Čočky a další optika

IB016 Seminář z funkcionálního programování

Mnoho autorů napříč věky

Fakulta informatiky, Masarykova univerzita

Jaro 2021

Motivace

```haskell
type User = String

data Issue = Issue { reporter :: User
                   , assignee :: Maybe User
                   , weight :: Int
                   , closed :: Bool
                   } deriving Show

data Repo = Repo { maintainer :: User
                 , issues :: [Issue]
                 } deriving Show

data Project = Project { codeRepo :: Repo
                       , wikiRepo :: Maybe Repo
                       } deriving Show
```

Motivace

```
data Issue = Issue { reporter :: User, assignee :: Maybe User
                   , weight :: Int, closed :: Bool }
data Repo = Repo { maintainer :: User , issues :: [Issue] }
data Project = Project { codeRepo :: Repo , wikiRepo :: Maybe Repo }
```

Úkol: v projektu `p` zavřít první issue v repositáři s kódem.

Motivace

```haskell
data Issue = Issue { reporter :: User, assignee :: Maybe User
                   , weight :: Int, closed :: Bool }
data Repo = Repo { maintainer :: User , issues :: [Issue] }
data Project = Project { codeRepo :: Repo , wikiRepo :: Maybe Repo }
```

Úkol: v projektu `p` zavřít první issue v repositáři s kódem.

- V „běžných“ jazycích (C, Java, . . .):
  `p.codeRepo.issues[0].closed = true;`

Motivace

```haskell
data Issue = Issue { reporter :: User, assignee :: Maybe User
                   , weight :: Int, closed :: Bool }
data Repo = Repo { maintainer :: User , issues :: [Issue] }
data Project = Project { codeRepo :: Repo , wikiRepo :: Maybe Repo }
```

Úkol: v projektu `p` zavřít první issue v repositáři s kódem.

- V „běžných“ jazycích (C, Java, ...):
  ```
  p.codeRepo.issues[0].closed = true;
  ```
- V Haskellu (s využitím syntaxe záznamů):
  ```haskell
  p { codeRepo = (codeRepo p) { issues =
        i { closed = True } : is
    } } where (i:is) = issues (codeRepo p)
  ```
- Posláním čoček je dělat podobné věci krásněji.

Hezčí řešení: settery a updatery

Napíšeme si „setter“:

```
setClosed :: Bool -> (Issue -> Issue)
```

Potřebujeme ho aplikovat hluboko uvnitř; pomůžeme si updatery:

```
overNth :: Int -> (Issue -> Issue) -> ([Issue] -> [Issue])
overIssues :: ([Issue] -> [Issue]) -> (Repo -> Repo)
overCodeRepo :: (Repo -> Repo) -> (Project -> Project)
```

Hezčí řešení: settery a updatery

Napíšeme si „setter“:

```
setClosed :: Bool -> (Issue -> Issue)
```

Potřebujeme ho aplikovat hluboko uvnitř; pomůžeme si updatery:

```
overNth :: Int -> (Issue -> Issue) -> ([Issue] -> [Issue])
overIssues :: ([Issue] -> [Issue]) -> (Repo -> Repo)
overCodeRepo :: (Repo -> Repo) -> (Project -> Project)
```

Řešení úkolu nyní:

```
(overCodeRepo . overIssues . overNth 0 . setClosed) True p
```

Hezčí řešení: settery a updatery

Napíšeme si „setter“:

```
setClosed :: Bool -> (Issue -> Issue)
```

Potřebujeme ho aplikovat hluboko uvnitř; pomůžeme si updatery:

```
overNth :: Int -> (Issue -> Issue) -> ([Issue] -> [Issue])
overIssues :: ([Issue] -> [Issue]) -> (Repo -> Repo)
overCodeRepo :: (Repo -> Repo) -> (Project -> Project)
```

Řešení úkolu nyní:

```
(overCodeRepo . overIssues . overNth 0 . setClosed) True p
```

