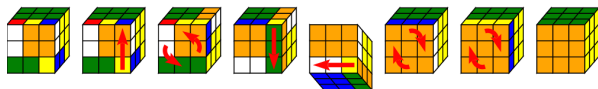


Lenses und Zauberwürfel



Tim Baumann

Curry Club Augsburg
13. August 2015



Welche Bibliothek?



Abbildung: Picking a Lens Library (Cartesian Closed Comic)



Welche Bibliothek?



Abbildung: Picking a Lens Library (Cartesian Closed Comic)

Die von Edward Kmett natürlich!



Welche Bibliothek?



Abbildung: Picking a Lens Library (Cartesian Closed Comic)

Die von Edward Kmett natürlich!

\$ cabal update



Welche Bibliothek?

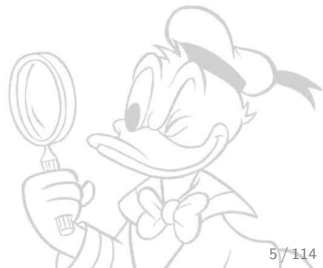


Abbildung: Picking a Lens Library (Cartesian Closed Comic)

Die von Edward Kmett natürlich!

\$ cabal update

\$ cabal install lens



Welche Bibliothek?



Abbildung: Picking a Lens Library (Cartesian Closed Comic)

Die von Edward Kmett natürlich!

\$ cabal update

\$ cabal install lens

Building profunctors...



Welche Bibliothek?



Abbildung: Picking a Lens Library (Cartesian Closed Comic)

Die von Edward Kmett natürlich!

`$ cabal update`

`$ cabal install lens`

Building profunctors...

Configuring semigroupoids...



Welche Bibliothek?



Abbildung: Picking a Lens Library (Cartesian Closed Comic)

Die von Edward Kmett natürlich!

`$ cabal update`

`$ cabal install lens`

`Building profunctors...`

`Configuring semigroupoids...`

`Downloading kan-extensions...`



Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.



Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine  :: String
  }
```

```
data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address   :: Address
  }
```



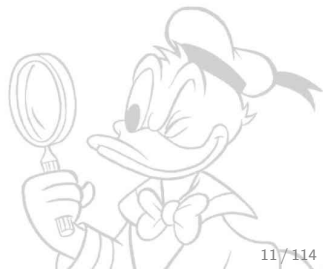
Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine  :: String
  }
```

```
data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address   :: Address
  }
```

```
data Lens† s a = Lens†
  { getter :: s -> a
  , setter :: a -> s -> s
  }
```



Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

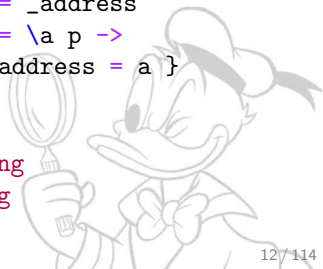
```
data Address = Address
  { _streetLine :: String
  , _townLine  :: String
  }
```

```
data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address   :: Address
  }
```

```
streetLine, townLine :: Lens† Address String
firstName,  lastName :: Lens† Person String
```

```
data Lens† s a = Lens†
  { getter :: s -> a
  , setter :: a -> s -> s
  }
```

```
address :: Lens† Person Address
address = Lens†
  { getter = _address
  , setter = \a p ->
      p { _address = a }
  }
```



Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine  :: String
  }
```

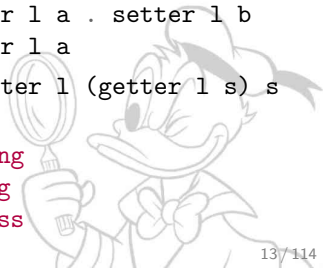
```
data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address   :: Address
  }
```

```
streetLine, townLine :: Lens† Address String
firstName,  lastName :: Lens† Person String
address    :: Lens† Person Address
```

```
data Lens† s a = Lens†
  { getter :: s -> a
  , setter :: a -> s -> s
  }
```

Lens-Gesetze:

- ① $a = \text{getter } l \ (\text{setter } l \ a \ s)$
- ② $\text{setter } l \ a \ . \ \text{setter } l \ b = \text{setter } l \ a$
- ③ $s = \text{setter } l \ (\text{getter } l \ s) \ s$



Was sind Lenses?

Was sind Lenses?

Eine Lens beschreibt eine (feste) Position in einer Datenstruktur, an der ein Wert eines bestimmten Typs gespeichert ist. Mit einer Lens ist es möglich, diesen Wert auszulesen und zu überschreiben.

```
data Address = Address
  { _streetLine :: String
  , _townLine  :: String
  }

data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address   :: Address
  }

data Lens' a b = Lens'
  { getter :: a -> b
  , setter :: a -> a -> b
  }

Lens-Gesetze:
  @ a = getter 1 (setter 1 a a)
  @ setter 1 a . setter 1 b
    = setter 1 a
  @ a = setter 1 (getter 1 a) a

streetLine, townLine :: Lens' Address String
firstName, lastName  :: Lens' Person String
address              :: Lens' Person Address
```

Die Lens-Gesetze sind die “costate comonad coalgebra”-Gesetze

Komponieren von Lenses

```
compose :: Lens† s a -> Lens† a b -> Lens† s b
```

```
personTownLine :: Lens† Person String  
personTownLine = compose address townLine
```



Komponieren von Lenses

```
compose :: Lens† s a -> Lens† a b -> Lens† s b
```

```
compose l m = Lens†  
  { getter = getter m . getter l  
  , setter = \b s -> setter l (setter m b (getter l s)) s  
  }
```

```
personTownLine :: Lens† Person String
```

```
personTownLine = compose address townLine
```



Komponieren von Lenses

```
compose :: Lens† s a -> Lens† a b -> Lens† s b
compose l m = Lens†
  { getter = getter m . getter l
  , setter = \b s -> setter l (setter m b (getter l s)) s
  }
```

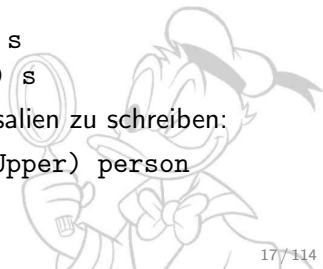
```
personTownLine :: Lens† Person String
personTownLine = compose address townLine
```

Folgende Hilfsfunktion ist oft nützlich:

```
modify :: Lens† s a -> (a -> a) -> s -> s
modify l f s = setter l (f (getter l s)) s
```

Zum Beispiel, um die Stadt in der Adresse in Versalien zu schreiben:

```
person' = modify personTownLine (map toUpper) person
```



Alles wunderbar? Leider nein

Problem Nr. 1: Bei der Auswertung

```
modify (compose l m) f s
```

```
= setter (compose l m) (f (getter (compose l m) s)) s
```

```
= setter l (setter m (f (getter m (getter l s)))) (getter l s)) s
```

wird `getter l s` zweimal berechnet. Besser wäre

```
modify (compose l m) f s
```

```
= let a = getter l s in setter l (setter m (f (getter m a)) a) s
```



Alles wunderbar? Leider nein

Problem Nr. 1: Bei der Auswertung

```
modify (compose l m) f s
= setter (compose l m) (f (getter (compose l m) s)) s
= setter l (setter m (f (getter m (getter l s)))) (getter l s) s
```

wird `getter l s` zweimal berechnet. Besser wäre

```
modify (compose l m) f s
= let a = getter l s in setter l (setter m (f (getter m a)) a) s
```

Problem Nr. 2: In `modify` wird die Datenstruktur zweimal durchlaufen: Einmal, um den gesuchten Wert zu extrahieren, dann nochmal, um den neuen Wert abzulegen. Das kann kostspielig sein, z. B. bei der `Lens` rechts.

```
data NonEmpty a =
  Cons a (NonEmpty a) | Last a
last :: Lens† (NonEmpty a) a
last = Lens† getter setter
where
  getter (Cons _ xs) = getter xs
  getter (Last x) = x
  setter a (Cons x xs) =
    Cons x (setter a xs)
  setter a (Last _) = Last a
```

└ Alles wunderbar? Leider nein

Alles wunderbar? Leider nein

Problem Nr. 1: Bei der Auswertung

```

modify (compose l m) f x
= setter (compose l m) (f (getter (compose l m) x)) x
= setter l (setter m (f (getter m (getter l x)))) (getter l x) x
wird getter l 2x zweimal berechnet. Besser wäre
modify (compose l m) f x
= let a = getter l x in setter l (setter m (f (getter m a)) a) x

```

Problem Nr. 2: In `modify`

```

wird die Datenstruktur zweimal
durchlaufen: Einmal, um den
gesuchten Wert zu extrahieren,
dann nochmal, um den neuen
Wert abzulegen.
Das kann kostspielig sein, z.B.
bei der Lens rechts.

```

```

data ReadOnly a =
  Cons a (ReadOnly a) | Last a
Last :: Lens' (ReadOnly a) a
last = Lens' getter setter
where
  getter (Cons _ _) = getter xs
  getter (Last x) = x
  setter a (Cons x xs) =
    Cons x (setter a xs)
  setter a (Last _) = Last a

```

Man kann diese Probleme auch anders lösen: Nämlich, indem man Lenses über die Store-Komonade definiert:

```

newtype Lens a b = Lens (a -> Store b a)
data Store b a = Store (b -> a) b

```

Das ermöglicht, die Datenstruktur nur einmal zu durchlaufen.

Dies macht das Paket `data-lens` von Edward Kmett. (Das ist der Vorgänger seiner `lens`-Library.)

Alles wunderbar? Leider nein

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.

