

Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 7: *Records, Lenses & Testing*
Stand: 27. Mai 2016



Record Syntax

Record-Syntax (I)

Angenommen, wir haben einen größeren Produkttypen wie diesen:

```
data Configuration = Configuration
  String    -- User name
  String    -- Local host
  String    -- Remote host
  Bool      -- Is guest?
  Bool      -- Is superuser?
  String    -- Current directory
  String    -- Home directory
  Integer   -- Time connected
deriving (Eq, Show)
```

Record-Syntax (I)

Angenommen, wir haben einen größeren Produkttypen wie diesen:

```
data Configuration = Configuration
  String    -- User name
  String    -- Local host
  String    -- Remote host
  Bool      -- Is guest?
  Bool      -- Is superuser?
  String    -- Current directory
  String    -- Home directory
  Integer   -- Time connected
deriving (Eq, Show)
```

Um auf einzelne Felder zugreifen zu können, müssen wir dann Funktionen wie diese von Hand schreiben:

```
getUserName (Configuration un _ _ _ _ _ _) = un
getLocalHost (Configuration _ lh _ _ _ _ _) = lh
getRemoteHost (Configuration _ _ rh _ _ _ _) = rh
getIsGuest (Configuration _ _ _ ig _ _ _ _) = ig
-- And so on...
```

Record-Syntax (II)

Haskells so genannte *record syntax* erlaubt uns, auch folgendes:

```
data Configuration = Configuration
  { username      :: String
  , localhost    :: String
  , remoteHost   :: String
  , isGuest      :: Bool
  , isSuperuser  :: Bool
  , currentDir   :: String
  , homeDir      :: String
  , timeConnected :: Integer
  }
```

Record-Syntax (II)

Haskells so genannte *record syntax* erlaubt uns, auch folgendes:

```
data Configuration = Configuration
  { username      :: String
  , localhost    :: String
  , remoteHost   :: String
  , isGuest      :: Bool
  , isSuperuser  :: Bool
  , currentDir   :: String
  , homeDir      :: String
  , timeConnected :: Integer
  }
```

Hierbei werden folgende Funktionen *automatisch* generiert:

```
username  :: Configuration -> String
localhost :: Configuration -> String
-- etc.
```

Record Updates

Durch diese Syntax kriegen wir auch eine angenehme Art, (pure) Updates durchzuführen:

```
changeDir :: Configuration -> String -> Configuration
changeDir cfg newDir =
    if directoryExists newDir -- make sure the directory exists
    then cfg { currentDir = newDir }
    else error "Directory does not exist"
```

Record Updates

Durch diese Syntax kriegen wir auch eine angenehme Art, (pure) Updates durchzuführen:

```
changeDir :: Configuration -> String -> Configuration
changeDir cfg newDir =
    if directoryExists newDir -- make sure the directory exists
    then cfg { currentDir = newDir }
    else error "Directory does not exist"
```

Alles hier ist nur *syntaktischer Zucker*. Unter der Haube läuft nichts anders als im ersten Beispiel. Aber es ist angenehm, auf diese Syntax zurück greifen zu können.

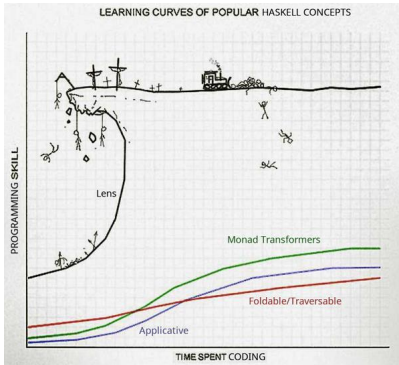
Lenses

Edward Kmett, die Erste

Lens ist eine Bibliothek, geschrieben von Edward Kmett, einem Ikon der Haskell-Community. Lenses sind in vielen größeren Projekten nahezu unersetzlich und werden euch auf jeden Fall noch häufiger begegnen, wenn ihr weiter Haskell macht.

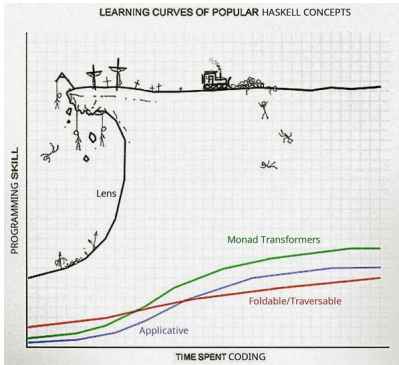


Regel 1: Keine Panik!



Sich über die Komplexität der Lens-Bibliothek lustig zu machen, ist zu einem gewissen *inside joke* der Community geworden. . .

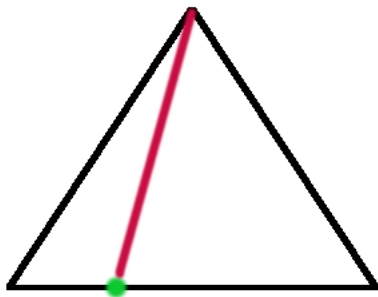
Regel 1: Keine Panik!



