

# Monoidy, zpracování argumentů příkazové řádky

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2016

# Motivace: zpracování argumentů příkazové řádky

Chtěli bychom zpracovat argumenty příkazové řádky jako nastavení programu.

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

- `-v` (*verbose*) zapíná ladící výstupy
- `-q` (*quiet*) vypíná ladící výstupy
- program se vždy chová podle posledního přepínače `-v/-q`
- `-opt` umožňuje přidat programu libovolný textový argument
- výchozí nastavení je bez ladících 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 vlastně zodpovídá nějakému nastavení
- nedaly by se jednoduše „spojit“ nějakou vhodnou funkcí?

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

# Představa zpracování

- každý přepínač vlastně zodpovídá nějakému nastavení
- nedaly by se jednoduše „spojit“ nějakou vhodnou funkcí?

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

```
Config {  
  verbose  
  = True  
}
```

$\oplus$

```
Config {  
  options  
  = ["o1"]  
}
```

$\oplus$

```
Config {  
  verbose  
  = False  
}
```

$\oplus$

```
Config {  
  options  
  = ["o2"]  
}
```

# Představa zpracování

- každý přepínač vlastně zodpovídá nějakému nastavení
- nedaly by se jednoduše „spojit“ nějakou vhodnou funkcí?

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

```
Config {  
  verbose  
  = True  
}
```

$\oplus$

```
Config {  
  options  
  = ["o1"]  
}
```

$\oplus$

```
Config {  
  verbose  
  = False  
}
```

$\oplus$

```
Config {  
  options  
  = ["o2"]  
}
```

- Jaké vlastnosti má funkce  $\oplus$ ?

## Vlastnosti operace $\oplus$ : uzavřenost

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

# Vlastnosti operace $\oplus$ : uzavřenost

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

Ano, protože:

- Operace  $\oplus$  má správný typ.  
 $\oplus :: \text{Config} \rightarrow \text{Config} \rightarrow \text{Config}$
- Můžeme ji zdefinovat tak, aby jejím výsledkem nikdy nebyla neplatná konfigurace.



# Vlastnosti operace $\oplus$ : asociativita

Je operace  $\oplus$  asociativní?

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

# Vlastnosti operace $\oplus$ : asociativita

Je operace  $\oplus$  asociativní?

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

Ano, pořadí zpracování by nemělo ovlivnit výslednou konfiguraci.

# Vlastnosti operace $\oplus$ : komutativita

Je operace  $\oplus$  komutativní?

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

# Vlastnosti operace $\oplus$ : komutativita

Je operace  $\oplus$  komutativní?

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

Ne, na pořadí záleží.

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

# 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

Ano, neutrálním prvkem by měla být výchozí konfigurace.

```
defaultConfig :: Config
defaultConfig = Config
  { verbose = NotSet
  , options = []
  }
```

## Config: nová definice

Upravíme tedy datový typ Config následovně:

```
data Flag a = Set a
            | NotSet
            deriving (Eq, Show)
```

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

# Monoidy: matematická definice

Monoid<sup>1</sup> je algebraická struktura. Je to grupoid  $(M; \cdot)$ , tedy množina  $M$  s binární operací  $\cdot : M \times M \rightarrow M$ , a těmito axiomy:

- Asociativita:  $\forall x, y, z \in M . (x \cdot y) \cdot z = x \cdot (y \cdot z)$
- Neutrální prvek:  $\exists e \in M \forall x \in M . x \cdot e = e \cdot x = x$

Někdy se uvádí i následující axiom plynoucí z definice binární operace grupoidu:

- $\forall x, y \in M . x \cdot y \in M$

---

<sup>1</sup>Definice převzata z <https://cs.wikipedia.org/wiki/Monoid>



# Typová třída Monoid

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

- definováno v Data.Monoid
- <> je infixové synonymum pro mappend (v Data.Monoid)
- mnoho užitečných knihovních instancí
- opět několik pravidel, která musí platit
  - levá identita:  $\text{mempty} \langle \rangle x \equiv x$
  - pravá identita:  $x \langle \rangle \text{mempty} \equiv x$
  - asociativita:  $x \langle \rangle (y \langle \rangle z) \equiv (x \langle \rangle y) \langle \rangle z$

Udělejme tedy instanci typu `Config` pro třídu `Monoid`:

# Instance pro konfigurace

Udělejme tedy instanci typu `Config` pro třídu `Monoid`:

```
instance Monoid Config where
  mempty = defaultConfig
  c1 `mappend` c2 =
    Config (verbose c1 `mappend` verbose c2)
           (options c1 `mappend` options c2)
```

Teď ale potřebujeme ještě:

- instanci pro datový typ `Flag`
- instanci pro seznamy

Jak chceme, aby se chovala instance pro Flag a?

Jak chceme, aby se chovala instance pro Flag a?

```
instance Monoid (Flag a) where
  mempty = NotSet
  _ `mappend` (Set x) = Set x
  x `mappend` NotSet = x
```

Jak chceme, aby se chovala instance pro seznamy?

Jak chceme, aby se chovala instance pro seznamy?

```
instance Monoid [a] where  
  mempty = []  
  x `mappend` y = x ++ y
```

(je součástí knihovny)

A teď máme elegantní, rozšiřitelný systém parsující argumenty příkazové řádky `:-}`.

... který navyše využívá pěknou algebraickou teorii.



# Monoid: knihovní instance

V knihovně je vícero instancí pro Monoid:

- `[a]` – instance stejná, jako jsme vytvořili my
- `Last a` – odpovídá našemu `Flag a` (je to vlastně instance pro `Maybe a`)
- `First a` – jako `Flag a`, jenom prioritu má první výskyt definované hodnoty (ne poslední)
- `Any` – `Bool` s operací `||`
- `All` – `Bool` s operací `&&`
- `Num a => (Product a)` – numerická instance kolem operace násobení
- `Num a => (Sum a)` – numerická instance kolem sčítání
- ...

<https://wiki.haskell.org/Typeclassopedia>

*The goal of this document is to serve as a starting point for the student of Haskell wishing to gain a firm grasp of its standard type classes. The essentials of each type class are introduced, with examples, commentary, and extensive references for further reading.*

- třídy, které už známe (Functor, Applicative, ...)
- třídy, které na nich staví (Monoid  $\rightarrow$  Foldable)
- třídy ještě vyšší abstrakce (MonadTrans, Arrow)

# Úkol: rozšíření pro další volby

Rozšiřte program o možnost následujících argumentů:

- `-printer=lj4a` nastaví tiskárnu, identifikovanou řetězcem. Vždy se použije první zadaná tiskárna.
- Předělejte příznak *verbose* tak, aby úroveň výstupů byla charakterizována přirozeným číslem. Použije se vždy maximální z nastavených hodnot, pokud není použit přepínač `-q`. Tedy argumenty `-v=1` `-v=5` `-v=2` nastaví level na 5 a argumenty `-v=1` `-q` `-v=6` na 0.

Zkuste si v *Typeclassopedia* přečíst o typové třídě `Foldable`, která zobecňuje struktury, přes které se dá „foldovat“.