```
data Lens† s a = Lens†  
  { getter  :: s -> a  
  , setter  :: a -> s -> s  
  , modify  ::  
  }
```

`(a -> a) -> s -> s`



Alles wunderbar? Leider nein `{-# LANGUAGE Rank2Types #-}`

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.

Wir verallgemeinern auch gleich `modify` auf effektvolle Updatefunktionen, d. h. solche, die beispielsweise `IO` verwenden:

```
data Lens‡ s a = Lens‡
  { getter  :: s -> a
  , setter  :: a -> s -> s
  , modifyF :: ∀ f. Functor f => (a -> f a) -> s -> f s
  }
```



Alles wunderbar? Leider nein `{-# LANGUAGE Rank2Types #-}`

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.

Wir verallgemeinern auch gleich `modify` auf effektvolle Updatefunktionen, d. h. solche, die beispielsweise `IO` verwenden:

```
data Lens† s a = Lens†
  { getter  :: s -> a
  , setter  :: a -> s -> s
  , modifyF :: ∀ f. Functor f => (a -> f a) -> s -> f s
  }
```

Bahnbrechende Einsicht von Twan van Laarhoven:

`modifyF` umfasst `getter` und `setter`!

Alles wunderbar? Leider nein `{-# LANGUAGE Rank2Types #-}`

Idee: Erweitere die Definition einer Lens um die `modify`-Funktion.

Wir verallgemeinern auch gleich `modify` auf effektvolle Updatefunktionen, d. h. solche, die beispielsweise `IO` verwenden:

```
data Lens† s a = Lens†
  { {-getter  :: s -> a
    , setter  :: a -> s -> s
    , -} modifyF :: ∀ f. Functor f => (a -> f a) -> s -> f s
  }
```

Bahnbrechende Einsicht von Twan van Laarhoven:

`modifyF` umfasst `getter` und `setter`!

└ Alles wunderbar? Leider nein

```
Alles wunderbar? Leider nein  [-* LANGUAGE Rank2Types *]  
Idee: Erweitere die Definition einer Lens um die modify-Funktion.  
Wir verallgemeinern auch gleich modify auf effektvolle Updatefunktionen,  
d.h. solche, die beispielweise IO verwenden.  
data Lens1 s a = Lens1  
{ -getter :: s -> a  
, -setter :: a -> s -> s  
, -} modifyF :: ∀ f. Functor f => (a -> f a) -> s -> f a  
}
```

Bahnbrechende Einsicht von Twan van Laarhoven:

modifyF umfasst getter und setter!

Diese Einsicht hat Twan van Laarhoven 2009 im Blogartikel “CPS based functional references” veröffentlicht.

modifyF umfasst getter und setter

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



modifyF umfasst 1. getter und setter

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

① ($\hat{\cdot}$) :: s -> Lens' s a -> a



modifyF umfasst 1. getter und 2. setter

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

① ($\hat{\cdot}$) :: s -> Lens' s a -> a

② ($\cdot\sim$) :: Lens' s a -> a -> s -> s



modifyF umfasst 1. getter und 2. setter

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

① ($\hat{\cdot}$) :: s -> Lens' s a -> a

② ($\cdot \sim$) :: Lens' s a -> a -> s -> s

```
newtype Id a = Id { getId :: a }  
instance Functor Id where  
  fmap f (Id a) = Id (f a)
```



modifyF umfasst 1. getter und 2. setter

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

① ($\hat{\cdot}$) :: s -> Lens' s a -> a

② (\sim) :: Lens' s a -> a -> s -> s
(\sim) l a s = getId (l (_ -> Id a) s)

```
newtype Id a = Id { getId :: a }  
instance Functor Id where  
  fmap f (Id a) = Id (f a)
```



modifyF umfasst 1. getter und 2. setter

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

① ($\hat{\cdot}$) :: s -> Lens' s a -> a

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

② ($\tilde{\cdot}$) :: Lens' s a -> a -> s -> s
($\tilde{\cdot}$) l a s = getId (l (_ -> Id a) s)

```
newtype Id a = Id { getId :: a }  
instance Functor Id where  
  fmap f (Id a) = Id (f a)
```



modifyF umfasst 1. getter und 2. setter

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

① $(\hat{\cdot}) :: s \rightarrow \text{Lens}' s a \rightarrow a$

```
s ^. l = getConst (l Const s)
```

```
newtype Const a b = Const { getConst :: a }
```

```
instance Functor (Const a) where
```

```
  fmap _ (Const b) = Const b
```

② $(\tilde{\cdot}) :: \text{Lens}' s a \rightarrow a \rightarrow s \rightarrow s$

```
( $\tilde{\cdot}$ ) l a s = getId (l (\_ -> Id a) s)
```

```
newtype Id a = Id { getId :: a }
```

```
instance Functor Id where
```

```
  fmap f (Id a) = Id (f a)
```



Komponieren von **Lens'**es

Gegeben: 1 :: **Lens'** s a

und m :: **Lens'** a b

Gesucht: ? :: **Lens'** s b



Komponieren von **Lens**'es

Gegeben: $l :: \forall f. \text{Functor } f \Rightarrow (a \rightarrow f a) \rightarrow s \rightarrow f s$
und $m :: \forall f. \text{Functor } f \Rightarrow (b \rightarrow f b) \rightarrow a \rightarrow f a$
Gesucht: $? :: \forall f. \text{Functor } f \Rightarrow (b \rightarrow f b) \rightarrow s \rightarrow f s$



Komponieren von **Lens**'es

Gegeben: $l :: \forall f. \text{Functor } f \Rightarrow (a \rightarrow f a) \rightarrow (s \rightarrow f s)$
und $m :: \forall f. \text{Functor } f \Rightarrow (b \rightarrow f b) \rightarrow (a \rightarrow f a)$
Gesucht: $? :: \forall f. \text{Functor } f \Rightarrow (b \rightarrow f b) \rightarrow (s \rightarrow f s)$



Komponieren von **Lens**'es

Gegeben: $l :: \forall f. \text{Functor } f \Rightarrow (a \rightarrow f a) \rightarrow (s \rightarrow f s)$

und $m :: \forall f. \text{Functor } f \Rightarrow (b \rightarrow f b) \rightarrow (a \rightarrow f a)$

Gesucht: $l.m :: \forall f. \text{Functor } f \Rightarrow (b \rightarrow f b) \rightarrow (s \rightarrow f s)$



Komponieren von `Lens`'es

Gegeben: `l :: ∀ f. Functor f => (a -> f a) -> (s -> f s)`

und `m :: ∀ f. Functor f => (b -> f b) -> (a -> f a)`

Gesucht: `l.m :: ∀ f. Functor f => (b -> f b) -> (s -> f s)`

Dabei ist `.` die stinknormale Funktionsverkettung aus der `Prelude`!



Komponieren von `Lens`'es

Gegeben: `l :: ∀ f. Functor f => (a -> f a) -> (s -> f s)`

und `m :: ∀ f. Functor f => (b -> f b) -> (a -> f a)`

Gesucht: `l.m :: ∀ f. Functor f => (b -> f b) -> (s -> f s)`

Dabei ist `.` die stinknormale Funktionsverkettung aus der `Prelude`!

Im Beispiel vom Anfang:

```
address :: Lens' Person Address
```

```
address f (Person first last addr) =  
  fmap (Person first last) (f addr)
```



Komponieren von `Lens`'es

Gegeben: `l :: ∀ f. Functor f => (a -> f a) -> (s -> f s)`
und `m :: ∀ f. Functor f => (b -> f b) -> (a -> f a)`
Gesucht: `l.m :: ∀ f. Functor f => (b -> f b) -> (s -> f s)`

Dabei ist `.` die stinknormale Funktionsverkettung aus der Prelude!

Im Beispiel vom Anfang:

```
address :: Lens' Person Address
```

```
address f (Person first last addr) =  
  fmap (Person first last) (f addr)
```

```
streetLine, townLine :: Lens' Address String
```

```
firstName, lastName :: Lens' Person String
```



Komponieren von `Lens`'es

Gegeben: `l :: ∀ f. Functor f => (a -> f a) -> (s -> f s)`
und `m :: ∀ f. Functor f => (b -> f b) -> (a -> f a)`
Gesucht: `l.m :: ∀ f. Functor f => (b -> f b) -> (s -> f s)`

Dabei ist `.` die stinknormale Funktionsverkettung aus der Prelude!

Im Beispiel vom Anfang:

```
address :: Lens' Person Address
```

```
address f (Person first last addr) =  
  fmap (Person first last) (f addr)
```

```
streetLine, townLine :: Lens' Address String
```

```
firstName, lastName :: Lens' Person String
```

Dann haben wir `address.townLine :: Lens' Person String`



Was ist ein Traversal?

```
class (Functor t, Foldable t) => Traversable t where  
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```



Was ist ein Traversal?

```
class (Functor t, Foldable t) => Traversable t where  
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

Ein **Traversal**'s `a` ermöglicht das Durchlaufen und Abändern von mehreren Werten vom Typ `a` in einem vom Typ `s` mit applik. Effekten.



Was ist ein Traversal?

```
class (Functor t, Foldable t) => Traversable t where  
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

Ein `Traversable` `s a` ermöglicht das Durchlaufen und Abändern von mehreren Werten vom Typ `a` in einem vom Typ `s` mit applik. Effekten.

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

```
traverse :: Traversable t => Traversal' (t a) a
```



Was ist ein Traversal?

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

Ein `Traversal' s a` ermöglicht das Durchlaufen und Abändern von mehreren Werten vom Typ `a` in einem vom Typ `s` mit applik. Effekten.