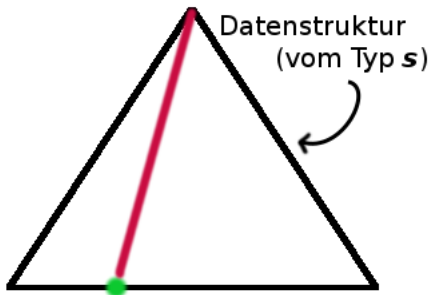
Sich über die Komplexität der Lens-Bibliothek lustig zu machen, ist zu einem gewissen *inside joke* der Community geworden. . .

Das bedeutet aber auch, dass es (größtenteils) nicht so schlimm ist, wie Leute behaupten.

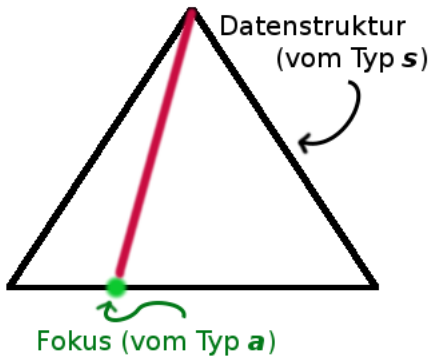
Die Grundidee (I)



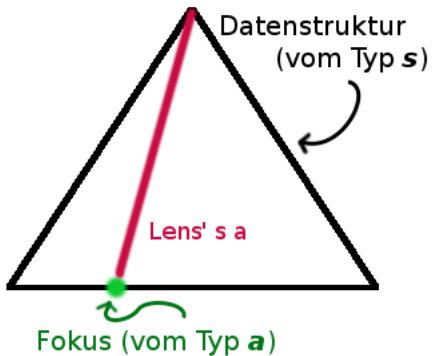
Die Grundidee (I)



Die Grundidee (I)



Die Grundidee (I)



Die Grundidee (II)

Eine Lens gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

Die Grundidee (II)

Eine Lens gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

„Zugriff“ bedeutet hier. . .

- lesen, schreiben, modifizieren. . .

Die Grundidee (II)

Eine Lens gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

„Zugriff“ bedeutet hier. . .

- lesen, schreiben, modifizieren. . .
- aber auch falten, traversieren usw.

Die Grundidee (II)

Eine Lens gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

„Zugriff“ bedeutet hier. . .

- lesen, schreiben, modifizieren. . .
- aber auch falten, traversieren usw.

Lenses sind „first-class values“ (können also umhergereicht, in Datenstrukturen gepackt oder zurückgegeben werden). Die simple Variante hat den Typ `Lens' s a`.

Die Grundidee (II)

Eine Lens gibt Zugriff auf einen bestimmten Teil eines Container oder einer sonstigen Datenstruktur.

„Zugriff“ bedeutet hier. . .

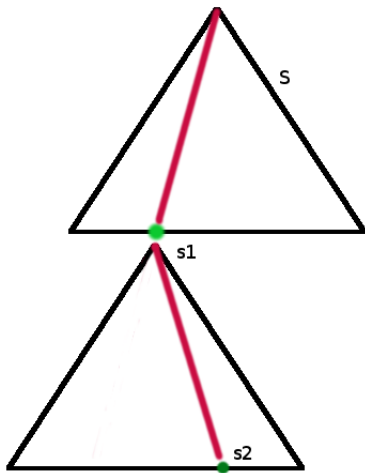
- lesen, schreiben, modifizieren. . .
- aber auch falten, traversieren usw.

Lenses sind „first-class values“ (können also umhergereicht, in Datenstrukturen gepackt oder zurückgegeben werden). Die simple Variante hat den Typ `Lens' s a`.

Beispiele:

```
Lens' DateTime Hour  
Lens' DateTime Minute  
...
```

Composability



Was wir gerne hätten: Lenses, die sich einfach miteinander kombinieren lassen.

```
composeL :: Lens' s s1  
          -> Lens' s1 s2  
          -> Lens' s s2
```

Wir wissen bereits, dass Composability ein großer Vorteil für funktionale Konstrukte ist.

„Puzzle Programming“ macht es uns einfacher, korrekte und elegante Programme zu schreiben.

Warum Lenses?

Aber warum brauchen wir sowas? Geht das nicht alles schon mit Pattern-Matching?

Warum Lenses?

Aber warum brauchen wir sowas? Geht das nicht alles schon mit Pattern-Matching?

```
data Person = Person { name :: String
                      , addr :: Address }
```

```
data Address = Address { road :: String
                       , city :: String
                       , pstc :: Int }
```

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

```
setPostcode :: Int -> Person -> Person
setPostcode pc p = p { addr = addr p { pstc = pc } }
```

Ja, aber es wird schnell ermüdend. Es geht auch eleganter!

Was wäre wenn?

Angenommen, wir hätten jetzt eine Lens für jedes Feld, ...

```
lname :: Person -> String  
laddr :: Person -> Address  
lpstc :: Address -> Int
```

Was wäre wenn?

