LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Fortgeschrittene Funktionale Programmierung
Wintersemester 2021/22

Funktionale Optiken

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 21. Januar 2022 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

Motivation für Funktionale Referenzen

- Programmieraufgabe: **Kleine Änderung** an **großer Datenstruktur**
- In rein funktionalen Programmiersprachen:
  - Eine Möglichkeit: Echte Referenzen mit einer Monade verwenden (z.B. `STRef`)
  - Andere Lösungen erfordern Kopieren der Struktur
    Kann durch Sharing der unveränderten Teile recht effizient implementiert werden
- Ziel der funktionalen Referenzen:
  - Eleganter, kurzer Code für solche Änderungen
  - Weniger im Fokus ist die Effizienz.

Motivation für Funktionale Referenzen

- Funktionale Referenz: Referenz auf das zu ändernde Element
- Anderer Name: Linse
- Verallgemeinerung oder Einschränkung von Linsen:

  andere Optiken, wie Prismen, Traversal, Isos, . . .
- Linsen für abstrakte Datentypen machen auch z.B. in imperativen Sprachen Sinn,
  es gibt z.B. Linsen-Bibliotheken für Javascript.

Problembeispiel

Datentypen und Daten:

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
data Faction = Faction { faction :: String, members :: [Person]}

cersei = Person "Cersei" $ Wealth 222 22

tyrion = Person "Tyrion" tw1

tw1    = Wealth 100 1

lannisters = Faction "Lannisters" [cersei,tyrion]
```

Problembeispiel (2)

Problemstellung: Tyrion gibt 7 Goldstücke aus

Lösungsmöglichkeit mit Recordsyntax

```
payGold  :: Int -> Person -> Person
payGold  m p = let w  = wealth p
                   w' = w { gold=(gold w) - m}
               in  p { wealth=w' }
```

Ausführung:

```
*> payGold  7 tyrion
Tyrion(93g,1d)
```


Problembeispiel (3)

Variante ohne Record-Syntax:

```
payGold' :: Int -> Person -> Person
payGold' m (Person n (Wealth g d)) =
  let w = (Wealth (g-m) d)
  in Person n w
```

Ausführung:

```
*> payGold  7 tyrion
Tyrion(93g,1d)
```


Problembeispiel (4)

- Nachteil: Komplettes Auspacken und wieder Einpacken
- Funktioniert, aber sehr umständlich mit Record-Update-Syntax!
- Ohne Record-Syntax sieht es einfach aus, bei vielen Feldern aber sehr umständlich!

Problembeispiel: Vorschau So geht es mit Linsen

```
*> over (wealth.gold) (subtract 7) tyrion
Tyrion(93g,1d)

*> over (members.(ix 1).wealth.gold) (subtract 7) lannisters
Lannisters[Cersei(222g,22d),Tyrion(93g,1d)]

*> over (members.traverse.wealth.gold) (subtract 7) lannisters
Lannisters[Cersei(215g,22d),Tyrion(93g,1d)]
```

lens Package

- Es gab und gibt verschiedene Versuche, Linsen zu implementieren.
- Durchbruch gelang Edward Kmett MIRI Berkeley (US) 2012 mit dem Package `lens`.
- `https://hackage.haskell.org/package/lens`
- Video-Empfehlung: `http://youtu.be/cefnmjtAolY`
  (Edward Kmett erläutert seine Linsen)
- Buch zu Linsen: Chris Penner: Optics by Example, 2019-2020,
  `https://leanpub.com/optics-by-example`

lens Package (2)

Damit der gezeigte Code funktioniert, ist folgendes notwendig:

- `import Control.Lens`
- Feldern ein Underscore voranstellen – Konvention, kein muss
- Linsen automatisch erzeugen mit Template Haskell:

```
makeLenses ''Wealth
makeLenses ''Person
makeLenses ''Faction
```

  Dadurch werden für die drei Typen `Wealth`, `Person` und `Faction` viele verschiedene Linsen erstellt.
  ansehen mit `-ddump-splices` oder `-dth-dec-file`

Beispiel: komplett

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens
data Wealth  = Wealth { _gold:: Int, _diamonds :: Int }
data Person  = Person { _name :: String, _wealth :: Wealth }
data Faction = Faction{ _faction :: String, _members :: [Person]}

makeLenses ''Wealth
makeLenses ''Person
makeLenses ''Faction
```