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

```
traverse :: Traversable t => Traversal' (t a) a
```

```
evenIxs :: Traversal' [a] a
```

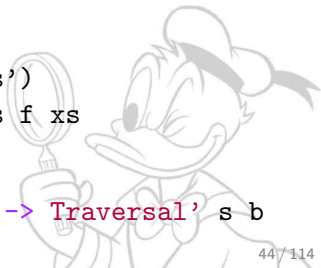
```
evenIxs f [] = pure []
```

```
evenIxs f [x] = (:[]) <$> f x
```

```
evenIxs f (x:y:xs) = (\x' xs' -> x':y:xs')
  <$> f x <*> evenIxs f xs
```

Traversals lassen sich wie Lenses verknüpfen:

```
(.) :: Traversal' s a -> Traversal' a b -> Traversal' s b
```

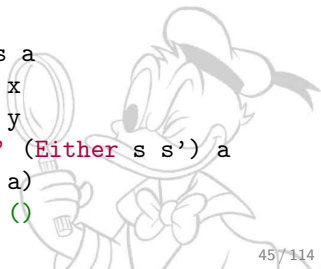


Weitere optische Gerätschaften aus `lens`

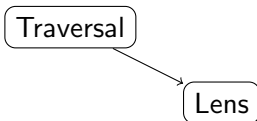
Lens

`Lens` ein `s` besteht aus einem `a` und anderen Daten

```
lens :: (s -> a) -> (s -> a -> s) -> Lens' s a
_1 :: Lens' (x,y) x      _1 :: Lens' (x,y,z) x
_2 :: Lens' (x,y) y      _2 :: Lens' (x,y,z) y
choosing :: Lens' s a -> Lens' s' a -> Lens' (Either s s') a
inside :: Lens' s a -> Lens' (e -> s) (e -> a)
devoid :: Lens' Void a    united :: Lens' a ()
```



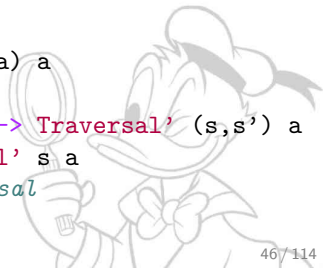
Weitere optische Gerätschaften aus `lens`



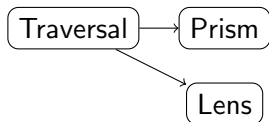
Traversal ein `s` besteht aus `a`'s und anderen Daten

Lens ein `s` besteht aus einem `a` und anderen Daten

```
traverse :: Traversable t => Traversal' (t a) a
both :: Traversal' (s,s) s
beside :: Traversal' s a -> Traversal' s' a -> Traversal' (s,s') a
taking :: Int -> Traversal' s a -> Traversal' s a
ignored :: Traversal' s a -- trivial traversal
```



Weitere optische Gerätschaften aus `lens`

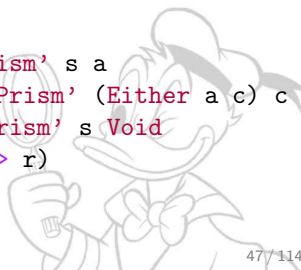


`Traversal` ein `s` besteht aus `a`'s und anderen Daten

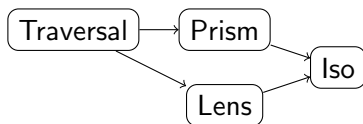
`Lens` ein `s` besteht aus einem `a` und anderen Daten

`Prism` ein `s` ist ein `a` oder etwas anderes

```
prism :: (a -> s) -> (s -> Either s a) -> Prism' s a
_Left :: Prism' (Either a c) a    _Right :: Prism' (Either a c) c
_Just :: Prism' (Maybe a) a      _Void :: Prism' s Void
outside :: Prism' s a -> Lens' (s -> r) (a -> r)
_Show :: (Read a, Show a) => Prism' String a
```



Weitere optische Gerätschaften aus `lens`



`Traversal` ein `s` besteht aus `a`'s und anderen Daten

`Lens` ein `s` besteht aus einem `a` und anderen Daten

`Prism` ein `s` ist ein `a` oder etwas anderes

`Iso` ein `s` ist dasselbe wie ein `a`

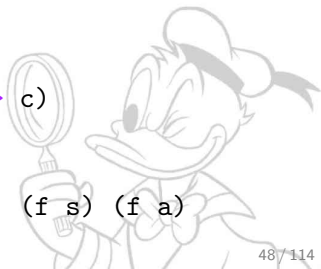
```
iso :: (s -> a) -> (a -> s) -> Iso' s a
```

```
curried :: Iso' ((a, b) -> c) (a -> b -> c)
```

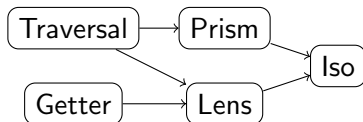
```
packed :: Iso' String Text
```

```
from :: Iso' s a -> Iso' a s
```

```
mapping :: Functor f => Iso' s a -> Iso' (f s) (f a)
```



Weitere optische Gerätschaften aus `lens`



Getter Funktion `s -> a`

Traversal ein `s` besteht aus `a`'s und anderen Daten

Lens ein `s` besteht aus einem `a` und anderen Daten

Prism ein `s` ist ein `a` oder etwas anderes

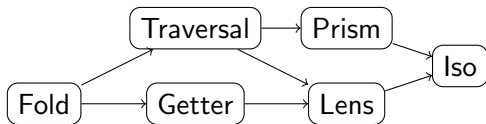
Iso ein `s` ist dasselbe wie ein `a`

```
to :: (s -> a) -> Getter s a
```

```
to length :: Getter [a] Int
```



Weitere optische Gerätschaften aus `lens`



Fold Funktion `s -> [a]`

Getter Funktion `s -> a`

Traversal ein `s` besteht aus `a`'s und anderen Daten

Lens ein `s` besteht aus einem `a` und anderen Daten

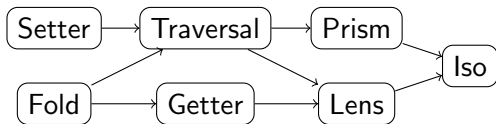
Prism ein `s` ist ein `a` oder etwas anderes

Iso ein `s` ist dasselbe wie ein `a`

```
unfolded :: (b -> Maybe (a, b)) -> Fold b a
folding  :: Foldable f => (s -> f a) -> Fold s a
folded   :: Foldable f => Fold (f a) a
replicated :: Int -> Fold a a
```



Weitere optische Gerätschaften aus `lens`



Setter in `s` gibt es veränderbare `a`'s

Fold Funktion `s -> [a]`

Getter Funktion `s -> a`

Traversal ein `s` besteht aus `a`'s und anderen Daten

Lens ein `s` besteht aus einem `a` und anderen Daten

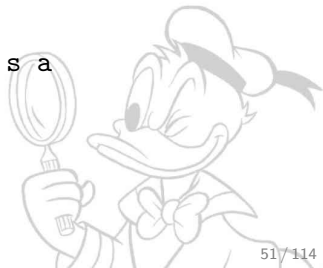
Prism ein `s` ist ein `a` oder etwas anderes

Iso ein `s` ist dasselbe wie ein `a`

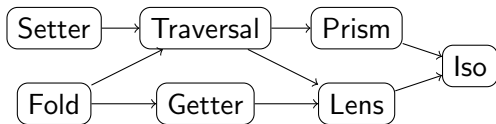
`sets :: ((a -> a) -> s -> s) -> Setter' s a`

`mapped :: Functor f => Setter' (f a) a`

`mapped :: Setter' (x -> a) a`



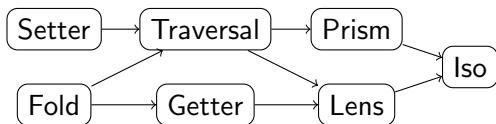
Weitere optische Gerätschaften aus `lens`



```
Setter' s a = (a -> Id a) -> s -> Id s
Fold s a = (Contrav't f, Applicative f) => (a -> f a) -> s -> f s
Getter s a = (Contrav't f, Functor f) => (a -> f a) -> s -> f s
Traversal' s a = Applicative f => (a -> f a) -> s -> f s
Lens' s a = Functor f => (a -> f a) -> s -> f s
Prism' s a = (Choice p, Applicative f) => p a (f a) -> p s (f s)
Iso' s a = (Profunctor p, Functor f) => p a (f a) -> p s (f s)
```



Weitere optische Gerätschaften aus `lens`



`Setter' s a = (a -> Id a) -> s -> Id s`

`Fold' s a = (Contrav't f, Applicative f) => (a -> f a) -> s -> f s`

`Getter' s a = (Contrav't f, Functor f) => (a -> f a) -> s -> f s`

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

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

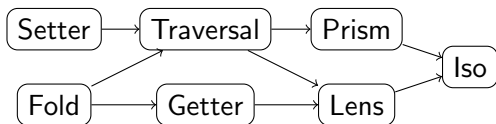
`Prism' s a = (Choice p, Applicative f) => p a (f a) -> p s (f s)`

`Iso' s a = (Profunctor p, Functor f) => p a (f a) -> p s (f s)`

- Durch Subtyping ist jeder Iso eine Lens, jedes Prism ein Traversal ...

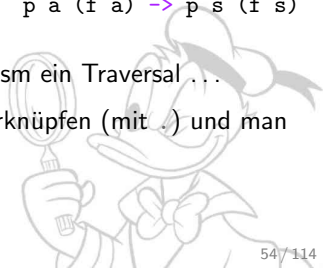


Weitere optische Gerätschaften aus `lens`

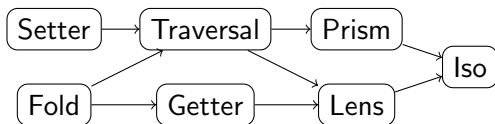


```
Setter' s a = (a -> Id a) -> s -> Id s
Fold s a = (Contrav't f, Applicative f) => (a -> f a) -> s -> f s
Getter s a = (Contrav't f, Functor f) => (a -> f a) -> s -> f s
Traversal' s a = Applicative f => (a -> f a) -> s -> f s
Lens' s a = Functor f => (a -> f a) -> s -> f s
Prism' s a = (Choice p, Applicative f) => p a (f a) -> p s (f s)
Iso' s a = (Profunctor p, Functor f) => p a (f a) -> p s (f s)
```

- Durch Subtyping ist jeder Iso eine Lens, jedes Prism ein Traversal ...
- Man kann z. B. eine Lens mit einem Traversal verknüpfen (mit `.`) und man bekommt ein Traversal.



Weitere optische Gerätschaften aus `lens`



```
Setter' s a = (a -> Id a) -> s -> Id s
Fold s a = (Contrav't f, Applicative f) => (a -> f a) -> s -> f s
Getter s a = (Contrav't f, Functor f) => (a -> f a) -> s -> f s
Traversal' s a = Applicative f => (a -> f a) -> s -> f s
Lens' s a = Functor f => (a -> f a) -> s -> f s
Prism' s a = (Choice p, Applicative f) => p a (f a) -> p s (f s)
Iso' s a = (Profunctor p, Functor f) => p a (f a) -> p s (f s)
```

- Durch Subtyping ist jeder Iso eine Lens, jedes Prism ein Traversal ...
- Man kann z. B. eine Lens mit einem Traversal verknüpfen (mit `.`) und man bekommt ein Traversal.
- Viele Beispielfunktionen haben einen allgemeineren Typ als angegeben.