Angenommen, wir hätten jetzt eine Lens für jedes Feld, ...

```
lname :: Person -> String
laddr :: Person -> Address
lpstc :: Address -> Int
```

...Funktionen, die Lenses zum lesen und schreiben benutzen, ...

```
view :: Lens' s a -> s -> a
set  :: Lens' s a -> a -> s -> s
```

Was wäre wenn?

Angenommen, wir hätten jetzt eine Lens für jedes Feld, ...

```
lname :: Person -> String
laddr :: Person -> Address
lpstc :: Address -> Int
```

...Funktionen, die Lenses zum lesen und schreiben benutzen, ...

```
view :: Lens' s a -> s -> a
set  :: Lens' s a -> a -> s -> s
```

...dann könnten wir (zusammen mit der composeL-Funktion) deutlich eleganteren und effizienteren Code schreiben:

```
setPostcode :: Int -> Person -> Person
setPostcode pc p = set (laddr 'composeL' lpstc) pc p
```

Naïve Implementation (I)

Der erste Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter in einem Datentypen zu bündeln:

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


Naïve Implementation (I)

Der erste Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter in einem Datentypen zu bündeln:

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

Mit etwas Hirnschmalz kriegen wir sogar composeL:

```
composeL :: LensR s s1 -> LensR s1 s2 -> LensR s s2
composeL (L v1 u1) (L v2 u2) = L (\s -> v2 (v1 s))
                                   (\a s -> u1 (u2 a (v1 s)) s)
```

... all das ist aber sehr ineffizient. Falls wir over haben wollen

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

... müssten wir erst getten, dann setzen. Das ist leider sehr ineffizient!

Naïve Implementation (II)

Wir könnten jetzt einfach eine `modify`-Funktion hinzufügen:

```
data LensR s a = L { view    :: s -> a
                    , set    :: a -> s -> s
                    , modify :: (a -> a) -> s -> s }
```

Naïve Implementation (II)

Wir könnten jetzt einfach eine `modify`-Funktion hinzufügen:

```
data LensR s a = L { view    :: s -> a
                    , set    :: a -> s -> s
                    , modify :: (a -> a) -> s -> s }
```

Das Problem dabei ist nur, dass wir sehr schnell zu viele Funktionen haben. Was ist mit effektvollen Veränderungen? Oder mit Veränderungen, die Fehlschlagen können?

Naïve Implementation (II)

Wir könnten jetzt einfach eine `modify`-Funktion hinzufügen:

```
data LensR s a = L { view    :: s -> a
                    , set    :: a -> s -> s
                    , modify :: (a -> a) -> s -> s }
```

Das Problem dabei ist nur, dass wir sehr schnell zu viele Funktionen haben. Was ist mit effektvollen Veränderungen? Oder mit Veränderungen, die Fehlschlagen können?

```
data LensR s a =
  L { view      :: s -> a
    , set       :: a -> s -> s
    , modify    :: (a -> a) -> s -> s
    , modifyIO  :: (a -> IO a) -> s -> IO s
    , modifyMaybe :: (a -> Maybe a) -> s -> Maybe s }
```

Diese Datenstruktur wächst uns schnell über den Kopf und ist dafür nicht mal sehr flexibel.



The End

...OR IS IT?

Abstraction to the rescue!

Das geübte Auge findet zumindest für den letzten Schritt noch einen Ausweg.

Abstraction to the rescue!

Das geübte Auge findet zumindest für den letzten Schritt noch einen Ausweg.

Wir könnten immerhin die Funktionen `modifyMaybe` und `modifyIO` (und alle, die dem gleichen Muster folgen) zusammenfassen:

```
data LensR s a =  
  L { view      :: s -> a  
    , set       :: a -> s -> s  
    , modify    :: (a -> a) -> s -> s  
    , modifyF   :: Functor f => (a -> f a) -> s -> f s }
```

Und das ist eine wirklich gute Idee!

Edward's big insight

Eine *noch* bessere Idee ist es allerdings (und das ist die große Idee hinter `Lens`), auch die Funktionen `view`, `set` und `modify` über die Funktion `modifyF` auszudrücken!

Edward's big insight

Eine *noch* bessere Idee ist es allerdings (und das ist die große Idee hinter Lens), auch die Funktionen `view`, `set` und `modify` über die Funktion `modifyF` auszudrücken!

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

Edward's big insight

Eine *noch* bessere Idee ist es allerdings (und das ist die große Idee hinter Lens), auch die Funktionen `view`, `set` und `modify` über die Funktion `modifyF` auszudrücken!

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

Das ist nur noch ein `type`, also ein Alias von einem Typen auf einen anderen. Mehr brauchen wir nicht.

Lens Isomorphismus (I)

Fun Fact: Lens' und LensR sind *isomorph*!

Lens Isomorphismus (I)

Fun Fact: `Lens'` und `LensR` sind *isomorph*!

Das bedeutet wir können folgende Funktionen schreiben:

```
lensR2Lens :: LensR s a -> Lens' s a  
lens2LensR :: Lens' s a -> LensR s a
```

Lens Isomorphismus (I)

Fun Fact: `Lens'` und `LensR` sind *isomorph*!

