Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 14: *How deep the rabbit hole goes. . .*
Stand: 22. Juli 2016

Was machen wir heute?

Was machen wir heute?

Zum Abschluss der Vorlesung wollen wir euch einen groben Überblick über das geben, was wir gerne angesprochen hätten.

Was machen wir heute?

Zum Abschluss der Vorlesung wollen wir euch einen groben Überblick über das geben, was wir gerne angesprochen hätten.

Themen:

- Fixpunkte

Was machen wir heute?

Zum Abschluss der Vorlesung wollen wir euch einen groben Überblick über das geben, was wir gerne angesprochen hätten.

Themen:

- Fixpunkte
- Recursion-Schemes

Was machen wir heute?

Zum Abschluss der Vorlesung wollen wir euch einen groben Überblick über das geben, was wir gerne angesprochen hätten.

Themen:

- Fixpunkte
- Recursion-Schemes
- Co-Monaden

Was machen wir heute?

Zum Abschluss der Vorlesung wollen wir euch einen groben Überblick über das geben, was wir gerne angesprochen hätten.

Themen:

- Fixpunkte
- Recursion-Schemes
- Co-Monaden
- Free und Co-Free

Was machen wir heute?

Zum Abschluss der Vorlesung wollen wir euch einen groben Überblick über das geben, was wir gerne angesprochen hätten.

Themen:

- Fixpunkte
- Recursion-Schemes
- Co-Monaden
- Free und Co-Free
- Weitere Technologien

Fixpunkte

Fixpunkte

Mathematisch: eine Funktion f hat einen Fixpunkt x, wenn gilt $f(x) = x$.

Fixpunkte

Mathematisch: eine Funktion f hat einen Fixpunkt x, wenn gilt $f(x) = x$.

Ein Fixpunkt ist abstrakt gesprochen ein Punkt, an dem eine weitere Anwendung der Funktion keinen Unterschied mehr macht.

Fixpunkte

Mathematisch: eine Funktion f hat einen Fixpunkt x, wenn gilt $f(x) = x$.

Ein Fixpunkt ist abstrakt gesprochen ein Punkt, an dem eine weitere Anwendung der Funktion keinen Unterschied mehr macht.
Alle Verfahren, die durch wiederholte Anwendung derselben Funktion zu einem Ergebnis kommen, sind iterative Fixpunkt-Suchen, wie zum Beispiel das Newton-Verfahren.

Fixpunkte

Mathematisch: eine Funktion f hat einen Fixpunkt x, wenn gilt $f(x) = x$.

Ein Fixpunkt ist abstrakt gesprochen ein Punkt, an dem eine weitere Anwendung der Funktion keinen Unterschied mehr macht.
Alle Verfahren, die durch wiederholte Anwendung derselben Funktion zu einem Ergebnis kommen, sind iterative Fixpunkt-Suchen, wie zum Beispiel das Newton-Verfahren.

Wenn wir in `Haskell` eine Struktur haben, die sich selbst enthalten kann, dann können wir darüber immer einen Funktor erstellen:

```haskell
data Typename a = Konstruktor {- .... irgendwo wird a verwendet ... -}
```

In der Mathematik unterscheidet man noch zwischen „least“ und „greatest“ Fixpunkt, welche in `Haskell` aber identisch sind uns uns daher nicht weiter interessieren.[1]

[1] Es gibt ein Kata, wo man diesen Isomorphismus implementieren muss: `https://www.codewars.com/kata/folding-through-a-fixed-point`

In der Mathematik unterscheidet man noch zwischen „least“ und „greatest“ Fixpunkt, welche in `Haskell` aber identisch sind uns uns daher nicht weiter interessieren.[1]

Wie immer, wenn wir ein Konzept in `Haskell` haben, bauen wir entweder einen Typen oder ein Typklasse, die dieses einfängt.

[1] Es gibt ein Kata, wo man diesen Isomorphismus implementieren muss: `https://www.codewars.com/kata/folding-through-a-fixed-point`

In der Mathematik unterscheidet man noch zwischen „least“ und „greatest“ Fixpunkt, welche in `Haskell` aber identisch sind uns uns daher nicht weiter interessieren.[1]

Wie immer, wenn wir ein Konzept in `Haskell` haben, bauen wir entweder einen Typen oder ein Typklasse, die dieses einfängt.

