Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 5: *Foldable/Traversable und GUI*
Stand: 14. Mai 2016

Wiederholung am praktischen Beispiel:
Lineare Algebra

Was ist lineare Algebra?

Euch allen sollte im Studium bereits lineare Algebra über den Weg gelaufen sein.

Was ist lineare Algebra?

Euch allen sollte im Studium bereits lineare Algebra über den Weg gelaufen sein.

Konkret geht es um das Rechnen mit

- Matrizen
- Vektoren

und die damit einhergehenden Rechenregeln und Gesetze.

Recap

Konkret ist ein Vektorraum über einem Körper K aufgespannt. Ein Körper setzt eine Addition $(+)$, Multiplikation $(\cdot)$ und ein Distributivgesetz vorraus.

Recap

Konkret ist ein Vektorraum über einem Körper K aufgespannt. Ein Körper setzt eine Addition ($+$), Multiplikation ($\cdot$) und ein Distributivgesetz vorraus.
Außerdem brauchen wir zwei abgeschlossene Operationen:

$$\oplus : V \times V \to V$$

$$\odot : K \times V \to V$$

Recap

Konkret ist ein Vektorraum über einem Körper K aufgespannt. Ein Körper setzt eine Addition ($+$), Multiplikation ($\cdot$) und ein Distributivgesetz vorraus.
Außerdem brauchen wir zwei abgeschlossene Operationen:

$$\oplus : V \times V \to V$$

$$\odot : K \times V \to V$$

Da wir hier in `Haskell` reden: Ein Körper entspricht der Typklasse `Num` mit den „gewöhnlichen“ Regeln.

Recap

Konkret ist ein Vektorraum über einem Körper K aufgespannt. Ein Körper setzt eine Addition ($+$), Multiplikation ($\cdot$) und ein Distributivgesetz vorraus.
Außerdem brauchen wir zwei abgeschlossene Operationen:

$$\oplus : V \times V \to V$$

$$\odot : K \times V \to V$$

Da wir hier in `Haskell` reden: Ein Körper entspricht der Typklasse `Num` mit den „gewöhnlichen“ Regeln.

Im Folgenden betrachten wir nur einen 3-dimensionalen Vektorraum und abstrahieren anschließend über die Dimensionszahl.

Was brauchen wir konkret?

Beginnen wir mit der Definition eines 3D-Vektors:

```
data V3 a = V3 a a a
          deriving (Show, Eq)
```

Was brauchen wir konkret?

Beginnen wir mit der Definition eines 3D-Vektors:

```
data V3 a = V3 a a a
            deriving (Show, Eq)
```

An Funktionen brauchen wir diese:

```
vadd  :: Num a => V3 a -> V3 a -> V3 a
vmul  :: Num a =>    a -> V3 a -> V3 a
vscal :: Num a => V3 a -> V3 a ->    a
```

Instanzen

Bevor wir mit den Details der Implementation anfangen, schauen wir erstmal, was für Instanzen wir schreiben können.

Instanzen

Bevor wir mit den Details der Implementation anfangen, schauen wir erstmal, was für Instanzen wir schreiben können.

Offensichtlich ist `V3` ein Functor:

```
instance Functor V3 where
  fmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

Instanzen

Bevor wir mit den Details der Implementation anfangen, schauen wir erstmal, was für Instanzen wir schreiben können.

Offensichtlich ist `V3` ein Functor:

```
instance Functor V3 where
  fmap f (V3 x y z) = V3 (f x) (f y) (f z)
```

In den Übungen haben wir die `V3` `Applicative`-Instanz gesehen:

```
instance Applicative V3 where
  pure a = V3 a a a
  (V3 f g h) <*> (V3 x y z) = V3 (f x) (g y) (h z)
```

Implementation

Versuchen wir hiermit nun die Funktionen zu schreiben:

```
vadd :: Num a => V3 a -> V3 a -> V3 a
```

Implementation

Versuchen wir hiermit nun die Funktionen zu schreiben:

```
vadd :: Num a => V3 a -> V3 a -> V3 a

vadd (V3 a b c) (V3 x y z) = V3 (a+x) (b+y) (c+z)
```

Implementation

Versuchen wir hiermit nun die Funktionen zu schreiben:

```
vadd :: Num a => V3 a -> V3 a -> V3 a
```

```
vadd (V3 a b c) (V3 x y z) = V3 (a+x) (b+y) (c+z)
```

Dies ist die naive Variante. Was wir eigentlich tun wollen ist `(+)` auf alle Elemente anwenden. Dafür haben wir uns ja die Mühe mit dem `Applicative` gemacht:

Implementation

Versuchen wir hiermit nun die Funktionen zu schreiben:

```
vadd :: Num a => V3 a -> V3 a -> V3 a

vadd (V3 a b c) (V3 x y z) = V3 (a+x) (b+y) (c+z)
```

Dies ist die naive Variante. Was wir eigentlich tun wollen ist `(+)` auf alle Elemente anwenden. Dafür haben wir uns ja die Mühe mit dem `Applicative` gemacht:

```
vadd x y = (+) <$> x <*> y
```

Implementation

Was ist denn jetzt der Typ von

```
vadd x y = (+) <$> x <*> y
```

Implementation

Was ist denn jetzt der Typ von

```
vadd x y = (+) <$> x <*> y
```

```
vadd :: (Applicative f, Num a) => f a -> f a -> f a
```

Implementation

Was ist denn jetzt der Typ von

```
vadd x y = (+) <$> x <*> y
```

```
vadd :: (Applicative f, Num a) => f a -> f a -> f a
```

Das gilt für alle Vektoren und nicht nur für unseren `V3`, wenn wir die `Applicative`-Instanz schreiben.

Implementation

Was ist denn jetzt der Typ von

```
vadd x y = (+) <$> x <*> y
```

```
vadd :: (Applicative f, Num a) => f a -> f a -> f a
```

Das gilt für alle Vektoren und nicht nur für unseren `V3`, wenn wir die `Applicative`-Instanz schreiben.

Die Anwendung einer binären Funktion auf ein `Applicative` ist sogar so häufig, dass hierfür in `Control.Applicative` die Funktion `liftA2` implementiert ist.

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
```

Implementation

Wenn das mit `liftA2` für jede Funktion klappt, gilt das dann auch für `(*)`, `(/)`, `(-)`?

