Intermediate Functional Programming in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

Wozu brauchen wir lens überhaupt?

Die Idee dahinter ist, dass man Zugriffsabstraktionen über Daten verknüpfen kann. Als einfachen Datenstruktur kann man einen Record mit der entsprechenden Syntax nehmen.

Nehmen wir folgende Datenstruktur an:

```
data Person = P { name :: String
                , addr :: Address
                , salary :: Int }
data Address = A { road :: String
                 , city :: String
                 , postcode :: String }
-- autogeneriert unten anderem: addr :: Person -> Address

setName :: String -> Person -> Person
setName n p = p { name = n } --record update notation

setPostcode :: String -> Person -> Person
setPostcode pc p = p { addr = addr p { postcode = pc } }
-- update of a record inside a record
```

Probleme mit diesem Code:

- für 1-Dimensionale Felder ist die record-syntax ok.
- tiefere Ebenen nur umständlich zu erreichen
- eigentlich wollen wir nur pc in p setzen, müssen aber über addr etc. gehen.
- wir brauchen wissen über die “Zwischenstrukturen”, an denen wir nicht interessiert sind

Was wir gerne hätten:

```
data Person = P { name :: String
                , addr :: Address
                , salary :: Int }
-- a lens for each field
lname   :: Lens' Person String
laddr   :: Lens' Person Adress
lsalary :: Lens' Person Int
-- getter/setter for them
view    :: Lens' s a -> s -> a
set     :: Lens' s a -> a -> s -> s
-- lens-composition
composeL :: Lens' s1 s2 -> Lens s2 a -> Lens' s1 a
```

Mit diesen Dingen (wenn wir sie hätten) könnte man dann

```
data Person = P { name :: String
                , addr :: Address
                , salary :: Int }
data Address = A { road :: String
                 , city :: String
                 , postcode :: String }
setPostcode :: String -> Person -> Person
setPostcode pc p
    = set (laddr `composeL` lpostcode) pc p
```

machen und wäre fertig.

```haskell
data LensR s a = L { viewR :: s -> a
                   , setR  :: a -> s -> s }

composeL (L v1 u1) (L v2 u2)
  = L (\s -> v2 (v1 s))
      (\a s -> u1 (u2 a (v1 s)) s)
```

Wieso ist das schlecht?

Wieso ist das schlecht?

Dies ist extrem ineffizient:
Auslesen traversiert die Datenstruktur, dann wird die Funktion angewendet und zum setzen wird die Datenstruktur erneut traversiert:

```
over :: LensR s a -> (a -> a) -> s -> s
over ln f s = setR l (f (viewR l s)) s
```

Wieso ist das schlecht?

Dies ist extrem ineffizient:
Auslesen traversiert die Datenstruktur, dann wird die Funktion angewendet und zum setzen wird die Datenstruktur erneut traversiert:

```
over :: LensR s a -> (a -> a) -> s -> s
over ln f s = setR l (f (viewR l s)) s
```

Lösung: modify-funktion hinzufügen

```haskell
data LensR s a
   = L { viewR :: s -> a
       , setR  :: a -> s -> s
       , mod   :: (a->a) -> s -> s
       , modM  :: (a->Maybe a) -> s -> Maybe s
       , modIO :: (a->IO a) -> s -> IO s }
```

```haskell
data LensR s a
   = L { viewR :: s -> a
       , setR  :: a -> s -> s
       , mod   :: (a->a) -> s -> s
       , modM  :: (a->Maybe a) -> s -> Maybe s
       , modIO :: (a->IO a) -> s -> IO s }
```

Neues Problem: Für jeden Spezialfall muss die Lens erweitert werden.

Man kann alle Monaden abstrahieren. Functor reicht schon:

```
data LensR s a
   = L { viewR :: s -> a
       , setR  :: a -> s -> s
       , mod   :: (a->a) -> s -> s
       , modF  :: Functor f => (a->f a) -> s -> f s }
```

Man kann alle Monaden abstrahieren. Functor reicht schon:

```
data LensR s a
   = L { viewR :: s -> a
       , setR  :: a -> s -> s
       , mod   :: (a->a) -> s -> s
       , modF  :: Functor f => (a->f a) -> s -> f s }
```

Idee: Die 3 darüberliegenden durch modF ausdrücken.