Das bedeutet wir können folgende Funktionen schreiben:

```
lensR2Lens :: LensR s a -> Lens' s a  
lens2LensR :: Lens' s a -> LensR s a
```

Hier werden wir eine Richtung zeigen, die andere ist eine nette Übungsaufgabe. ;-)

Lens Isomorphismus (II)

Wir brauchen folgende Funktion:

```
set :: Lens' s a -> a -> s -> s
```

Lens Isomorphismus (II)

Wir brauchen folgende Funktion:

```
set :: Lens' s a -> a -> s -> s
```

In, wenn angewendet, gibt irgendein f s zurück, wir wollen aber eigentlich nur ein s . Also müssen wir uns ein passendes f wählen:

Lens Isomorphismus (II)

Wir brauchen folgende Funktion:

```
set :: Lens' s a -> a -> s -> s
```

In, wenn angewendet, gibt irgendein `f s` zurück, wir wollen aber eigentlich nur ein `s`. Also müssen wir uns ein passendes `f` wählen:

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

```
newtype Identity a = Identity { runIdentity :: a }
```

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

Lens Isomorphismus (II)

Wir brauchen folgende Funktion:

```
set :: Lens' s a -> a -> s -> s
```

In, wenn angewendet, gibt irgendein `f s` zurück, wir wollen aber eigentlich nur ein `s`. Also müssen wir uns ein passendes `f` wählen:

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

```
newtype Identity a = Identity { runIdentity :: a }
```

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

Jetzt nur noch in Code ausformulieren:

```
set :: Lens' s a -> a -> s -> s  
set ln x s = _magic
```

Lens Isomorphismus (II)

Wir brauchen folgende Funktion:

```
set :: Lens' s a -> a -> s -> s
```

In, wenn angewendet, gibt irgendein `f s` zurück, wir wollen aber eigentlich nur ein `s`. Also müssen wir uns ein passendes `f` wählen:

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

```
newtype Identity a = Identity { runIdentity :: a }
```

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

Jetzt nur noch in Code ausformulieren:

```
set :: Lens' s a -> a -> s -> s  
set ln x s = runIdentity (ln set_fld s)  
  where  
    set_fld :: a -> Identity a  
    set_fld k = _magic
```

Lens Isomorphismus (II)

Wir brauchen folgende Funktion:

```
set :: Lens' s a -> a -> s -> s
```

In, wenn angewendet, gibt irgendein `f s` zurück, wir wollen aber eigentlich nur ein `s`. Also müssen wir uns ein passendes `f` wählen:

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

```
newtype Identity a = Identity { runIdentity :: a }
```

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

Jetzt nur noch in Code ausformulieren:

```
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 -- discard current  
                        -- return new value x
```


Lens Isomorphismus (III)

```
view :: Lens' s a -> s -> a
view ln = undefined
```

Lens Isomorphismus (III)

```
view :: Lens' s a -> s -> a  
view ln = undefined
```

Wieder einmal gibt uns die Lens ein `f s` zurück, wir wollen aber einen Wert vom Typ `a`.

Lens Isomorphismus (III)

```
view :: Lens' s a -> s -> a
view ln = undefined
```

Wieder einmal gibt uns die Lens ein `f s` zurück, wir wollen aber einen Wert vom Typ `a`.

Erinnerung:

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

```
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
```

Lens Isomorphismus (III)

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

Die Idee: Wir packen das `a` in das `f`.

`Const` hat hier den Typen `a -> Const a a`. Auf was wird also `f` instanziiert?

Erinnerung:

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

```
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
```

Lens Isomorphismus (IV)

Nach point-free-style umgeschrieben und zusammengesteckt:

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

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

-- one way of the isomorphism
lens2LensR :: Lens' s a -> LensR s a
lens2LensR ln = L { viewR = view ln, setR = set ln }

-- the other way of the isomorphism
lensR2Lens :: LensR s a -> Lens' s a
lensR2Lens = error "ToDo"
```

Selber coden macht schlau!

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen.
Wir wollen aber auch noch sehen, wie wir uns welche *bauen*
können.

Selber coden macht schlau!

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen.
Wir wollen aber auch noch sehen, wie wir uns welche *bauen*
können.

Zur Erinnerung:

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

Selber coden macht schlau!

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

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

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}
```


Selber coden macht schlau!

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

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

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}

-- name :: Functor f => (String -> f String)
--                               -> Person   -> f Person
name :: Lens' Person String
```

.

Selber coden macht schlau!

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

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

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}

-- name :: Functor f => (String -> f String)
--                               -> Person   -> f Person
name :: Lens' Person String
name fn (P n b) = undefined
```

Selber coden macht schlau!

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

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

```
-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}
```

```
-- name :: Functor f => (String -> f String)
--                               -> Person -> f Person
```

```
name :: Lens' Person String
```

```
name fn (P n b) = fmap (\n' -> P n' b) (fn n)
```

Selber coden macht schlau!

Bisher haben wir uns nur angeschaut, wie wir Lenses benutzen. Wir wollen aber auch noch sehen, wie wir uns welche *bauen* können.

