# Monoidy, `Foldable`

Martin Jonáš, Martin Kurečka, Adam Matoušek

(původní autoři slajdů Vladimír Štill, Martin Ukrop)

Fakulta informatiky, Masarykova univerzita

jaro 2023

# Monoidy

# Motivace I: zpracovávání argumentů příkazové řádky

Chtěli bychom zpracovat argumenty příkazové řádky jako nastavení programu.

```
./run -v -opt=o1 -q -opt=o2
```

- `-v` (*verbose*) zapíná ladicí výstupy
- `-q` (*quiet*) vypíná ladicí výstupy
- program se vždy chová podle posledního přepínače `-v`/`-q`
- každý výskyt `-opt` přidává programu libovolný textový argument
- výchozí nastavení je bez ladicích výstupů a bez dalších argumentů

# Datový typ pro konfiguraci

Nachystejme si na nastavení vhodný datový typ:

```
data Config = Config
    { verbose :: Bool
    , options :: [String]
    } deriving (Eq, Show)
```

# Představa zpracování

- každý přepínač je jedna změna oproti výchozímu nastavení
- každý přepínač reprezentuje elementární validní nastavení
- tato nastavení můžeme sloučit nějakou vhodnou funkcí

```
-v          ⊕   -opt=o1     ⊕   -q          ⊕   -opt=o2
```

```
def {       def {       def {       def {
verbose     options     verbose     options
= True  ⊕   = ["o1"]  ⊕   = False   = ["o2"]
}           }           }           }
```

- `def = Config { verbose = False, options = []}`
- Jaké vlastnosti by měla mít funkce $\oplus$?

# Vlastnosti funkce $\oplus$: uzavřenost

Je množina všech platných konfigurací uzavřená na funkci $\bigoplus$?

# Vlastnosti funkce $\oplus$: uzavřenost

Je množina všech platných konfigurací uzavřená na funkci $\oplus$?

Ano, protože:

- Funkce $\oplus$ by měla vracet opět platné konfigurace.
  $\oplus$ `:: Config -> Config -> Config`
- Říkáme, že se jedná o *operaci* na množině konfigurací.

# Vlastnosti operace $\oplus$: asociativita

Záleží na pořadí zpracovávání parametrů?

`(-v` $\oplus$ `-opt=o1)` $\oplus$ `-q`

vs.

`-v` $\oplus$ `(-opt=o1` $\oplus$ `-q)`

# Vlastnosti operace $\oplus$: asociativita

Záleží na pořadí zpracovávání parametrů?

(`-v` $\oplus$ `-opt=o1`) $\oplus$ `-q`

vs.

`-v` $\oplus$ (`-opt=o1` $\oplus$ `-q`)

Ne, pořadí zpracování by nemělo ovlivnit výslednou konfiguraci.
Říkáme, že operace $\oplus$ je *asociativní*.

# Vlastnosti operace $\oplus$: komutativita

Záleží na pořadí samotných parametrů?

`-v` $\oplus$ `-opt=o1` $\oplus$ `-q`

vs.

`-q` $\oplus$ `-opt=o1` $\oplus$ `-v`

# Vlastnosti operace $\oplus$: komutativita

Záleží na pořadí samotných parametrů?

`-v` $\oplus$ `-opt=o1` $\oplus$ `-q`

vs.

`-q` $\oplus$ `-opt=o1` $\oplus$ `-v`

Ano!

- argumenty `-q -v` produkují jinou konfiguraci než `-v -q`

Operace proto není *komutativní*.

# Vlastnosti operace $\oplus$: neutrální prvek

Má operace $\oplus$ neutrální prvek?

N $\oplus$ `-v` $\oplus$ `-opt=o1` $\oplus$ `-opt=o2` $\oplus$ N

# Vlastnosti operace $\oplus$: neutrální prvek

Má operace $\oplus$ neutrální prvek?