Die Definition sieht folgendermaßen aus:

```
newtype Fix f = Fix { unfix :: f (Fix f) }
```

[1] Es gibt ein Kata, wo man diesen Isomorphismus implementieren muss: `https://www.codewars.com/kata/folding-through-a-fixed-point`

In der Mathematik unterscheidet man noch zwischen „least" und „greatest" Fixpunkt, welche in `Haskell` aber identisch sind uns uns daher nicht weiter interessieren.[1]

Wie immer, wenn wir ein Konzept in `Haskell` haben, bauen wir entweder einen Typen oder ein Typklasse, die dieses einfängt.

Die Definition sieht folgendermaßen aus:

```
newtype Fix f = Fix { unfix :: f (Fix f) }
```

Wir können somit ein weiteres `Fix f` erzeugen, indem wir eine weitere „Schicht" des Funktors hinzufügen.

[1] Es gibt ein Kata, wo man diesen Isomorphismus implementieren muss: https://www.codewars.com/kata/folding-through-a-fixed-point

Wieso interessiert uns das alles?

Wieso interessiert uns das alles?
Funktoren sind sehr mächtig und wir können in ihnen vieles Speichern und somit sehr generische Algorithmen schreiben.

Wieso interessiert uns das alles?
Funktoren sind sehr mächtig und wir können in ihnen vieles Speichern und somit sehr generische Algorithmen schreiben.

Daher kommen wir nun zu einem konkreten Beispiel aus der letzten Vorlesung:

```
-- Haskell type of System T expressions
data Lang exp = Z                        -- Zero
              | Succ exp                 -- Successor
              | Var String               -- Variables
              | Lambda Typ exp exp       -- Lambda
              | Rec exp exp exp exp exp  -- Recursor
              | Ap  exp exp              -- Application
              deriving (Functor, Show, Eq)
```

Fixpunkte

Wieso interessiert uns das alles?
Funktoren sind sehr mächtig und wir können in ihnen vieles Speichern und somit sehr generische Algorithmen schreiben.

Daher kommen wir nun zu einem konkreten Beispiel aus der letzten Vorlesung:

```haskell
-- Haskell type of System T expressions
data Lang exp = Z                          -- Zero
              | Succ exp                   -- Successor
              | Var String                 -- Variables
              | Lambda Typ exp exp         -- Lambda
              | Rec exp exp exp exp exp    -- Recursor
              | Ap  exp exp                -- Application
              deriving (Functor, Show, Eq)
```

Alle gültigen Programme sind Fixpunkte dieses Funktors.
Wenn wir uns das als einen Baum vorstellen, dann darf in den Blättern kein `exp` mehr vorkommen, sondern nur noch `Var` und `Z`.

Fixpunkte

Nun, da wir wissen, wie eine Fixpunkt-Struktur aussieht, stellen sich die zwei üblichen Fragen:

Nun, da wir wissen, wie eine Fixpunkt-Struktur aussieht, stellen sich die zwei üblichen Fragen:

- Wie generieren wir einen solchen Typen?

Nun, da wir wissen, wie eine Fixpunkt-Struktur aussieht, stellen sich die zwei üblichen Fragen:

- Wie generieren wir einen solchen Typen?
- Wie verbrauchen wir so einen Typen?

Nun, da wir wissen, wie eine Fixpunkt-Struktur aussieht, stellen sich die zwei üblichen Fragen:

- Wie generieren wir einen solchen Typen?
- Wie verbrauchen wir so einen Typen?

Kurzum, wir brauchen

```
create  :: Functor f => creationRules    -> a     -> Fix f
consume :: Functor f => consumptionRules -> Fix f -> b
```

Recursion-Schemes[2]

[2] Implementation u.v.m.: `https://hackage.haskell.org/package/recursion-schemes`

Recursion-Schemes

Recursion-Schemes fassen alle rekursive Vorgänge, die über einen Funktor operieren, unter einem Dach zusammen.

Recursion-Schemes fassen alle rekursive Vorgänge, die über einen Funktor operieren, unter einem Dach zusammen.