Zur Erinnerung:

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

-- field names with underscores so lenses can have the names
data Person = P { _name :: String, _balance :: Integer}

-- name :: Functor f => (String -> f String)
--                               -> Person -> f Person
name :: Lens' Person String
name fn (P n b) = fmap (\n' -> P n' b) (fn n)
```

Mit etwas mentaler Gymnastik merken wir: Die Typen stimmen!

Lens in action

Wir können genau diese Lens jetzt verwenden.

Lens in action

Wir können genau diese Lens jetzt verwenden.

```
ghci> let fred = P { _name = "Fred", _balance = 1000 }
ghci> view name fred
"Fred"
ghci> set name "Bill" fred
P { _name = "Bill", _balance = 1000 }
```

Lens in action

Wir können genau diese Lens jetzt verwenden.

```
ghci> let fred = P { _name = "Fred", _balance = 1000 }
ghci> view name fred
"Fred"
ghci> set name "Bill" fred
P { _name = "Bill", _balance = 1000 }
```

Aber wie funktioniert das genau?

Lens in action

Wir können genau diese Lens jetzt verwenden.

```
ghci> let fred = P { _name = "Fred", _balance = 1000 }
ghci> view name fred
"Fred"
ghci> set name "Bill" fred
P { _name = "Bill", _balance = 1000 }
```

Aber wie funktioniert das genau?

```
ghci> view name P { _name = "Fred", _balance = 1000 }
-- inline view
= getConst (name Const (P { _name = "Fred", _balance = 1000 }))
-- inline name
= getConst (fmap (\n' -> P n' 1000) (Const "Fred"))
-- fmap f (Const x) = Const x
= getConst (Const "Fred")
-- getConst (Const x) = x
= "Fred"
```


Gesetze für Lens

Wenn wir eigene Lenses bauen, müssen wir ein paar Regeln beachten. Dann wie bestimmte Typklassen (Functor, Applicative, Monad), haben auch Lenses ihre eigenen Regeln.

Gesetze für Lens

Wenn wir eigene Lenses bauen, müssen wir ein paar Regeln beachten. Dann wie bestimmte Typklassen (Functor, Applicative, Monad), haben auch Lenses ihre eigenen Regeln.

Glücklicherweise sind sie nicht sehr kompliziert:

- Man bekommt heraus, was man rein tut:

```
view l (set l v s) == v
```

Gesetze für Lens

Wenn wir eigene Lenses bauen, müssen wir ein paar Regeln beachten. Dann wie bestimmte Typklassen (Functor, Applicative, Monad), haben auch Lenses ihre eigenen Regeln.

Glücklicherweise sind sie nicht sehr kompliziert:

- Man bekommt heraus, was man rein tut:

```
view l (set l v s) == v
```

- Zurücklegen was man bekam ändert nichts:

```
set l (view l s) s == s
```

Gesetze für Lens

Wenn wir eigene Lenses bauen, müssen wir ein paar Regeln beachten. Dann wie bestimmte Typklassen (Functor, Applicative, Monad), haben auch Lenses ihre eigenen Regeln.

Glücklicherweise sind sie nicht sehr kompliziert:

- Man bekommt heraus, was man rein tut:

```
view l (set l v s) == v
```

- Zurücklegen was man bekam ändert nichts:

```
set l (view l s) s == s
```

- Zweimal setzen ist das gleiche wie einmal setzen:

```
set l v' (set l v s) == set l v' s
```

Automatische Lenses

```
data Person = P { _name :: String, _balance :: Integer }
```

```
name :: Lens' Person String
```

```
name fn (P n b) = fmap (\n' -> P n' b) (fn n)
```

Jedes Mal alle Lenses von Hand zu schreiben, wenn wir einen Datentypen anlegen wäre ziemlich schreibaufwändig. Aber genau davon wollen wir doch eigentlich weg.

Automatische Lenses

```
data Person = P { _name :: String, _balance :: Integer}

name :: Lens' Person String
name fn (P n b) = fmap (\n' -> P n' b) (fn n)
```

Jedes Mal alle Lenses von Hand zu schreiben, wenn wir einen Datentypen anlegen wäre ziemlich schreibaufwändig. Aber genau davon wollen wir doch eigentlich weg.

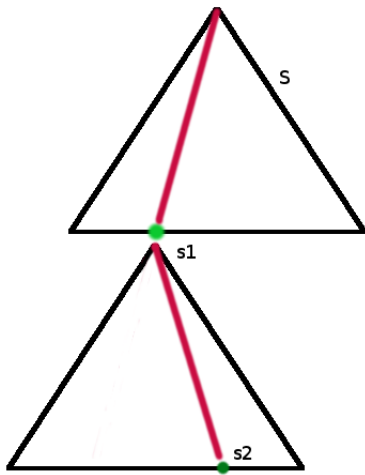
Die Lösung: Statt dem Code oben können wir schreiben:

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

$(makeLenses ''Person)
```