N $\oplus$ `-v` $\oplus$ `-opt=o1` $\oplus$ `-opt=o2` $\oplus$ N

Ne (zatím), neutrálním prvkem by měla být výchozí konfigurace.

```
def :: Config
def = Config
    { verbose = False  -- yikes! /o\
    , options = []
    }
```

# Vlastnosti operace $\oplus$: neutrální prvek

Má operace $\oplus$ neutrální prvek?

N $\oplus$ `-v` $\oplus$ `-opt=o1` $\oplus$ `-opt=o2` $\oplus$ N

Ne (zatím), neutrálním prvkem by měla být výchozí konfigurace.

```
def :: Config
def = Config
    { verbose = False  -- yikes! /o\
    , options = []
    }
```

Jak zajistit, aby výchozí konfigurace nepřepsala případné `-v`?

# Config: nová definice

Upravíme datový typ **Config** následovně:

```
data Config = Config
    { verbose :: Maybe Bool
    , options :: [String]
    } deriving (Eq, Show)


def :: Config
def = Config { verbose = Nothing , options = [] }
```

- Hodnota **Nothing** $\sim$ dosud nedefinovaný parametr verbose.
- Je přepsána libovolnou hodnotou **Just**.

# Motivace II: průchod adresářovou strukturou

`Filesystem` reprezentuje stromovou strukturu adresářového systému.

```
data Filesystem = File Name Size
                | Folder Name [Filesystem]
```

- Chtěli bychom celý strom projít a do `Map Name Size` ukládat soubory splňující zadaný regex.

# Motivace II: průchod adresářovou strukturou

`Filesystem` reprezentuje stromovou strukturu adresářového systému.

```
data Filesystem = File Name Size
                | Folder Name [Filesystem]
```

- Chtěli bychom celý strom projít a do `Map Name Size` ukládat soubory splňující zadaný regex.

- Pro každý přidaný prvek lze vytvořit jednoprvkovou `Map`u.
- `singleton :: k -> v -> Map k v`
- `union :: Ord k => Map k v -> Map k v -> Map k v`

# Vlastnosti funkce union

- Spojení dvou struktur (`Map k v`) vytvoří novou strukturu (`Map k v`).
  - `union :: Ord k => Map k v -> Map k v -> Map k v`
- Nezáleží na „uzávorkování“ spojení `Map` ⇒ asociativita
- Záleží na pořadí spojovaných `Map` ⇒ **NE** komutativita
- Prázdná struktura je neutrální prvek vůči spojení
- `empty :: Map k v`

⇒ V každém uzlu lze vrátit prázdnou nebo jednoprvkovou strukturu. Pomocí operace `union` je následně všechny spojíme.

# Algebraické okénko

# Pologrupy a monoidy

**Grupoid** $(M, \circ)$ je algebraická struktura. Sestává z nosné množiny $M$ a binární operace $\circ\colon M \times M \to M$.

**Pologrupa** je grupoid, jehož operace je asociativní:

- Asociativita: $\forall x, y, z \in M.\ (x \circ y) \circ z = x \circ (y \circ z)$

**Monoid** je pologrupa s neutrálním prvkem:

- Neutrální prvek: $\exists e \in M\, \forall x \in M.\ x \circ e = e \circ x = x$

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním
  $\Rightarrow$ Ano, (komutativní) monoid.
- $(\mathbb{N}, -)$, přirozená čísla s odčítáním

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním
  $\Rightarrow$ Ano, (komutativní) monoid.
- $(\mathbb{N}, -)$, přirozená čísla s odčítáním
  $\Rightarrow$ Ne (není ani grupoidem).
- $(\mathbb{N}, \textbf{min})$, přirozená čísla s minimem

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním
  $\Rightarrow$ Ano, (komutativní) monoid.
- $(\mathbb{N}, -)$, přirozená čísla s odčítáním
  $\Rightarrow$ Ne (není ani grupoidem).
- $(\mathbb{N}, \textbf{min})$, přirozená čísla s minimem
  $\Rightarrow$ Ne (neexistuje neutrální prvek), ale je pologrupa.
- $(\mathbb{N}, \textbf{max})$, přirozená čísla s maximem