Hle! Updatery se skvěle skládají do hlouběji zanořených updaterů:

```
overFirstIssue :: (Issue -> Issue) -> (Project -> Project)
overFirstIssue = overCodeRepo . overIssues . overNth 0

(overFirstIssue . setClosed) True p
```

První polovina čočky

Updatery můžeme použít i bez setteru:

```
overWeight :: (Int -> Int) -> (Issue -> Issue)
(overFirstIssue . overWeight) (+10) p       -- zvýší váhu
```

První polovina čočky

Updatery můžeme použít i bez setteru:

```
overWeight :: (Int -> Int) -> (Issue -> Issue)
(overFirstIssue . overWeight) (+10) p       -- zvýší váhu
```

Máme-li updater, samostatný setter vlastně vůbec nepotřebujeme:

```
(overFirstIssue . overWeight) (\_ -> 42) p -- nastaví váhu
```

První polovina čočky

Updatery můžeme použít i bez setteru:

```
overWeight :: (Int -> Int) -> (Issue -> Issue)
(overFirstIssue . overWeight) (+10) p       -- zvýší váhu
```

Máme-li updater, samostatný setter vlastně vůbec nepotřebujeme:

```
(overFirstIssue . overWeight) (\_ -> 42) p -- nastaví váhu
```

Ale je to užitečná pomocná funkce:

```
set updater newval obj = updater (\_ -> newval) obj
set (overFirstIssue . overWeight) 42 p
```

První polovina čočky

Updatery můžeme použít i bez setteru:

```
overWeight :: (Int -> Int) -> (Issue -> Issue)
(overFirstIssue . overWeight) (+10) p       -- zvýší váhu
```

Máme-li updater, samostatný setter vlastně vůbec nepotřebujeme:

```
(overFirstIssue . overWeight) (\_ -> 42) p -- nastaví váhu
```

Ale je to užitečná pomocná funkce:

```
set updater newval obj = updater (\_ -> newval) obj
set (overFirstIssue . overWeight) 42 p
```

Pro každý atribut napíšeme updater a „zadarmo“ můžeme vcelku hezky měnit libovolně zanořené hodnoty!

Co gettery?

```haskell
data Issue = Issue { reporter :: User, assignee :: Maybe User
                   , weight :: Int, closed :: Bool }
data Repo = Repo { maintainer :: User , issues :: [Issue] }
data Project = Project { codeRepo :: Repo , wikiRepo :: Maybe Repo }
```

Nový úkol: zjistit, zda je první issue v repositáři s kódem zavřená.

- V „běžných“ jazycích:
  `p.codeRepo.issues[0].closed`
- V Haskellu (s využitím getterů vyrobených ze záznamů):
  `closed . head . issues . codeRepo $ p`
- Hezké, ale pořadí je opačné než při nastavování hodnot. Nepřijatelné!

Otáčíme gettery

Samotný getter už máme díky použití záznamů:

```
closed :: Issue -> Bool
```

Získanou hodnotu je potřeba vytáhnout z hlubin struktury:

```
fromNth :: Int -> (Issue -> b) -> ([Issue] -> b)
fromIssues :: ([Issue] -> b) -> (Repo -> b)
fromCodeRepo :: (Repo -> b) -> (Project -> b)
```

Otáčíme gettery

Samotný getter už máme díky použití záznamů:

```
closed :: Issue -> Bool
```

Získanou hodnotu je potřeba vytáhnout z hlubin struktury:

```
fromNth :: Int -> (Issue -> b) -> ([Issue] -> b)
fromIssues :: ([Issue] -> b) -> (Repo -> b)
fromCodeRepo :: (Repo -> b) -> (Project -> b)
```

Stav první issue nyní získáme pomocí:

```
(fromCodeRepo . fromIssues . fromNth 0) closed p
```