Abgesehen vom nicht-verändern der Struktur (`fmap`), gibt es genau zwei Endzustände einer wiederholten Anwendung einer Regel:

Recursion-Schemes

Recursion-Schemes fassen alle rekursive Vorgänge, die über einen Funktor operieren, unter einem Dach zusammen.

Abgesehen vom nicht-verändern der Struktur (`fmap`), gibt es genau zwei Endzustände einer wiederholten Anwendung einer Regel:

- Wir bauen eine Struktur **auf** und **entfalten** mit jeder Anwendung den Funktor weiter **zu seinem Fixpunkt**

Recursion-Schemes fassen alle rekursive Vorgänge, die über einen Funktor operieren, unter einem Dach zusammen.

Abgesehen vom nicht-verändern der Struktur (`fmap`), gibt es genau zwei Endzustände einer wiederholten Anwendung einer Regel:

- Wir bauen eine Struktur **auf** und **entfalten** mit jeder Anwendung den Funktor weiter **zu seinem Fixpunkt**
- Wir bauen eine Struktur **ab** und **falten** mit jeder Anwendung den Fixpunkt des Funktors weiter **auf ein Ergebnis**

In der Mathematik nennt man Regeln, die Vorgeben, wie eine Struktur reduziert wird, eine **Algebra**. Analog geschieht das Aufbauen mit einer **CoAlgebra**.

Recursion-Schemes

In der Mathematik nennt man Regeln, die Vorgeben, wie eine Struktur reduziert wird, eine **Algebra**. Analog geschieht das Aufbauen mit einer **CoAlgebra**.

Da wir bereits etabliert haben, dass ein Fixpunkt aus Ebenen besteht, brauchen wir nur Regeln für „die nächste Ebene“.

In der Mathematik nennt man Regeln, die Vorgeben, wie eine Struktur reduziert wird, eine **Algebra**. Analog geschieht das Aufbauen mit einer **CoAlgebra**.

Da wir bereits etabliert haben, dass ein Fixpunkt aus Ebenen besteht, brauchen wir nur Regeln für „die nächste Ebene“.

In `Haskell` bedeutet das dann:

```
type FAlgebra   f a = Functor f => f a ->   a
type FCoAlgebra f a = Functor f =>   a -> f a
```

In der Mathematik nennt man Regeln, die Vorgeben, wie eine Struktur reduziert wird, eine **Algebra**. Analog geschieht das Aufbauen mit einer **CoAlgebra**.

Da wir bereits etabliert haben, dass ein Fixpunkt aus Ebenen besteht, brauchen wir nur Regeln für „die nächste Ebene“.

In `Haskell` bedeutet das dann:

```
type FAlgebra   f a = Functor f => f a ->   a
type FCoAlgebra f a = Functor f =>   a -> f a
```

Zusammengesetzt ergeben sich zwei grundlegende Konzepte:

```
cata :: Functor f => FAlgebra  f a -> Fix f ->     a
ana  :: Functor f => FCoAlgera f a ->     a -> Fix f
```

genannt **Catamorphismus** und **Anamorphismus**.

Ein praktisches Beispiel:

```
data Tree  a   = Leaf  a
               | Branch  (Tree a) (Tree a)
                deriving (Show,Eq,Functor)
data TreeF a t = LeafF a
               | BranchF t t
                deriving (Show,Eq,Functor)
```

Ein praktisches Beispiel:

```
data Tree  a   = Leaf  a
               | Branch  (Tree a) (Tree a)
                deriving (Show,Eq,Functor)
data TreeF a t = LeafF a
               | BranchF t t
                deriving (Show,Eq,Functor)
```

Wir wollen nun erstmal einen solchen Baum konstruieren. Als Beispiel hätten wir gerne den String `"Hallo"` in einen Baum geschrieben. Somit ergeben die Typen:

```
ana :: (a -> f a) -> a -> Fix f

ana :: (String -> TreeF String String) -> String -> Fix (TreeF String)
```

Ein praktisches Beispiel:

```
data Tree  a   = Leaf  a
               | Branch  (Tree a) (Tree a)
                deriving (Show,Eq,Functor)
data TreeF a t = LeafF a
               | BranchF t t
                deriving (Show,Eq,Functor)
```