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním
  $\Rightarrow$ Ano, (komutativní) monoid.
- $(\mathbb{N}, -)$, přirozená čísla s odčítáním
  $\Rightarrow$ Ne (není ani grupoidem).
- $(\mathbb{N}, \textbf{min})$, přirozená čísla s minimem
  $\Rightarrow$ Ne (neexistuje neutrální prvek), ale je pologrupa.
- $(\mathbb{N}, \textbf{max})$, přirozená čísla s maximem
  $\Rightarrow$ Ano, (komutativní) monoid.
- $([...], ++)$, seznamy se zřetězením

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním
  $\Rightarrow$ Ano, (komutativní) monoid.
- $(\mathbb{N}, -)$, přirozená čísla s odčítáním
  $\Rightarrow$ Ne (není ani grupoidem).
- $(\mathbb{N}, \textbf{min})$, přirozená čísla s minimem
  $\Rightarrow$ Ne (neexistuje neutrální prvek), ale je pologrupa.
- $(\mathbb{N}, \textbf{max})$, přirozená čísla s maximem
  $\Rightarrow$ Ano, (komutativní) monoid.
- $([...], ++)$, seznamy se zřetězením
  $\Rightarrow$ Ano, (NEkomutativní) monoid.
- $(\{f \mid f :: a \to a\}, .)$, funkce typu a `->` a se skládáním

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním
  $\Rightarrow$ Ano, (komutativní) monoid.
- $(\mathbb{N}, -)$, přirozená čísla s odčítáním
  $\Rightarrow$ Ne (není ani grupoidem).
- $(\mathbb{N}, \textbf{min})$, přirozená čísla s minimem
  $\Rightarrow$ Ne (neexistuje neutrální prvek), ale je pologrupa.
- $(\mathbb{N}, \textbf{max})$, přirozená čísla s maximem
  $\Rightarrow$ Ano, (komutativní) monoid.
- $([...], ++)$, seznamy se zřetězením
  $\Rightarrow$ Ano, (NEkomutativní) monoid.
- $(\{f \mid f :: a \to a\}, .)$, funkce typu a `->` a se skládáním
  $\Rightarrow$ Ano, (NEkomutativní) monoid.
- (`Config`, $\bigoplus$), datový typ konfigurace s operací $\bigoplus$

# Příklady monoidů

Jsou následující struktury monoidy?

- $(\mathbb{N}, +)$, přirozená čísla se sčítáním
  $\Rightarrow$ Ano, (komutativní) monoid.
- $(\mathbb{N}, -)$, přirozená čísla s odčítáním
  $\Rightarrow$ Ne (není ani grupoidem).
- $(\mathbb{N}, \textbf{min})$, přirozená čísla s minimem
  $\Rightarrow$ Ne (neexistuje neutrální prvek), ale je pologrupa.
- $(\mathbb{N}, \textbf{max})$, přirozená čísla s maximem
  $\Rightarrow$ Ano, (komutativní) monoid.
- $([...], ++)$, seznamy se zřetězením
  $\Rightarrow$ Ano, (NEkomutativní) monoid.
- $(\{f \mid f :: a \to a\}, .)$, funkce typu a `->` a se skládáním
  $\Rightarrow$ Ano, (NEkomutativní) monoid.
- (`Config`, $\bigoplus$), datový typ konfigurace s operací $\bigoplus$
  $\Rightarrow$ Ano, (NEkomutativní) monoid!