Beispiel: komplett

```
$> stack exec -- ghci -ddump-splices L1
GHCi, version 8.8.3: https://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( L1.hs, interpreted )
L1.hs:7:1-19: Splicing declarations
    makeLenses ''Wealth
  ======>
    diamonds :: Lens' Wealth Int
    diamonds f_a7a4 (Wealth x1_a7a5 x2_a7a6)
      = (fmap (\ y1_a7a7 -> (Wealth x1_a7a5) y1_a7a7)) (f_a7a4 x2_a7a6)
    {-# INLINE diamonds #-}
    gold :: Lens' Wealth Int
    gold f_a7a8 (Wealth x1_a7a9 x2_a7aa)
      = (fmap (\ y1_a7ab -> (Wealth y1_a7ab) x2_a7aa)) (f_a7a8 x1_a7a9)
    {-# INLINE gold #-}
...
```

Hintergrund: lens-Package

- Die Idee von Linsen als Put/Get-Paar, oder auch View/Update, stammt aus der Datenbankforschung Ende 70er/Anfang 80er.
- 2005 taucht der Begriff „Linse“ in einer Arbeit von Benjamin Pierce auf
- 2009 beschrieb Twan van Laarhoven, in seinem Blog einen Typ, der jetzt `Lens'` genannt wird.
- 2012 erkannte Russell O'Connor die Verbindung zwischen seinen Multi-Linsen und den van Laarhoven-Linsen, was zu Traversals führte.
- 2012 formulierte Edward Kmett die benötigten Gesetze und schrieb die inzwischen populärste Linsen Bibliothek für Haskell
- Es gibt viel andere Bibliotheken mit anderen Ansätzen!

Linsen-Implementierung

Grundidee, Linsen als Paare, van Laarhoven-Linsen

Grundidee

Eine Linse ist primär ein Set/Get-Paar bzw.View/Update-Paar.

In der `lens`-Bibliothek: `view` und `set`

```
*> view wealth tyrion
(100g,1d)

*> set wealth (Wealth 0 0) tyrion
Tyrion(0g,0d)
```


Grundidee (2)

Hier wird auch gerne eine Schreibweise genommen,
bei der das Funktionsargument vorne steht:

```
*> tyrion & view wealth
(100g,1d)
*> tyrion & set wealth (Wealth 0 0)
Tyrion(0g,0d)
```

Bemerkung: `(&)` ist `flip ($)`. Im Modul `Modul Data.Function`:

```
(&) :: a -> (a -> b) -> b
x & f = f x
```

Linsen als Paare

Modellierung von Linsen durch getter/setter-Paare ist prinzipiell möglich, z.B.

```
data LensPair s a = LensPair { view :: s -> a
                             , set  :: a -> s -> s }

over :: LensPair s a -> (a -> a) -> s -> s
over lens f s = set lens (f $ view lens s) s

lensPersonName :: LensPair Person String
lensPersonName   = LensPair name   (\x s-> s{name=x})

lensPersonWealth :: LensPair Person Wealth
lensPersonWealth = LensPair wealth (\x s-> s{wealth=x})
```

Aufrufe dazu:

```
*> view lensPersonWealth tyrion
(100g,1d)
*> set lensPersonWealth (Wealth 9 1) tyrion
Tyrion(9g,1d)
*> over lensPersonName (map toUpper) tyrion
```


Linsen als Paare (2)

- Obwohl grundsätzlich möglich, wird diese Modellierung **nicht** verwendet.
- Ein Problem: Bei dieser Modellierung lassen sich Linsen nicht einfach komponieren
- Z.B. Fokussiere Gold einer Person
- Lösung: van Laarhoven-Linsen (gleich)

Einschub/Wiederholung: Rank-N types

Spracherweiterung `RankNTypes` erlaubt Typen, bei denen ∀-Quantoren für Typvariablen mitten im Typ auftauchen.

```
--  foo :: (forall a . a) -> b
  foo :: (forall a . a -> a) -> (b -> b)
  foo x = x x

*> (foo (foo id)) 42
42
*> foo id 42
42
*> foo toUpper 42
error ...
-- (toUpper :: Char -> Char) zu speziell!
```

- Beide Typen sind möglich, keine kann automatisch vom GHC inferiert werden
- Funktion hat keinen allgemeinsten Typ und die Typisierung ist unentscheidbar
- Typsignaturen werden notwendig.