Wir wollen nun erstmal einen solchen Baum konstruieren. Als Beispiel hätten wir gerne den String `"Hallo"` in einen Baum geschrieben. Somit ergeben die Typen:

```
ana :: (a -> f a) -> a -> Fix f
```

```
ana :: (String -> TreeF String String) -> String -> Fix (TreeF String)
```

Den String haben wir, fehlt nur noch die CoAlgebra.

```haskell
treeCoAlg :: String -> TreeF String String
treeCoAlg s = case len of
        1         -> LeafF s
        otherwise -> BranchF (take len2 s) (drop len2 s)
    where
        len = length s
        len2 = len `div` 2
```

Diese CoAlgebra schaut, ob der String die Länge 1 hat und generiert dann ein `LeafF` mit dem Zeichen oder Teilt dieses in einen `BranchF` auf.

```haskell
treeCoAlg :: String -> TreeF String String
treeCoAlg s = case len of
        1          -> LeafF s
        otherwise -> BranchF (take len2 s) (drop len2 s)
    where
        len = length s
        len2 = len `div` 2
```

Diese CoAlgebra schaut, ob der String die Länge 1 hat und generiert dann ein `LeafF` mit dem Zeichen oder Teilt dieses in einen `BranchF` auf.

Somit können wir nun einfach ana aufrufen:

```haskell
createTree :: String -> Fix (TreeF String)
createTree s = ana treeCoAlg s
```

Da wir gerade den Fixpunkt so schön aufgebaut haben, reißen wir ihn nun wieder ein.
In diesem Beispiel hätten wir gerne den Fixpunkt auf einen `Int` gefaltet.

Da wir gerade den Fixpunkt so schön aufgebaut haben, reißen wir ihn nun wieder ein.
In diesem Beispiel hätten wir gerne den Fixpunkt auf einen `Int` gefaltet.
Die Typen sagen uns somit:

```
cata :: (f a -> a) -> Fix f -> a

cata :: (TreeF String Int -> Int) -> Fix (TreeF String) -> Int
```

Da wir gerade den Fixpunkt so schön aufgebaut haben, reißen wir ihn nun wieder ein.
In diesem Beispiel hätten wir gerne den Fixpunkt auf einen `Int` gefaltet.
Die Typen sagen uns somit:

```
cata :: (f a -> a) -> Fix f -> a

cata :: (TreeF String Int -> Int) -> Fix (TreeF String) -> Int
```

Interessant ist: Der Fixpunkt weiss nicht, welchen Typen wir zum generieren oder zum akkumulieren nehmen.
Wir müssen jetzt nur noch die Algebra bestimmen.

```
treeAlg :: TreeF String Int -> Int
treeAlg (LeafF (c:_)) = if isLower c then 1 else 0
treeAlg (BranchF l r) = l + r
```

Diese Algebra schaut, ob der String mit einem Groß- oder Kleinbuchstaben anfängt und Summiert die Teillösungen auf.

```
treeAlg :: TreeF String Int -> Int
treeAlg (LeafF (c:_)) = if isLower c then 1 else 0
treeAlg (BranchF l r) = l + r
```

Diese Algebra schaut, ob der String mit einem Groß- oder Kleinbuchstaben anfängt und Summiert die Teillösungen auf.

Somit können wir nun einfach cata aufrufen:

```
countLowerInTree :: Fix (TreeF String) -> Int
countLowerInTree t = cata treeAlg t
```

```
treeAlg :: TreeF String Int -> Int
treeAlg (LeafF (c:_)) = if isLower c then 1 else 0
treeAlg (BranchF l r) = l + r
```

Diese Algebra schaut, ob der String mit einem Groß- oder Kleinbuchstaben anfängt und Summiert die Teillösungen auf.

Somit können wir nun einfach cata aufrufen:

```
countLowerInTree :: Fix (TreeF String) -> Int
countLowerInTree t = cata treeAlg t
```

Wir haben somit die Anwendung der Algebra vom Schema der Rekursion getrennt und müssen immer nur genau einen Schritt vorgeben.

Ist das alles nicht furchtbar kompliziert und hat keinen Vorteil gegenüber der „klassischen“ Weise dieses zu tun?