Implementation

Wenn das mit `liftA2` für jede Funktion klappt, gilt das dann auch für `(*)`, `(/)`, `(-)`? JA!

Implementation

Wenn das mit `liftA2` für jede Funktion klappt, gilt das dann auch für `(*), (/), (-)`? JA!
Wir können sogar alles in `Num` implementieren, indem wir alles einfach „durchreichen“:

Implementation

Wenn das mit `liftA2` für jede Funktion klappt, gilt das dann auch für `(*)`, `(/)`, `(-)`? JA!
Wir können sogar alles in `Num` implementieren, indem wir alles einfach „durchreichen“:

```
instance Num a => Num (V3 a) where
  (+) = liftA2 (+)
  (*) = liftA2 (*)
  (-) = liftA2 (-)
  (/) = liftA2 (/)
  fromInteger = pure . fromInteger
  negate = fmap negate
  signum = fmap signum
  abs = fmap abs
```

Wenn wir ein `Num a` als Inhalt haben, können wir auf den Vektoren auch alles machen, was wir auf den `Num`-Sachen machen können.

Implementation

Müssen wir dann `vadd` noch schreiben?

Implementation

Müssen wir dann `vadd` noch schreiben? Stellt sich raus: Nein. `vadd` können wir erhaten durch:

```
-- casually adding vectors
let sum = v + w
```

Implementation

Müssen wir dann `vadd` noch schreiben? Stellt sich raus: Nein. `vadd` können wir erhaten durch:

```
-- casually adding vectors
let sum = v + w
```

Dies bedeutet aber auch, dass

```
let prod = v * w
```

punktweise multipliziert.

Implementation

Müssen wir dann `vadd` noch schreiben? Stellt sich raus: Nein. `vadd` können wir erhaten durch:

```
-- casually adding vectors
let sum = v + w
```

Dies bedeutet aber auch, dass

```
let prod = v * w
```

punktweise multipliziert. Punktweise bedeutet hierbei, dass wir jede Komponente einzeln betrachten:

```
(V3 a b c) * (V3 x y z) = V3 (a*x) (b*y) (c*z)
```

Implementation

Müssen wir dann `vadd` noch schreiben? Stellt sich raus: Nein. `vadd` können wir erhaten durch:

```
-- casually adding vectors
let sum = v + w
```

Dies bedeutet aber auch, dass

```
let prod = v * w
```

punktweise multipliziert. Punktweise bedeutet hierbei, dass wir jede Komponente einzeln betrachten:

```
(V3 a b c) * (V3 x y z) = V3 (a*x) (b*y) (c*z)
```

Dieses sollte man *nicht* mit der üblichen Skalarmultiplikation verwechseln.

Implementation

Wie machen wir denn jetzt die Skalarmultiplikation?

```
vmul :: Num a => a -> V3 a -> V3 a
```

Implementation

Wie machen wir denn jetzt die Skalarmultiplikation?

```
vmul :: Num a => a -> V3 a -> V3 a
```

Eigentlich wollen wir ja mit a in jeder Komponente multiplizieren.
Oder anders gesagt, (a*) auf jede Komponente anwenden

Implementation

Wie machen wir denn jetzt die Skalarmultiplikation?

```
vmul :: Num a => a -> V3 a -> V3 a
```

Eigentlich wollen wir ja mit `a` in jeder Komponente multiplizieren.
Oder anders gesagt, `(a*)` auf jede Komponente anwenden

```
vmul s v = (s*) <$> v
```

Implementation

Wie machen wir denn jetzt die Skalarmultiplikation?

```
vmul :: Num a => a -> V3 a -> V3 a
```

Eigentlich wollen wir ja mit `a` in jeder Komponente multiplizieren.
Oder anders gesagt, `(a*)` auf jede Komponente anwenden

```
vmul s v = (s*) <$> v
```

Somit ändert sich die Signatur in

```
vmul :: (Functor f, Num a) => a -> f a -> f a
```

und `vmul` gilt wieder für alle Vektoren - egal welcher Dimension.

Was fehlt?

Wir haben nun noch die Funktion

```
vscal :: Num a => V3 a -> V3 a -> a
```

Was fehlt?

Wir haben nun noch die Funktion

```
vscal :: Num a => V3 a -> V3 a -> a
```

Wiederholung: Das Standardskalarprodukt im `V3` bedeutet eine punktweise Multiplikation und anschließende Aufsummierung:

Was fehlt?

Wir haben nun noch die Funktion

```
vscal :: Num a => V3 a -> V3 a -> a
```

Wiederholung: Das Standardskalarprodukt im `V3` bedeutet eine punktweise Multiplikation und anschließende Aufsummierung:

```
vscal v w = sum $ v * w
    where
      sum (V3 x y z) = x + y + z
```

Was fehlt?

Wir haben nun noch die Funktion

```
vscal :: Num a => V3 a -> V3 a -> a
```

Wiederholung: Das Standardskalarprodukt im `V3` bedeutet eine punktweise Multiplikation und anschließende Aufsummierung:

```
vscal v w = sum $ v * w
    where
      sum (V3 x y z) = x + y + z
```

Das ist schon recht nett. Aber man muss so häufig etwas aufsummieren oder anders zusammenfassen. . .

Was fehlt?

Wir haben nun noch die Funktion

```
vscal :: Num a => V3 a -> V3 a -> a
```

Wiederholung: Das Standardskalarprodukt im `V3` bedeutet eine punktweise Multiplikation und anschließende Aufsummierung:

```
vscal v w = sum $ v * w
    where
      sum (V3 x y z) = x + y + z
```

Das ist schon recht nett. Aber man muss so häufig etwas aufsummieren oder anders zusammenfassen. . .

Außerdem kann man auch

```
foldr (+) [x,y,z]
```

machen, wenn man es in eine Liste packt.

Was ist nun dieses `fold`?

Foldable / Traversable

Was ist eine Faltung?

Was ist eine Faltung?

Ein „Faltung“ im mathematischen Sinne ist die Reduktion eines „Dings“ mittels einer Funktion auf ein weiteres „Dings“, welche „eine Schicht“ kleiner ist.