Einschub/Wiederholung: Rank-N types (2)

- ∀-Quantor auf der **rechten** Seite eines Pfeiles **kann hochgeschoben** werden

  `forall a. a-> (forall b. b -> a)`
  ist äquivalent zu
  `forall a b. a -> b -> a`.
- Beides sind Rank-1 Typen, deren Bedeutung identisch ist zu dem bisherigen Typ `a -> b -> a`

Einschub/Wiederholung: Rank-N types (3)

- ∀-Quantor auf der **linken Seite** eines Pfeiles kann **nicht verschoben** werden und erhöht den Rank des Typen:

```
forall a. (forall b.b -> b) -> a -> a
```

Bedeutung:

- Hier wird eine Funktion als Argument gefordert, welche mit jeden beliebigen Typen umgehen kann! (z.B. `id`)
- Bei `forall a b. (b -> b) -> a -> a` wird als Argument nur eine Funktion verlangt, welche irgendein spezielles `b` verarbeiten kann. Für jedes `b` darf eine andere Funktion übergeben werden.

Beispiele Rank-N Types

Beispiele, wobei die Nummer im Namen dem Rang entspricht:

```
f1 :: forall a b. a -> b -> a
g1 :: forall a b. (Ord a, Eq  b) => a -> b -> a

f2 :: (forall a. a -> a) -> Int -> Int
g2 :: (forall a. Eq a => [a] -> a -> Bool) -> Int -> Int

f3 :: ((forall a. a -> a) -> Int) -> Bool -> Bool
```


Van Laarhoven-Linsen

Twan van Laarhoven schlug 2009 folgende Definition vor:

```
{-# LANGUAGE Rank2Types #-}

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

Lens' sind Funktionen, die für alle Funktoren `f` funktionieren müssen!

Der Typ `Lens'` selbst ist ein Rank-1-Typ, aber er wird innerhalb der Operationen (view, set, over, ...) links unter dem Funktionspfeil verwendet (nächste Folie)

Van Laarhoven-Linsen: view, set, over

```
view :: Lens' s a -> s -> a
view l = getConst . l Const

set :: Lens' s a -> a -> s -> s
set l a = over l (const a)

over :: Lens' s a -> (a -> a) -> s -> s
over l m = runIdentity . l (Identity . m)
```

Beachte: `view`, `set`, `over` haben Rank-2-Typen!
Um die Implementierungen zu verstehen: Wiederholung `Identity` und `Const`

Die leere Identity-Monade

```
newtype Identity a = Identity { runIdentity :: a }
instance Functor Identity where
    fmap :: (a -> b) -> Identity a -> Identity b
    fmap f (Identity x) = Identity $ f x
instance Applicative Identity where
    pure :: a -> Identity a
    pure = Identity
    (<*>) :: Identity (a->b) -> Identity a -> Identity b
    Identity f <*> Identity x = Identity $ f x
instance Monad Identity where
    (>>=) :: Identity a -> (a -> Identity b) -> Identity b
    Identity x >>= mf = mf x
```

- `>>=` entspricht `flip ($)`, vom newtype-Konstruktor abgesehen.
- `(<$>)` entspricht `($)`, vom newtype-Konstruktor abgesehen.

Der „konstante" Const Functor

```
newtype Const a b = Const {getConst :: a}
instance Functor (Const m) where
    fmap :: (a -> b) -> Const m a -> Const m b
    fmap _ (Const x) = Const x
instance Monoid m => Applicative (Const m) where
    pure :: a -> Const m a
    pure = const $ Const mempty
    (<*>) :: Const m (a->b) -> Const m a -> Const m b
    Const x <*> Const y = Const $ x `mappend` y
instance Foldable (Const m) where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap _ _ = mempty
instance Traversable (Const m) where
    traverse :: Applicative f=> (a->f b)-> t a-> f(t b)
    traverse _ (Const m) = pure $ Const m
```


Der „konstante" Const Functor (2)

- `Const a b` ist ein applikativer Funktor,
- Container über `b`, der kein Wert von `b` enthält, aber dafür als Kontext einen Wert des Typs `a` hat.
- `b` ist Phantom-Typ, da Werte von `Const a b` nie `b`-Werte enthalten.
- `Const a b` ist keine Monade, aber Instanz der Klassen `Foldable` und `Traversable`.
- Dank `Const` kann man aus jeder `Traversable`-Instanz eine `Foldable`-Instanz ableiten:

```
foldMapDefault::(Traversable t,Monoid c)=>(a->c)->t a->c

foldMapDefault f = getConst . traverse (Const . f)
```


Bemerkungen zu view für die van Laarhoven-Linsen

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

view :: Lens' s a -> s -> a
view l = getConst . l Const
```

- Der Funktor `Const a` wird für `f` im Typ `Lens'` verwendet
- D.h. `l :: (a -> Const a a) -> (s -> Const a s)`
- Daher liefert `(l Const) :: s -> Const a s`
- Wendet man dies auf eine Struktur `s` an, so erhält man `Const a s`
- Struktur ist dann schon gelöscht, nur Typ existiert noch
- Als Wert steht hier `Const x` mit `x` vom Typ `a`
- D.h. mit `getConst :: Const a b -> a` erhält man dieses `x` vom Typ `a`.
- Insgesamt: `view l` ist Funktion von `s` nach `a`

Bemerkungen zu over für die van Laarhoven-Linsen

Für die Definition

```
over :: Lens' s a -> (a -> a) -> s -> s
over l m = runIdentity . l (Identity . m)
```

- Der Funktor `Identity` wird für `f` im Typ `Lens'` verwendet
- D.h. `l :: (a -> Identity a) -> (s -> Identity s)`
- Für `m :: a -> a`, die den Ausschnitt der Struktur `s` ändern soll, gilt `(Identity . m)::(a -> Identity a)`
- Daher `(l (Identity . m)):: s -> Identity s`
- Anwenden auf Struktur, dann mit `runIdentity` auf Ergebnis vom Typ `s` zugreifen

Bemerkungen zu set für die van Laarhoven-Linsen

Die Funktion

```
set :: Lens' s a -> a -> s -> s
set l a = over l (const a)
```

- verwendet `over` und damit `Identity` für `f` im Typ `Lens'`.
- Für `over` wird eine Funktion benötigt, die den Ausschnitt verändert.
- Hier wird `const a` verwendet, die konstant den Wert `a` zurück liefert.

Definition eigener Linsen

```
data Wealth  = Wealth { gold, diamonds :: Int }
data Person  = Person { name :: String, wealth :: Wealth }
```

Die van-Laarhoven-Linsen für unsere Datentypen sind:

```
-- forall f. Functor f => (Int-> f Int)-> Wealth-> f Wealth
lWgold :: Lens' Wealth Int
lWgold fun (Wealth g d) = (\g' -> Wealth g' d)  <$> (fun g)
lWdiamonds :: Lens' Wealth Int
lWdiamonds fun (Wealth g d) = (\d' -> Wealth g d') <$> (fun d)
-- forall f.Functor f => (String-> f String)-> Person-> f Person
lPname :: Lens' Person String
lPname  fun (Person n w) = (\n' -> Person n' w) <$> fun n
-- forall f.Functor f => (Wealth-> f Wealth)-> Person-> f Person
lPwealth :: Lens' Person Wealth
lPwealth  fun (Person n w) = (\w' -> Person n w') <$> fun w
```


Definition eigener Linsen (2)

Idee in allen Fällen analog:

- Zugriff mit Pattern Matching auf Feld
- Update auf das entsprechende Feld anwenden
- Anschließend unveränderte Struktur herum aufbauen
- dies geschieht mit `fmap` aus dem Funktor, indem die Struktur „auf den geänderten Wert gemappt wird".

Sämtliche Funktionen sind gleich aufgebaut

→ lassen sich daher algorithmisch erzeugen lassen

→ Verwendung von Template Haskell möglich

Definition eigener Linsen (3)

- `lens` stellt Template Haskell-Metafunktionen zur Verfügung, um diese Linsen automatisch zu erzeugen
- Eine Linse für jedes Record-Feld, welches mit Unterstrich beginnt.

```
data Wealth  = Wealth { _gold, _diamonds :: Int }
data Person  = Person { _name :: String, _wealth :: Wealth }
data Faction = Faction{ _faction :: String, _members :: [Person]}
makeLenses ''Wealth
$(makeLenses (mkName "Person"))
makeLenses ''Faction
```


