Monoidy, `Foldable`, `Traversable`

IB016 Seminář z funkcionálního programování

Martin Kurečka, Adam Matoušek

(původní autoři slajdů Vladimír Štill, Martin Ukrop)

Fakulta informatiky, Masarykova univerzita

Jaro 2022

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 ⊕?

Vlastnosti funkce $\bigoplus$: uzavřenost

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

Ano, protože:

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

Vlastnosti operace $\oplus$: asociativita

Záleží na pořadí zpracovávání parametrů?

$$(\texttt{-v} \oplus \texttt{-opt=o1}) \oplus \texttt{-q}$$

vs.

$$\texttt{-v} \oplus (\texttt{-opt=o1} \oplus \texttt{-q})$$

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`

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

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.

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

Vlastnosti funkce `union`

- Spojení dvou struktur (`Map k a`) vytvoří novou strukturu (`Map k a`).
  - `union :: Ord k => Map k a -> Map k a -> Map k a`
- Nezáleží na „uzávorkování“ spojení `Map` $\Rightarrow$ asociativita
- Nezáleží na pořadí spojovaných `Map` $\Rightarrow$ komutativita
- Prázdná struktura je neutrální prvek vůči spojení
- `empty :: Map k a`

$\Rightarrow$ 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
  $\Rightarrow$ 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

```
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?

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

Monoid `Maybe a`

- Jak bude vypadat instance pro `Maybe a`?

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

- 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])
```

★ Zkuste napsat řešení pomocí `Endo Config`.

`Foldable`

Typová třída `Foldable`

```
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`.
- Kontejner, který lze lineárně projít a hodnoty „splácnout“.
- Nemusí splňovat žádné rovnosti (na rozdíl od jiných základních tříd).
- Definováno v `Prelude`, další funkce v `Data.Foldable`.

`foldMap` vs. `foldr`

- Obě funkce jsou ekvivalentní
  - `foldMap :: Monoid m => (a -> m) -> t a -> m`
  - `foldr :: (a -> b -> b) -> b -> t a -> b`
- Je zřejmé jak 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.

Definice pro `BinTree`

```
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:

- `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.

Traversable

Motivace III: vyhodnocení `IO` operací

Mějme `Foldable` a `Functor` kontejner (např. `[]`), do kterého chceme načíst hodnoty uživatele.

- Začínáme se seznamem `[String]` obsahujícím otázky.
- V každém uzlu seznamu můžeme vytvořit akci typu `IO String`, která vypíše otázku a načte odpověď.
  - Využíváme toho, že `[]` je funktor.
- Potřebovali bychom vytvořit jedinou akci typu `IO [String]`.
- To se nám nepodaří, pokud nevyužíváme dalších vlastností `IO`.

- `fmap` zachovává strukturu, hodnoty spolu neinteragují.
- `foldMap` nezachovává strukturu, hodnoty spolu interagují.
- Potřebujeme funkci, která umí obojí.

Typová třída `Traversable`

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f =>
        (a -> f b) -> t a -> f (t b)
    sequenceA :: Applicative f =>
        t (f a) -> f (t a)
```

- Minimální definice `traverse` | `sequenceA`.
- Kontejner na data typu `a`, kterým lze projít a zachovat strukturu dat.
- Definováno v `Prelude`, další funkce v `Data.Foldable`.

Rovnosti pro `Traversable`

Musí platit rovnosti, které kontroluje programátor

- Obě funkce by se měly chovat hezky k transformacím `Applicative` (tj. funkce respektující `<*>`).
- Pokud je funktor `f` identita (`Identity`), pak by se měla `sequenceA` chovat jako identita a `traverse` by nemělo měnit strukturu kontejneru `t`.
- Přesné rovnosti v dokumentaci.

Definice pro seznam

```haskell
instance Traversable [] where
    traverse :: ... => (a -> f b) -> [a] -> f [b]
    traverse f (x:xs) =
        liftA2 (:) (f x) (traverse f xs)
    traverse _ [] = pure []

    traverse' f (x:xs) = do
        h <- f x
        rest <- traverse f xs
        return $ h : rest
    traverse' _ [] = pure []
```

Definice pro binární stromy

```
instance Traversable BinTree where
    traverse :: ... => (a -> f b) -> BT a -> f (BT b)
    traverse f (Node a l r) =
        liftA3
            Node
            (f a)
            (traverse f l)
            (traverse f r)
    traverse _ Leaf = pure Leaf
```

★ Souvislost s `Foldable`

- Není vidět spojení s `Foldable`.
- Existuje instance `Applicative`, která se chová jako akumulátor → můžeme definovat `foldMap` pomocí `traverse`.

- Aplikativní funktor `Const m`.
- Podobá se již zmiňované monádě písaře `([],)`, která má akumulátor v první složce.
- `Const m` ∼ `(m, ())`

★ Aplikativní funktor `Const m`

```haskell
newtype Const a b = Const { getConst :: a }

instance Functor (Const a) where
  fmap f (Const a) = Const a

instance Monoid a => Applicative (Const a) where
  pure _ = Const mempty
  Const x <*> Const y = Const (x `mappend` y)

foldMap :: (...) => (a -> m) -> t a -> m
foldMap f = getConst . traverse (Const . f)
```

Užitečné funkce

- `sequence :: Monad m => t (m a) -> m (t a)`
  - Stejné jako `sequenceA` pro monády – historický relikt.
- `mapM :: Monad m => (a -> m b) -> t a -> m (t b)`
  - Stejné jako `traverse` pro monády.
- `traverse_ :: (Applicative f, Foldable t) => (a -> f b) -> t a -> f ()`
- `sequence_ :: (Applicative f, Foldable t) => t (f a) -> f ()`
  - Jako odpovídající funkce, ale zapomínají výsledek.
  - Tj. jenom spouští dané akce (např. IO).
  - Nepotřebují si pamatovat strukturu → `t` nemusí být `Traversable`.

★ 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.