Was ist eine Faltung?

Ein „Faltung“ im mathematischen Sinne ist die Reduktion eines „Dings“ mittels einer Funktion auf ein weiteres „Dings“, welche „eine Schicht“ kleiner ist.

Beispiele:

- Vektorraum auf den unterliegenden Körper
- Eine Liste von etwas auf ein Element
- Die Umgebung eines Pixels auf einen neuen Wert
- ...

Faltungen

Das Konzept „Dinge kleiner zu machen“ ist natürlich sehr abstrakt. Dennoch begegnet es uns in vielen Bereichen.

Faltungen

Das Konzept „Dinge kleiner zu machen“ ist natürlich sehr abstrakt. Dennoch begegnet es uns in vielen Bereichen.

Wir haben dies sogar schon benutzt: Man kann

```
join :: Monad m => m (m a) -> m a
```

als „Faltung einer Monade“ auffassen, wenn man weiß, dass man zu jeder Monade einen Monoiden definieren kann

Faltungen

Das Konzept „Dinge kleiner zu machen“ ist natürlich sehr abstrakt. Dennoch begegnet es uns in vielen Bereichen.

Wir haben dies sogar schon benutzt: Man kann

```
join :: Monad m => m (m a) -> m a
```

als „Faltung einer Monade“ auffassen, wenn man weiß, dass man zu jeder Monade einen Monoiden definieren kann, mittels:

```
instance Monad m => Monoid (a -> m a) where
  mempty  = return
  mappend = (>=>)
```

wobei >=> den sogennanten „Kleisli“-Operator darstellt:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
m >=> n = \a -> m a >>= n
```

Faltungen

Das Konzept „Dinge kleiner zu machen“ ist natürlich sehr abstrakt. Dennoch begegnet es uns in vielen Bereichen.

Wir haben dies sogar schon benutzt: Man kann

```
join :: Monad m => m (m a) -> m a
```

als „Faltung einer Monade“ auffassen, wenn man weiß, dass man zu jeder Monade einen Monoiden definieren kann, mittels:

```
instance Monad m => Monoid (a -> m a) where
  mempty  = return
  mappend = (>=>)
```

wobei >=> den sogennanten „Kleisli“-Operator darstellt:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
m >=> n = \a -> m a >>= n
```

Häufig ist dies aber uninteressant, da man sich durch den Monoiden auf einen konkreten Typen `m a` festlegen muss.

Faltungen

Und was hat `join` jetzt mit Faltungen zu tun?

Faltungen

Und was hat `join` jetzt mit Faltungen zu tun? Viel! Vergleichen wir nur mal:

```
fold :: (Foldable t, Monoid m) => t    m  ->   m
join ::               Monad m  => m (m a) -> m a
```

```
foldMap :: (Foldable t, Monoid m) => (a ->   m) -> t a ->   m
(=<<)   ::               Monad m  => (a -> m b) -> m a -> m b
```

Faltungen

Und was hat `join` jetzt mit Faltungen zu tun? Viel! Vergleichen wir nur mal:

```
fold :: (Foldable t, Monoid m) => t    m  ->   m
join ::               Monad m  => m (m a) -> m a
```

```
foldMap :: (Foldable t, Monoid m) => (a ->   m) -> t a ->   m
(=<<)   ::               Monad m  => (a -> m b) -> m a -> m b
```

Somit ist eine Monade auch ein faltbarer Monoid.

But wait! There is more!

Ein komplett anderes Beispiel wäre eine Faltung auf Bildern:

Wir nehmen einfach aus jeder Zeile bzw. jeder Spalte den durchschnittlichen Farbwert. Dieses ist eine „Faltung“ um ein 2D-Bild auf eine Linie zu reduzieren.

But wait! There is more!

Ein komplett anderes Beispiel wäre eine Faltung auf Bildern:

Wir nehmen einfach aus jeder Zeile bzw. jeder Spalte den durchschnittlichen Farbwert. Dieses ist eine „Faltung“ um ein 2D-Bild auf eine Linie zu reduzieren.

Wir können jetzt noch ein zweites Mal falten und erhalten dann den Durchschnitt auf dem gesamten Bild.

But wait! There is more!

Ein komplett anderes Beispiel wäre eine Faltung auf Bildern:

Wir nehmen einfach aus jeder Zeile bzw. jeder Spalte den durchschnittlichen Farbwert. Dieses ist eine „Faltung“ um ein 2D-Bild auf eine Linie zu reduzieren.

Wir können jetzt noch ein zweites Mal falten und erhalten dann den Durchschnitt auf dem gesamten Bild.
Alternativ können wir z.B. auch ein äußeres Produkt bilden um wieder ein Bild zu erhalten:

$$Pixel_{ij} = avg(c_i, r_j)$$

But wait! There is more!

Noch eine weitere Art wären sämtliche Bildfilter:

```
imgfilter :: Coord -> Img a -> a
imgfilter c i = foldr1 effect (neighbours c i)
  where
    effect :: a -> a -> a
    neighbours :: Coord -> Img a -> [a]
```

So (oder sehr ähnlich) ist es möglich, ganz einfach Effekte wie Kantenfilter, Gaußfilter, Durchschnitt, etc. realisieren.

Was ist eine Faltung in `Haskell`?