Definition eigener Linsen (4)

- Beispiele:

```
*> :type gold
gold :: Functor f => (Int -> f Int) -> Wealth -> f Wealth
*> :type wealth.gold
wealth.gold :: Functor f => (Int -> f Int) -> Person -> f Person
```

- Dies ist die Standard-Option; eigene Benennungen wählbar mit

```
makeLensesWith :: LensRules -> Name -> DecsQ
makeLensesFor :: [(String, String)] -> Name -> DecsQ
```


Beispiel: view mit Const

```
Const    :: a -> Const a b
getConst :: Const a b -> a
view::(forall f.Functor f => ((a -> f a)->(s -> f s))) -> s -> a
view l = getConst . l Const
```

```
data Person  = Person { name :: String, wealth :: Wealth }
tyrion = Person "Tyrion" tw1    -- für ein tw1::Wealth
```

```
lPname :: Functor f=> (String-> f String)-> Person-> f Person
lPname k (Person n w) = (\n' -> Person n' w) <$> k n
```

Nachrechnen:

```
view lPname tyrion
= (getConst . lPname Const) tyrion
= getConst  (lPname Const (Person "Tyrion" tw1))
= getConst (\n'-> Person n' tw1) <$> Const "Tyrion")
= getConst (Const "Tyrion")
= "Tyrion"
```


Beispiel: over mit Identity

```
Identity    :: a -> Identity a
runIdentity :: Identity a -> a
over::(forall f.Functor f => ((a -> f a) -> (s -> f s))) -> (a -> a) -> s -> s
over l m = runIdentity . l (Identity . m)
```

```
data Person  = Person { name :: String, wealth :: Wealth }
tyrion = Person "Tyrion" tw1    -- für ein tw1::Wealth
```

```
lPname :: Functor f=> (String-> f String)-> Person-> f Person
lPname k (Person n w) = (\n' -> Person n' w) <$> k n
```

Nachrechnen

```
over lPname reverse tyrion
= (runIdentity . lPname (Identity.m)) tyrion
= runIdentity (lPname (Identity . reverse) (Person "Tyrion" tw1))
= runIdentity ((\n'-> Person n' tw1) <$> (Identity $ reverse "Tyrion"))
= runIdentity (Identity $ Person (reverse "Tyrion") tw1)
= Person (reverse "Tyrion") tw1
```

Beispiel: set

```
set :: Lens' s a -> a -> s -> s
set l a = over l (const a)
```

Wir deuten die Berechnung von `set lPname "TYRION" (Person "Tyrion" tw1)` an:

```
set lPname "TYRION" (Person "Tyrion" tw1)
= over lPname (const "TYRION") (Person "Tyrion" tw1)
= ...
= runIdentity (Identity (Person (const "TYRION" "Tyrion") tw1))
= Person (const "TYRION" "Tyrion") tw1
```


Van Laarhoven-Linsen-Komposition

Komposition von van-Laarhoven-Linsen entspricht einfach umgedrehter Funktionskomposition:

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

compose :: Lens' b c -> Lens' a b -> Lens' a c
compose r s = s . r

(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

Einsetzen des Typsynonyms ergibt einfach:

```
compose :: forall a b c f. Functor f
        => ((c -> f c) -> (b -> f b))
        -> ((b -> f b) -> (a -> f a))
        -> ((c -> f c) -> (a -> f a))
```

Beachte: Statt `compose` wird meist `(.)` verwendet (mit umgedrehter Reihenfolge)

Van Laarhoven-Linsen-Komposition (2)

- Man kann Linsen mit Funktionskomposition zusammensetzen:

```
*> tyrion & view (lPwealth.lWgold)
  100
*> :type (lPwealth.lWgold)
  (lPwealth.lWgold) :: Functor f =>
    (Int -> f Int) -> Person -> f Person
```

- Allerdings mit verdrehter Reihenfolge: In der funktionalen Welt ist die äußere Funktion normalerweise auf der rechten Seite des Punktes:

```
*> negate.length $ [1..3]
-3
```

- Komposition von Linsen erinnert an Accessor- Verkettung in objektorientierten Sprachen wie etwa Java.
- Mit Hilfe des Infix-Synonym `(^.)` für `view` sieht das auch in Haskell so aus:

```
*> tyrion^.wealth.gold
100
```