So erstellt TemplateHaskell Lenses für *alle* Felder. Danke, Edward Kmett. :D

Lens composition

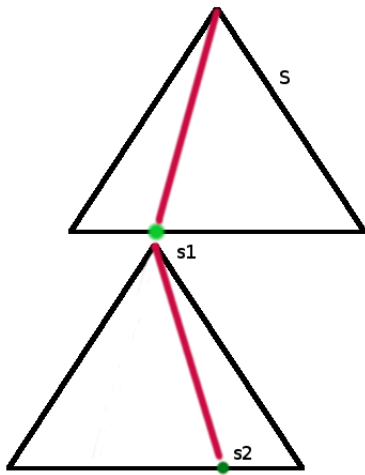


Ihr erinnert euch: Wir wollten gerne eine Funktion `composeL` haben, mit der wir zwei verschiedene Lenses aneinander kleben können.

```
composeL :: Lens' s s1  
         -> Lens' s1 s2  
         -> Lens' s s2
```

Glücklicherweise ist das dieses Mal kein großer Aufwand...

Lens composition



Ihr erinnert euch: Wir wollten gerne eine Funktion `composeL` haben, mit der wir zwei verschiedene Lenses aneinander kleben können.

```
composeL :: Lens' s s1  
-> Lens' s1 s2  
-> Lens' s s2
```

Glücklicherweise ist das dieses Mal kein großer Aufwand...

Lens composition is just function composition!

Modify für Funktoren

Wir wollten auch gerne solche Funktionen in unseren Lenses haben wie `modifyMaybe` oder `modifyIO`.

```
modifyMaybe :: Lens' s a -> (a -> Maybe a) -> s -> Maybe s  
modifyIO     :: Lens' s a -> (a -> IO a) -> s -> IO a
```

Modify für Funktoren

Wir wollten auch gerne solche Funktionen in unseren Lenses haben wie `modifyMaybe` oder `modifyIO`.

```
modifyMaybe :: Lens' s a -> (a -> Maybe a) -> s -> Maybe s
modifyIO     :: Lens' s a -> (a -> IO a) -> s -> IO a
```

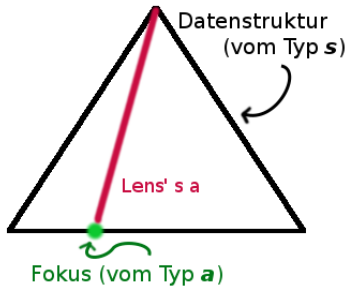
Aber eine `Lens` ist schon so eine Funktion!

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

Und so sehen wir auch, dass es sinnig ist, `f` mit anderen Funktoren als `Const` oder `Identity` zu instanziiieren.

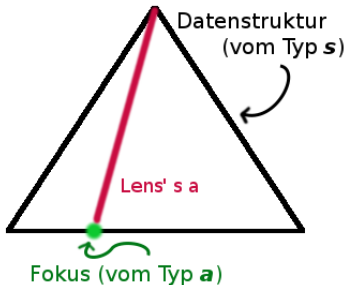
Edward's zweite große Einsicht

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



Edward's zweite große Einsicht

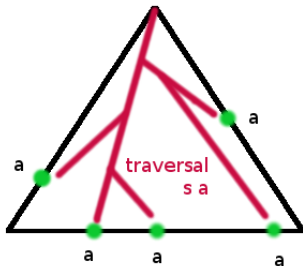
```
type Lens' s a = forall f. Functor f =>
  (a -> f a) -> s -> f s
```



Was passiert, wenn wir statt Functor ein Applicative fordern?

Edward's zweite große Einsicht

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



Wir bekommen eine „multi-lens“, genannt Traversal!

Ein Geständnis:

Ich hab euch die ganze Zeit angelogen!

Ein Geständnis:

Ich hab euch die ganze Zeit angelogen!

```
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)
```

Ein Geständnis:

Ich hab euch die ganze Zeit angelogen!

```
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)
```

over sieht auch nicht besser aus!

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


Ein Geständnis:

Ich hab euch die ganze Zeit angelogen!

```
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)
```

over sieht auch nicht besser aus!

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

„Edward is deeply in thrall to abstractionitis!“ (SPJ)

Heute Lens, morgen die Welt!

Es gibt natürlich noch mehr. Viel mehr.

Heute Lens, morgen die Welt!

Es gibt natürlich noch mehr. Viel mehr.

- Prisms (indexed Lenses)
- Rays (Lenses nach außen)
- Generic Programming
- Interaktionen mit State
- ...

Heute Lens, morgen die Welt!

Es gibt natürlich noch mehr. Viel mehr.

- Prisms (indexed Lenses)
- Rays (Lenses nach außen)
- Generic Programming
- Interaktionen mit State
- ...

Die nach-Hause-Message ist, dass wir mit ein paar cleveren Typsynonymen und Typklassen in Haskell uns ein Framework basteln können, dass enorm viel Ausdruckskraft hat.

That is the power of abstraction!