Otáčíme gettery

Samotný getter už máme díky použití záznamů:

```
closed :: Issue -> Bool
```

Získanou hodnotu je potřeba vytáhnout z hlubin struktury:

```
fromNth :: Int -> (Issue -> b) -> ([Issue] -> b)
fromIssues :: ([Issue] -> b) -> (Repo -> b)
fromCodeRepo :: (Repo -> b) -> (Project -> b)
```

Stav první issue nyní získáme pomocí:

```
    (fromCodeRepo . fromIssues . fromNth 0) closed p
p & (fromCodeRepo . fromIssues . fromNth 0) closed
```

`(&) = flip ($)`, z modulu `Data.Function`

Otáčíme gettery

Samotný getter už máme díky použití záznamů:

```
closed :: Issue -> Bool
```

Získanou hodnotu je potřeba vytáhnout z hlubin struktury:

```
fromNth :: Int -> (Issue -> b) -> ([Issue] -> b)
fromIssues :: ([Issue] -> b) -> (Repo -> b)
fromCodeRepo :: (Repo -> b) -> (Project -> b)
```

Stav první issue nyní získáme pomocí:

```
    (fromCodeRepo . fromIssues . fromNth 0) closed p
p & (fromCodeRepo . fromIssues . fromNth 0) closed
```

`(&) = flip ($)`, z modulu `Data.Function`

Hle! „Extraktory“ se skvěle skládají do hlouběji zanořených extraktorů:

```
fromFirstIssue :: (Issue -> b) -> (Project -> b)
fromFirstIssue = fromCodeRepo . fromIssues . fromNth 0

fromFirstIssue closed p
```

Druhá polovina čočky

Nic nám nebrání mít extraktor i pro „listové“ atributy:

```
fromWeight :: (Int -> b) -> (Issue -> b)
fromWeight g issue = g (weight issue)
```

Druhá polovina čočky

Nic nám nebrání mít extraktor i pro „listové“ atributy:

```
fromWeight :: (Int -> b) -> (Issue -> b)
fromWeight g issue = g (weight issue)
```

Máme-li extraktor, samostatný getter vlastně vůbec nepotřebujeme:

```
(fromFirstIssue . fromWeight) id p
```

Druhá polovina čočky

Nic nám nebrání mít extraktor i pro „listové“ atributy:

```
fromWeight :: (Int -> b) -> (Issue -> b)
fromWeight g issue = g (weight issue)
```

Máme-li extraktor, samostatný getter vlastně vůbec nepotřebujeme:

```
(fromFirstIssue . fromWeight) id p
```

Ale je to užitečná pomocná funkce:

```
view extractor obj = extractor id obj
view (fromFirstIssue . fromWeight) p
```

Druhá polovina čočky

Nic nám nebrání mít extraktor i pro „listové“ atributy:

```
fromWeight :: (Int -> b) -> (Issue -> b)
fromWeight g issue = g (weight issue)
```

Máme-li extraktor, samostatný getter vlastně vůbec nepotřebujeme:

```
(fromFirstIssue . fromWeight) id p
```

Ale je to užitečná pomocná funkce:

```
view extractor obj = extractor id obj
```

```
view (fromFirstIssue . fromWeight) p
```

Pro každý atribut napíšeme extraktor a můžeme vcelku hezky číst libovolně zanořené hodnoty!

Myšlenka čočky

Máme:

```
set  (overCodeRepo . overIssues . overNth 0 . overClosed) True
view (fromCodeRepo . fromIssues . fromNth 0 . fromClosed)
```

Myšlenka čočky

Máme:

```
set  (overCodeRepo . overIssues . overNth 0 . overClosed) True
view (fromCodeRepo . fromIssues . fromNth 0 . fromClosed)
```

Chceme:

```
set  (codeRepo' . issues' . nth' 0 . closed') True
view (codeRepo' . issues' . nth' 0 . closed')
```

Myšlenka čočky

Máme:

```
set  (overCodeRepo . overIssues . overNth 0 . overClosed) True
view (fromCodeRepo . fromIssues . fromNth 0 . fromClosed)
```