Wenn man das berücksichtigt, dann hat einen Lens folgenden Typ:

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

Wenn man das berücksichtigt, dann hat einen Lens folgenden Typ:

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

Allerdings haben wir dann noch unseren getter/setter:

```
data LensR s a = L { viewR :: s -> a
                   , setR :: a -> s -> s }
```

Wenn man das berücksichtigt, dann hat einen Lens folgenden Typ:

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

Allerdings haben wir dann noch unseren getter/setter:

```
data LensR s a = L { viewR :: s -> a
                   , setR :: a -> s -> s }
```

Stellt sich raus: Die sind isomorph! Auch wenn die von den Typen her komplett anders aussehen.

```haskell
set :: Lens' s a -> (a -> s -> s)
set ln a s = --...umm...
--:t ln => (a -> f a) -> s -> f s
--              => get s out of f s to return it
```

Wir können für f einfach die “Identity”-Monade nehmen, die wir nachher wegcasten können.

Zur Erinnerung kurz nochmal die Definition:

```haskell
newtype Identity a = Identity a
-- Id :: a -> Identity a

runIdentity :: Identity s -> s
runIdentity   (Identity x) = x

instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
```

somit ist set einfach nur

```haskell
set :: Lens' s a -> (a -> s -> s)
set ln x s
   = runIdentity (ln set_fld s)
   where
     set_fld :: a -> Identity a
     set_fld _ = Identity x
     -- a was the OLD value.
     -- We throw that away and set the new value
```

somit ist set einfach nur

```haskell
set :: Lens' s a -> (a -> s -> s)
set ln x s
   = runIdentity (ln set_fld s)
   where
     set_fld :: a -> Identity a
     set_fld _ = Identity x
     -- a was the OLD value.
     -- We throw that away and set the new value
```

oder kürzer (für nerds wie den Autor der Lens-Lib)

```haskell
set :: Lens' s a -> (a -> s -> s)
set ln x = runIdentity . ln (Identity . const x)
```

Wie nutzen wir das nun als modify?

Dasselbe wie set, nur dass wir den Parameter nicht entsorgen, sondern in die mitgelieferte Funktion stopfen.

```
over :: Lens' s a -> (a -> a) -> s -> s
over ln f = runIdentity . ln (Identity . f)
```

.. und als getter?

```
view :: Lens' s a -> (s -> a)
view ln s = --...umm...
--:t ln => (a -> f a) -> s -> f s
--              => get a out of the (f s) return-value
--              Wait, WHAT?
```

.. und als getter?

```
view :: Lens' s a -> (s -> a)
view ln s = --...umm...
--:t ln => (a -> f a) -> s -> f s
--              => get a out of the (f s) return-value
--              Wait, WHAT?
```

Auch hier gibt es einen netten Funktor. Wir packen das a einfach in das f und werfen das s am Ende weg.

.. und als getter?

```
view :: Lens' s a -> (s -> a)
view ln s = --...umm...
--:t ln => (a -> f a) -> s -> f s
--              => get a out of the (f s) return-value
--              Wait, WHAT?
```

Auch hier gibt es einen netten Funktor. Wir packen das a einfach in das f und werfen das s am Ende weg.

```
newtype Const v a = Const v

getConst :: Const v a -> v
getConst (Const x) = x

instance Functor (Const v) where
    fmap f (Const x) = Const x
    -- throw f away. Nothing changes our const!
```

somit ergibt sich

```haskell
view :: Lens' s a -> (s -> a)
view ln s
  = getConst (ln Const s)
               -- Const :: s -> Const a s
```

somit ergibt sich

```haskell
view :: Lens' s a -> (s -> a)
view ln s
  = getConst (ln Const s)
               -- Const :: s -> Const a s
```

oder nerdig

```haskell
view :: Lens' s a -> (s -> a)
view ln = getConst . ln Const
```

Nochmal kurz der Typ:

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

Nochmal kurz der Typ:

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

Für unser Personen-Beispiel vom Anfang:

```
data Person = P { _name :: String, _salary :: Int }

name :: Lens' Person String
-- name :: Functor f => (String -> f String)
--                   -> Person -> f Person
name elt_fn (P n s)
  = fmap (\n' -> P n' s) (elt_fn n)
-- fmap :: Functor f => (a->b) -> f a -> f b
-- - der Funktor, der alles verknpft
-- \n' -> .. :: String -> Person
-- - Funktion um das Element zu lokalisieren (WO wird ersetzt/gelesen/...)
-- elt_fn n  :: f String
-- - Funktion um das Element zu veraendern (setzen, aendern, ...)
```

Die Lambda-Funktion ersetzt einfach den Namen.

Nochmal kurz der Typ:

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

Für unser Personen-Beispiel vom Anfang:

```
data Person = P { _name :: String, _salary :: Int }

name :: Lens' Person String
-- name :: Functor f => (String -> f String)
--                   -> Person -> f Person
name elt_fn (P n s)
  = fmap (\n' -> P n' s) (elt_fn n)
-- fmap :: Functor f => (a->b) -> f a -> f b
-- - der Funktor, der alles verknpft
-- \n' -> .. :: String -> Person
-- - Funktion um das Element zu lokalisieren (WO wird ersetzt/gelesen/...)
-- elt_fn n  :: f String
-- - Funktion um das Element zu veraendern (setzen, aendern, ...)
```

Die Lambda-Funktion ersetzt einfach den Namen.

Häufig sieht man auch

```
name elt_fn (P n s)
  = (\n' -> P n' s) <$> (elt_fn n)
```

Ist das nicht alles ein ziemlicher Overhead?

Ist das nicht alles ein ziemlicher Overhead?

```
view name (P {_name="Fred", _salary=100})
    -- inline view-function
= getConst (name Const (P {_name="Fred", _salary=100})
    -- inline name
= getConst (fmap (\n' -> P n' 100) (Const "Fred"))
    -- fmap f (Const x) = Const x - Definition von Const
= getConst (Const "Fred")
    -- getConst (Const x) = x
= "Fred"
```

Ist das nicht alles ein ziemlicher Overhead?

```
view name (P {_name="Fred", _salary=100})
    -- inline view-function
= getConst (name Const (P {_name="Fred", _salary=100})
    -- inline name
= getConst (fmap (\n' -> P n' 100) (Const "Fred"))
    -- fmap f (Const x) = Const x - Definition von Const
= getConst (Const "Fred")
    -- getConst (Const x) = x
= "Fred"
```

Dieser Aufruf hat KEINE Runtime-Kosten, weil der Compiler direkt die Adresse des Feldes einsetzen kann. Der gesamte Boilerplate-Code wird vom Compiler wegoptimiert.
Dies gilt für jeden Funktor mit newtype, da das nur ein Typalias ist.

Wie verknüpfen wir lenses nun?

Wie verknüpfen wir lenses nun?

Schauen wir uns die Typen an:
Wir wollen ein

```
Lens' s1 s2 -> Lens' s2 a -> Lens' s1 a
```

Wie verknüpfen wir lenses nun?

Schauen wir uns die Typen an:
Wir wollen ein

```
Lens' s1 s2 -> Lens' s2 a -> Lens' s1 a
```

Wir haben 2 Lenses

```
ln1 :: (s2 -> f s2) -> (s1 -> f s1)
ln2 :: (a -> f a) -> (s2 -> f s2)
```

Wie verknüpfen wir lenses nun?

Schauen wir uns die Typen an:
Wir wollen ein

```
Lens' s1 s2 -> Lens' s2 a -> Lens' s1 a
```

Wir haben 2 Lenses

```
ln1 :: (s2 -> f s2) -> (s1 -> f s1)
ln2 :: (a -> f a) -> (s2 -> f s2)
```

wenn man scharf hinsieht, kann man die verbinden

```
ln1 . ln2 :: (a -> f s) -> (s1 -> f s1)
```

Wie verknüpfen wir lenses nun?

