Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Outline I

Übersicht für Heute:

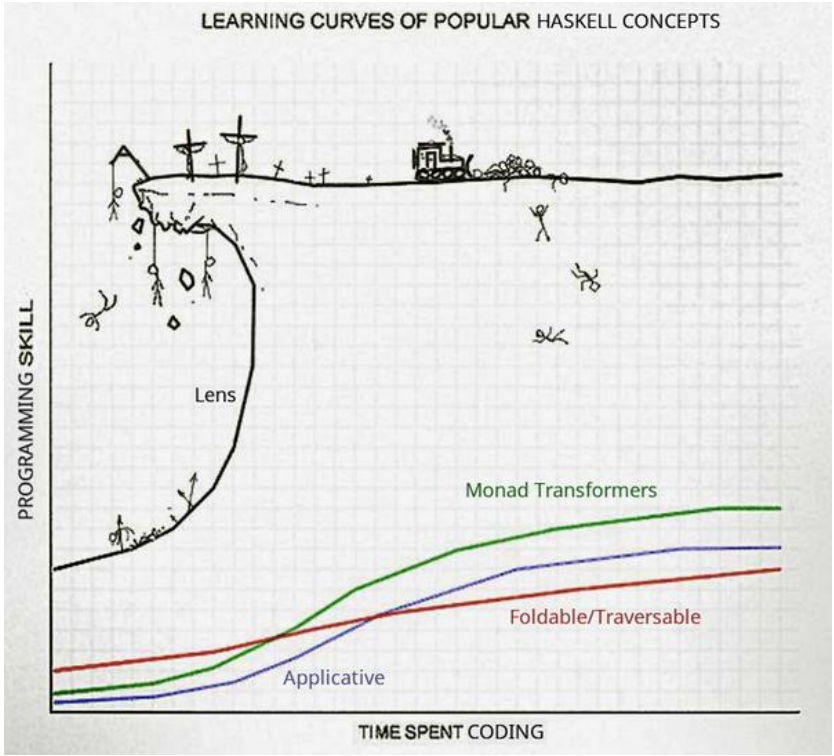
1. Lens
   - Grundidee
   - Motivation & Anwendungen
   - Fortgeschrittenes
   - Traversals
2. QuickCheck
   - Motivation
   - Idee und Anwendung

Lenses

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

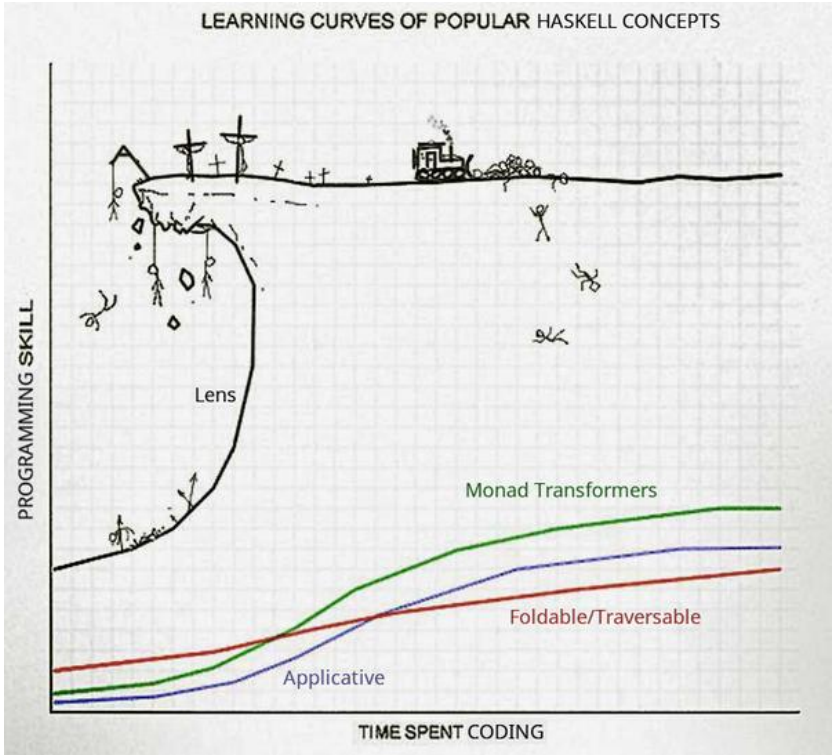
Regel 1: Keine Panik!