QuickCheck

(Randomised Property-Based Testing)

Unit Testing: The Basics

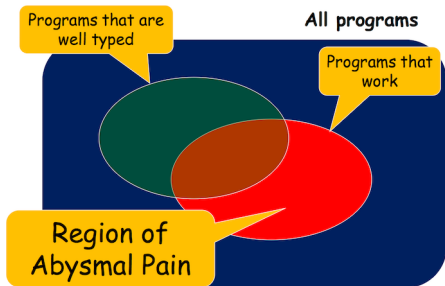
Unit Testing ist eine Vorgehensmethode in der Softwareentwicklung um Fehler vorzubeugen bzw. früh zu finden (und bessere Dokumentation zu haben, oft auch besseres Design etc.).

Ein Entwickler schreibt zunächst *Tests* (d.h. Code, der funktionieren *sollte*) die Funktionalität benutzen, die noch nicht implementiert wurde, und gibt ein erwartetes Ergebnis an.

Bei jedem Build können diese Tests dann automatisch durchgeführt werden, um einen freien Überblick darüber zu erhalten, was funktioniert und was nicht.

Types vs. Tests

In Haskell steht uns natürlich schon das Typsystem zur Seite, wenn es darum geht, korrekten Code zu schreiben. Und an vielen Stellen ist es auch sehr hilfreich, weil es viele Fehler bereits zur *compile time* abfängt, die sonst in der *run time* landen würden.



Allerdings ist das Typsystem von Haskell nicht stark genug, wirklich alle Fehler abzufangen. Oft genug kann man sich leicht dran vorbei schummeln.

Tricks vs. Typsystem

Man schaue sich die Typsignatur von `sort` (aus `Data.List`) an:

```
sort :: Ord a => [a] -> [a]
```

Alles, was wir wissen, ist, dass eine Liste von `as` auf eine Liste von `as` abgebildet wird. Mehr nicht.

Tricks vs. Typsystem

Man schaue sich die Typsignatur von `sort` (aus `Data.List`) an:

```
sort :: Ord a => [a] -> [a]
```

Alles, was wir wissen, ist, dass eine Liste von `as` auf eine Liste von `as` abgebildet wird. Mehr nicht.

Hier sind ein paar Implementationen, die erfolgreich typchecken:

```
sort = const []  
sort = id  
sort = reverse  
sort = \xs -> (permutations xs) !! 19  
sort = take 5
```

Tricks vs. Typsystem

Man schaue sich die Typsignatur von `sort` (aus `Data.List`) an:

```
sort :: Ord a => [a] -> [a]
```

Alles, was wir wissen, ist, dass eine Liste von `as` auf eine Liste von `as` abgebildet wird. Mehr nicht.

Hier sind ein paar Implementationen, die erfolgreich typchecken:

```
sort = const []  
sort = id  
sort = reverse  
sort = \xs -> (permutations xs) !! 19  
sort = take 5
```

... und wenn diese Beispiele durchlaufen, dann auch euer beinahe-korrektes Programm mit einem kritischen off-by-one-error. Aber dafür sind Unit Tests gedacht!

Lesson Learned (I)

Unit Tests sind wichtig. Warum?

Lesson Learned (I)

Unit Tests sind wichtig. Warum?



Das Problem mit Unit Tests

Wenn Unit Tests schreiben also so toll ist, wo ist denn dann die Problematik? Warum brauchen wir die Bibliothek?

Das Problem mit Unit Tests

Wenn Unit Tests schreiben also so toll ist, wo ist denn dann die Problematik? Warum brauchen wir die Bibliothek?

Auch Unit Testing ist keine silberne Kugel gegen Fehler in der eigenen Software!

Das Problem mit Unit Tests

Wenn Unit Tests schreiben also so toll ist, wo ist denn dann die Problematik? Warum brauchen wir die Bibliothek?

Auch Unit Testing ist keine silberne Kugel gegen Fehler in der eigenen Software!

- Es ist ein großer Arbeitsaufwand, der signifikante Mengen an Zeit benötigt

Das Problem mit Unit Tests

Wenn Unit Tests schreiben also so toll ist, wo ist denn dann die Problematik? Warum brauchen wir die Bibliothek?

Auch Unit Testing ist keine silberne Kugel gegen Fehler in der eigenen Software!

- Es ist ein großer Arbeitsaufwand, der signifikante Mengen an Zeit benötigt
- ... es sei denn man macht es halbherzig. Dann verliert das Ganze aber seinen Sinn.

Das Problem mit Unit Tests

Wenn Unit Tests schreiben also so toll ist, wo ist denn dann die Problematik? Warum brauchen wir die Bibliothek?

Auch Unit Testing ist keine silberne Kugel gegen Fehler in der eigenen Software!

- Es ist ein großer Arbeitsaufwand, der signifikante Mengen an Zeit benötigt
- ... es sei denn man macht es halbherzig. Dann verliert das Ganze aber seinen Sinn.
- Oft schreibt die Person, die ein Stück Code entwickelt, auch die Tests für dieses Stück Code. Edge Cases, die eins hier übersieht, übersieht eins oft auch dort.

Das Problem mit Unit Tests

Wenn Unit Tests schreiben also so toll ist, wo ist denn dann die Problematik? Warum brauchen wir die Bibliothek?

Auch Unit Testing ist keine silberne Kugel gegen Fehler in der eigenen Software!