Schauen wir uns die Typen an:
Wir wollen ein

```
Lens' s1 s2 -> Lens' s2 a -> Lens' s1 a
```

Wir haben 2 Lenses

```
ln1 :: (s2 -> f s2) -> (s1 -> f s1)
ln2 :: (a -> f a) -> (s2 -> f s2)
```

wenn man scharf hinsieht, kann man die verbinden

```
ln1 . ln2 :: (a -> f s) -> (s1 -> f s1)
```

und erhält eine Lens. Sogar die Gewünschte!

Somit ist Lens-Composition einfach nur Function-Composition (.).

Der Code um die Lenses zu bauen ist für records immer Identisch:

```
data Person = P { _name :: String, _salary :: Int }

name :: Lens' Person String
name elt_fn (P n s) = (\n' -> P n' s) <$> (elt_fn n)
```

Der Code um die Lenses zu bauen ist für records immer Identisch:

```
data Person = P { _name :: String, _salary :: Int }

name :: Lens' Person String
name elt_fn (P n s) = (\n' -> P n' s) <$> (elt_fn n)
```

Daher kann man einfach

```
import Control.Lens.TH
data Person = P { _name :: String, _salary :: Int }

$(makeLenses ''Person)
```

nehmen, was einem eine Lens für "name" und eine Lens für "salary" generiert.
Mit anderen Templates kann man auch weitere Dinge steuern (etwa wofür Lenses generiert werden, welches Prefix (statt _) man haben will etc. pp.).
Will man das aber haben, muss man selbst in den Control.Lens.TH-Code schauen.

Noch ein Beispiel:

```
import Control.Lens.TH

data Person = P { _name :: String
                , _addr :: Address
                , _salary :: Int }
data Address = A { _road :: String
                 , _city :: String
                 , _postcode :: String }

$(makeLenses ''Person)
$(makeLenses ''Address)

setPostcode :: String -> Person -> Person
setPostcode pc p = set (addr . postcode) pc p
```

Für alle gängigen Funktionen gibt es auch infix-Operatoren:

```
setPostcode :: String -> Person -> Person
setPostcode pc p = addr . postcode .~ pc     $ p
--                      |   Focus     |set|to what|in where

getPostcode :: Person -> String
getPostcode p = p    ^. $ addr . postcode
--              |from|get|    Focus       |
```

Es gibt momentan viele weitere Infix-Operatoren (für Folds, Listenkonvertierungen, -traversierungen, . . .). Aktueller Stand: 80+

Man kann mit Lenses sogar Felder emulieren, die gar nicht da sind.

Man kann mit Lenses sogar Felder emulieren, die gar nicht da sind.
Angenommen folgender Code:

```
data Temp = T { _fahrenheit :: Float }

$(makeLenses ''Temp)
-- liefert Lens: fahrenheit :: Lens Temp Float

centigrade :: Lens Temp Float
centigrade centi_fn (T faren)
  = (\centi' -> T (cToF centi'))
    <$> (centi_fn (fToC faren))
-- cToF & fToC as Converter-Functions defined someplace else
```

Man kann mit Lenses sogar Felder emulieren, die gar nicht da sind.
Angenommen folgender Code:

```
data Temp = T { _fahrenheit :: Float }

$(makeLenses ''Temp)
-- liefert Lens: fahrenheit :: Lens Temp Float

centigrade :: Lens Temp Float
centigrade centi_fn (T faren)
  = (\centi' -> T (cToF centi'))
    <$> (centi_fn (fToC faren))
-- cToF & fToC as Converter-Functions defined someplace else
```

Hiermit kann man dann auch Funktionen, die auf Grad-Celsius rechnen auf Daten anwenden, die eigenlich nur Fahrenheit speichern, aber eine Umrechnung bereitstellen.
Analog kann man auch einen Zeit-Datentypen definieren, der intern mit Sekunden rechnet (und somit garantiert frei von Fehlern wie -3 Minuten oder 37 Stunden ist)