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:

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

Die Grundidee:

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

Zugriff bedeutet hier...

- lesen, schreiben, modifizieren...

Die Grundidee:

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:

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:

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

Zugriff bedeutet hier...

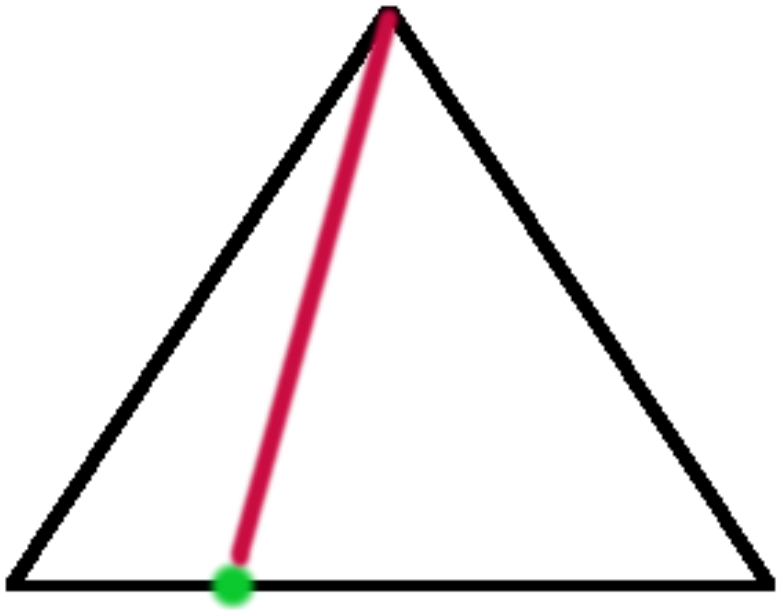
- lesen, schreiben, modifizieren...
- aber auch falten, traversieren usw.

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

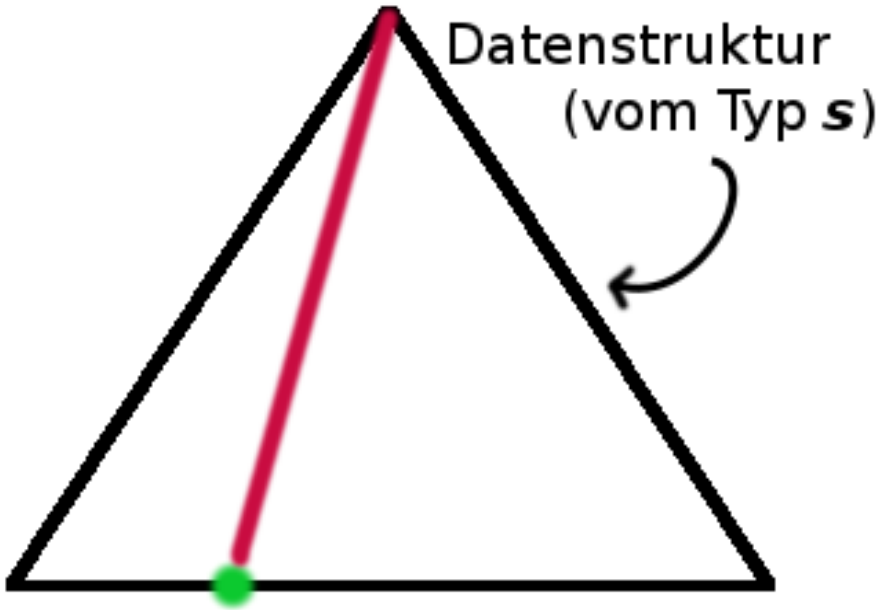
Beispiele:

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

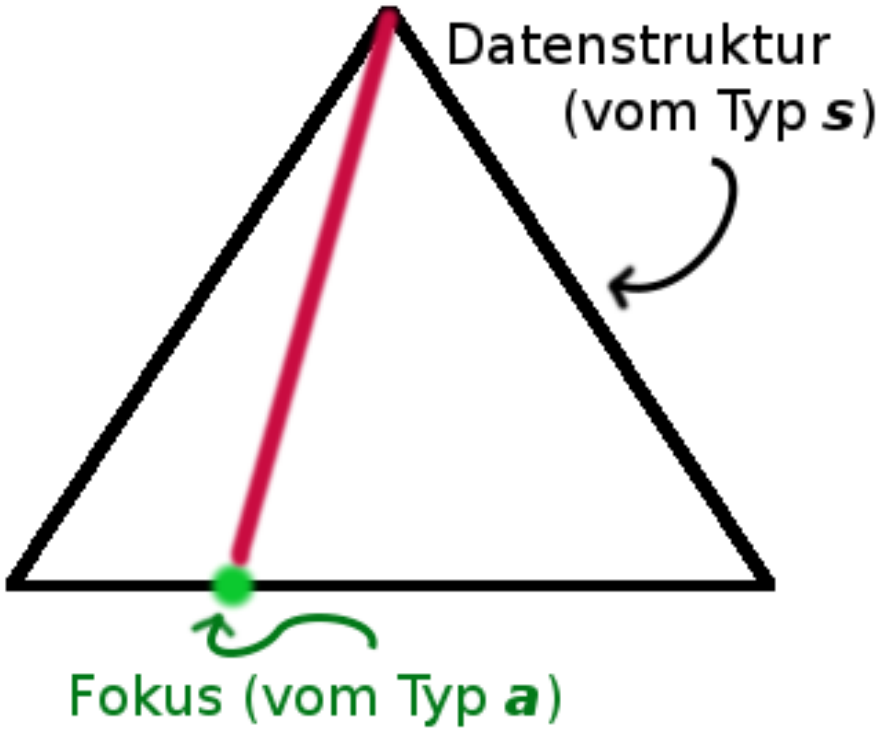
Die Grundidee:



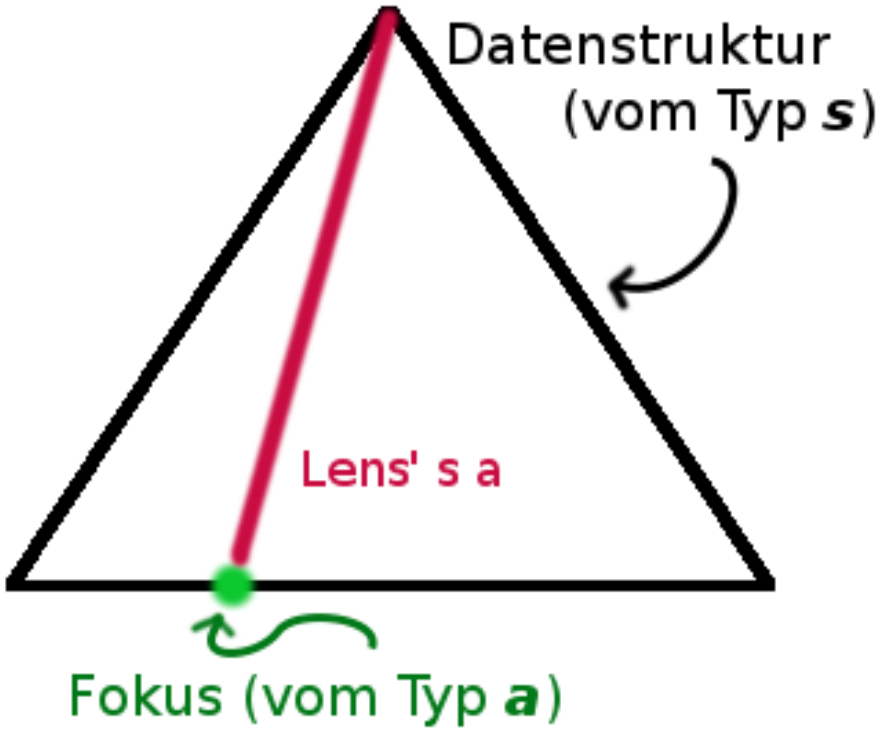
Die Grundidee:

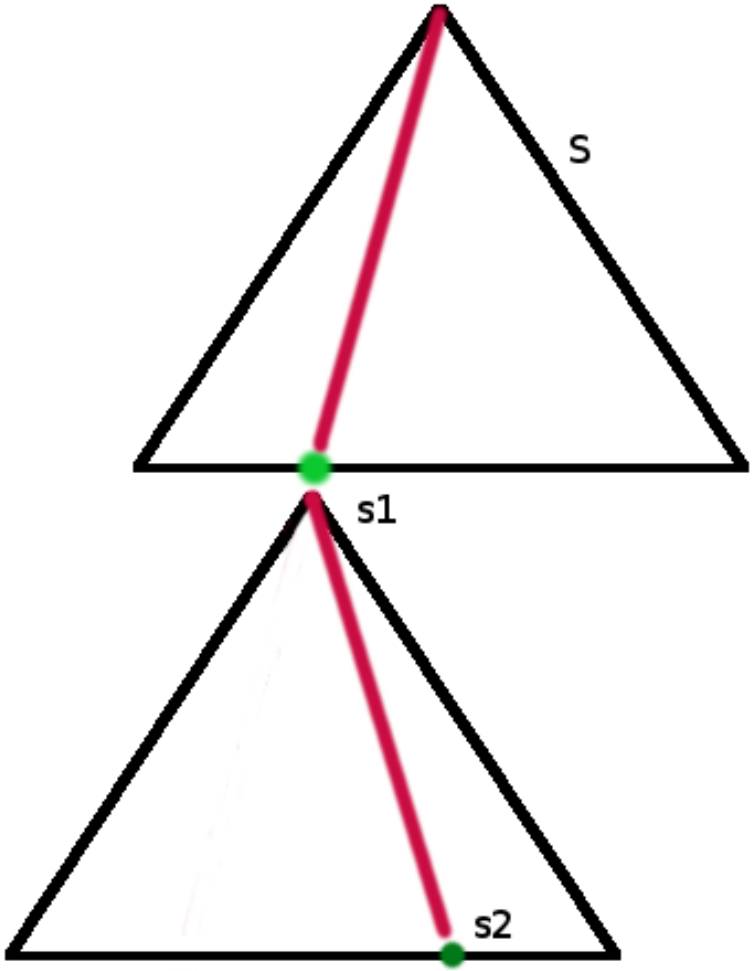


Die Grundidee:



Die Grundidee:





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 funktioniale Konstrukte ist.

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

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

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

```haskell
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, das geht. Aber es wird schnell ermüdend. Wer beim Erklären zu oft „blah-blah“ sagt, sollte sich um elegantere Wege oder Automatisierung bemühen.

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

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

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

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

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

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

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

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

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

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

Der naive Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter zu bündeln:

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

Der naive Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter 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)
```

Der naive Ansatz für so eine Struktur wäre wahrscheinlich, einfach feste Getter und Setter 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 setten. *Nicht cool*.

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 }
```

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?

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
      , modidyIO     :: (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?

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

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
      , modidyF :: Functor f => (a -> f a) -> s -> f s }
```

Und das ist eine wirklich gute Idee.

Edward's big insight:

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' = 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' = 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.

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

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
```

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 vortanzen, die andere Richtung ist Übungsaufgabe. ;-)

```
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 }
```

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

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

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' = 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}
```

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' = 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
.
```

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' = 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) = ... ähm ...
```

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' = 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)
```

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' = 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` laws:

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

`Lens` laws:

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
```

`Lens` laws:

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
```

`Lens` laws:

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
```

Wir können genau diese `Lens` jetzt verwenden.