└ Weitere optische Gerätschaften aus `lens`

Weitere optische Gerätschaften aus `lens`

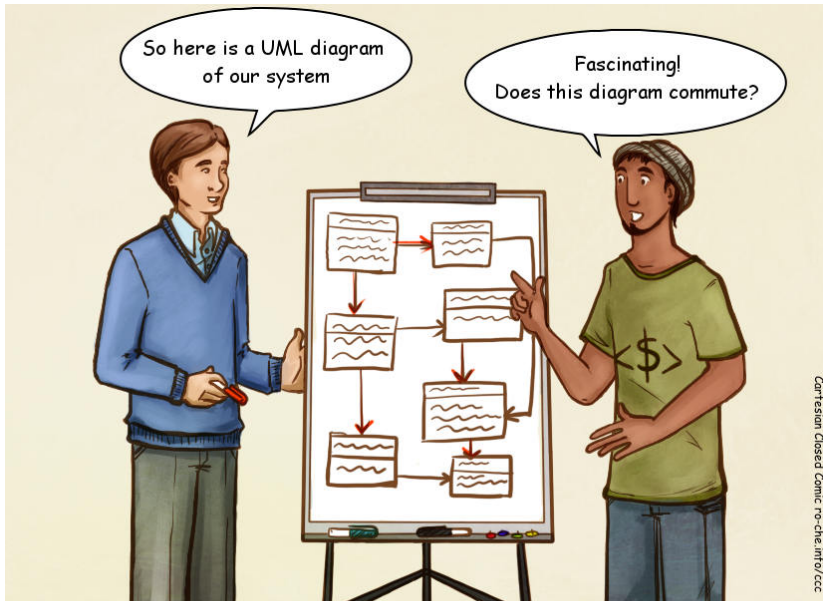
```

Setter' a a = (a -> Id a) -> a -> Id a
Fold a a = (Comonad' t f, Applicative f) -> (a -> f a) -> a -> f a
Getter a a = (Comonad' t f, Functor f) -> (a -> f a) -> a -> f a
Traversal' a a = (Comonad' t f, Applicative f) -> (a -> f a) -> a -> f a
Lens' a a a = (Functor f) -> (a -> f a) -> a -> f a
Prism' a a a = (Choice p, Applicative f) -> p a (f a) -> p a (f a)
Iso' a a = (Product p, Functor f) -> p a (f a) -> p a (f a)
  
```

- ↳ Durch Subtyping ist jeder `Iso` eine `Lens`, jedes `Prism` ein `Traversal`...
- ↳ Man kann z. B. eine `Lens` mit einem `Traversal` verknüpfen (mit `~`) und man bekommt ein `Traversal`.
- ↳ Viele Beispielfunktionen haben einen allgemeineren Typ als angegeben.

Prismen sind dual zu Lenses: Während eine `Lens' s a` den Typ `s` als Produkt von `a` und einem zweiten Typ darstellt, gibt ein `Prism' s a` eine Darstellung von `s` als Summe von `a` und einem zweiten Typ an. Es gibt außerdem noch den Typ `Review t b`. Er entspricht einer Funktion `b -> t`. Unter der Dualität von `Lens'` und `Prism'` entspricht ein `Getter` einem `Review`. Insbesondere ist jedes `Prism' b t` ein `Review t b`.

<https://github.com/ekmett/lens/wiki/Derivation> Das Lens-Wiki beschreibt die Herleitung dieser Typen.



~~Transition from Haskell to Java can be awkward~~
Learning the lens library

Polymorphe Updates

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



Polymorphe Updates

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



Polymorphe Updates

```
type Lens s t a b = forall f.  
  Functor f => (a -> f b) -> s -> f t  
  
type Lens' s a = Lens s s a a  
  
_1 :: Lens (x,y) (x',y) x x'
```



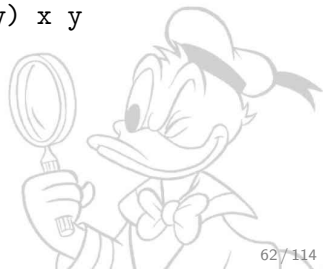
Polymorphe Updates

```
type Lens s t a b = forall f.  
  Functor f => (a -> f b) -> s -> f t  
  
type Lens' s a = Lens s s a a  
  
_1 :: Lens (x,y) (x',y) x x'  
  
set (_2._1) 42 ("hello",("world","!!!"))  
~> ("hello",(42,"!!!"))
```



Polymorphe Updates

```
type Lens s t a b = forall f.  
  Functor f => (a -> f b) -> s -> f t  
  
type Lens' s a = Lens s s a a  
  
_1 :: Lens (x,y) (x',y) x x'  
  
set (_2._1) 42 ("hello",("world","!!!"))  
~> ("hello",(42,"!!!"))  
  
type Setter s t a b = (a -> Id b) -> (s -> Id t)  
type Setter' s a = Setter s s a a  
mapped :: Functor f => Setter (f x) (f y) x y
```

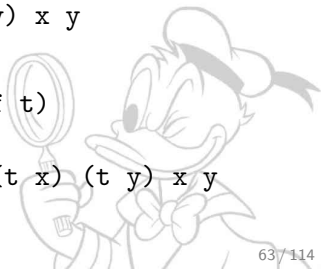


Polymorphe Updates

```
type Lens s t a b = forall f.  
  Functor f => (a -> f b) -> s -> f t  
  
type Lens' s a = Lens s s a a  
  
_1 :: Lens (x,y) (x',y) x x'  
  
set (_2._1) 42 ("hello",("world","!!!"))  
~> ("hello", (42, "!!!"))
```

```
type Setter s t a b = (a -> Id b) -> (s -> Id t)  
type Setter' s a = Setter s s a a  
mapped :: Functor f => Setter (f x) (f y) x y
```

```
type Traversal s t a b = forall f.  
  Applicative f => (a -> f b) -> (s -> f t)  
type Traversal' s a = Traversal s s a a  
traverse :: Traversable t => Traversal (t x) (t y) x y
```



Polymorphe Updates

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

```
type Prism' s a = Prism s s a a
```

```
_Right :: Prism (Either x y) (Either x y') y y'
```

```
set (_Right._2) "world" (Right ("hello",42))  
~> Right ("hello", "world")
```

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

```
type Setter' s a = Setter s s a a
```

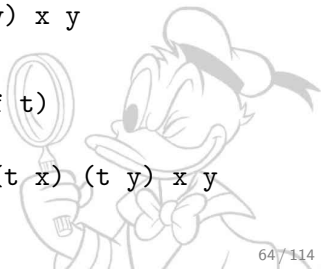
```
mapped :: Functor f => Setter (f x) (f y) x y
```

```
type Traversal s t a b = forall f.
```

```
  Applicative f => (a -> f b) -> (s -> f t)
```

```
type Traversal' s a = Traversal s s a a
```

```
traverse :: Traversable t => Traversal (t x) (t y) x y
```



└ Polymorphe Updates

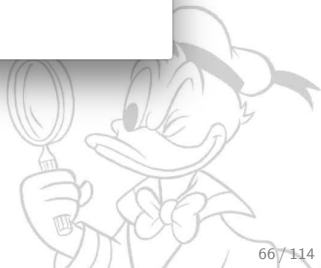
Polymorphe Updates

```
type Prism s t a b = forall p f.  
  (Choice p, Applicative f) => p a (f b) -> p a (f t)  
type Prism' s a = Prism s s a a  
_Right :: Prism (Either x y) (Either x y') y y'  
set (_Right_2) "world" (Right ("hello",42))  
  == Right ("hello","world")  
type Setter s t a b = (a -> Id b) -> (a -> Id t)  
type Setter' s a = Setter s s a a  
mapped :: Functor f => Setter (f a) (f y) x y  
type Traversal s t a b = forall f.  
  Applicative f => (a -> f b) -> (a -> f t)  
type Traversal' s a = Traversal s s a a  
traverse :: Traversable t => Traversal (t a) (t y) x y
```

Polymorphe Updates wurden zum ersten Mal 2012 in einem Blogartikel von Russell O'Connor beschrieben.

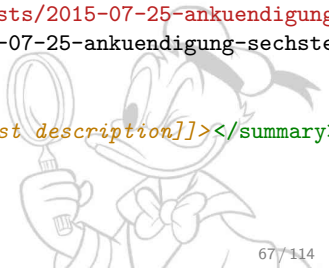
Beispiel: Curry-Feed parsen

```
fish /Users/tim/Projects/presentations/lens -- fish -- 60x15
~/P/p/lens λ curry-termine
* Programm für das sechste Treffen am 10. September 2015
* Drittes Treffen des Curry Clubs
* Programm für das fünfte Treffen am 13. August 2015
* Programm für das vierte Treffen am 16. Juli 2015
* Programm für das dritte Treffen am 18. Juni 2015
* Programm für das zweite Treffen am 21. Mai 2015
* Wir bauen einen Parserkombinator
* Erstes Treffen des Curry Clubs
* Programm für das erste Treffen am 23. April 2015
* Hallo Welt!
~/P/p/lens λ █
```



Beispiel: Curry-Feed parsen

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Curry Club Augsburg</title>
  <link href="http://curry-club-augsburg.de/atom.xml" rel="self" />
  <link href="http://curry-club-augsburg.de" />
  <id>http://curry-club-augsburg.de/atom.xml</id>
  <author>
    <name>Curry Club Augsburg</name>
    <email>post@curry-club-augsburg.de</email>
  </author>
  <updated>2015-07-25T00:00:00Z</updated>
  <entry>
    <title>Programm für das sechste Treffen am 10. September 2015</title>
    <link href="http://curry-club-augsburg.de/posts/2015-07-25-ankuendigung" />
    <id>http://curry-club-augsburg.de/posts/2015-07-25-ankuendigung-sechste</id>
    <published>2015-07-25T00:00:00Z</published>
    <updated>2015-07-25T00:00:00Z</updated>
    <summary type="html"><![CDATA[This is the post description]]></summary>
  </entry>
  <!-- weitere <entry>'s -->
</feed>
```



Beispiel: Curry-Feed parsen

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Data.Monoid ((<>))
import Data.Text.IO as T
import Text.XML
import Text.XML.Lens
import Network.Wreq

main :: IO ()
main = do
  res <- get "http://curry-club-augsburg.de/atom.xml"
  forOf_ (responseBody . to (parseLBS def) . _Right . entryTitles)
    res
    (T.putStrLn . ("* " <>))
  where
    entryTitles = root . childEl "entry" . childEl "title" . text
    childEl tag = nodes . traverse . _Element . named tag
```