Was ist eine Faltung in Haskell?

```
class Foldable t where
    fold    :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldr   :: (a -> b -> b) -> b -> t a -> b
    foldr'  :: (a -> b -> b) -> b -> t a -> b
    foldl   :: (b -> a -> b) -> b -> t a -> b
    foldl'  :: (b -> a -> b) -> b -> t a -> b
    foldr1  :: (a -> a -> a) -> t a -> a
    foldl1  :: (a -> a -> a) -> t a -> a
    toList  :: t a -> [a]
    null    :: t a -> Bool
    length  :: t a -> Int
    elem    :: Eq a => a -> t a -> Bool
    maximum :: forall a . Ord a => t a -> a
    minimum :: forall a . Ord a => t a -> a
    sum     :: Num a => t a -> a
    product :: Num a => t a -> a
```

Was ist eine Faltung in `Haskell`?

```
class Foldable t where
    fold    :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldr   :: (a -> b -> b) -> b -> t a -> b
    foldr'  :: (a -> b -> b) -> b -> t a -> b
    foldl   :: (b -> a -> b) -> b -> t a -> b
    foldl'  :: (b -> a -> b) -> b -> t a -> b
    foldr1  :: (a -> a -> a) -> t a -> a
    foldl1  :: (a -> a -> a) -> t a -> a
    toList  :: t a -> [a]
    null    :: t a -> Bool
    length  :: t a -> Int
    elem    :: Eq a => a -> t a -> Bool
    maximum :: forall a . Ord a => t a -> a
    minimum :: forall a . Ord a => t a -> a
    sum     :: Num a => t a -> a
    product :: Num a => t a -> a
```

Kein Angst, alle Funktionen sind vorimplementiert. Es genügt, lediglich `foldMap` oder `foldr` definieren.

Implemantation

Versuchen wir doch einfach mal `Foldable` für unseren `V3` mittels `foldMap` zu implementieren:

```
instance Foldable V3 where
    foldMap tm (V3 x y z) = tm x <> tm y <> tm z
```

Implemantation

Versuchen wir doch einfach mal `Foldable` für unseren `V3` mittels `foldMap` zu implementieren:

```
instance Foldable V3 where
    foldMap tm (V3 x y z) = tm x <> tm y <> tm z
```

Nun können wir `vscal` schreiben als:

```
vscal :: (Foldable f, Num f, Num a) => f a -> f a -> a
vscal v w = sum $ v * w
```

Implemantation

Versuchen wir doch einfach mal `Foldable` für unseren `V3` mittels `foldMap` zu implementieren:

```
instance Foldable V3 where
    foldMap tm (V3 x y z) = tm x <> tm y <> tm z
```

Nun können wir `vscal` schreiben als:

```
vscal :: (Foldable f, Num f, Num a) => f a -> f a -> a
vscal v w = sum $ v * w
```

oder, wenn wir die Beschränkung auf `Num f` loswerden wollen:

```
vscal :: (Foldable f, Applicative f, Num a) => f a -> f a -> a
vscal v w = sum $ liftA2 (*) v w
```

Probleme mit `Foldable`

Die `Foldable`-Abstraktion hat meist ein paar mehr Freiheiten, als wir wollen. Es wird nämlich keine Reihenfolge festgelegt.

Probleme mit `Foldable`

Die `Foldable`-Abstraktion hat meist ein paar mehr Freiheiten, als wir wollen. Es wird nämlich keine Reihenfolge festgelegt.
Einfach Beispiele dafür sind z.B. `foldl (-)` und `foldr (-)`.

Probleme mit `Foldable`

Die `Foldable`-Abstraktion hat meist ein paar mehr Freiheiten, als wir wollen. Es wird nämlich keine Reihenfolge festgelegt.
Einfach Beispiele dafür sind z.B. `foldl (-)` und `foldr (-)`.
Dafür haben wir die Klasse `Traversable` - diese geht von „links nach rechts“ durch die Datenstruktur.

Probleme mit `Foldable`

Die `Foldable`-Abstraktion hat meist ein paar mehr Freiheiten, als wir wollen. Es wird nämlich keine Reihenfolge festgelegt.
Einfach Beispiele dafür sind z.B. `foldl (-)` und `foldr (-)`.
Dafür haben wir die Klasse `Traversable` - diese geht von „links nach rechts“ durch die Datenstruktur.
Jedes `Traversable` ist nicht nur ein `Foldable`, sondern auch ein `Functor`.

Probleme mit `Foldable`

Die `Foldable`-Abstraktion hat meist ein paar mehr Freiheiten, als wir wollen. Es wird nämlich keine Reihenfolge festgelegt.
Einfach Beispiele dafür sind z.B. `foldl (-)` und `foldr (-)`.
Dafür haben wir die Klasse `Traversable` - diese geht von „links nach rechts“ durch die Datenstruktur.
Jedes `Traversable` ist nicht nur ein `Foldable`, sondern auch ein `Functor`.

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f
    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id
    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse
    sequence :: Monad m => t (m a) -> m (t a)
    sequence = sequenceA
```

Implemantation

Die Anwendung auf unser V3-Beispiel ist auch einfach:

```
instance Traversable V3 where
  traverse f (V3 x y z) = V3 <$> f x <*> f y <*> f z
```

Implemantation

Die Anwendung auf unser V3-Beispiel ist auch einfach:

```
instance Traversable V3 where
  traverse f (V3 x y z) = V3 <$> f x <*> f y <*> f z
```

Was bringt uns das?

Implemantation

Die Anwendung auf unser V3-Beispiel ist auch einfach:

```
instance Traversable V3 where
  traverse f (V3 x y z) = V3 <$> f x <*> f y <*> f z
```

Was bringt uns das?

```
askV3 :: (Read a) => IO (V3 a)
askV3 = sequenceA $ ask <$> V3 "x?" "y?" "z?"
  where
    ask :: Read a => String -> IO a
```

Wir können so z.B. einfach interaktiv Vektoren einlesen.

Implemantation

Die Anwendung auf unser V3-Beispiel ist auch einfach:

```
instance Traversable V3 where
  traverse f (V3 x y z) = V3 <$> f x <*> f y <*> f z
```

Was bringt uns das?