Chceme:

```
set  (codeRepo' . issues' . nth' 0 . closed') True
view (codeRepo' . issues' . nth' 0 . closed')
```

Čočka (lens) $\approx$ kombinace „updateru“ a „extraktoru“

- ulehčuje práci se zanořenými datovými strukturami
- toutéž čočkou lze číst i zapisovat
- čočky lze skládat do větších, brát jako argumenty, vracet,...
- při změně datového typu stačí změnit pár čoček

Spojení polovin čočky

První řešení: datový typ pro čočky.

```
data Lens s a = Lens { over :: (a -> a) -> s -> s
                     , from :: (a -> b) -> s -> b }
```

(Zjednodušený kód, není validní Haskell)

Čočka typu `Lens s a` „ostří“ ve struktuře `s` na atribut typu `a`.

Jak vypadá čočka ostřící v `Project` na `codeRepo`?

Spojení polovin čočky

První řešení: datový typ pro čočky.

```
data Lens s a = Lens { over :: (a -> a) -> s -> s
                     , from :: (a -> b) -> s -> b }
```

(Zjednodušený kód, není validní Haskell)

Čočka typu `Lens s a` „ostří“ ve struktuře `s` na atribut typu `a`.

Jak vypadá čočka ostřící v `Project` na `codeRepo`?

```
codeRepo' :: Lens Project Repo
codeRepo' = Lens
  { over = ( \f p -> p { codeRepo = f (codeRepo p) } )
  , from = ( \g p -> g (codeRepo p) }
```

Příklad použití: převzetí moci správcem jiného projektu:

```
p1 & set (over codeRepo' . over maintainer')
         (p2 & view (from codeRepo' . from maintainer'))
```

Proč ne takto

```
data Lens s a = Lens { over :: (a -> a) -> s -> s
                     , from :: (a -> b) -> s -> b }
p1 & set (over codeRepo' . over maintainer')
         (p2 & view (from codeRepo' . from maintainer'))
```

To jsme si moc nepomohli – skládáme funkce získané přes `over` nebo `from` podle toho, jestli zapisujeme nebo čteme. Neskládáme celé čočky.

- napíšeme vlastní skládání
- upravíme příslušně `set` a `view`
- ```
  codeMaintainer :: Lens Project User
  codeMaintainer = codeRepo' <.> maintainer'
  p1 & set codeMaintainer
           (p2 & view codeMaintainer)
  ```
  ```

Proč ne takto

```
data Lens s a = Lens { over :: (a -> a) -> s -> s
                     , from :: (a -> b) -> s -> b }
p1 & set (over codeRepo' . over maintainer')
         (p2 & view (from codeRepo' . from maintainer'))
```

To jsme si moc nepomohli – skládáme funkce získané přes `over` nebo `from` podle toho, jestli zapisujeme nebo čteme. Neskládáme celé čočky.

- napíšeme vlastní skládání
- upravíme příslušně `set` a `view`
- ```
  codeMaintainer :: Lens Project User
  codeMaintainer = codeRepo' <.> maintainer'
  p1 & set codeMaintainer
           (p2 & view codeMaintainer)
  ```

. . . nebo se pokusíme updater (`over`) a extraktor (`from`) zobecnit do jediné, snadno skladatelné funkce!
  ```

Čočka, druhý pokus

Čočkou bude jakési „vylepšené `over`“:

```
type Lens s a = Functor f => (a -> f a) -> s -> f s
```

(Taková funkce je užitečná sama o sobě. Můžeme třeba chtít, aby „updater“ selhal při zadání neplatné hodnoty (`Maybe`) nebo zaprotokoloval do database, že došlo ke změně (`IO`).)

Čočka, druhý pokus

Čočkou bude jakési „vylepšené `over`“:

```
type Lens s a = Functor f => (a -> f a) -> s -> f s
```

(Taková funkce je užitečná sama o sobě. Můžeme třeba chtít, aby „updater“ selhal při zadání neplatné hodnoty (`Maybe`) nebo zaprotokoloval do database, že došlo ke změně (`IO`).)