Das ganze kann man auch parametrisieren und auf Non-Record-Strukturen anwenden. Beispielhaft an einer Map verdeutlicht:

```
-- from Data.Lens.At
at :: Ord k => k -> Lens' (Map k v) (Maybe v)

-- oder identisch, wenn man die Lens' aufloest:
at :: Ord k, forall f. Functor f =>
      k -> (Maybe v -> f Maybe v) -> Map k v -> f Map k v

at k mb_fn m
  = wrap <$> (mb_fn mv)
  where
    mv = Map.lookup k m

    wrap :: Maybe v -> Map k v
    wrap (Just v') = Map.insert k v' m
    wrap Nothing   = case mv of
                       Nothing -> m
                       Just _  -> Map.delete k m

-- mb_fn :: Maybe v -> f Maybe v
```

- Bitfields auf Strukturen die Bits haben (Ints, . . .) in Data.Bits.Lens

- Bitfields auf Strukturen die Bits haben (Ints, . . .) in Data.Bits.Lens
- Web-scraper in Package hexpat-lens

```
p ^.. _HTML' . to allNodes
             . traverse . named "a"
             . traverse . ix "href"
             . filtered isLocal
             . to trimSpaces
```

Zieht alle externen Links aus dem gegebenen HTML-Code in p um weitere ziele fürs crawlen zu finden.

Bisher hatten wir Lenses nur auf Funktoren F. Die nächstmächtigere Klasse ist Applicative.

Bisher hatten wir Lenses nur auf Funktoren F. Die nächstmächtigere Klasse ist Applicative.

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

Bisher hatten wir Lenses nur auf Funktoren F. Die nächstmächtigere Klasse ist Applicative.

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

Da wir den Container identisch lassen (weder s noch a wurde angefasst) muss sich etwas anderes ändern. Statt eines einzelnen Focus erhalten wir viele Foci.

Recap: Was macht eine Lens:

```
data Adress = A { _road :: String
                , _city :: String
                , _postcode :: String }

road :: Lens' Adress String
road elt_fn (A r c p) = (\r' -> A r' c p) <$> (elt_fn r)
--                       |    "Hole"     |      | Thing to put in|
```

Recap: Was macht eine Lens:

```
data Adress = A { _road :: String
                , _city :: String
                , _postcode :: String }

road :: Lens' Adress String
road elt_fn (A r c p) = (\r' -> A r' c p) <$> (elt_fn r)
--                        |    "Hole"     |      | Thing to put in|
```

Wenn man nun road & city gleichzeitig bearbeiten will:

```
addr_strs :: Traversal' Address String
addr_strs elt_fn (A r c p)
  = ... (\r' c' -> A r' c' p)    .. (elt_fn r) .. (elt_fn c) ..
--      | function with 2 "Holes"|  first Thing |  second Thing
```

fmap kann nur 1 Loch stopfen, aber nicht mit n Löchern umgehen.
Applicative mit <*> kann das.
Somit gibt sich

fmap kann nur 1 Loch stopfen, aber nicht mit n Löchern umgehen.

Applicative mit <*> kann das.

Somit gibt sich

```
addr_strs :: Traversal' Address String
addr_strs elt_fn (A r c p)
  = pure           (\r' c' -> A r' c' p)  <*> (elt_fn r) <*> (elt_fn c)
--  lift in Appl. | function with 2 "Holes"|  first Thing |  second Thing
-- oder krzer
addr_strs :: Traversal' Address String
addr_strs elt_fn (A r c p)
  = (\r' c' -> A r' c' p)  <$> (elt_fn r) <*> (elt_fn c)
-- pure x <*> y == x <$> y
```

Wie würd eine modify-funktion aussehen?