# A zpátky k Haskellu...

# Typová třída Semigroup

```
class Semigroup a where
    (<>) :: a -> a -> a
    sconcat :: GHC.Base.NonEmpty a -> a
    stimes :: Integral b => b -> a -> a
```

- nejmenší nezbytná definice: (<>)
- musí splňovat pravidlo asociativity:
  x <> (y <> z) ≡ (x <> y) <> z
- v **Prelude** jen (<>), více v **Data.Semigroup**
- lze použít alternativní předdefinované implementace `stimes` pro monoidy a/nebo idempotentní operaci; např.:

  ```
  stimes = stimesMonoid
  ```

# Typová třída Monoid

```haskell
class Semigroup a => Monoid a where
    mempty :: a
    mappend :: a -> a -> a -- = (<>)
    mconcat :: [a] -> a    -- = foldr mappend mempty
```

- musí splňovat pravidla:
  - levá identita: `mempty <> x ≡ x`
  - pravá identita: `x <> mempty ≡ x`
  - (asociativita: `x <> (y <> z) ≡ (x <> y) <> z`)
  - řetězení: `mconcat ≡ foldr (<>) mempty`
- užitečné knihovní instance v **`Data.Monoid`**

★ **`Semigroup`** je nadtřídou teprve od `base-4.11.0.0` (GHC 8.4)

# Monoid [a]

- Jak vypadá instance **Monoid**u pro seznamy?

# Monoid [a]

- Jak vypadá instance **Monoid**u pro seznamy?

```haskell
instance Semigroup [a] where
    (<>) = (++)
instance Monoid [a] where
    mempty  = []
```

# Monoid `Maybe a`

- Jak bude vypadat instance pro **`Maybe`** a?

# Monoid Maybe a

- Jak bude vypadat instance pro `Maybe a`?

```
instance Semigroup (Maybe a) where
    Nothing  <> b          = b
    (Just a) <> Nothing    = Just a
    (Just a) <> (Just b)   = Just (a <> b)
instance Monoid (Maybe a) where
    mempty = Nothing
```

# Monoid Maybe a

- Jak bude vypadat instance pro **Maybe** a?

```
instance Semigroup (Maybe a) where
    Nothing  <> b          = b
    (Just a) <> Nothing    = Just a
    (Just a) <> (Just b)   = Just (a <> b)
instance Monoid (Maybe a) where
    mempty = Nothing
```

- **Just** „přebíjí“ **Nothing**
- Neutrálním prvkem je **Nothing**
- Co musí splňovat typ a?

# Monoid Maybe a

- Jak bude vypadat instance pro **Maybe** a?

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing  <> b          = b
    (Just a) <> Nothing    = Just a
    (Just a) <> (Just b)   = Just (a <> b)
instance Semigroup a => Monoid (Maybe a) where
    mempty = Nothing
```

- **Just** „přebíjí“ **Nothing**
- Neutrálním prvkem je **Nothing**
- Co musí splňovat typ a?
  - Musí se jednat o instanci třídy **Semigroup**.

# Pologrupa Last

Knihovní pologrupa **Last** (z modulu **Data.Semigroup**):

```
newtype Last a = Last { getLast :: a }
instance Semigroup (Last a) where
    _ <> b = b
```

# Pologrupa Last

Knihovní pologrupa **Last** (z modulu **Data.Semigroup**):

```
newtype Last a = Last { getLast :: a }
instance Semigroup (Last a) where
    _ <> b = b
```

- Lze zúplnit na monoid obalením v **Maybe**.
- Analogicky existuje **First** a, kde (<>) = const.

# Pologrupa Last

Knihovní pologrupa **Last** (z modulu **Data.Semigroup**):

```
newtype Last a = Last { getLast :: a }
instance Semigroup (Last a) where
    _ <> b = b
```

- Lze zúplnit na monoid obalením v **Maybe**.
- Analogicky existuje **First** a, kde (<>) = const.

**Pozor.**

- neplést s knihovním *monoidem* **Data.Monoid.Last**:
  **newtype Last** a = **Last** { getLast :: **Maybe** a }
- zanedlouho bude z knihovny odstraněn
- doporučení: **import Data.Monoid hiding** (**First**, **Last**)

# Další knihovní monoidy

Existuje-li více monoidů nad jedním typem, používá se **newtype**:

- `Num a => (Product a)` – monoid vzhledem k násobení
- `Num a => (Sum a)` – monoid vzhledem ke sčítání
- `Any` – (`Bool`, `||`)
- `All` – (`Bool`, `&&`)
- `(Ord a, Bounded a) => (Max a)` – monoid vzhledem k operaci maximum
- `(Ord a, Bounded a) => (Min a)` – monoid vzhledem k operaci minimum
- `(Monoid a, Monoid b) => (a, b)` – kartézský součin monoidů `a` a `b`

★ `newtype Endo a = Endo { appEndo :: a -> a }`

# Řešení příkladu I

Původní:

```
data Config = Config
    { verbose :: Bool
    , options :: [String]
    } deriving (Eq, Show)
```

Nové:

```
type Config' = (Maybe (Last Bool), [String])
```

# Řešení příkladu I

Původní:

```
data Config = Config
    { verbose :: Bool
    , options :: [String]
    } deriving (Eq, Show)
```

Nové:

```
type Config' = (Maybe (Last Bool), [String])
```

★ Zkuste napsat řešení pomocí `Endo Config`.

# Foldable

# Typová třída `Foldable`

Kontejner, který lze lineárně projít a hodnoty „splácnout“.

```
class Foldable t where
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldMap :: Monoid m => (a -> m) -> t a -> m
    fold :: Monoid m => t m -> m
```

- Nejmenší nezbytná definice: `foldMap` | `foldr`.
- Nemusí splňovat žádné rovnosti (na rozdíl od jiných základních tříd).
- Definováno v `Prelude`, další funkce v **`Data.Foldable`**.

# Definice pro BinTree

```haskell
instance Foldable BinTree where
    foldMap f (Node a l r) = f a <> foldMap f l <> foldMap f r
    foldMap _ Leaf = mempty
```

Pozor! Obecný *fold* připojuje vždy jen jednu hodnotu.
Tj. ke každé struktuře se chová jako k seznamu.
Tj. `foldr` má v argumentu ve skutečnosti jen binární funkci `f` (na rozdíl od `treeFold` z kurzu IB015).

# Užitečné funkce

Automaticky získáme mnoho třídních funkcí pro každé **Foldable** t:

- `foldl :: (b -> a -> b) -> b -> t a -> b`
- `foldl' :: (b -> a -> b) -> b -> t a -> b`
  - Operátor je aplikován striktně.
- `toList :: t a -> [a]`
- `null :: t a -> Bool`
- `length :: t a -> Int`
- `elem :: Eq a => a -> t a -> Bool`
- `maximum, minimum :: Ord a => t a -> a`
- `sum, product :: Num a => t a -> a`

Základní definice funkcí nemusí být nutně optimální. Například `elem` pro **Map** je předefinován.

# foldMap vs. foldr

- Obě funkce jsou ekvivalentní
  - `foldMap :: Monoid m => (a -> m) -> t a -> m`
  - `foldr :: (a -> b -> b) -> b -> t a -> b`
- Není těžké vytvořit `foldMap`, pokud je `foldr` definováno.
- Jak vytvořit `foldr` pomocí `foldMap`?

# foldMap vs. foldr

- Obě funkce jsou ekvivalentní
  - `foldMap :: Monoid m => (a -> m) -> t a -> m`
  - `foldr :: (a -> b -> b) -> b -> t a -> b`
- Není těžké vytvořit `foldMap`, pokud je `foldr` definováno.
- Jak vytvořit `foldr` pomocí `foldMap`?
  - Stačí přeuzávorkovat typ `foldr`.
  - `(a -> b -> b) -> b -> t a -> b`
    $\rightarrow$ `(a -> (b -> b)) -> t a -> (b -> b)`
  - Víme že funkce `b -> b` tvoří monoid – v knihovně typ **`Endo`**.

$\Rightarrow$ celý koncept „foldování“ lze definovat v řeči monoidů.

- **`Foldable`** popisuje způsob procházení.
- **`Monoid`** popisuje způsob skládání hodnot.

# ★ Automatické odvozování instancí

U některých typů jsou instance „jasné“, „triviální“ a „mechanické“.

Příklad: instance monoidu pro součin monoidů:

```
data Config = Config { verbose :: Maybe (Last Bool)
                     , options :: [String]
                     } -- deriving Monoid :(
(Config v1 o1) <> (Config v2 o2) = Config (v1 <> v2) (o1 <> o2)
mempty = Config mempty mempty
```

# ★ Automatické odvozování instancí

U některých typů jsou instance „jasné“, „triviální“ a „mechanické“.

Příklad: instance monoidu pro součin monoidů:

```
data Config = Config { verbose :: Maybe (Last Bool)
                     , options :: [String]
                     } -- deriving Monoid :(
(Config v1 o1) <> (Config v2 o2) = Config (v1 <> v2) (o1 <> o2)
mempty = Config mempty mempty
```

- GHC zavádí typovou třídu **`Generic`**.
- Rozšíření DeriveGeneric umožňuje odvodit její instanci
- Balík `generic-deriving` umožňuje z instance **`Generic`** odvodit instance některých tříd, mj. monoidu či **`Traversable`**.
- ★ Existují i další rozšíření umožňující odvozování instancí.

# Samostatné programování

# Wrapper Xor

Podobně jako knihovní wrappery **Any** a **All** nad **Bool** napište vlastní wrapper **MyXor** a implementujte pro něj instance **Semigroup MyXor** a **Monoid MyXor** tak, aby například

```
getMyXor $ foldMap MyXor [b1, b2, ..., bn]
```

bylo rovno aplikaci operace `xor` na hodnoty `b1`, `b2`, ..., `bn` typu **Bool**.

# Foldable pro stromy

Mějme následující datový typ pro stromy libovolné arity

```
data RoseTree a = Node a [RoseTree a] deriving (Show, Eq)
```

Napište instanci **Foldable RoseTree**.

# Součet hodnot ve stromě libovolné arity

Pomocí funkce `foldMap` a instance `Monoid (Sum a)` napište funkci

```
roseTreeSum :: RoseTree Int -> Int
```

která vypočítá součet hodnot v zadaném stromě.

# Vlastní akumulační funkce

Funkce `roseTreeSum :: RoseTree Int -> Int` jde ve skutečnosti napsat jako

```
roseTreeSum :: Ord a => RoseTree a -> a
roseTreeSum = sum
```

protože knihovní funkce `sum` je definovaná přesně pomocí `foldMap` a **`Sum`**.

Napište podobně pomocí `foldMap` vlastní verze knihovních funkcí

- `myMaximum, myMinimum :: Ord a => t a -> a`
- `mySum, myProduct :: Num a => t a -> a`
- `myLength :: t a -> Int`
- `myToList :: t a -> [a]`

které fungují nejen pro **`RoseTree`**, ale pro libovolný **`Foldable`** typ.

# Skládání hodnocení řetězců (Twitter algorithm lite)

Zjistěte, jak funguje instance `Monoid b => Monoid (a -> b)`.

Poté uvažte seznam funkcí typu `String -> Int`, které každému řetězci přiřadí nějaké hodnocení.

Pomocí funkce `foldMap` a instancí `Monoid (a -> b)`, `Monoid (Max a)` a `Monoid (Sum a)` napište funkce

- `maxVal :: [String -> Int] -> String -> Int`, která pro zadaný řetězec vypočítá největší hodnocení, které vrátila nějaká vstupní funkce.
- `totalVal :: [String -> Int] -> String -> Int`, která pro zadaný řetězec vypočítá součet všech hodnocení, které vrátila nějaká vstupní funkce.

★ Co kdyby některé funkce mohly být aplikovatelné a jiné ne (tj. byly by typu `String -> Maybe Int`) a ty, které vrací hodnocení `Nothing`, bychom chtěli vynechávat a vracet `Maybe Int`?