Beispiel: Curry-Feed parsen

```
{-# LANGUAGE OverloadedStrings #-}
```

```
module Main where
```

```
import Data.Monoid ((<>))
import Data.Text.IO as T
import Text.XML
import Text.XML.Lens
import Network.Wreq
```

```
main :: IO ()
main = do
```

```
  res <- get "http://curry-club-augsburg.de/atom.xml"
  forOf_ (responseBody . to (parseLBS def) . _Right . entryTitles)
    res
    (T.putStrLn . ("* " <>))
```

```
where
```

```
  entryTitles = root . childEl "entry" . childEl "title" . text
  childEl tag = nodes . traverse . _Element . named tag
```

```
responseBody
  :: Lens' (Response body) body
root :: Lens' Document Element
nodes :: Lens' Element [Node]
_Element :: Prism' Node Element
named :: CI Text
  -> Traversal' Element Element
text :: Traversal' Element Text
```

Beispiel: Curry-Feed parsen

```
{-# LANGUAGE OverloadedStrings #-}
```

```
module Main where
```

```
import Data.Monoid ((<>))
import Data.Text.IO as T
import Text.XML
import Text.XML.Lens
import Network.Wreq
```

```
main :: IO ()
main = do
  res <- get "http://www.rubico-augsburg.de/atom.xml"
  forOf_ (responseBody . to (parseLBS def) . _Right . entryTitles)
    res
    (T.putStrLn . ("* " <>))
```

```
where
```

```
entryTitles = root . childEl "entry" . childEl "title" . text
childEl tag = nodes . traverse . _Element . named tag
```

```
responseBody
```

```
  :: Lens' (Response body) body
```

```
root  :: Lens' Document Element
```

```
nodes :: Lens' Element [Node]
```

```
Element
```

```
Element
```

```
Element
```

```
Text
```

Lens ist jQuery für Haskell!

└ Beispiel: Curry-Feed parsen


```

Beispiel: Curry-Feed parsen
(→ # LANGUAGE OverloadedStrings #-)

module Main where

import Data.Monoid ((<>))
import Data.Text.IO as T
import Text.XML
import Text.XML.Lens
import Network.HTTP

main = do
  res <- get "http://www.sputnig.de/atom.xml"
  forOf_ (responseBody <- to (parseES def) <_Right . entryTitles
        (T.putStrLn . ("* * <>)))
  where
    entryTitles = root < childEl "entry" < childEl "title" < text
    childEl tag = nodes < traverse <_Element < named tag
  
```



Verwendete Packages: `lens`, `text`, `xml-conduit`, `wreq`, `xml-lens` Es gibt auch Lenses für JSON-Daten. Diese waren früher in `lens` enthalten, jetzt sind sie Teil des Pakets `lens-aeson`.

Wie bekomme ich Lenses für meine Datentypen?

```
data Address = Address {...}
```

```
data Person = Person  
  { _firstName :: String  
  , _lastName  :: String  
  , _address   :: Address  
  }
```



Wie bekomme ich Lenses für meine Datentypen?

-- keine Imports nötig

```
data Address = Address {...}
```

```
data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address   :: Address
  }
```

```
address :: Functor f
  => (Address -> f Address)
  -> Person -> f Person
address f (Person first last addr) =
  fmap (Person first last) (f addr)
{-# INLINE address #-} -- empfohlen
-- und so weiter ...
```

① Lenses selber schreiben

Vorteil: Keine Library benötigt!

Nachteil: Boilerplate-Code



Wie bekomme ich Lenses für meine Datentypen?

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

```
data Address = Address {...}
```

```
data Person = Person  
  { _firstName :: String  
  , _lastName  :: String  
  , _address   :: Address  
  }
```

```
makeLenses ''Address
```

```
makeLenses ''Person
```

- ① Lenses selber schreiben
Vorteil: Keine Library benötigt!
Nachteil: Boilerplate-Code
- ② Lenses generieren mit Template Haskell-Funktionen aus `lens`
Vorteil: Komfortabel
Nachteil: Template Haskell



Wie bekomme ich Lenses für meine Datentypen?

```
{-# LANGUAGE TemplateHaskell #-}  
import Lens.Micro.TH  
-- aus 'microlens-th' (Beispiel)  
  
data Address = Address {...}  
  
data Person = Person  
  { _firstName :: String  
  , _lastName  :: String  
  , _address   :: Address  
  }  
  
makeLenses ''Address  
makeLenses ''Person
```

- ① Lenses selber schreiben
Vorteil: Keine Library benötigt!
Nachteil: Boilerplate-Code
- ② Lenses generieren mit Template Haskell-Funktionen aus `lens`
Vorteil: Komfortabel
Nachteil: Template Haskell
- ③ Lenses generieren mit einer anderen Bibliothek und TH
Vorteil: Komfortabel, keine Dependency auf `lens`
Nachteil: Template Haskell



Wie bekomme ich Lenses für meine Datentypen?

```
{-# LANGUAGE DeriveDataTypeable #-}
import Data.Data
```

```
data Address = Address {...}
  deriving (Typeable, Data)
data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address   :: Address
  } deriving (Typeable, Data)
```

-- Im Client-Code: Importiere Lens

```
import Data.Data.Lens
```

-- und benutze dann die Funktion:

```
biplate :: (Data s, Typeable a)
         => Traversal' s a
biplate :: Traversal' Person Address
```

- ① Lenses selber schreiben
Vorteil: Keine Library benötigt!
Nachteil: Boilerplate-Code
- ② Lenses generieren mit Template Haskell-Funktionen aus `lens`
Vorteil: Komfortabel
Nachteil: Template Haskell
- ③ Lenses generieren mit einer anderen Bibliothek und TH
Vorteil: Komfortabel, keine Dependency auf `lens`
Nachteil: Template Haskell
- ④ `Derive Data` und `Typeable` und benutze `Data.Data.Lens`
Nachteil: nur typgesteuerte Traversals möglich

└─ Wie bekomme ich Lenses für meine Datentypen?

Wie bekomme ich Lenses für meine Datentypen?

```
(=> LANGUAGE DerivDataTypable #
import Data.Data

data Address = Address (...)
  deriving (Typeable, Data)
data Person = Person
  { _firstName :: String
  , _lastName  :: String
  , _address  :: Address
  } deriving (Typeable, Data)

-- In Ghci-Code: Importiere Lens
import Data.Data.Lens
-- und benutze dann die Funktion:
biplate :: (Data a, Typeable a)
=> Traversal' a a
biplate :: Traversal' Person Address
```

- ⊗ Lenses selber schreiben
Vorteil: Keine Library benötigt!
Nachteil: Boilerplate-Code
- ⊗ Lenses generieren mit Template Haskell-Funktionen aus `lens`
Vorteil: Komfortabel
Nachteil: Template Haskell
- ⊗ Lenses generieren mit einer anderen Bibliothek und TH
Vorteil: Komfortabel, keine Dependency auf `lens`
Nachteil: Template Haskell
- ⊗ Derive `Data` und `Typeable` und benutze `Data.Data.Lens`
Nachteil: nur typotestete Traversals möglich

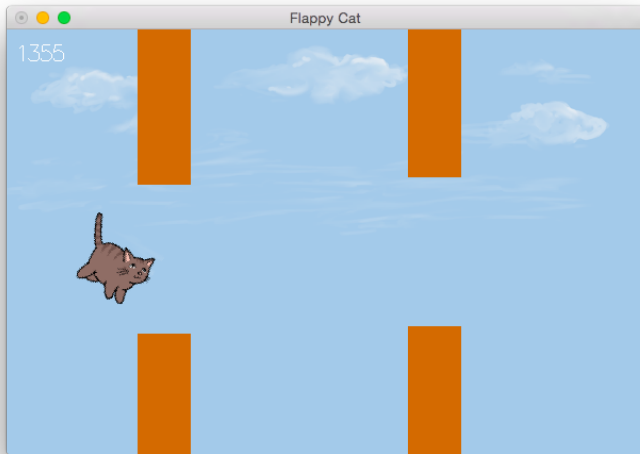
Im Lens-Wiki auf <https://github.com/ekmett/lens/wiki/How-can-I-write-lenses-without-depending-on-lens%3F> wird erklärt, wie man Lenses, Traversals, Folds etc. schreiben kann, ohne auf Lens zu dependen.

Lenses bieten die Möglichkeit, die konkrete Implementierung eines Datentyps nach außen hin zu verstecken, aber dennoch Zugriff auf Felder zu gewähren. Dadurch gewinnt man die Freiheit, die Implementierung des Datentyps zu verbessern, ohne das Interface nach außen zu verändern.

`Data.Data.Lens` und `Control.Lens.Plated` basieren auf der `uniplate`-Library für generisches Programmieren von Neil Mitchell.

Die `Plated`-Funktionalität von Lens kann besonders hilfreich für Compiler-Programmierer sein: Damit kann man nämlich z.B. mit wenig Code auf einem AST Transformationen durchführen.