Wie würd eine modify-funktion aussehen?

```
over :: Lens' s a -> (a -> a) -> s -> s
over ln f = runIdentity . ln (Identity . f)
```

Wie würd eine modify-funktion aussehen?

```
over :: Lens' s a -> (a -> a) -> s -> s
over ln f = runIdentity . ln (Identity . f)

over :: Traversal' s a -> (a -> a) -> s -> s
over ln f = runIdentity . ln (Identity . f)
```

Wie würd eine modify-funktion aussehen?

```
over :: Lens' s a -> (a -> a) -> s -> s
over ln f = runIdentity . ln (Identity . f)

over :: Traversal' s a -> (a -> a) -> s -> s
over ln f = runIdentity . ln (Identity . f)
```

Der Code ist derselbe - nur der Typ ist generischer. Auch die anderen Dinge funktioniert diese Erweiterung (für Identity und Const muss man noch ein paar dummy-Instanzen schreiben um sie von Functor auf Applicative oder Monad zu heben - konkret reicht hier die Instanzierung von Monoid). In der Lens-Library ist daher meist Monad m statt Functor f gefordert.

Man kann mit Foci sehr selektiv vorgehen. Auch kann man diese durch Funktionen steuern. Beispisweise eine Funktion anwenden auf

- Jedes 2. Listenelement
- Alle graden Elemente in einem Baum
- Alle Namen in einer Tabelle, deren Gehalt $>$ 10.000 EUR ist

Man kann mit Foci sehr selektiv vorgehen. Auch kann man diese durch Funktionen steuern. Beispisweise eine Funktion anwenden auf

- Jedes 2. Listenelement
- Alle graden Elemente in einem Baum
- Alle Namen in einer Tabelle, deren Gehalt $>$ 10.000 EUR ist

Traversals und Lenses kann man trivial kombinieren (lens . lens $\Rightarrow$ lens, lens . traversal $\Rightarrow$ traversal etc.)

In dieser Vorlesung wurde nur auf Monomorphic Lenses eingegangen. In der richtigen Library ist eine Lens

```
type Lens' s a = Lens s s a a
type Lens s t a b = forall f. Functor f => (a -> f b) -> (s -> f t)
```

In dieser Vorlesung wurde nur auf Monomorphic Lenses eingegangen. In der richtigen Library ist eine Lens

```
type Lens' s a = Lens s s a a
type Lens s t a b = forall f. Functor f => (a -> f b) -> (s -> f t)
```

sodass sich auch die Typen ändern können um z.B. automatisch einen Konvertierten (sicheren) Typen aus einer unsicheren Datenstruktur zu geben.

In dieser Vorlesung wurde nur auf Monomorphic Lenses eingegangen. In der richtigen Library ist eine Lens

```
type Lens' s a = Lens s s a a
type Lens s t a b = forall f. Functor f => (a -> f b) -> (s -> f t)
```

sodass sich auch die Typen ändern können um z.B. automatisch einen Konvertierten (sicheren) Typen aus einer unsicheren Datenstruktur zu geben.
Die modify-Funktion over ist auch

```
over :: Profunctor p => Setting p s t a b -> p a b -> s -> t
```

Edward is deeply in thrall to abstractionitis - Simon Peyton Jones

Edward is deeply in thrall to abstractionitis - Simon Peyton Jones

Lens alleine definiert 39 newtypes, 34 data-types und 194 Typsynonyme...

Edward is deeply in thrall to abstractionitis - Simon Peyton Jones

Lens alleine definiert 39 newtypes, 34 data-types und 194 Typsynonyme...

Ausschnitt

```
-- traverseOf :: Functor f => Iso s t a b -> (a -> f b) -> s -> f t
-- traverseOf :: Functor f => Lens s t a b -> (a -> f b) -> s -> f t
-- traverseOf :: Applicative f => Traversal s t a b -> (a -> f b) -> s -> f t

traverseOf :: Over p f s t a b -> p a (f b) -> s -> f t
```

Edward is deeply in thrall to abstractionitis - Simon Peyton Jones

Lens alleine definiert 39 newtypes, 34 data-types und 194 Typsynonyme...

Ausschnitt

```
-- traverseOf :: Functor f => Iso s t a b -> (a -> f b) -> s -> f t
-- traverseOf :: Functor f => Lens s t a b -> (a -> f b) -> s -> f t
-- traverseOf :: Applicative f => Traversal s t a b -> (a -> f b) -> s -> f t

traverseOf :: Over p f s t a b -> p a (f b) -> s -> f t
```

dafuq?