```
askV3 :: (Read a) => IO (V3 a)
askV3 = sequenceA $ ask <$> V3 "x?" "y?" "z?"
  where
    ask :: Read a => String -> IO a
```

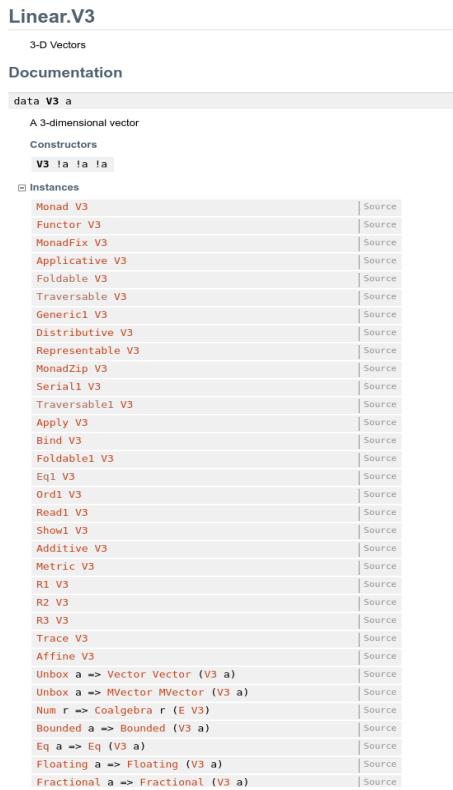
Wir können so z.B. einfach interaktiv Vektoren einlesen.

All diese Abstraktionen gibt es natürlich auch schon implementiert, damit ihr das nicht alles selbst schreiben müsst.

linear auf Hackage

V3 in linear

Die Definition von `V3` ist gleich. Allerdings gibt es ein paar mehr Instanzen[1]:



Linear.V3

3-D Vectors

Documentation