Beispiel: Imperative (Spiele-) Programmierung



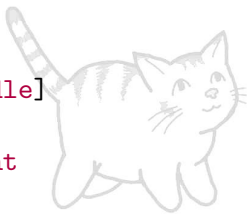
Beispiel: Imperative (Spiele-) Programmierung

```
data Pos = Pos { _x :: Float, _y :: Float }
makeLenses ''Pos

newtype Hurdle = Hurdle { _hurdlePos :: Pos }
makeLenses ''Hurdle

data GameState = Running | Paused | GameOver

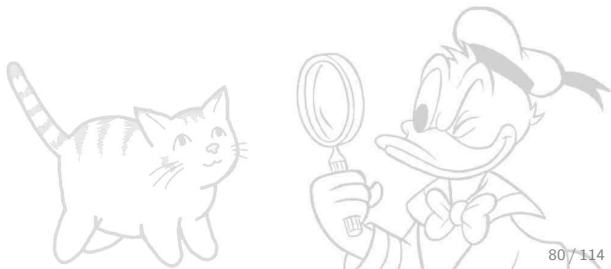
data FlappyCat =
  FlappyCat
  { _gen :: StdGen
  , _gameState :: GameState
  , _catPos :: Pos
  , _velY :: Float
  , _hurdles :: [Hurdle]
  }
makeLenses ''FlappyCat
```



Beispiel: Imperative (Spiele-) Programmierung

Ein paar Hilfsfunktionen:

```
randomHurdle :: (RandomGen g, MonadState g m)  
              => Float -> m Hurdle  
passes      :: Pos -> Hurdle -> Bool  
catExtremePoints :: FlappyCat -> [Pos]
```



Beispiel: Imperative (Spiele-) Programmierung

Ein paar Hilfsfunktionen:

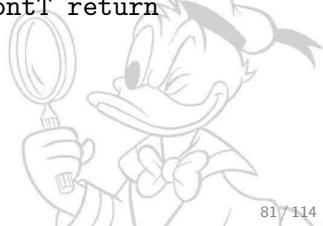
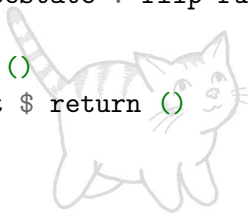
```
randomHurdle :: (RandomGen g, MonadState g m)
              => Float -> m Hurdle
passes      :: Pos -> Hurdle -> Bool
catExtremePoints :: FlappyCat -> [Pos]
```

Eine Monade:

```
type FlappyMonad = ContT () (State FlappyCat)

execFlappyMonad :: FlappyMonad () -> FlappyCat -> FlappyCat
execFlappyMonad = execState . flip runContT return

abort :: FlappyMonad ()
abort = ContT $ const $ return ()
```



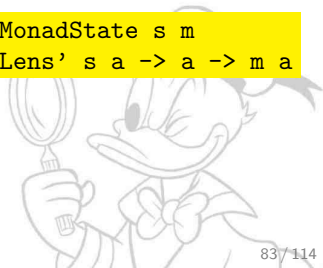
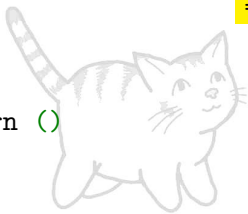
Beispiel: Imperative (Spiele-) Programmierung

```
handleInput :: Event -> FlappyMonad ()
handleInput (EventKey (Char 'p') Down _ _) =
  gameState %= \case
    Running -> Paused
    Paused -> Running
    GameOver -> GameOver
handleInput (EventKey (SpecialKey key) Down _ _)
| key `elem` jumpKeys = do
  velY .= jumpVel
  oldState <- gameState <<.= Running
  when (oldState == GameOver) $ do
    catPos.x .= 0
    catPos.y .= 0
    hurdles .= []
handleInput _ = return ()
```



Beispiel: Imperative (Spiele-) Programmierung

```
handleInput :: Event -> FlappyMonad ()
handleInput (EventKey (Char 'p') Down _ _) =
  gameState %= \case
    Running -> Paused
    Paused -> Running
    GameOver -> GameOver
handleInput (EventKey (SpecialKey key) Down _ _)
  | key `elem` jumpKeys = do
    vely .|= jumpVel
    oldState <- gameState <<.= Running
    when (oldState == GameOver) $ do
      catPos.x .= 0
      catPos.y .= 0
      hurdles .= []
handleInput _ = return ()
```



Beispiel: Imperative (Spiele-) Programmierung

```
step :: Float -> FlappyMonad ()
step dt = do
  state <- use gameState
  when (state /= Running) abort
  vy <- velY <+= dt*gravity
  px <- catPos.x <+= dt*velX
  py <- catPos.y <+= dt*vy
  when (py <= -h/2) $ gameState .= GameOver
  hs <- hurdles <%= filter ((> (px-w)) . (^ .hurdlePos.x))
  let lastX = fromMaybe (d+w) $
      lastOf (traverse.hurdlePos.x) hs
  when (lastX < px + 2*w) $ do
    hurdle <- lift $ zoom gen $ randomHurdle lastX
    hurdles .= hs ++ [hurdle]
  eps <- use $ to catExtremePoints
  unless (all id $ passes <$> eps <*> hs) $
    gameState .= GameOver
```

Beispiel: Imperative (Spiele-) Programmierung

```
step :: Float -> FlappyMonad ()
```

```
:: MonadState s m  
=> Getter s a -> m a
```

```
step dt = do
```

```
  state <- use gameState
```

```
:: (MonadState s m, Num a)
```

```
  when (state /= Running) abort
```

```
=> Lens' s a -> a -> m a
```

```
  vy <- vely <+= dt*gravity
```

```
  px <- catPos.x <+= dt*velX
```

```
:: MonadState s m
```

```
  py <- catPos.y <+= dt*vy
```

```
=> Lens' s a -> (a -> a) -> m a
```

```
  when (py <= -h/2) $ gameState .= GameOver
```

```
  hs <- hurdles <%= filter ((> (px-w)) . (^..hurdlePos.x))
```

```
  let lastX = fromMaybe (d+w) $
```

```
      lastOf (traverse.hurdlePos.x) hs
```

```
  when (lastX < px + 2*w) $ do
```

```
    hurdle <- lift $ zoom gen $ randomHurdle lastX
```

```
    hurdles .= hs ++ [hurdle]
```

```
:: Monad m => Lens' s t
```

```
  eps <- use $ to catExtremePo -> StateT t m a -> StateT s m a
```

```
  unless (all id $ passes <$> eps <*> hs) $
```

```
    gameState .= GameOver
```

Operatoren in Lens

```
(%~)  :: Setter s t a b -> (a -> b) -> s ->      t
(<%~) :: Lens   s t a b -> (a -> b) -> s -> (b, t)
(<<%~) :: Lens   s t a b -> (a -> b) -> s -> (a, t)
(=%)  :: MonadState s m => Setter s s a b -> (a -> b) -> m ()
(<%=) :: MonadState s m => Lens   s s a b -> (a -> b) -> m b
(<<%=) :: MonadState s m => Lens   s s a b -> (a -> b) -> m a
```

Funktional	%~	.~	+~	<>~	~
Funktional mit Ergebnis	<%~	<.~	<+~	<<>~	< ~
Funktional mit vorh. Wert	<<%~	<<.~	<<+~	<<<>~	<< ~
Monadisch	%=	.=	=	<>=	=
Monadisch mit Ergebnis	<%=	<.=	<+=	<<>=	< =
Monadisch mit vorh. Wert	<<%=	<<.=	<<+=	<<<>=	<< =

(Vollständige Liste auf <https://github.com/ekmett/lens/wiki/Operators>)

Operatoren in Lens

```
(.~) :: Setter s t a b -> b -> s -> t  
(<.~) :: Setter s t a b -> b -> s -> (b, t)  
(<<.~) :: Lens s t a b -> b -> s -> (a, t)  
(.=) :: MonadState s m => Setter s s a b -> b -> m ()  
(<.=) :: MonadState s m => Setter s s a b -> b -> m b  
(<<.=) :: MonadState s m => Lens' s s a b -> b -> m a
```

Funktional	%~	.~	+~	<>~	~
Funktional mit Ergebnis	<%~	<.~	<+~	<<>~	< ~
Funktional mit vorh. Wert	<<%~	<<.~	<<+~	<<<>~	<< ~
Monadisch	%=	.=	=	<>=	=
Monadisch mit Ergebnis	<%=	<.=	<+=	<<>=	< =
Monadisch mit vorh. Wert	<<%=	<<.=	<<+=	<<<>=	<< =

(Vollständige Liste auf <https://github.com/ekmett/lens/wiki/Operators>)

Operatoren in Lens

```
(+~) :: Num a => Setter s t a a -> a -> s -> t
(<+~) :: Num a => Lens s t a a -> a -> s -> (a, t)
(<<+~) :: Num a => Lens s t a a -> a -> s -> (a, t)
(+=) :: (Num a, MonadState s m) => Setter' s a -> a -> m ()
(<+=) :: (Num a, MonadState s m) => Lens' s a -> a -> m a
(<<+=) :: (Num a, MonadState s m) => Lens' s a -> a -> m a
```

Funktional	%~	.~	+~	<>~	~
Funktional mit Ergebnis	<%~	<.~	<+~	<<>~	< ~
Funktional mit vorh. Wert	<<%~	<<.~	<<+~	<<<>~	<< ~
Monadisch	%=	.=	+=	<>=	=
Monadisch mit Ergebnis	<%=	<.=	<+=	<<>=	< =
Monadisch mit vorh. Wert	<<%=	<<.=	<<+=	<<<>=	<< =

(Vollständige Liste auf <https://github.com/ekmett/lens/wiki/Operators>)

Operatoren in Lens

```
(<>~) :: Monoid a => Setter s t a a -> a -> s -> t
(<<>~) :: Monoid a => Lens s t a a -> a -> s -> (a, t)
(<<<>~) :: Monoid a => Lens s t a a -> a -> s -> (a, t)
(<>=) :: (Monoid a, MonadState s m) => Setter' s a -> a -> m ()
(<<>=) :: (Monoid a, MonadState s m) => Lens' s a -> a -> m a
(<<<>=) :: (Monoid a, MonadState s m) => Lens' s a -> a -> m a
```

Funktional	%~	.~	+~	<>~	~
Funktional mit Ergebnis	<%~	<.~	<+~	<<>~	< ~
Funktional mit vorh. Wert	<<%~	<<.~	<<+~	<<<>~	<< ~
Monadisch	%=	.=	=	<>=	=
Monadisch mit Ergebnis	<%=	<.=	<+=	<<>=	< =
Monadisch mit vorh. Wert	<<%=	<<.=	<<+=	<<<>=	<< =

(Vollständige Liste auf <https://github.com/ekmett/lens/wiki/Operators>)

Operatoren in Lens

```
(||~) :: Setter s t Bool Bool -> Bool -> s -> t
(<||~) :: Lens s t Bool Bool -> Bool -> s -> (Bool, t)
(<<||~) :: Lens s t Bool Bool -> Bool -> s -> (Bool, t)
(==) :: MonadState s m => Setter' s Bool -> Bool -> m ()
(<==) :: MonadState s m => Lens' s Bool -> Bool -> m Bool
(<<==) :: MonadState s m => Lens' s Bool -> Bool -> m Bool
```