Ist das alles nicht furchtbar kompliziert und hat keinen Vorteil gegenüber der „klassischen“ Weise dieses zu tun?
Das Interessante ist, dass wir jetzt kombinieren können:

```
cata :: Functor f => FAlgebra  f a -> Fix f ->     a
ana  :: Functor f => FCoAlgera f a ->     a -> Fix f

hylo :: Functor f => FAlgebra  f a -> FCoAlgebra f b -> a -> b
hylo phi psi = cata phi . ana psi
-- oder
hylo f   g   = h where h = f . fmap h . g
```

Ist das alles nicht furchtbar kompliziert und hat keinen Vorteil gegenüber der „klassischen“ Weise dieses zu tun?
Das Interessante ist, dass wir jetzt kombinieren können:

```
cata :: Functor f => FAlgebra  f a -> Fix f ->     a
ana  :: Functor f => FCoAlgera f a ->     a -> Fix f

hylo :: Functor f => FAlgebra  f a -> FCoAlgebra f b -> a -> b
hylo phi psi = cata phi . ana psi
-- oder
hylo f   g   = h where h = f . fmap h . g
```

Mehr noch: Da `Haskell` lazy ist, wird die Struktur nur soweit aufgebaut, wie es minimal nötig ist, damit sie wieder abgebaut werden kann.

Mit dem, was wir bisher haben, können wir nun eine Funktion `String -> Int` angeben:

```
countLower :: String -> Int
countLower = hylo treeAlg treeCoAlg
```

Mit dem, was wir bisher haben, können wir nun eine Funktion `String -> Int` angeben:

```
countLower :: String -> Int
countLower = hylo treeAlg treeCoAlg
```

Das allein mag nun nicht sehr hilfreich klingen, aber ermöglicht es uns viele Dinge zu trennen und separat zu betrachten (und wiederzuverwenden).

Mit dem, was wir bisher haben, können wir nun eine Funktion `String -> Int` angeben:

```
countLower :: String -> Int
countLower = hylo treeAlg treeCoAlg
```

Das allein mag nun nicht sehr hilfreich klingen, aber ermöglicht es uns viele Dinge zu trennen und separat zu betrachten (und wiederzuverwenden). Durch den Austausch der Algebra können wir aber auch ganz andere Dinge tun. z.B.:

```
drawTreeAlg :: Show s => Fix (TreeF s Picture) -> Picture

drawStringAsTree :: String -> Picture
drawStringAsTree = hylo drawTreeAlg treeCoAlg

drawTree :: Show s => (a -> TreeF s a) -> a -> Picture
drawTree = hylo drawTreeAlg
```

Hier geben wir nur noch eine Regel an, wie wir den Input aufsplitten und was die Beschreibung davon ist und schon können wir jedes a als Baum visualisieren.

`Ana` und `Cata` machen noch nichts besonderes gegenüber „normaler“ Rekursion - allerdings können wir den tiefergehenden Stoff nur anreißen.

`Ana` und `Cata` machen noch nichts besonderes gegenüber „normaler“ Rekursion - allerdings können wir den tiefergehenden Stoff nur anreißen.

Zu einem genaueren Studium gehört eine gehörige Portion Kategorientheorie und ein hohes Abstraktionsvermögen.

`Ana` und `Cata` machen noch nichts besonderes gegenüber „normaler“ Rekursion - allerdings können wir den tiefergehenden Stoff nur anreißen.

Zu einem genaueren Studium gehört eine gehörige Portion Kategorientheorie und ein hohes Abstraktionsvermögen.

Viele Programme lassen sich zusammenfassen als:

1. Eingaben in Datenstrukturen einlesen
2. Datenstrukturen bearbeiten und transformieren
3. Datenstrukturen zu einem Ergebnis zusammenfalten

`Ana` und `Cata` machen noch nichts besonderes gegenüber „normaler“ Rekursion - allerdings können wir den tiefergehenden Stoff nur anreißen.

Zu einem genaueren Studium gehört eine gehörige Portion Kategorientheorie und ein hohes Abstraktionsvermögen.