```
data V3 a
```

A 3-dimensional vector

Constructors

```
V3 !a !a !a
```

Instances

`Monad V3`	Source
`Functor V3`	Source
`MonadFix V3`	Source
`Applicative V3`	Source
`Foldable V3`	Source
`Traversable V3`	Source
`Generic1 V3`	Source
`Distributive V3`	Source
`Representable V3`	Source
`MonadZip V3`	Source
`Serial1 V3`	Source
`Traversable1 V3`	Source
`Apply V3`	Source
`Bind V3`	Source
`Foldable1 V3`	Source
`Eq1 V3`	Source
`Ord1 V3`	Source
`Read1 V3`	Source
`Show1 V3`	Source
`Additive V3`	Source
`Metric V3`	Source
`R1 V3`	Source
`R2 V3`	Source
`R3 V3`	Source
`Trace V3`	Source
`Affine V3`	Source
`Unbox a => Vector Vector (V3 a)`	Source
`Unbox a => MVector MVector (V3 a)`	Source
`Num r => Coalgebra r (E V3)`	Source
`Bounded a => Bounded (V3 a)`	Source
`Eq a => Eq (V3 a)`	Source
`Floating a => Floating (V3 a)`	Source
`Fractional a => Fractional (V3 a)`	Source

[1] `https://hackage.haskell.org/package/linear-1.20.4/docs/Linear-V3.html`

Module in linear

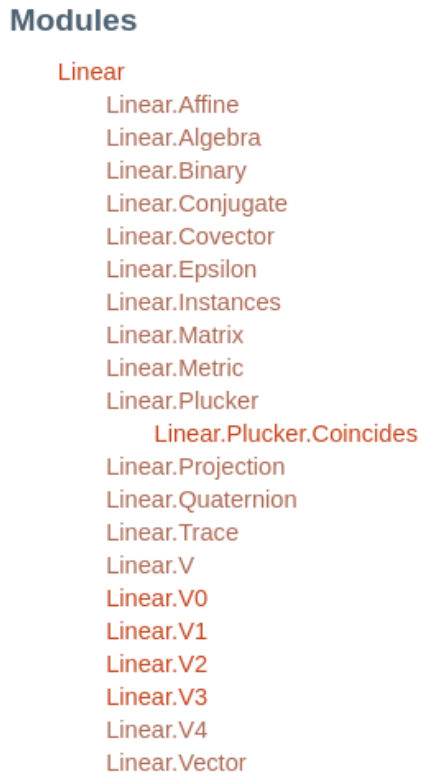
Die Module in `linear` sind viel aufschlussreicher:

Modules

- Linear
  - Linear.Affine
  - Linear.Algebra
  - Linear.Binary
  - Linear.Conjugate
  - Linear.Covector
  - Linear.Epsilon
  - Linear.Instances
  - Linear.Matrix
  - Linear.Metric
  - Linear.Plucker
    - Linear.Plucker.Coincides
  - Linear.Projection
  - Linear.Quaternion
  - Linear.Trace
  - Linear.V
  - Linear.V0
  - Linear.V1
  - Linear.V2
  - Linear.V3
  - Linear.V4
  - Linear.Vector

Module in linear

Die Module in `linear` sind viel aufschlussreicher:



Wir gehen sie nun in einer logischen Reihenfolge nach durch um uns einen Überblick zu verschaffen.

Module (I)

V, Vx, Vector sind diverse Repräsentationen
verschieden-dimensionaler Vektoren.

Module (I)

V, Vx, Vector sind diverse Repräsentationen verschieden-dimensionaler Vektoren.

Metric definiert metrische Räume (Norm, Skalarprodukt, Distanz, ...)

Module (I)

V, Vx, Vector sind diverse Repräsentationen verschieden-dimensionaler Vektoren.

Metric definiert metrische Räume (Norm, Skalarprodukt, Distanz, ...)

Matrix enthält alle gängigen Matrix-Operationen (invertieren, diagonalisieren, ...) und ist als „Vektor von Vektoren“ angelegt.

Module (I)

V, Vx, Vector sind diverse Repräsentationen verschieden-dimensionaler Vektoren.

Metric definiert metrische Räume (Norm, Skalarprodukt, Distanz, ...)

Matrix enthält alle gängigen Matrix-Operationen (invertieren, diagonalisieren, ...) und ist als „Vektor von Vektoren“ angelegt.

Trace definiert die Spur einer Matrix.

Module (I)

V, Vx, Vector sind diverse Repräsentationen verschieden-dimensionaler Vektoren.

Metric definiert metrische Räume (Norm, Skalarprodukt, Distanz, ...)

Matrix enthält alle gängigen Matrix-Operationen (invertieren, diagonalisieren, ...) und ist als „Vektor von Vektoren“ angelegt.

Trace definiert die Spur einer Matrix.

Projection stellt verschiedenste Projektions-Matrizen für den 3D-Spiele-Bereich zur Verfügung.

Module (II)

Conjugate stellt alle komplexen Operationen zur Verfügung (Adjunktion, etc.)

[2]Wikipedia: `https://en.wikipedia.org/wiki/Quaternion`

[3]Numberphile: `https://www.youtube.com/watch?v=3BR8tK-LuB0`

Module (II)

Conjugate stellt alle komplexen Operationen zur Verfügung (Adjunktion, etc.)

Quarternion bietet Quaternione[2,3].

[2] Wikipedia: `https://en.wikipedia.org/wiki/Quaternion`

[3] Numberphile: `https://www.youtube.com/watch?v=3BR8tK-LuB0`

Module (II)

Conjugate stellt alle komplexen Operationen zur Verfügung (Adjunktion, etc.)

Quarternion bietet Quaternione[2,3].

Binary ist zum serialisieren/deserialisieren.

[2] Wikipedia: `https://en.wikipedia.org/wiki/Quaternion`

[3] Numberphile: `https://www.youtube.com/watch?v=3BR8tK-LuB0`

Module (II)

Conjugate	stellt alle komplexen Operationen zur Verfügung (Adjunktion, etc.)
Quarternion	bietet Quaternione[2,3].
Binary	ist zum serialisieren/deserialisieren.
Epsilon	hilft beim Kampf mit `Float`-Ungenauigkeiten.

[2] Wikipedia: `https://en.wikipedia.org/wiki/Quaternion`
[3] Numberphile: `https://www.youtube.com/watch?v=3BR8tK-LuB0`

Module (II)

Conjugate stellt alle komplexen Operationen zur Verfügung (Adjunktion, etc.)

Quarternion bietet Quaternione[2,3].

Binary ist zum serialisieren/deserialisieren.

Epsilon hilft beim Kampf mit `Float`-Ungenauigkeiten.

Affine, Algebra, Covector, Plucker sind weitere Konzepte, auf die wir hier nicht weiter eingehen.

---

[2] Wikipedia: `https://en.wikipedia.org/wiki/Quaternion`

[3] Numberphile: `https://www.youtube.com/watch?v=3BR8tK-LuB0`

Fazit

Man kann mittels `linear` sehr gut an Vektoren und Matrizen rumspielen und so zum Beispiel direkt in Mathe gelerntes einsetzen.

Fazit

Man kann mittels `linear` sehr gut an Vektoren und Matrizen rumspielen und so zum Beispiel direkt in Mathe gelerntes einsetzen.

Die meiste Verwendung fidet diese Bibliothek in 3D-Applikationen (sprich Spiele) und in sehr vielen Machine-Learning-Algorithmen.

Fazit

Man kann mittels `linear` sehr gut an Vektoren und Matrizen rumspielen und so zum Beispiel direkt in Mathe gelerntes einsetzen.

Die meiste Verwendung fidet diese Bibliothek in 3D-Applikationen (sprich Spiele) und in sehr vielen Machine-Learning-Algorithmen.

Wenn man jetzt noch eine Möglichkeit hat diese Sachen einfach auszugeben, dann ist man auf sehr gutem Weg.

GUI

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.
Dies ist aber allgemeiner gehalten, als man normalerweise denkt und umfasst auch:

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.
Dies ist aber allgemeiner gehalten, als man normalerweise denkt und umfasst auch:

- Websites

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.
Dies ist aber allgemeiner gehalten, als man normalerweise denkt und umfasst auch:

- Websites
- Terminals

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.
Dies ist aber allgemeiner gehalten, als man normalerweise denkt und umfasst auch:

- Websites
- Terminals
- Kaffeeautomaten

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.
Dies ist aber allgemeiner gehalten, als man normalerweise denkt und umfasst auch:

- Websites
- Terminals
- Kaffeeautomaten
- ....

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.
Dies ist aber allgemeiner gehalten, als man normalerweise denkt und umfasst auch:

- Websites
- Terminals
- Kaffeeautomaten
- ....
- so ziemlich alles, was einen Bildschirm hat.

Für jedes „Zielmedium“ gibt es hierbei natürlich eigene Frameworks, die einem die eigentliche Arbeit abnehmen.

Was ist GUI?

`GUI` bedeutet „Graphical User Interface“ - ist also das, was angezeigt wird.
Dies ist aber allgemeiner gehalten, als man normalerweise denkt und umfasst auch:

- Websites
- Terminals
- Kaffeeautomaten
- ....
- so ziemlich alles, was einen Bildschirm hat.

Für jedes „Zielmedium“ gibt es hierbei natürlich eigene Frameworks, die einem die eigentliche Arbeit abnehmen.
Wir werden uns heute mit dem sehr einfachen Haskell-Framework `gloss` auseinandersetzen.

Was ist `gloss`?

Die Selbstbeschreibung zu `gloss` auf Hackage ist schon sehr treffend:

> *Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions. Gloss uses OpenGL under the hood, but you won't need to worry about any of that. Get something cool on the screen in under 10 minutes.*

Was ist `gloss`?

Die Selbstbeschreibung zu `gloss` auf Hackage ist schon sehr treffend:

> *Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions. Gloss uses OpenGL under the hood, but you won't need to worry about any of that. Get something cool on the screen in under 10 minutes.*

Es ist somit geeignet für:

Was ist `gloss`?

Die Selbstbeschreibung zu `gloss` auf Hackage ist schon sehr treffend:

> *Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions. Gloss uses OpenGL under the hood, but you won't need to worry about any of that. Get something cool on the screen in under 10 minutes.*

Es ist somit geeignet für:

- „schnell mal was anzeigen“

Was ist `gloss`?

Die Selbstbeschreibung zu `gloss` auf Hackage ist schon sehr treffend:

> *Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions. Gloss uses OpenGL under the hood, but you won't need to worry about any of that. Get something cool on the screen in under 10 minutes.*

Es ist somit geeignet für:

- „schnell mal was anzeigen“
- „simple“ Dinge, wie Visualisierungen

Was ist `gloss`?

Die Selbstbeschreibung zu `gloss` auf Hackage ist schon sehr treffend:

> *Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions. Gloss uses OpenGL under the hood, but you won't need to worry about any of that. Get something cool on the screen in under 10 minutes.*

Es ist somit geeignet für:

- „schnell mal was anzeigen“
- „simple“ Dinge, wie Visualisierungen
- **aber nicht** für sowas wie 3D-Spiele

Beispiel

Am besten lernt man am Beispiel. Hierzu sehen wir uns zunächst die Funktionen an, die das tun, was wir wollen.

Beispiel

Am besten lernt man am Beispiel. Hierzu sehen wir uns zunächst die Funktionen an, die das tun, was wir wollen.

Anschließend gehen wir nach dem bewährten Verfahren vor:

Beispiel

Am besten lernt man am Beispiel. Hierzu sehen wir uns zunächst die Funktionen an, die das tun, was wir wollen.

Anschließend gehen wir nach dem bewährten Verfahren vor:

1. Welche Funktion brauche ich?

Beispiel

Am besten lernt man am Beispiel. Hierzu sehen wir uns zunächst die Funktionen an, die das tun, was wir wollen.

Anschließend gehen wir nach dem bewährten Verfahren vor:

1. Welche Funktion brauche ich?
2. Welche Typen nimmt die Funktion?

Beispiel

Am besten lernt man am Beispiel. Hierzu sehen wir uns zunächst die Funktionen an, die das tun, was wir wollen.

Anschließend gehen wir nach dem bewährten Verfahren vor:

1. Welche Funktion brauche ich?
2. Welche Typen nimmt die Funktion?
3. Wie erzeuge ich diese Typen?

Beispiel

Wir finden in `Graphics.Gloss` folgende Dinge:

```
display  :: Display -> Color -> Picture -> IO ()
animate  :: Display -> Color -> (Float -> Picture) -> IO ()
simulate :: {- lange Typsignatur -}
play     :: {- noch laengere Typsignatur -}
```

Beispiel

Wir finden in `Graphics.Gloss` folgende Dinge:

```
display  :: Display -> Color -> Picture -> IO ()
animate  :: Display -> Color -> (Float -> Picture) -> IO ()
simulate :: {- lange Typsignatur -}
play     :: {- noch laengere Typsignatur -}
```

Somit scheint der sinnvollste Anfang `display` zu sein.

Beispiel

Wir finden in `Graphics.Gloss` folgende Dinge:

```
display  :: Display -> Color -> Picture -> IO ()
animate  :: Display -> Color -> (Float -> Picture) -> IO ()
simulate :: {- lange Typsignatur -}
play     :: {- noch laengere Typsignatur -}
```

Somit scheint der sinnvollste Anfang `display` zu sein.
Was ist nun dieses `Display`?

Beispiel

Wir finden in `Graphics.Gloss` folgende Dinge:

```
display  :: Display -> Color -> Picture -> IO ()
animate  :: Display -> Color -> (Float -> Picture) -> IO ()
simulate :: {- lange Typsignatur -}
play     :: {- noch laengere Typsignatur -}
```

Somit scheint der sinnvollste Anfang `display` zu sein.

Was ist nun dieses `Display`? Sehen wir nach:

```
data Display
        -- | Display in a window with the given name, size and position.
        = InWindow   String (Int, Int) (Int, Int)
        -- | Display full screen with a drawing area of the given size.
        | FullScreen (Int, Int)
        deriving (Eq, Read, Show)
```

Beispiel

Und `Color`?

Beispiel

Und `Color`?

An abstract color value. We keep the type abstract so we can be sure that the components are in the required range. To make a custom color use makeColor.

Beispiel

Und `Color`?

> *An abstract color value. We keep the type abstract so we can be sure that the components are in the required range. To make a custom color use makeColor.*

Also sehen wir uns `makeColor` an:

```
makeColor :: Float -- Red component.
          -> Float -- Green component.
          -> Float -- Blue component.
          -> Float -- Alpha component.
          -> Color
--Make a custom color. All components are clamped to the range [0..1].
```

Beispiel

Und `Color`?

> *An abstract color value. We keep the type abstract so we can be sure that the components are in the required range. To make a custom color use makeColor.*

Also sehen wir uns `makeColor` an:

```
makeColor :: Float -- Red component.
          -> Float -- Green component.
          -> Float -- Blue component.
          -> Float -- Alpha component.
          -> Color
--Make a custom color. All components are clamped to the range [0..1].
```

Sollte auch klar sein. Ansonsten kann man sich aus den vordefinierten (in `Graphics.Gloss.Data.Color`) auch noch eine aussuchen:
black, white, red, green, blue, yellow, cyan, magenta, orange, ...

Beispiel

Kommen wir zum letzten: Wie funktioniert `Picture`?

Beispiel

Kommen wir zum letzten: Wie funktioniert `Picture`?

```
data Picture = Blank
             | Polygon Path
             | Line Path
             | Circle Float
             | ThickCircle Float Float
             | Arc Float Float Float
             | ThickArc Float Float Float Float
             | Text String
             | Bitmap Int Int BitmapData Bool
             | Color Color Picture
             | Translate Float Float Picture
             | Rotate Float Picture
             | Scale Float Float Picture
             | Pictures [Picture]
```

Beispiel

Kommen wir zum letzten: Wie funktioniert `Picture`?

```
data Picture = Blank
             | Polygon Path
             | Line Path
             | Circle Float
             | ThickCircle Float Float
             | Arc Float Float Float
             | ThickArc Float Float Float Float
             | Text String
             | Bitmap Int Int BitmapData Bool
             | Color Color Picture
             | Translate Float Float Picture
             | Rotate Float Picture
             | Scale Float Float Picture
             | Pictures [Picture]
```

Hierin müssen wir nun speichern, was wir darstellen wollen und können komplexe Formen z.B. durch Rekursion aufbauen.

Beispiel (Code)

Alles zusammen gibt dann in einem praktischen Beispiel:

```
import Graphics.Gloss
import Graphics.Gloss.Data.Color (makeColor, red)

main :: IO ()
main = display (InWindow "Hello World" (500,500) (100,100))
               (makeColor 0.9 0.9 0.9 1)
               (Pictures [ Color red (Circle 1000)
                         , Text "Hello World Text"
                         ])
```

Beispiel (Ergebnis)

Wenn man es laufen lässt, sieht das wie folgt aus:

Weitere Funktionen

Was machen nun die Anderen Funktionen?

Weitere Funktionen

Was machen nun die Anderen Funktionen?