Funktional	%~	.~	+~	<>~	~
Funktional mit Ergebnis	<%~	<.~	<+~	<<>~	< ~
Funktional mit vorh. Wert	<<%~	<<.~	<<+~	<<<>~	<< ~
Monadisch	%=	.=	=	<>=	=
Monadisch mit Ergebnis	<%=	<.=	<+=	<<>=	< =
Monadisch mit vorh. Wert	<<%=	<<.=	<<+=	<<<>=	<< =

(Vollständige Liste auf <https://github.com/ekmett/lens/wiki/Operators>)

Operatoren in Lens

```
(||~) :: Setter s t Bool Bool -> Bool -> s -> t
(<||~) :: Lens s t Bool Bool -> Bool -> s -> (Bool, t)
(<<||~) :: Lens s t Bool Bool -> Bool -> s -> (Bool, t)
(==) :: MonadState s m => Setter' s Bool -> Bool -> m ()
(<==) :: MonadState s m => Lens' Bool Bool -> m Bool
(<<==) :: MonadState s m => Lens' Bool Bool -> m Bool
```

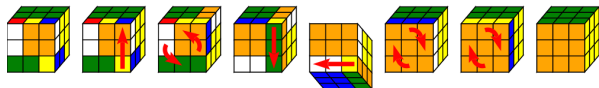
*Du willst in Haskell write-only
Code wie in Perl schreiben?
Dann versuche Lens!*

Funktion					~
Funktion					< ~
					<< ~
					<>~
					<< ~
Monadisch mit Ergebnis	<%=	<.=	<+=	<<>=	=
Monadisch mit vorh. Wert	<<%=	<<.=	<<+=	<<<>=	<< =

(Vollständige Liste auf <https://github.com/ekmett/lens/wiki/Operators>)

Beispiel: diagrams-rubiks-cube

Ziel: Zauberwürfel und Lösungsalgorithmen zeichnen. *Beispiel:*



```
import Diagrams.RubiksCube
import Control.Lens
```

```
diagram :: RubiksCubeBackend n b => Diagram b
diagram =
  let moves      = [B, R, F', R', D', F, F]
      endPos     = solvedRubiksCube
      settings   = with & showStart .~ True
  in drawMovesBackward settings endPos moves
```



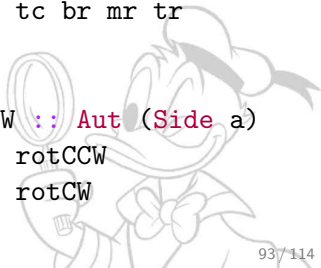
Beispiel: diagrams-rubiks-cube

```
data Side a = Side
  {   _topLeft   :: a,   _topCenter  :: a,   _topRight   :: a
    , _middleLeft :: a, _middleCenter :: a, _middleRight :: a
    , _bottomLeft :: a, _bottomCenter :: a, _bottomRight :: a
  } deriving (Show, Eq, Functor, Foldable, Traversable)
instance Applicative Side
makeLenses ''Side
```

tl	tc	tr
ml	mc	mr
bl	bc	br

```
rotCW, rotCCW :: Side a -> Side a
rotCW (Side tl tc tr ml mc mr bl bc br) =
  Side bl ml tl bc mc tc br mr tr
```

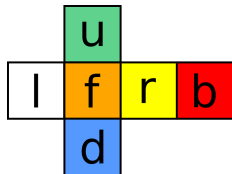
```
type Aut a = Iso' a a
rotateSideCW, rotateSideCCW :: Aut (Side a)
rotateSideCW = iso rotCW rotCCW
rotateSideCCW = iso rotCCW rotCW
```



Beispiel: diagrams-rubiks-cube

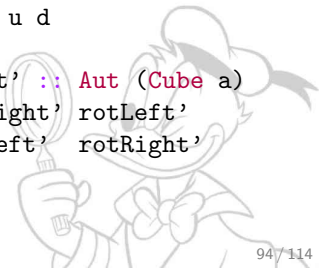
```
data Cube a = Cube
  { _frontSide :: a, _backSide :: a
  , _leftSide  :: a, _rightSide :: a
  ,   _upSide  :: a, _downSide :: a
  } deriving (Show, Eq, Functor, Foldable, Traversable)
```

```
instance Applicative Cube
makeLenses ''Cube
```

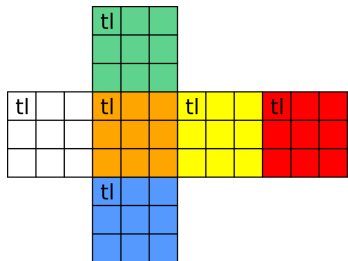


```
rotRight', rotLeft'  :: Cube a -> Cube a
rotRight' (Cube f b l r u d) =
  Cube l r b f u d
```

```
rotateRight', rotateLeft' :: Aut (Cube a)
rotateRight' = iso rotRight' rotLeft'
rotateLeft'  = iso rotLeft'  rotRight'
```



Beispiel: diagrams-rubiks-cube



```
newtype RubiksCube a = RubiksCube
  { _cube :: Cube (Side a) }
  deriving (Show, Eq, Functor)
```

```
instance Applicative RubiksCube
makeLenses ''RubiksCube
```



Beispiel: diagrams-rubiks-cube

-- Wende einen Automorphismus auf alle Komponenten an

```
cong :: Traversal' s a -> Aut a -> Aut s
```

```
cong t i = withIso i $ \f g -> iso (over t f) (over t g)
```

```
rotateRight, rotateLeft :: Aut (RubiksCube a)
```

```
rotateRight =
```

```
  cong cube $ rotateRight'
```

```
    . cong  upside rotateSideCCW
```

```
    . cong downside rotateSideCW
```

```
rotateLeft = from rotateRight
```

```
rotateUp, rotateDown :: Aut (RubiksCube a)
```

```
rotateCW, rotateCCW :: Aut (RubiksCube a)
```

```
rotateCW = rotateUp . rotateLeft . rotateDown
```

```
rotateCCW = from rotateCW
```



Beispiel: diagrams-rubiks-cube

```
data Vec4 a = Vec4 a a a a
  deriving (Show, Eq, Functor, Foldable, Traversable)
```

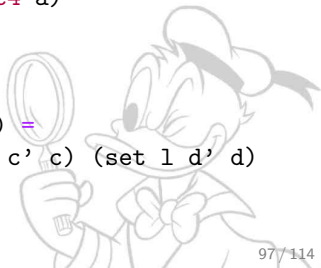
```
cycRight, cycLeft :: Vec4 a -> Vec4 a
cycRight (Vec4 a b c d) = Vec4 d a b c
```

```
cycleRight, cycleLeft :: Aut (Vec4 a)
cycleRight = iso cycRight cycLeft
cycleLeft  = from cycleRight
```

```
liftVec4 :: Lens' s a -> Lens' (Vec4 s) (Vec4 a)
liftVec4 l = lens getter setter
```

where

```
getter = fmap (^ . l)
setter (Vec4 a b c d) (Vec4 a' b' c' d') =
  Vec4 (set l a' a) (set l b' b) (set l c' c) (set l d' d)
```

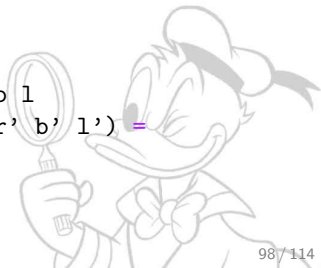


Beispiel: diagrams-rubiks-cube

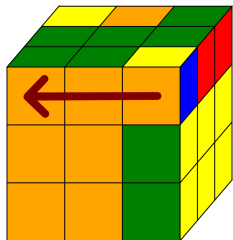
```
data Vec3 a = Vec3 a a a
  deriving (Show, Eq, Functor, Foldable, Traversable)

topRow :: Lens' (Side a) (Vec3 a)
topRow = lens getter setter
  where
    getter (Side t1 tc tr _ _ _ _ _ _) = Vec3 t1 tc tr
    setter (Side _ _ _ ml mc mr bl bc br) (Vec3 t1 tc tr) =
      Side t1 tc tr ml mc mr bl bc br

horizontalSides :: Lens' (Cube a) (Vec4 a)
horizontalSides = lens getter setter
  where
    getter (Cube f b l r _u _d) = Vec4 f r b l
    setter (Cube _f _b _l _r u d) (Vec4 f' r' b' l') =
      Cube f' b' l' r' u d
```



Beispiel: diagrams-rubiks-cube



```
upRows :: RowsLens a
```

```
upRows = horizontalSides . liftVec4 topRow
```

```
-- Drehe die oberste Ebene im Uhrzeigersinn
```

```
moveU :: Aut (RubiksCube a)
```

```
moveU =
```

```
  cong cube $ cong upRows cycleLeft
```

```
    . cong upSide rotateSideCW
```

Zur Erinnerung:

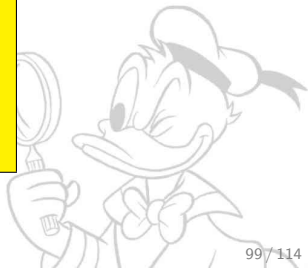
```
cong :: Traversal' s a -> Aut a -> Aut s
```

```
topRow :: Lens' (Side a) (Vec3 a)
```

```
horizontalSides :: Lens' (Cube a) (Vec4 a)
```

```
cycleLeft :: Aut (Vec4 a)
```

```
rotateSideCW :: Aut (Side a)
```



Wo kann ich mehr über `lens` erfahren?

- **Das Lens-Wiki:** <https://github.com/ekmett/lens/wiki>
- Blogserie “Lens over Tea” <http://artyom.me/lens-over-tea-1>
- Vortrag von Simon Peyton Jones bei Skills Matter
- Vortrag von Edward Kmett:
“The Unreasonable Effectiveness of Lenses for Business Applications”
- Blogpost: “Program imperatively using Haskell lenses”
- School of Haskell: “A Little Lens Starter Tutorial”
- Cheat Sheet für `Control.Lens`:
<https://github.com/anchor/haskell-cheat-sheets>



└─ Wo kann ich mehr über [lens](#) erfahren?