Jak pomocí tohoto „nad-updateru“ napíšeme...

- `over :: Lens s a -> (a -> a) -> s -> s`

- `from :: Lens s a -> (a -> b) -> s -> b`

Čočka, druhý pokus

Čočkou bude jakési „vylepšené `over`“:

```
type Lens s a = Functor f => (a -> f a) -> s -> f s
```

(Taková funkce je užitečná sama o sobě. Můžeme třeba chtít, aby „updater“ selhal při zadání neplatné hodnoty (`Maybe`) nebo zaprotokoloval do database, že došlo ke změně (`IO`).)

Jak pomocí tohoto „nad-updateru“ napíšeme...

- `over :: Lens s a -> (a -> a) -> s -> s`
  - pomocí triviálního knihovního funktoru `Identity`, který nic zajímavého nedělá a je „funktorem, aby byl funktor“.
- `from :: Lens s a -> (a -> b) -> s -> b`

Čočka, druhý pokus

Čočkou bude jakési „vylepšené `over`“:

```
type Lens s a = Functor f => (a -> f a) -> s -> f s
```

(Taková funkce je užitečná sama o sobě. Můžeme třeba chtít, aby „updater“ selhal při zadání neplatné hodnoty (`Maybe`) nebo zaprotokoloval do database, že došlo ke změně (`IO`).)

Jak pomocí tohoto „nad-updateru“ napíšeme...

- `over :: Lens s a -> (a -> a) -> s -> s`
  - pomocí triviálního knihovního funktoru `Identity`, který nic zajímavého nedělá a je „funktorem, aby byl funktor“.
- `from :: Lens s a -> (a -> b) -> s -> b`
  - pomocí knihovního funktoru `Const b`, který ukrývá hodnotu, na kterou `fmap` nemůže.

Čočka, druhý pokus

Čočkou bude jakési „vylepšené `over`“:

```
type Lens s a = Functor f => (a -> f a) -> s -> f s
```

(Taková funkce je užitečná sama o sobě. Můžeme třeba chtít, aby „updater“ selhal při zadání neplatné hodnoty (`Maybe`) nebo zaprotokoloval do database, že došlo ke změně (`IO`).)

Jak pomocí tohoto „nad-updateru“ napíšeme...

- `over :: Lens s a -> (a -> a) -> s -> s`
  - pomocí triviálního knihovního funktoru `Identity`, který nic zajímavého nedělá a je „funktorem, aby byl funktor“.
- ~~`from`~~ `:: Lens s a -> (a -> b) -> s -> b`
  `view :: Lens s a -> s -> a`
  - pomocí knihovního funktoru `Const b`, který ukrývá hodnotu, na kterou `fmap` nemůže.
  - loučíme se s `from` – bylo užitečné jen na skládání, a to teď čočka umí pěkně sama od sebe.

Identity a Const

```haskell
type Lens s a = Functor f => (a -> f a) -> s -> f s
```

```haskell
newtype Identity a = Identity { runIdentity :: a }
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
```

Identity a Const

```haskell
type Lens s a = Functor f => (a -> f a) -> s -> f s


newtype Identity a = Identity { runIdentity :: a }
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)

over :: Lens s a -> (a -> a) -> s -> s
over lens f obj = runIdentity $ lens (Identity . f) obj
```

Identity a Const

```haskell
type Lens s a = Functor f => (a -> f a) -> s -> f s

newtype Identity a = Identity { runIdentity :: a }
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)

over :: Lens s a -> (a -> a) -> s -> s
over lens f obj = runIdentity $ lens (Identity . f) obj

newtype Const c a = Const { getConst :: c }
instance Functor (Const c) where
    fmap _ (Const y) = Const y
```