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 }
```

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?

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' 100) (Const "Fred"))
      -- fmap f (Const x) = Const (f x)
= getConst (Const "Fred")
      -- getConst (Const x) = x
= "Fred"
```

Lenses generieren lassen:

```
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.

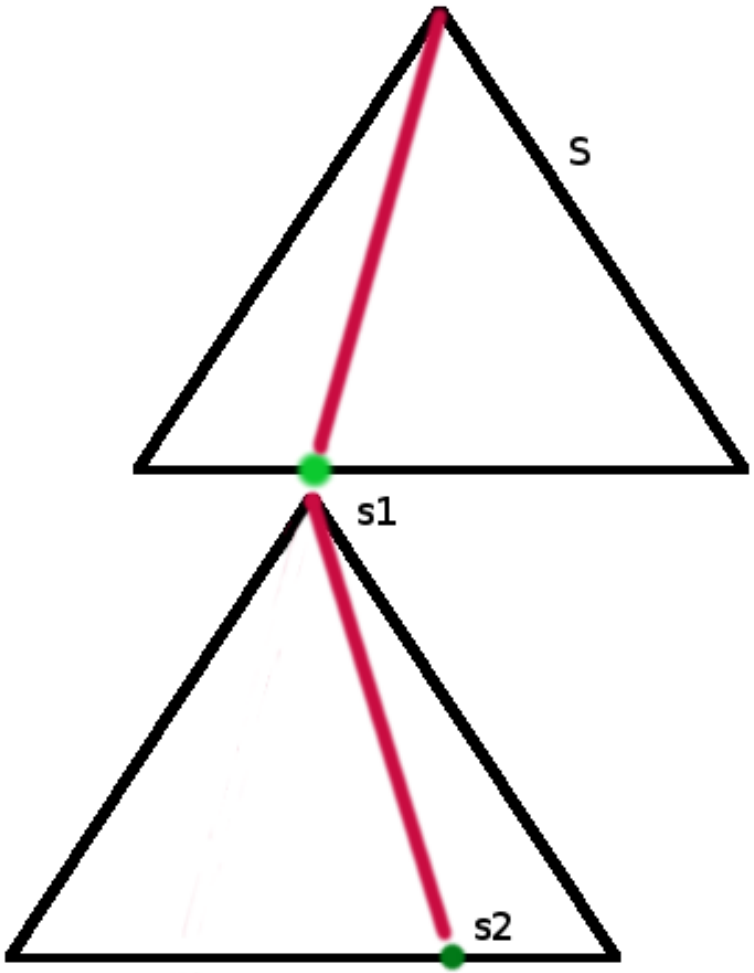
Lenses generieren lassen:

```
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

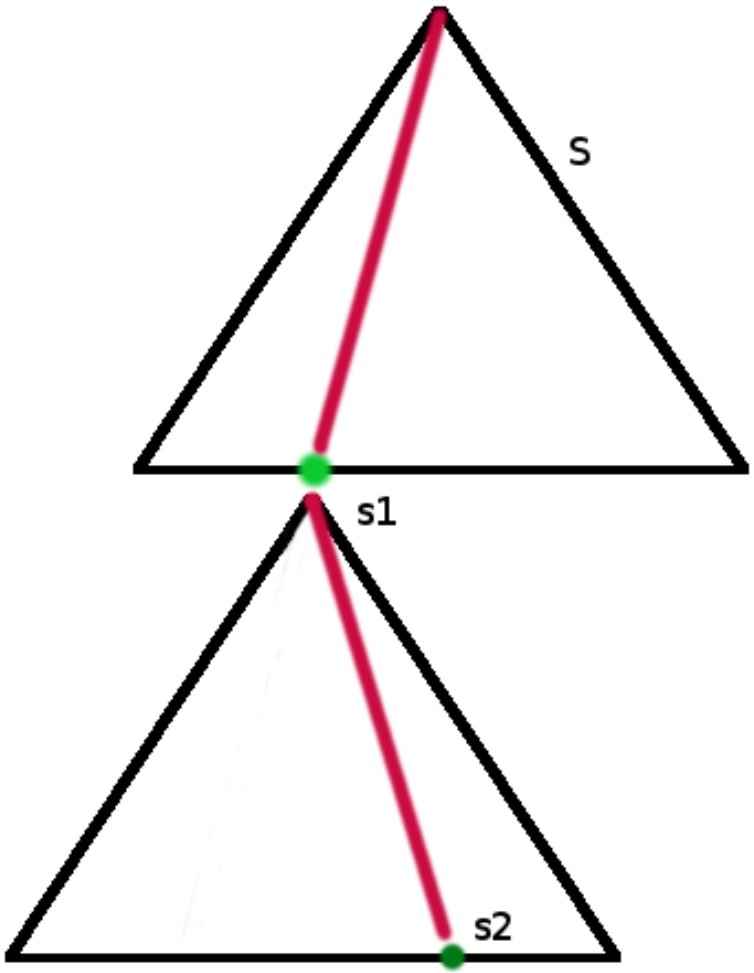


Lens composition:

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!

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
```

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
```

Total einfach! Eine `Lens` *ist* schon so eine Funktion!

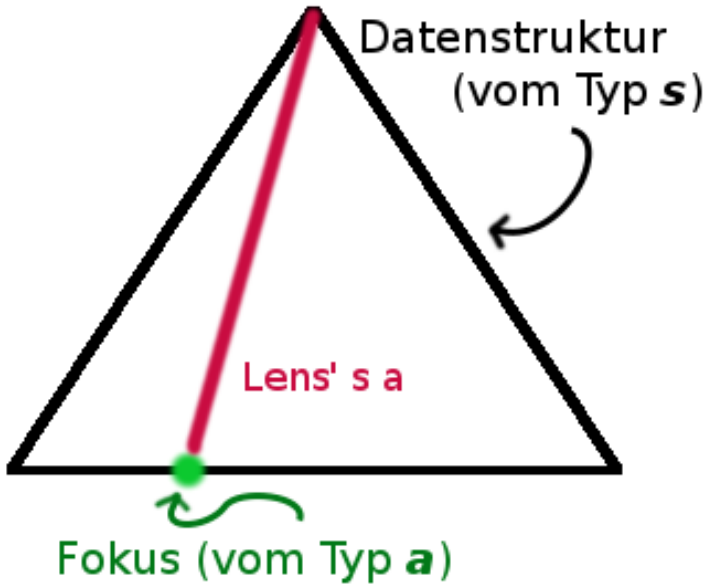
```
type Lens' = forall f. Functor f =>
                  (a -> f a) -> s -> f s
```

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

Edward's zweite große Einsicht:

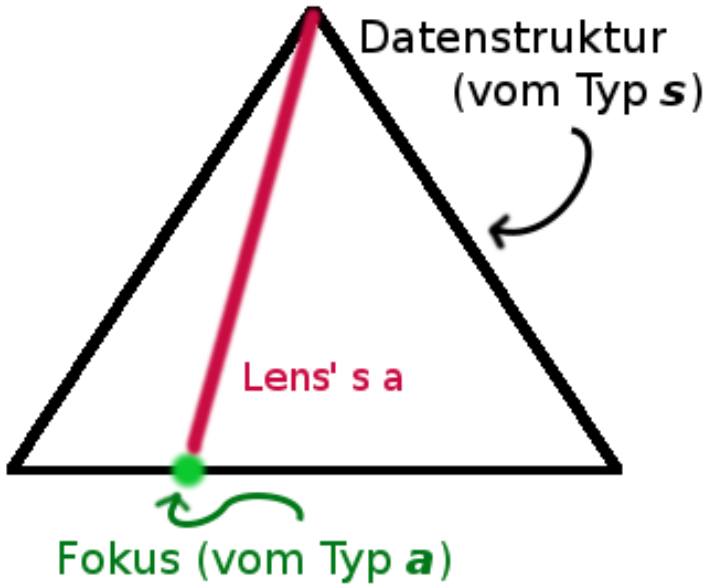
Edward's zweite große Einsicht:

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



Edward's zweite große Einsicht:

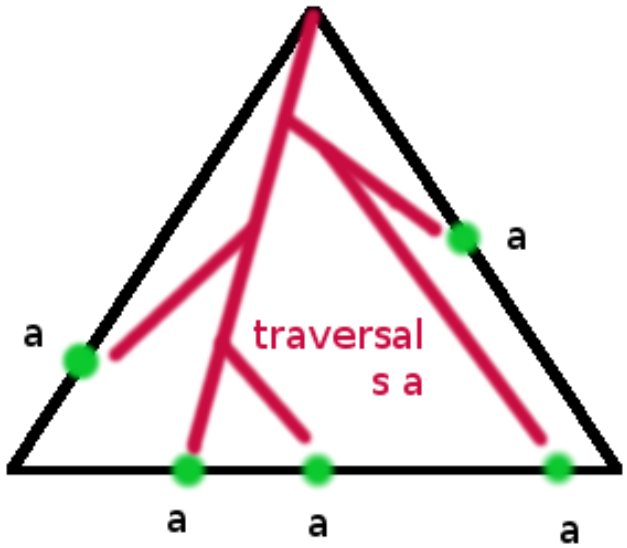
```
type Lens' = forall f. Functor f =>
                       (a -> f a) -> s -> f s
```



