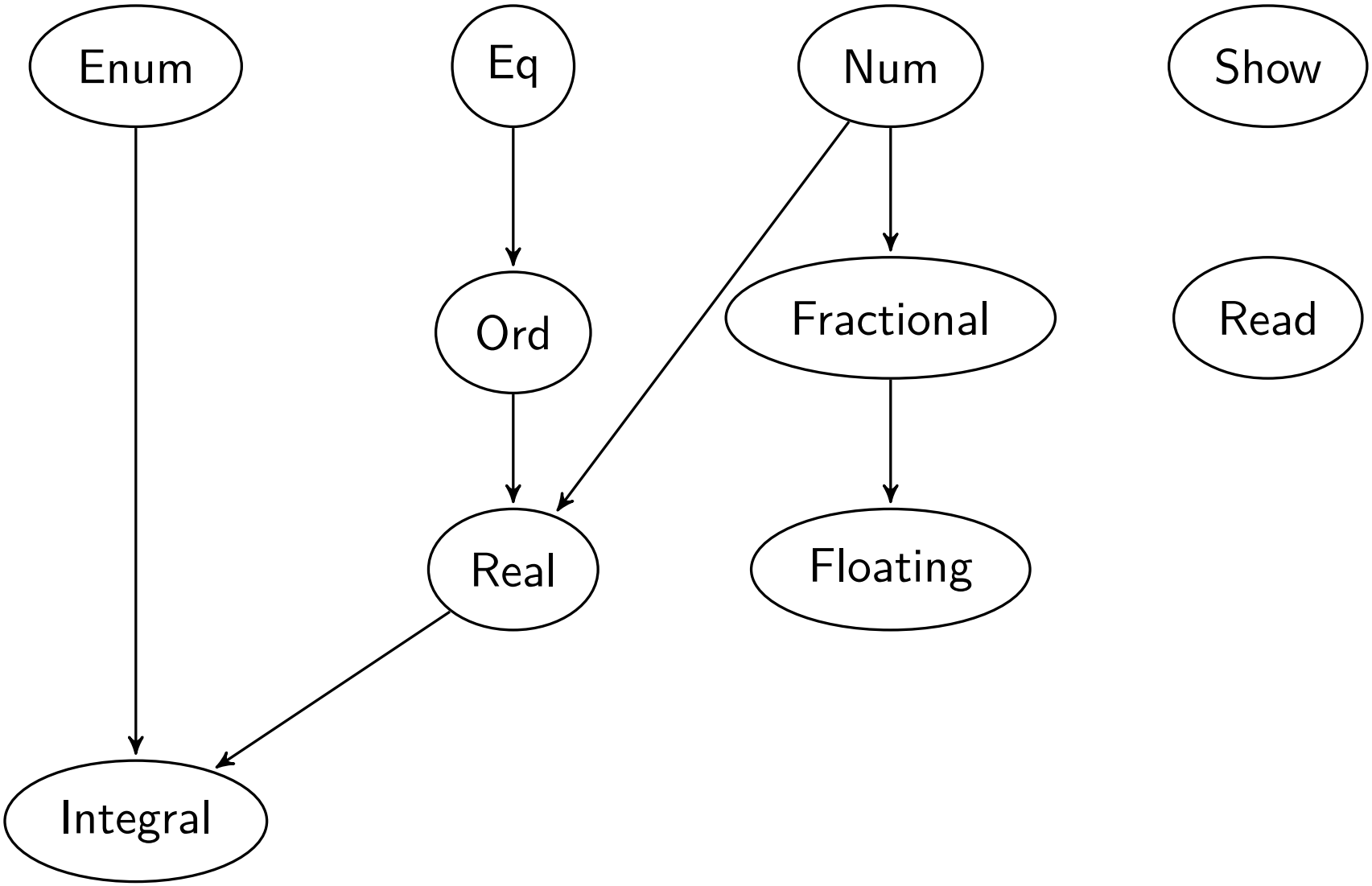
- vyjadřují nějakou vlastnost hodnot
  (porovnatelnost, převeditelnost na text, . . .)
- obsahuje funkce, které implementují tuto vlastnost
  (porovnávání, . . .)
- typ patří do třídy $\sim$ instance typové třídy
  - automatické instance (`deriving`)
  - ruční instance (pozor na odsazení!)

```
data Shape = P | C Double | R Double Double
instance Eq Shape where
    P == P                   = True
    (C r1) == (C r2)         = r1 == r2
    (R x1 y1) == (R x2 y2) = x1 == x2 && y1 == y2
    _ == _                   = False
```

Typové třídy

- často lze některé funkce odvodit z jiných
  - `(/=) z (==)`
  - `(>), (<), (>=) z (<=) a (==)`
- *minimální kompletní implementace*: skupina funkcí, které musíme implementovat, aby typová třída fungovala
  - pro `Eq`: `(==)`
  - pro `Ord`: `(<=)`
- základní informace o typové třídě vypíše interpret při povelu `:info <nazev-tridy>`
- typové třídy jsou v hierarchii
  - instance `Ord` vyžaduje instanci `Eq`
  - část hierarchie následuje

Hierarchie základních typových tříd



Typové třídy – příklad

Příklad 7.1.2: Uvažte datový typ představující semafor zadefinovaný níže.

```
data TrafficLight = Red | Orange | Green
```

Umožněte zobrazování hodnot tohoto typu, jejich vzájemné porovnávání a řazení (zelená $<$ oranžová $<$ červená). Řečeno jinak, napište instanci `TrafficLight` pro typové třídy `Show`, `Eq` a `Ord`.