Van Laarhoven-Linsen-Komposition (3)

- Andere Sichtweise: Komposition von Linsen ist wie Komposition von fmaps
- Bsp.

```
fmapMaybe :: (a -> b) -> Maybe a -> Maybe b
fmapList  :: (a -> b) -> [a] -> [b]
fmapPair  :: (a -> b) -> (c,a) -> (c,b)
```

- Dann ergibt `(fmapMaybe . fmapList . fmapPair)` ein `fmap` für den Typ `Maybe [(c,a)]`, d.h.

```
(fmapMaybe . fmapList . fmapPair) ::
           (a -> b) -> Maybe [(c, a)] -> Maybe [(c, b)]
```

- Typen werden genau wie bei der Komposition von Linsen von außen nach innen durchlaufen.

Infix-Operatoren

Das `lens` Paket definiert mehr als 100 Infix Operatoren:

```
view == (^.) :: s -> Lens' s a -> a
set  == (.~) :: Lens' s a -> a -> s -> s
over == (%~) :: Lens' s a -> (a -> a) -> s -> s
```

Die Benennung hält sich an folgende Konventionen:

- `^` kennzeichnet Getter-ähnliche Operatoren
- `~` Setter-ähnliche
- `.` grundlegende Operatoren
- `%` Operatoren mit Funktionen als Parameter
- `=` Setter-Variante für `State` Monade

Infix-Operatoren (2)

Viele Operatoren für kleine Bequemlichkeiten:

```
(&&~) :: ASetter' s Bool -> Bool -> s -> s
l &&~ n = over l (&& n)
(<>~) :: Monoid a => ASetter' s a -> a -> s -> s
l <>~ m = over l (`mappend` m)
```


Getter und Setter

Bis jetzt haben wir folgende Typen:

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

- Ein `Getter` verwendet für den Functor, den `Const`-Functor (wie bei `view`) und
- ein `Setter` verwendet (wie bei `over` und `set`), die `Identity`-Monade.
- Daher ist im `lens`-Paket definiert:

```
type AGetter' s a = forall r.(a -> Const r  a)-> s-> Const r s
type ASetter' s a =          (a -> Identity b)-> s-> Identity s
```


Polymorphe Linsen

```
type AGetter' s a = forall r.(a -> Const r  a)-> s-> Const r s
type ASetter' s a =          (a -> Identity b)-> s-> Identity s
type Lens'    s a = forall f. Functor f =>
                             (a -> f a)       -> s-> f s
```

Für polymorphe Datentypen wie z.B.

```
data Person nty wty = Person { name::nty, wealth::wty }
```

benötigen wir noch folgende Verallgemeinerung:

```
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)
type Setter s t a b  = (a -> Identity b) -> (s -> Identity t)
```

- Idee: Setter verwendet Funktion von $a \to b$, um Ausschnitt vom Typ a einer großen Struktur vom Typ s durch einen Ausschnitt vom Typ b zu ersetzen, wobei sich der Typ der Struktur von s auf t ändert.

Polymorphe Linsen (2)

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

Die Implementierung der Linsen ändert sich dabei nicht:

```
lPname :: Lens (Person n1 w) (Person n2 w) n1 n2
lPname  fun (Person n w) = (\n' -> Person n' w) <$> fun n
```


Linsen Gesetze

Pierce formulierte bereits Gesetze für „very well behaved lenses“:

Gesetze für „very well behaved lenses“

1. Man bekommt zurück, was man hineintut:
   `view l (set l v s)    == v`
2. Das vorhandene neu zu setzen ändert nichts:
   `set l (view l s) s    == s`
3. Nur das letzte Setzen zählt:
   `set l v2 (set l v1 s) == set l v2 s`

Edward Kmett betrachtete die Konsequenzen aus diesen Gesetzen und formulierte Varianten für weitere Optiken, welche wir jetzt noch betrachten werden.

Weitere Optiken

Linsen-Hierarchie, Traversal, Prismen, Isos

Traversal

Was passiert, wenn wir statt `Functor` `Applicative` einfordern?
Wir erhalten eine schwächere `Traversal`-Optik:

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

Achtung: Jede `Lens` ist auch ein `Traversal`, nicht umgekehrt, denn:

- `Traversal` ist eine Funktion, welche mit jedem beliebigen Typen `f` der Klasse `Applicative` umgehen können muss.
- `Lens` ist eine Funktion, welche mit jedem beliebigen Typen `f` der Klasse `Functor` umgehen können muss.