Identity a Const

```haskell
type Lens s a = Functor f => (a -> f a) -> s -> f s

newtype Identity a = Identity { runIdentity :: a }
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)

over :: Lens s a -> (a -> a) -> s -> s
over lens f obj = runIdentity $ lens (Identity . f) obj

newtype Const c a = Const { getConst :: c }
instance Functor (Const c) where
    fmap _ (Const y) = Const y

view :: Lens s a -> s -> a
view lens obj = getConst $ lens Const obj
```

Naše čočka – shrnutí

- Čočka typu `Lens s a` „ostří“ ve struktuře `s` na atribut typu `a`.
- Funkce `set`, `view` a `over` umí „zaostřený“ atribut číst a měnit.
- Skládaním čoček vzniká čočka „ostřící“ hlouběji do struktury.
- Čočka je vlastně funkce – „updater ve funktoru“:

```
type Lens s a = Functor f => (a -> f a) -> (s -> f s)
```

Naše čočka – shrnutí

- Čočka typu `Lens s a` „ostří“ ve struktuře `s` na atribut typu `a`.
- Funkce `set`, `view` a `over` umí „zaostřený“ atribut číst a měnit.
- Skládaním čoček vzniká čočka „ostřící“ hlouběji do struktury.
- Čočka je vlastně funkce – „updater ve funktoru“:

```
type Lens s a = Functor f => (a -> f a) -> (s -> f s)
```

Příklad čočky typu `Lens Project Repo`:

```
codeRepo' :: ... => (Repo -> f Repo) -> (Project -> f Project)
codeRepo' f p = flip Project (wikiRepo p) <$> f (codeRepo p)
```

Příklad použití: zvyšujeme prioritu

```
newp = p & over (codeRepo' . issues' . nth' 1 . weight')
                (+ 10)
```

Příklad použití: přidáváme správci povinnosti:

```
newp = p & set (codeRepo' . issues' . nth' 2 . assignee')
               (Just $ p & view (codeRepo' . maintainer'))
```

Balík `lens`

Balík `lens`

Knihovní balík `lens`:

- nejznámější Haskellová implementace čoček a jiné optiky
- „*batteries included*“ – mnoho funkcí i závislostí
- moduly `Control.Lens.*`
- definuje optiku pro knihovní typy
- umí být obecnější než naše čočky

Ukážeme si, jak s využitím balíku `lens`:

- . . . generovat čočky automaticky,
- . . . zpřehlednit kód pomocí operátorů,
- . . . číst a měnit víc hodnot najednou.

```
$ cabal new-update && cabal new-install --lib lens
```

lens: Automatické generování

```haskell
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens

type User = String
data Issue = Issue { _reporter :: User
                   , _assignee :: Maybe User
                   , _weight :: Int
                   , _closed :: Bool
                   } deriving Show

makeLenses ''Issue
```

- Nutno zapnout rozšíření TemplateHaskell.
- Vyrobí čočky pro atributy začínající podtržítkem.
- Čočky se jmenují bez podtržítka.
- Vše je možné nastavit.

`lens`: příklad použití

Čočky z `lens` můžeme používat skoro stejně[1] jako předtím:

```
root = p & view (codeRepo . maintainer)
newp = p & set (codeRepo . issues . ix 2 . assignee)
               (Just root)
newp = p & over (codeRepo . issues . ix 1 . weight)
                (+ 10)
```

[1] Ve jménech čoček nejsou apostrofy a místo `nth` existuje `ix` (jako „index“).

`lens`: příklad použití

Čočky z `lens` můžeme používat skoro stejně[1] jako předtím:

```
root = p & view (codeRepo . maintainer)
newp = p & set (codeRepo . issues . ix 2 . assignee)
               (Just root)
newp = p & over (codeRepo . issues . ix 1 . weight)
                (+ 10)
```

Nebo můžeme využít operátory:

```
root = p ^. codeRepo . maintainer
newp = p & codeRepo . issues . ix 2 . assignee .~ Just root
newp = p & codeRepo . issues . ix 1 . weight %~ (+ 10)
newp = p & codeRepo . issues . ix 1 . weight +~ 10
```

[1] Ve jménech čoček nejsou apostrofy a místo `nth` existuje `ix` (jako „index“).