Typové třídy – příklad

Příklad 7.1.3: Zadefinujme vlastní typ uspořádaných dvojic s názvem `PairT`. Tento typ bude mít pouze jeden binární datový konstruktor `PairD` (viz definice níže).

```
data PairT a b = PairD a b
```

Vytvořte instanci `PairT` pro typové třídy `Show`, `Eq` a `Ord`. Ať jsou si dvě dvojice rovny právě tehdy, pokud jsou si rovny po složkách. Uspořádání použijte lexikografické. Zobrazování hodnot tohoto typu nechť je slovní (tedy namísto obligátního `(1,2)` vypište třeba `"pair of 1 and 2"`).

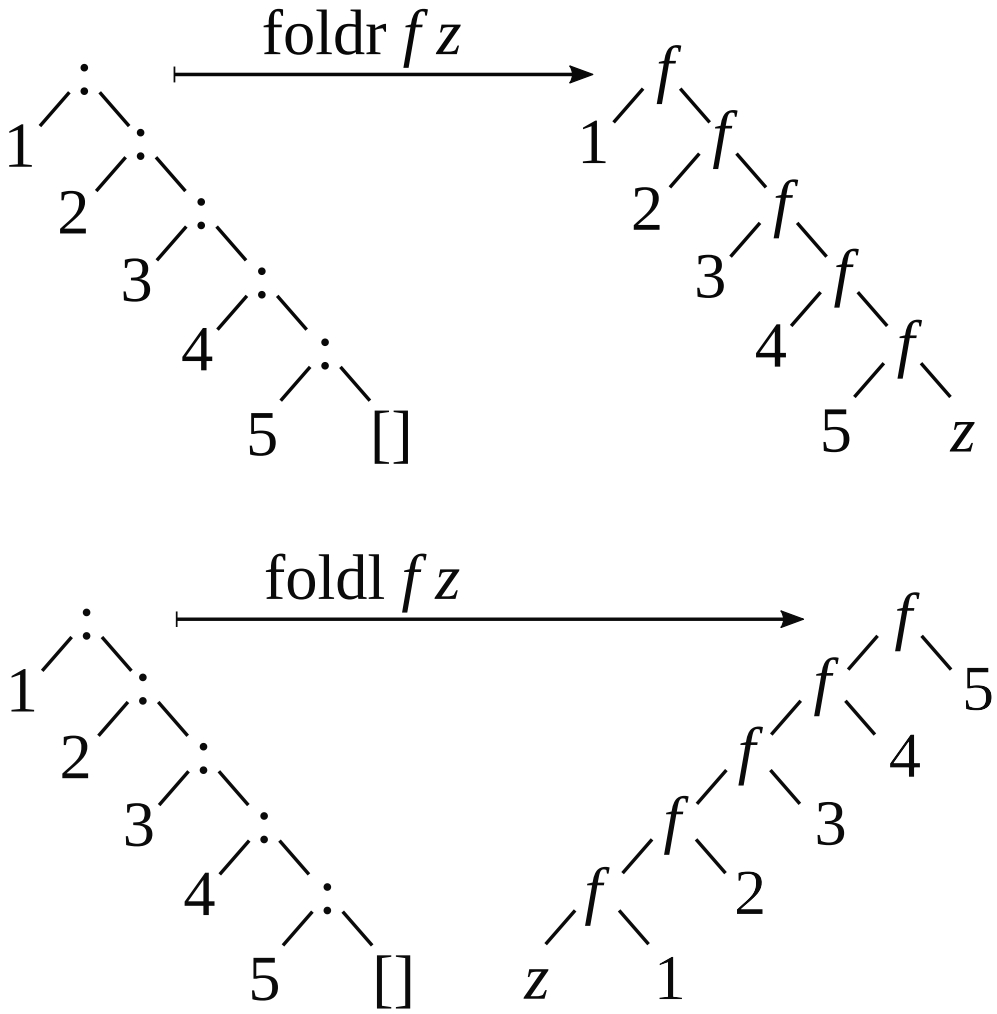
Akumulační funkce na seznamech – motivace

Příklad 7.2.1: Definujte následující funkce rekurzivně:

a) `product'` – součin prvků seznamu
b) `length'` – počet prvků seznamu
c) `map'` – funkci `map`

Co mají tyto definice společné? Jak by vypadalo jejich zobecnění?

Akumulační funkce na seznamech – ilustrace



Zdroj: Haskell wiki (`https://wiki.haskell.org/Fold`)

Akumulační funkce na seznamech

Seznamové akumulační funkce:[1] Akumulační funkce:

- zpracují seznam na jednu hodnotu
- transformace seznamu dle struktury

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`

- `foldr1 :: (a -> b -> b) -> [a] -> b`
- `foldl1 :: (b -> a -> b) -> [a] -> b`

„Praktická ukázka“ (vtip): `http://foldr.com/`

[1] Uvedené funkce jsou ve skutečnosti obecnější, jsou definované pro `Foldable t`. Pro kurz IB015 však můžete jakoukoliv `Foldable t` považovat za seznam (`[]`). Více o *foldable* například v IB016 Seminář z funkcionálního programování.

Akumulační funkce na seznamech I

Příklad 7.2.2: Určete, co dělají akumulační funkce s uvedenými argumenty. Najděte hodnoty, na které je lze tyto výrazy aplikovat, a ověřte pomocí interpretu.

a) `foldr (+) 0`
b) `foldr1 (\x s -> x + 10 * s)`
c) `foldl1 (\s x -> 10 * s + x)`

Akumulační funkce na seznamech II

Zhluboka se nadechněte, ve sbírce si otevřete příklad 7.2.8 (implementace funkcí pomocí akumulačních funkcí) a řešte jednotlivé podpříklady.