Wegen `Functor` $\supset$ `Applicative` muss `Lens` mehr Typen beherrschen!

Traversal

- Linse fokussiert immer **genau ein** Element
- `Traversal`-Optik hat **möglicherweise mehrere** Elemente im Focus.
- Funktion `traverse` aus der Klasse `Traversable` hat bereits einen zu van-Laarhoven-Linsen kompatiblen Typ:

```
traverse :: (Traversable t, Applicative f) =>
                      (a -> f b) ->  t a -> f (t b)
```

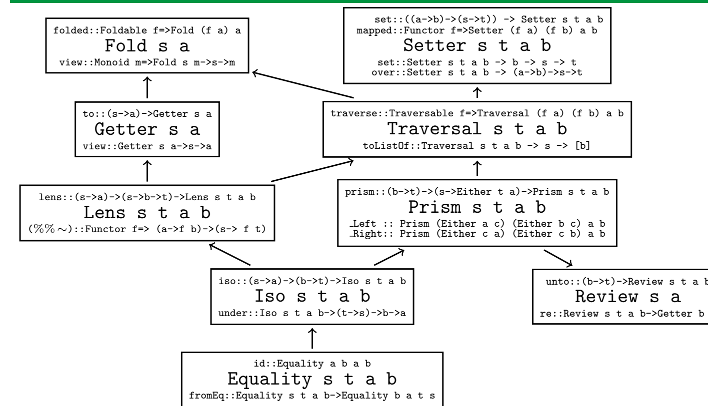
Anwendungsbeispiel:

```
data Faction = Faction{ faction::String, members::[Person] }
```

```
> over' (lFmembers.traverse.lPwealth.lWgold) (*10) lannisters
Lannisters[Cersei(2220g,22d),Tyrion(1000g,1d)]
```

- Wir nutzen hier natürlich aus, dass Listen `Traversable` sind.
- Beispiel nutzt selbst-definierte van-Laarhoven-Linsen und benötigt `over` mit speziellem Typen: `over' :: Traversal s s a a -> (a -> a) -> s -> s`
- `lens`-Paket definiert zur Umgehung solcher Probleme: Klassen und Typsynonyme

Linsen Hierarchie



- → entspricht "ist-auch-ein" mit `s=t` und `a=b` für Typen mit 2 Parametern.
- Traversal→Fold hatten wir schon gesehen!

Prismen

- Linsen fokussieren immer **genau ein** inneres Element.
- Prismen fokussieren dagegen **ein oder kein** Element.

```
type Prism s t a b= forall p f. (Choice p, Applicative f)=>
                                          p a (f b) -> p s (f t)
```

- Linsen arbeiten mit Produkt-Typen wir Paaren oder Records;
- Prismen kümmern sich um Summen-Typen wie `Maybe` oder `Either`.

Prismen (2)

- Ein Prisma kann also Pattern-Matching kodieren:

```
*> set _Just 7 (Just "Sieben")
Just 7
*> set _Just 7 Nothing
Nothing
*> over _Right (3*) (Right 7)
Right 21
*> over _Right (3*) (Left 7)
Left 7
```

Prismen (3)

Prismen sind keine Getter, da möglicherweise kein Fokus existiert.
Stattdessen erhalten wir ein `preview`:

```
> preview _Left (Left 42)
Just 42
> preview _Left (Right 42)
Nothing
```

Dafür lassen sich Prismen invertieren:

```
> review _Left 69
Left 69
```


Prismen (4)

Gesetze:

1. Eine Vorschau auf eine Rückschau gelingt immer mit dem gleichem Ergebnis:
   ```
   preview p (review p x)  == Just x
   ```
2. Eine Rückschau auf eine *erfolgreiche* Vorschau sollte ebenfalls gelingen:
   ```
   review p <$> preview p z == Just z
   ```


Iso

- Optik `Iso` ist für Isomorphismen
  und erlaubt es zwei Typen jederzeit ineinander zu überführen.
- Nützlich z.B. für
  - `newtype`-Wrapper
  - `Maybe a` und `Either () a` oder `(a,b)` und `(b,a)`, etc.
- Für einen Isomorphismus braucht man zwei Funktionen
  `fw :: s -> a` und `rw :: a -> s`, so dass gilt:
  ```
  fw . rw == id
  rw . fw == id
  ```
  damit können wir eine `Iso`-Optik erzeugen:
  `iso :: (s -> a) -> (b -> t) -> Iso s t a b`