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
```



Antwort: Wir bekommen eine „`multi-lens`“ , genannt `Traversal`, mit mehreren Fokussen!

Ein Geständnis:. . .

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)

Es gibt natürlich noch mehr. Viel mehr.

Es gibt natürlich noch mehr. Viel mehr.

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

Es gibt natürlich noch mehr. Viel mehr.

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

Eventuell sehen wir davon noch ein paar Sachen in späteren Vorlesungen.

Die nach-Hause-Message ist, dass wir mit ein paar cleveren Typsysonymen und Typklassen in `Haskell` uns ein Framework basteln können, dass enorm viel Ausdruckskraft hat. *The power of abstraction!*

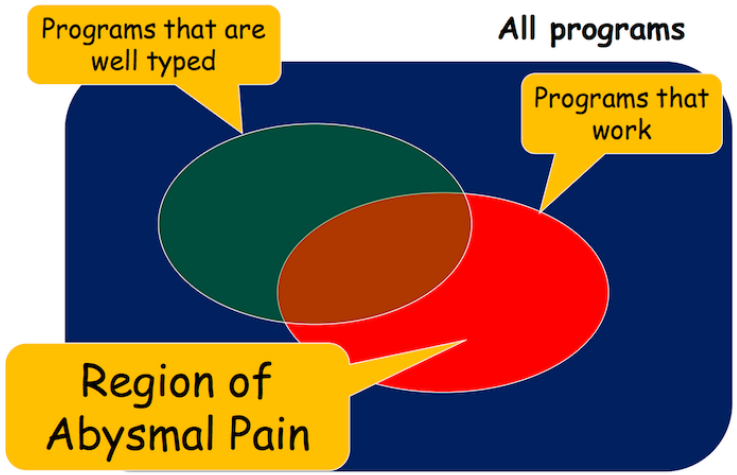
QuickCheck
(Randomised Property-Based Testing)

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* (i.e. Code, der funktionieren *sollte*) der Funktionalität benutzt, die noch nicht implementiert wurde, und gibt ein erwartetes Ergebnis an. Bei jedem Build können diese Tests dann automatisch durchgeführt werden, um einen 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.

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.

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

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 = []
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: Unit Tests sind wichtig. Warum?

Lesson learned: Unit Tests sind wichtig. Warum?
Weil wir dumme, fehlbare, stolze Menschen sind!

Das Problem mit Unit Tests:

Wenn Unit tests also so toll sind, wo ist denn dann bitteschön die Problematik? Wir schreiben die Tests, fertig aus. Wer braucht diese dumme Bibliothek?

Das Problem mit Unit Tests:

Wenn Unit tests also so toll sind, wo ist denn dann bitteschön die Problematik? Wir schreiben die Tests, fertig aus. Wer braucht diese dumme Bibliothek?

Auch Unit Testing ist keine Silberkugel gegen Fehler in der eigenen Software:

Das Problem mit Unit Tests:

Wenn Unit tests also so toll sind, wo ist denn dann bitteschön die Problematik? Wir schreiben die Tests, fertig aus. Wer braucht diese dumme Bibliothek?

Auch Unit Testing ist keine Silberkugel gegen Fehler in der eigenen Software:

- Es ist eine Menge Arbeit, die signifikante Mengen an Zeit benötigt

Das Problem mit Unit Tests:

Wenn Unit tests also so toll sind, wo ist denn dann bitteschön die Problematik? Wir schreiben die Tests, fertig aus. Wer braucht diese dumme Bibliothek?

Auch Unit Testing ist keine Silberkugel gegen Fehler in der eigenen Software:

- Es ist eine Menge Arbeit, die signifikante Mengen an Zeit benötigt
- . . . es sei denn man macht es halbherzig. Dann verliert es aber seinen ganzen Sinn.

Das Problem mit Unit Tests:

Wenn Unit tests also so toll sind, wo ist denn dann bitteschön die Problematik? Wir schreiben die Tests, fertig aus. Wer braucht diese dumme Bibliothek?

Auch Unit Testing ist keine Silberkugel gegen Fehler in der eigenen Software:

- Es ist eine Menge Arbeit, die signifikante Mengen an Zeit benötigt
- . . . es sei denn man macht es halbherzig. Dann verliert es aber seinen ganzen 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 also so toll sind, wo ist denn dann bitteschön die Problematik? Wir schreiben die Tests, fertig aus. Wer braucht diese dumme Bibliothek?

Auch Unit Testing ist keine Silberkugel gegen Fehler in der eigenen Software:

- Es ist eine Menge Arbeit, die signifikante Mengen an Zeit benötigt
- . . . es sei denn man macht es halbherzig. Dann verliert es aber seinen ganzen 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.
- . . .

Die Idee von QuickCheck: Automatisierte Testgenerierung

Der große Sprung von QuickCheck ist, dass Menschen ihre Tests

nicht mehr selbst schreiben, sondern wir das einer Bibliothek überlassen.

Die Idee von QuickCheck: Automatisierte Testgenerierung
Der große Sprung von QuickCheck ist, dass Menschen ihre Tests

nicht mehr selbst schreiben, sondern wir das einer Bibliothek überlassen.

Um das zu ermöglichen (und das Testschreiben allen kürzer und einfacher zu machen), wechseln wir von spezifischen Tests, die einzelne Werte überprüfen, zu *property based testing*. Das bedeutet, dass wir im Code nur noch Eigenschaften formulieren, die unser Code haben soll.

Ein paar Beispiele (zuerst `import Test.QuickCheck`):

- `reverse` doppelt angewendet hebt sich weg:

```
> quickCheck (\xs -> xs == (reverse . reverse) xs)
+++ OK, passed 100 tests.
```

Ein paar Beispiele (zuerst `import Test.QuickCheck`):

- `reverse` doppelt angewendet hebt sich weg:

```
> 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.
```

Ein paar Beispiele (zuerst `import Test.QuickCheck`):

- `reverse` doppelt angewendet hebt sich weg:

```
> 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 $\mathrm{M}_n = 2^n - 1$ 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
```

```haskell
-- 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)
```

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

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
```

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: Unit Tests zu automatisieren wichtig. Warum?

Lesson learned: Unit Tests zu automatisieren wichtig. Warum?
Weil wir dumme, fehlbare, stolze Menschen sind!