- Es ist ein großer Arbeitsaufwand, der signifikante Mengen an Zeit benötigt
- ... es sei denn man macht es halbherzig. Dann verliert das Ganze aber seinen Sinn.
- Oft schreibt die Person, die ein Stück Code entwickelt, auch die Tests für dieses Stück Code. Edge Cases, die eins hier übersieht, übersieht eins oft auch dort.
- ...

Das Problem als Bild:



Abbildung: All tests passed, 100% coverage

Automatisierte Testgenerierung

Der große Sprung von QuickCheck ist, dass Menschen ihre Tests nicht mehr selbst schreiben, sondern wir diese Aufgabe an eine Bibliothek auslagern.

Automatisierte Testgenerierung

Der große Sprung von QuickCheck ist, dass Menschen ihre Tests nicht mehr selbst schreiben, sondern wir diese Aufgabe an eine Bibliothek auslagern.

Um das zu ermöglichen (und das Testschreiben kürzer und einfacher zu machen), wechseln wir von spezifischen Tests, die einzelne, konkrete Werte überprüfen, zu *property based testing*.

Das bedeutet, dass wir im Code nur noch *Eigenschaften* (Properties) formulieren, die unser Code haben soll.

Beispiele

- reverse doppelt angewendet ist id:
> quickCheck (\xs -> xs == (reverse . reverse) xs)
+++ OK, passed 100 tests.

Beispiele

- reverse doppelt angewendet ist id:
> quickCheck (\xs -> xs == (reverse . reverse) xs)
+++ OK, passed 100 tests.
- Wenn n gerade ist, ist $n + 1$ ungerade (conditional):
> quickCheck(\n -> even(n) ==> odd(n+1))
+++ OK, passed 100 tests.

Beispiele

- reverse doppelt angewendet ist id:
> quickCheck (\xs -> xs == (reverse . reverse) xs)
+++ OK, passed 100 tests.
- Wenn n gerade ist, ist $n + 1$ ungerade (conditional):
> quickCheck(\n -> even(n) ==> odd(n+1))
+++ OK, passed 100 tests.
- Früher wurde geglaubt, dass wenn n prim ist, auch die n -te Mersenne-Zahl $M_n = 2^n - 1$ prim ist.
(Die Vermutung hält für M_2 , M_3 , M_5 und M_7)
> quickCheck(\n -> isPrime n ==> isPrime(2^n - 1))
*** Failed! Falsifiable (after 14 tests):
11

Properties for sorting

-- Sorting twice changes nothing

```
prop_idempotency :: Ord a => [a] -> Bool
```

```
prop_idempotency xs = qsort xs == qsort (qsort xs)
```

-- Sorting doesn't change the length

```
prop_len :: Ord a => [a] -> Bool
```

```
prop_len xs = length xs == length (qsort xs)
```

-- Sorted result is a permutation of input

```
prop_perm :: Ord a => [a] -> Bool
```

```
prop_perm xs = (qsort xs) `elem` (permutations xs)
```

-- Sorting produces sorted list

```
prop_sort :: Ord a => [a] -> Bool
```

```
prop_sort = isSorted . qsort
```

```
  where
```

```
    isSorted :: Ord a => [a] -> Bool
```

```
    isSorted [] = True
```

```
    isSorted [x] = True
```

```
    isSorted (x:y:zs) = (x <= y) && isSorted (y:zs)
```

Die Grenzen der Idee

Allerdings kann natürlich auch QuickCheck reingelegt werden. Und 100 Test Cases sind bei weitem nicht immer genug:

Die Grenzen der Idee

Allerdings kann natürlich auch QuickCheck reingelegt werden. Und 100 Test Cases sind bei weitem nicht immer genug:

```
> let notProduct n p q = n /= p * q
> quickCheck (notProduct 10)
+++ OK, passed 100 tests.
> quickCheck (notProduct 10)
+++ OK, passed 100 tests.
> quickCheck (notProduct 10)
+++ OK, passed 100 tests.
> quickCheck (notProduct 10)
*** Failed! Falsifiable (after 9 tests):
5
2
```

Die Grenzen der Idee

Allerdings kann natürlich auch QuickCheck reingelegt werden. Und 100 Test Cases sind bei weitem nicht immer genug:

```
> let notProduct n p q = n /= p * q
> quickCheck (notProduct 10)
+++ OK, passed 100 tests.
> quickCheck (notProduct 10)
+++ OK, passed 100 tests.
> quickCheck (notProduct 10)
+++ OK, passed 100 tests.
> quickCheck (notProduct 10)
*** Failed! Falsifiable (after 9 tests):
5
2
```

Es kommt leider immer noch auf die Person vor der Tastatur an.

Lesson Learned (II)

Unit Tests *zu automatisieren* ist wichtig. Warum?

Lesson Learned (II)

Unit Tests *zu automatisieren* ist wichtig. Warum?



Vorschau: Was machen wir nächste Woche?

- Parallelism
- Concurrency

Fragen?



xkcd by Randall Munroe, CC-BY-NC
<https://xkcd.com/1270/>