Iso (2)

- Beispiel: `swappedPair`
  ```
  swappedPair :: Iso' (a,b) (b,a)
  swappedPair = iso (\(a,b) -> (b,a)) (\(b,a) -> (a,b))
  ```
- Da Isos auch Prismen sind, können wir `review` und `preview` für diese verwenden.
  ```
  *> preview swappedPair (1,True)
  Just (True,1)
  *> review swappedPair (1,True)
  (True,1)
  *> set (swappedPair._2) 5 (1,True)
  (5,True)
  ```
- Beachte: Optiken zwischen Haskell-Datentypen und deren XML- und/oder JSON-Repräsentationen, wie z.B. `Data.Aeson.Lens._JSON`
  sind Prismen, da das Parsen von XML/JSON fehlschlagen kann!

Zusammenfassung

Vor- und Nachteile des `lens`-Paket, allgemeine Zusammenfassung

Vorteile am Paket `lens`

- Paket `lens` ist ein schweizer Taschenmesser, z.B. liefert es Typ-sichere Linsen, welche in beliebige Tupel fokussieren:

```
*> view _3 ('a','b','c')
  'c'
*> view _3 ('a','b','c','d','e','f','g')
  'c'
*> set _3 'z' ('a','b','c','d','e','f','g','h')
  ('a','b','z','d','e','f','g','h')
*> :type _3
  _3 ::(Field3 s t a b, Functor f)=> (a -> f b)-> s -> f t
```

- Trick liegt hier in der Klasse `Field3`, eine Klasse für alle Tupel-artigen Container, welche ein drittes Element haben.
- TemplateHaskell `makeClassy` generiert Linsen und solche Klassen.

Kritik am `lens` Paket

Die `lens` Bibliothek ist sehr komplex:

- Selbst einfache Beschreibungen wirken einschüchternd oder verwenden Kategorientheorie
- Verwirrung durch unüberschaubare Flut von Klassen, Typen, Funktionen und Operatoren
- Sehr komplexe Typen und zahlreiche Typsynonyme liefern hoffnungslose Typfehler
- Installation des Pakets dauert lange, aufgrund vieler Abhängigkeiten

⇒ Es gibt einige alternative Linsen-Bibliotheken, welche sich darauf konzentrieren, möglichst leichtgewichtig zu sein, z.B. `fclabels`

Nützliche vordefinierte Optiken

Linsen

```
(_1) :: Field1 s t a b => Lens s t a b -- Projektionen
(_2) :: Field2 s t a b => Lens s t a b -- Projektionen
```

Traversals

```
firstOf      :: Traversal' s a -> Maybe a -- == preview
ix:: Index m -> Traversal' m (IxValue m)
lastOf       :: Traversal' s a -> Maybe a
```

Prismen

```
_Nothing :: Prism' (Maybe a)          ()
_Just    :: Prism  (Maybe a) (Maybe b) a b
_Left    :: Prism (Either a c) (Either b c) a b
_Right   :: Prism (Either c a) (Either c b) a b
_Cons    :: Prism  [a] [b] (a,[a]) (b,[b])
```

Zusammenfassung

- Funktionale Referenzen, auch bekannt als Linsen, ermöglichen Zugriff auf tiefer liegende Teile von abstrakten Datenstrukturen
- Linsen sind Werte, d.h. Referenzen in Datenstrukturen können übergeben, manipuliert und mit `(.)` zusammengesetzt werden
  ⇒ First-Class Values
- `Lens s t a b`: Dabei ist `s` der äußere Datentyp `s`, und `a` der innere Datentyp.
  Man kann `a` zu `b` verändern. Dann ist `t` der Datentyp, den man erhält, wenn man in `s` den Typ `a` durch `b` ersetzt.
- Rank-N Types und Typ-Klassen ermöglichen etwas, dass sich sehr ähnlich wie Subtyping verhält
  z.B. jede Linse ist ein Getter, jedes Prisma ein Traversal,...
- Verschiedene Implementierungen von Linsen sind möglich/erhältlich, auch für andere Programmiersprachen