Operátory – přehled

Balík obsahuje veliké množství operátorů, například:

- `objekt ^. optika` (getter[2])
- `objekt ^.. optika` („multi-getter“)
- `objekt ^? optika` (Maybe-getter)
- `objekt & optika .~ hodnota` (setter)
- `objekt & optika %~ funkce` (updater)
- `objekt & optika %%~ funkce` (updater „s efekty“)
- `objekt & optika +~ číslo` (přičtení)
- `objekt & optika <>~ hodnota` (při<>ení, např. (++))

[2] je-li výsledků víc, pak je sloučí pomocí <>, není-li žádný, vrací `mempty`

Panoptikum

Čočky (*lenses*)

- umožňují „zaostřit“ na jednu složku typu.

Prismata (*prisms*)

- umožňují „zaostřit“ na jeden hodnotový konstruktor.

Traversals

- umožňují „zaostřit“ na libovolný počet hodnot.

Panoptikum

Čočky (*lenses*)

- umožňují „zaostřit“ na jednu složku typu.

Prismata (*prisms*)

- umožňují „zaostřit“ na jeden hodnotový konstruktor.

Traversals

- umožňují „zaostřit“ na libovolný počet hodnot.

Musíme počítat s neúspěchem!

- Hodnotový konstruktor nemusí být přítomen: `Nothing ^. _Just`
- Hodnota nemusí být přítomna: `[1..5] ^. ix 10`
- Lze řešit pomocí `Maybe` (`^?`), seznamu (`^..`) nebo monoidu (`^.`)
- Setter při neúspěchu neudělá nic.

Panoptikum

Čočky (*lenses*)

- umožňují „zaostřit“ na jednu složku typu.

Prismata (*prisms*)

- umožňují „zaostřit“ na jeden hodnotový konstruktor.

Traversals

- umožňují „zaostřit“ na libovolný počet hodnot.

Musíme počítat s neúspěchem!

- Hodnotový konstruktor nemusí být přítomen: `Nothing ^. _Just`
- Hodnota nemusí být přítomna: `[1..5] ^. ix 10`
- Lze řešit pomocí `Maybe` (`^?`), seznamu (`^..`) nebo monoidu (`^.`)
- Setter při neúspěchu neudělá nic.

Všechny druhy optik je možné řetězit prostým skládáním funkcí.

```
[Just 3, Nothing, Just 4] ^.. traverse . _Just ⇝ [3,4]
```

Prisms

Prisma je další druh optiky:

- „zaostřuje“ na jednu hodnotu s možností selhání
  - typicky argument (či jejich n-tici) hodnotového konstruktoru
  - ★ ale třeba taky `prefixed` z `Data.List.Lens`
- pokud konstuktor nenajde, vypropaguje `Nothing` nebo `mempty`.
- `makePrisms ''Typ` vyrábí prismata tvaru `_HKonstruktor`.

Vyzkoušejte a srovnejte chování různých operátorů:

```
Right 42 ^.  _Right ~>* ?      Left 66 ^.  _Right ~>* ?
Right 42 ^?  _Right ~>* ?      Left 66 ^?  _Right ~>* ?
Right 42 ^.. _Right ~>* ?      Left 66 ^.. _Right ~>* ?
Right "r" ^. _Right ~>* ?      Left "l" ^. _Right ~>* ?
Right 8 & _Right *~ 2 ~>* ?    Left 5 & _Right *~ 2 ~>* ?

Right (Right "foo") & _Right . _Left %~ map toUpper ~>* ?
Right (Left "bar")  & _Right . _Left %~ map toUpper ~>* ?
```

Traversals

Traversal je zobecněním čoček a prismat.