Wo kann ich mehr über [lens](#) erfahren?

- **Das Lens Wiki:** <https://github.com/kimmetz/lens/wiki>
- Blogserie "Lens over Tea" <http://artyom.me/lens-over-tea-1>
- Vortrag von Simon Peyton Jones bei Skills Matter
- Vortrag von Edward Kmett:
"The Unreasonable Effectiveness of Lenses for Business Applications"
- Blogpost: "Program imperatively using Haskell lenses"
- School of Haskell: "A Little Lens Starter Tutorial"
- Cheat Sheet für Control.Lens:
<https://github.com/anchor/haskell-cheat-sheets>

Auch ziemlich cool: total-1.0.0: Exhaustive pattern matching using traversals, prisms, and lenses



<http://timbaumann.info/lens>

<https://github.com/timjb/presentations/tree/gh-pages/lens>

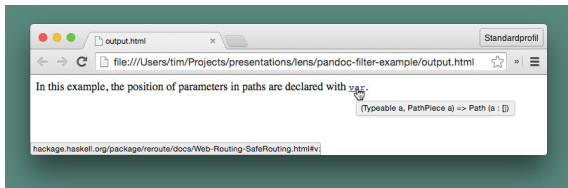
Beispiel: Haddock-Links mit Pandoc und Plated

Ziel: Verlinke Haskell-Funktionen in Markdown-Dokumenten

Markdown:

In this example, the position of parameters in paths are declared with 'Web.Routing.SafeRouting.var'.

```
$ pandoc --filter hayoo-links type-safe_routing.markdown > output.html
```



Beispiel: Haddock-Links mit Pandoc und Plated

```
data Inline
  = Str String
  | Emph [Inline]
  | Math MathType String
  | Link [Inline] Target
  | Image [Inline] Target
  ...
deriving (... , Typeable, Data, Generic)
```

```
data Block
  = Para [Inline]
  | BlockQuote [Block]
  | BulletList [[Block]]
  | Header Int Attr [Inline]
  ...
deriving (... , Typeable, Data, Generic)
```

```
data Pandoc = Pandoc Meta [Block]
deriving (... , Typeable, Data, Generic)
```



Beispiel: Haddock-Links mit Pandoc und Plated

```
data HsIdentifier = HsModule [String] | HsFunction [String] String
```

```
parseHsIdentifier :: String -> Maybe HsIdentifier
```

```
-- returns (str', url, title) pair
```

```
getIdentifierInfos :: HsIdentifier  
                    -> IO (Maybe (Maybe String, String, String))
```

```
hayooLinkModules :: Inline -> IO Inline
```

```
hayooLinkModules code@(Code attr str) =
```

```
  case parseHsIdentifier str of
```

```
    Nothing -> return code
```

```
    Just identifier ->
```

```
      maybe code (\(str',url,title) -> Link [maybe code (Code attr) str']  
              (url, title))
```

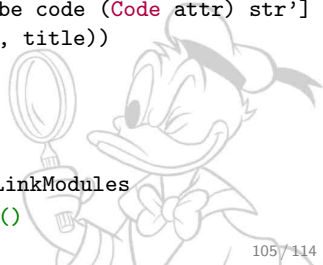
```
      <$> getIdentifierInfos identifier
```

```
hayooLinkModules x = return x
```

```
main :: IO ()
```

```
main = pipeline $ traverseOf template hayooLinkModules
```

```
  where pipeline :: (Pandoc -> IO Pandoc) -> IO ()
```



Beispiel: Haddock-Links mit Pandoc und Plated

```
data HsIdentifier = HsModule [String] | HsFunction [String] String
```

```
parseHsIdentifier :: String -> Maybe HsIdentifier
```

```
-- returns (str', url, title) pair
```

```
getIdentifierInfos :: HsIdentifier  
                    -> IO (Maybe (Maybe String, String, String))
```

```
hayooLinkModules :: Inline -> IO Inline
```

```
hayooLinkModules code@(Code attr str) =
```

```
  case parseHsIdentifier str of
```

```
    Nothing -> return code
```

```
    Just identifier ->
```

```
      maybe code (\(str',url,title) -> Link [maybe code (Code attr) str']  
                (url, title))
```

```
      <$> getIdentifierInfos identifier
```

```
hayooLinkModules x = return x
```

```
main :: IO ()
```

```
main = pipeline $ traverseOf template hayooLinkModules
```

```
  where pipeline :: (Pandoc -> IO Pandoc) -> IO ()
```

```
:: Applicative f => Traversal s t a b  
-> (a -> f b) -> s -> f t
```

```
:: forall s a. (Data s, Typeable a)  
=> Traversal' s a
```

Anwendung: Ausnahmebehandlung

```
class (Typeable e, Show e) => Exception e where ...  
data SomeException = forall e. Exception e => SomeException e  
handle :: Exception e => (e -> IO a) -> IO a -> IO a
```

```
handle (\(exc :: AssertionFailed) -> return "caught")  
      (assert (2+2 == 3) (return "uncaught"))  
~> "caught"
```

```
handle (\(exc :: AssertionFailed) -> return "caught")  
      (assert (2+2 == 4) (return "works"))  
~> "works"
```

Es ist doof, dass man das Argument im Exception-Handler mit einem Typ annotieren muss. Doch zum Glück gibt es `Control.Exception.Lens!`



Anwendung: Ausnahmebehandlung

```
import Control.Exception.Lens
catching :: MonadCatch m => Prism' SomeException a
         -> m r -> (a -> m r) -> m r
```

```
catching _AssertionFailed (assert (2+2 == 3) (return "uncaught"))
                          (const (return "caught"))
```

↪ "caught"

```
catching _AssertionFailed (assert (2+2 == 4) (return "works"))
                          (const (return "caught"))
```

↪ "works"

In `Control.Exception.Lens` sind ganz viele Prisms vordefiniert:

```
_IndexOutOfBounds :: Prism' SomeException String
_StackOverflow     :: Prism' SomeException ()
_UserInterrupt     :: Prism' SomeException ()
_DivideByZero     :: Prism' SomeException ArithException
_AssertionFailed   :: Prism' SomeException String
-- (usw)
```

Anwendung: Defaultparameter

Angenommen, wir schreiben eine HTTP-Library (Beispiel geklaut von Oliver Charles)

```
data HTTPSettings = HTTPSettings
  { _httpKeepAlive :: Bool
  , _httpCookieJar :: CookieJar
  } deriving (Show, ...)
makeLenses ''HTTPSettings

defaultHTTPSettings :: HTTPSettings
defaultHTTPSettings = HTTPSettings True emptyCookieJar

httpRequest :: HTTPSettings -> HTTPRequest -> IO Response
instance Default HTTPSettings where
  def = defaultHTTPSettings

httpRequest
  (def & httpKeepAlive .~ True
   & httpCookieJar .~ myCookieJar)
  aRequest
```

Kritik: Default ist eine gesetzlose Typklasse!



Anwendung: Defaultparameter

Angenommen, wir schreiben eine HTTP-Library (Beispiel geklaut von Oliver Charles)

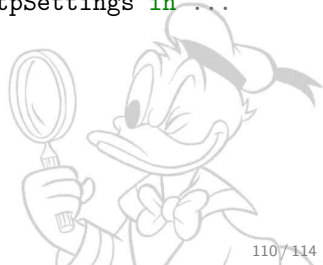
```
data HTTPSettings = HTTPSettings
  { _httpKeepAlive :: Bool
  , _httpCookieJar  :: CookieJar
  } deriving (Show, ...)
makeLenses ''HTTPSettings

defaultHTTPSettings :: HTTPSettings
defaultHTTPSettings = HTTPSettings True emptyCookieJar

httpRequest :: State HTTPSettings a -> HTTPRequest -> IO Response
httpRequest mkState req =
  let config = execState mkConfig defaultHttpSettings in ...

httpRequest
  (do httpKeepAlive .= True
      httpCookieJar  .= myCookieJar)
  aRequest
```

Besser!



Traversal1

Ein `Traversal1 s a` ist ein `Traversal s a`, das immer mindestens über ein `a` iteriert. Dies lässt sich mit der Typklasse `Apply` umsetzen:

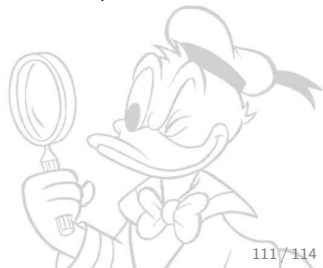
```
type Traversal1 s t a b =  $\forall$  f. Apply f => (a -> f b) -> s -> f t
type Traversal1' s a = Traversal1 s s a a
```

```
class Functor f => Apply f where
  (<.>) :: f (a -> b) -> f a -> f b
```

Zur Erinnerung:

```
type Traversal s t a b =  $\forall$  f. Applicative f => (a -> f b) -> s -> f t
type Traversal' s a = Traversal s s a a
```

```
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
  pure  :: a -> f a
```



Fold1

Analog gibt es auch Fold1:

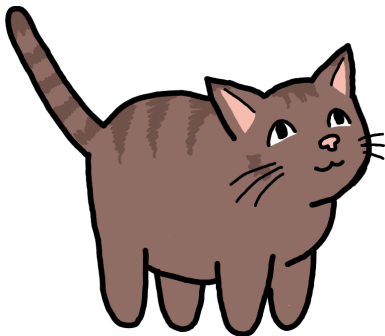
```
type Fold1 s a =  $\forall$  f. (Contravar't f, Apply f) => (a -> f a) -> s -> f s  
                $\cong$   $\forall$  m. Semigroup m => (a -> m) -> s -> m
```

```
type Fold s a =  $\forall$  f. (Contravar't f, App've f) => (a -> f a) -> s -> f s  
                $\cong$   $\forall$  m. Monoid m => (a -> m) -> s -> m
```

Es gilt also:

$$\frac{\text{Apply}}{\text{Applicative}} \approx \frac{\text{Semigroup}}{\text{Monoid}}$$





<http://timbaumann.info/lens>

<https://github.com/timjb/presentations/tree/gh-pages/lens>

Danke an Carina für die Erlaubnis, Socke zu verwenden!