```
animate :: Display -> Color -> (Float -> Picture) -> IO ()
```

Hier müssen wir kein Bild definieren, sondern eine Parameterfunktion `Float -> Picture`, welche uns das Bild nach `x` Sekunden generiert.

Weitere Funktionen

Was machen nun die Anderen Funktionen?

```
animate :: Display -> Color -> (Float -> Picture) -> IO ()
```

Hier müssen wir kein Bild definieren, sondern eine Parameterfunktion `Float -> Picture`, welche uns das Bild nach `x` Sekunden generiert.

Ein simples Beispiel wäre:

```
main :: IO ()
main = animate (InWindow "Spin around" (500,500) (100,100))
               (makeColor 0.9 0.9 0.9 1)
               (\time ->
                Pictures [ Color red $ ThickCircle 500 1000
                         , Rotate (time/60*360)
                           $ Text
                           $ show (round time) <> " Seconds"
                         ])
```

Beispiel (Ergebnis)

Oder ausgeführt:

Weitere Funktionen (I)

Die weiteren Dinge sind über die Typen auch selbsterklärend:

```
simulate :: Display --Display mode.
         -> Color   --Background color.
         -> Int     --Number of simulation steps to take
                    --for each second of real time.
         -> model   --The initial model.
         -> (model -> Picture)
                    --A function to convert the model
                    --to a picture.
         -> (ViewPort -> Float -> model -> model)
                    --A function to step the model one
                    --iteration. It is passed the current
                    --viewport and the amount of time for
                    --this simulation step (in seconds).
         -> IO ()
```

Run a finite-time-step simulation in a window. You decide how the model is represented, how to convert the model to a picture, and how to advance the model for each unit of time. This function does the rest.

Weitere Funktionen (II)

Außerdem gibt es noch `play`:

```
play :: Display --Display mode.
     -> Color   --Background color.
     -> Int     --Number of simulation steps to take
                --for each second of real time.
     -> world   --The initial world.
     -> (world -> Picture)
                --A function to convert the world a picture.
     -> (Event -> world -> world)
                --A function to handle input events.
     -> (Float -> world -> world)
                --A function to step the world one iteration.
                --It is passed the period of time (in seconds)
                --needing to be advanced.
     -> IO ()
```

Weitere Funktionen (II)

Außerdem gibt es noch `play`:

```
play :: Display --Display mode.
     -> Color   --Background color.
     -> Int     --Number of simulation steps to take
                --for each second of real time.
     -> world   --The initial world.
     -> (world -> Picture)
                --A function to convert the world a picture.
     -> (Event -> world -> world)
                --A function to handle input events.
     -> (Float -> world -> world)
                --A function to step the world one iteration.
                --It is passed the period of time (in seconds)
                --needing to be advanced.
     -> IO ()
```

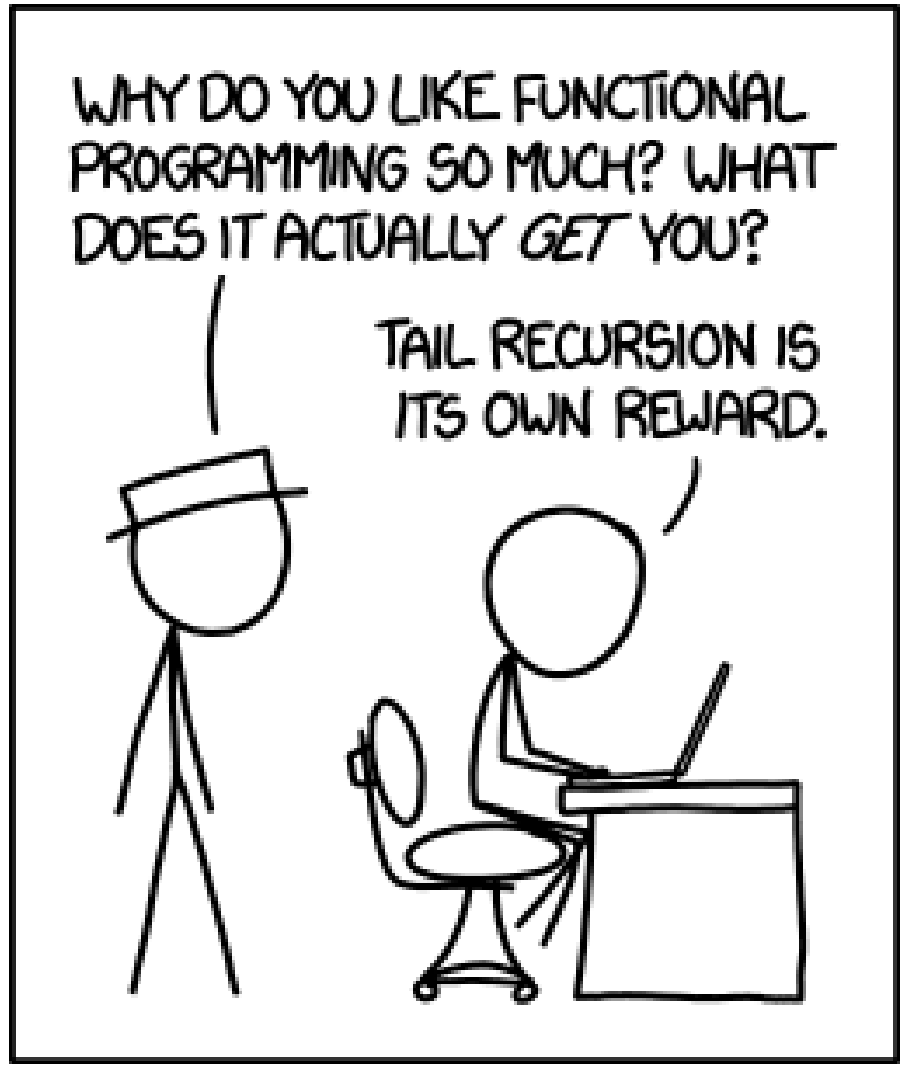
Man hat hier dafür keine vordefinierten Interaktionen (drag/drop/zoom/...) und muss alles selbst definieren.

Ausblick

Vorschau: Was machen wir nächste Woche?

- Wiederholung: Vorlesung 5
- Überblick über verschiedene Monaden
- mehr Typklassen?

Fragen?



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