- „zaostřuje“ na libovolný počet hodnot (klidně žádnou).
- výsledek podle použitého operátoru dostaneme jako seznam (`^..`), monoidový součin (`^.`) nebo jen první výsledek, existuje-li (`^?`).
- `traverse` – vybere všechny prvky seznamu (či jiného `Traversable`)

Vyzkoušejte a srovnejte chování různých operátorů:

```
[1..4] ^.  traverse ~> ?        [1..4] ^?  traverse ~> ?
[1..4] ^.. traverse ~> ?        [1..4] ^.  traverse . to Sum ~> ?
[1..4] & traverse +~ 10~> ?
[("a",1), ("b",2)] ^.  traverse . _1 ~>* ?
[("a",1), ("b",2)] ^.  traverse . _2 ~>* ?
[("a",1), ("b",2)] ^?  traverse . _2 ~>* ?
[("a",1), ("b",2)] ^.. traverse . _2 ~>* ?
[("a",1), ("b",2)] & traverse . _1 . traverse %~ toUpper ~>* ?
```

`_1` a `_2` jsou čočky ostřící na prvky n-tic.

Poznámka k psaní vlastních traversalů

Typ traversalů se liší od typu čoček požadavkem na `Applicative`:

```
type Traversal' s a = Applicative f => (a -> f a) -> s -> f s
```

Implementace bude v lecčem připomínat instanci třídy `Traversable`.
Příklad: výběr prvních dvou prvků trojice:

```
firstTwo :: Traversal' (a, a, b) a
-- " --  :: ... => (a -> f a) -> (a, a, b) -> f (a, a, b)
firstTwo f (x, y, z) = (,,) <$> f x <*> f y <*> pure z

(1, 2, 3) & firstTwo *~ 10 ⇝* (10, 20, 3)
```

Existující optika – přehled

Lze využít množství předdefinované optiky. Namátkou:

- `_1`, `_2`, ..., `both`, `each` – přístup k prvkům n-tic
- `_Just`, `_Right`, `_Left`, ... – výběr konstruktoru
- `ix n` – indexující traversal; umožňuje ostřit na
  - n-tý prvek seznamu
  - n-tý prvek m-tice[3]
  - prvek mapy s klíčem `n` (nemusí být číslo)
- `traverse` – výběr všech prvků kontejneru naráz
- `to f` – aplikuje na výsledek funkci
- `filtered p` – vybere hodnoty splňující predikát
- `_head`, `_last`, `_init` a `_tail` – traversaly pro části seznamu
- `worded`, `lined` – traversal přes slova/řádky řetězce

[3] Co pro m-tici musí platit, aby šel použít traversal `ix`?

Optika odjinud

K nemalému množství balíků existuje doprovodný balík s optikou.

- čočky na práci s datem a časem (`lens-datetime`)

```
postpone = issues . traverse . due . years +~ 1
```

- optika na parsování a procházení JSONu (`lens-aeson`):

```
jsonStr & key "flags" . nth 2 . _Bool .~ True
jsonStr & _Array . traverse . key "amount" . _Number -~ 1
```

- procházení XML à la XPath (`xml-lens`)

```
doc ^.. root . el "courses" ./ el "course" . attributeIs
↪  "faculty" "FI" ./ el "description" . text
```

- Podobně lze pracovat např. s YAMLem

Všechno se to sem nevleze. . .

Optika z balíku `lens` toho umí ještě víc

- čočky mohou měnit strukturu objektu:
  ```
  type Lens s t a b = Functor f => (a -> f b) -> s -> f t
  ```
- hierarchie optiky a pravidla pro validní optiku,
- práce s indexy výsledků („vypiš každou druhou issue“),
- „zpětný chod“ prismat (např. vkládání konstruktorů),
- isomorfismy (seznam a jeho obrácení, různé representace textu),
- operátory pro práci ve stavové monádě,
- ohromná spousta různých optik a kombinátorů,
- . . .

Další čtení v případě zájmu:

- tutoriál na School of Haskell
- balík Control.Lens.Tutorial