Haskell Workshop - Lenses

Carsten König

Juni 2021

Intro

`Lenses` sind zunächst nur die funktionale Variante eines **getter/setter** Paars.

Angenommen folgende Records:

```
data Name = Name
  { vorname :: String
  , nachname :: String
  } deriving Show


data Person = Person
  { name :: Name
  , email :: String
  } deriving Show


beispiel = Person (Name "Max" "Muster") "max.muster@mail.me"
```

Vorname ist dann einfach `vorname beispiel` und gesetzt werden kann er mittels Record-Update Syntax:

```
beispiel { vorname = "Karl" }
```

Bei *verschachtelten* Records wird das teilweise unangenehm.

Naive Lens

```
data Lens1 s a = Lens1
  { view1 :: s -> a
  , set1  :: a -> s -> s
  }

vornameLens1 :: Lens1 Name String
vornameLens1 = Lens1 vorname (\x n -> n { vorname = x })

nameLens1 :: Lens1 Person Name
nameLens1 = Lens1 name (\x p -> p { name = x })
```

Problem:

Was, wenn wir wir den Nachnamen einer Person ändern wollen?

```
setzeVorname :: String -> Person -> Person
setzeVorname n p =
  let alterName = view1 nameLens1 p
      neuerName =
        set1 vornameLens1 n alterName
  in set1 nameLens1 neuerName p
```

Wir können `set` mit `over :: (a -> a) -> s -> s` ersetzen:

```
data Lens2 s a = Lens2
  { view2 :: s -> a
  , over2 :: (a -> a) -> s -> s
  }


set2 :: Lens2 s a -> a -> s -> s
set2 l a s =
  over2 l (const a) s

vornameLens2 :: Lens2 Name String
vornameLens2 = Lens2 vorname (\f n -> n { vorname = f $ vorname n })

nameLens2 :: Lens2 Person Name
nameLens2 = Lens2 name (\f p -> p { name = f $ name p })

setzeVorname2 :: String -> Person -> Person
setzeVorname2 n p =
  over2 nameLens2 (set2 vornameLens2 n) p
```

Problem:

was wenn wir einen Seiteneffekt brauchen um den neuen Wert zu bestimmen?

```
data Lens3 s a = Lens3
  { view3 :: s -> a
  , over3 :: (a -> a) -> s -> s
  , overIO3 :: (a -> IO a) -> s -> IO s
  }
```

van Laarhoven Lens

die 3 Funktionen in `Lens3` lassen sich tatsächlich alle über eine einzige Funktion abbilden:

```
type VLLens s a =
  forall f . Functor f => (a -> f a) -> s -> f s


-- we can get over with the Identity Functor
overVL :: VLLens s a -> (a -> a) -> s -> s
overVL l f s = runIdentity $ l (Identity . f) s


-- and view with the Const Functor!
viewVL :: VLLens s a -> s -> a
viewVL l s = getConst $ l Const s


setVL :: VLLens s a -> a -> s -> s
setVL l a = overVL l (const a)


vornameLensVL :: VLLens Name String
vornameLensVL f n =
  fmap (\v -> n { vorname = v }) (f $ vorname n)
```

```haskell
nameLensVL :: VLLens Person Name
nameLensVL f p =
  fmap (\n -> p { name = n }) (f $ name p)


-- Komposition direkt mit `.` (umgekehrt!)
personVorname :: VLLens Person String
personVorname = nameLensVL . vornameLensVL


setVorname :: String -> Person -> Person
setVorname = setVL personVorname

getVorname :: Person -> String
getVorname = viewVL personVorname
```

lens - Library

Um *lenses* zu Definieren kann man `lens` benutzen:

```haskell
import Control.Lens

data Name = Name
  { vorname :: String
  , nachname :: String
  } deriving Show


vornameL :: Lens' Name String
vornameL = lens vorname (\n v -> n { vorname = v })
```

oder Template-Haskell benutzen:

```haskell
{-# LANGUAGE TemplateHaskell #-}

module Lenses where

import Control.Lens

-- ein paar Records

data Name = Name
  { _vorname :: String
  , _nachname :: String
  } deriving Show


makeLenses ''Name


beispielName :: Name
beispielName = Name "Max" "Muster"


data Person = Person
  { _name :: Name
  , _email :: String
  } deriving Show
```

```
makeLenses ''Person


maxMuster :: Person
maxMuster = Person (Name "Max" "Muster") "max.muster@mail.me"
```

Funktionen

`view`, `over` und `set` arbeiten wie bisher, haben aber etwas *einschüchternde* Signaturen (siehe Docs)

Außerdem ist `view` *magisch* die Reihenfolge egal!

Hintergrund ist, dass die nicht nur auf Lenses sondern auf mehr Typen der `lens` Lib. funktionieren.

Operatoren

siehe Cheat Sheet

- `^.` = `view`- Beispiel: `name . vorname ^. maxMuster`
- `.~` = `set` - Beispiel: `maxMuster & name . vorname .~ "Karl"`
- `%~` = `over` - Beispiel: `maxMuster & name . vorname %~ fmap toUpper`
- Operatoren mit `=` funktionieren im State-Monad:

```
> import Control.Monad.State
> flip runState (1,"Hallo") $ _2 %= fmap toUpper
((),(1,"HALLO"))
```

arbeiten mit Maps

siehe Doc-Modul

```
> import Data.Map.Strict
> fromList [(1,"Hallo")] ^. at 1
Just "Hallo"

> fromList [(1,"Hallo")] ^. at 2
Nothing

> fromList [(1,"Hallo")] & at 2 .~ Just "World"
fromList [(1,"Hallo"),(2,"World")]

>fromList [(1,"Hallo")] & at 1 .~ Nothing
fromList []

> fromList [(1,"Hallo")] & at 1 ?~ "Hey"
fromList [(1,"Hey")]
```

noch tiefer in den Kaninchenbau

bisher immer `Lens'` - ganz allgemein ist es aber

```
type Lens s t a b = forall f . Functor f => (a -> f b) -> s -> f t
```

die Operatoren sind aber allgemeiner:

```
> ("Hey","Du")
("Hey","Du") :: (String, String)

> over _1 length it
(3,"Du") :: (Int, "Du")
```

der Zieltyp hat sich hier also geändert!

Bisher:

```
type Lens' s a = Lens s s a a
```