Viele Programme lassen sich zusammenfassen als:

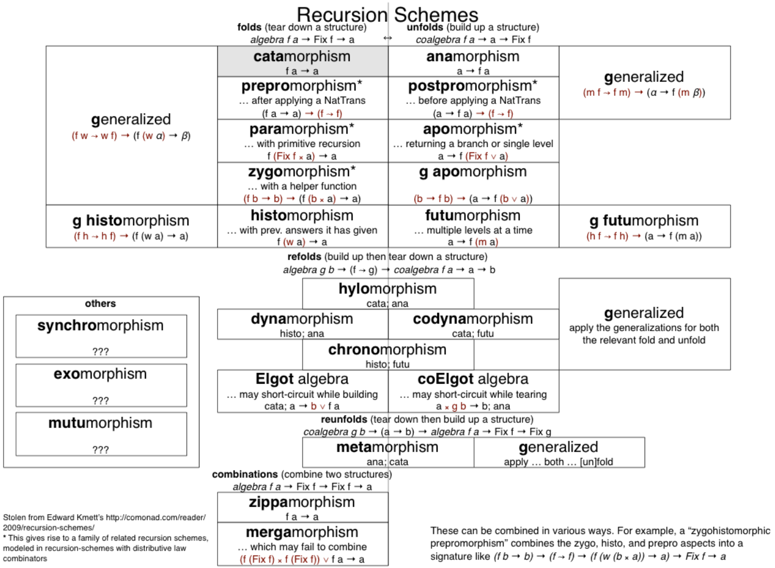
1. Eingaben in Datenstrukturen einlesen
2. Datenstrukturen bearbeiten und transformieren
3. Datenstrukturen zu einem Ergebnis zusammenfalten

Dieses können wir hiermit fast immer zusammenfassen auf ein:

```
program :: in -> out
program = cata alg . op . ana coalg

op :: (Functor f, Functor g) => Fix f -> Fix g
op = undefined
```

Recursion-Schemes



`https://github.com/slamdata/matryoshka/blob/master/resources/recursion-schemes.png`

Co-Monaden[3]

[3] All About Comonads - An incomprehensible guide: `http://comonad.com/haskell/Comonads_1.pdf`

Zunächst etwas Kategorientheorie:
Eine Kategorie besteht aus

1. Objekten
2. Pfeilen (Weg von Objekt a zu Objekt b, kurz: $a \to b$)
3. Kombinierbarkeit ($\forall f : a \to b, g : b \to c \; \exists \; g \circ f : a \to c$)

Zunächst etwas Kategorientheorie:
Eine Kategorie besteht aus

1. Objekten
2. Pfeilen (Weg von Objekt a zu Objekt b, kurz: $a \rightarrow b$)
3. Kombinierbarkeit ($\forall f : a \rightarrow b, g : b \rightarrow c \ \exists \ g \circ f : a \rightarrow c$)

Pfeile sind in der Kategorientheorie die einzige Möglichkeit, wie man von einem Objekt zu einem anderen kommt. Ein Pfeil hat hierbei genau eine Quelle und ein Ziel.

Zunächst etwas Kategorientheorie:
Eine Kategorie besteht aus

1. Objekten
2. Pfeilen (Weg von Objekt a zu Objekt b, kurz: $a \to b$)
3. Kombinierbarkeit ($\forall f : a \to b, g : b \to c \; \exists \; g \circ f : a \to c$)

Pfeile sind in der Kategorientheorie die einzige Möglichkeit, wie man von einem Objekt zu einem anderen kommt. Ein Pfeil hat hierbei genau eine Quelle und ein Ziel.

Bevor wir zu Co-Monaden kommen, schauen wir uns erstmal Funktoren genauer an:

Ein Funktor ist ein Pfeil in der Kategorie der (kleinen) Kategorien und weist jedem Objekt/Pfeil aus Kategorie $\mathcal{C}$ ein Objekt/Pfeil in Kategorie $\mathcal{D}$ zu.

In `Haskell` haben wir nur die Kategorie der `Haskell`-Typen (auch „Hask“ genannt), sodass wir hier nur noch Pfeile auf Pfeile mappen müssen.

In `Haskell` haben wir nur die Kategorie der `Haskell`-Typen (auch „Hask“ genannt), sodass wir hier nur noch Pfeile auf Pfeile mappen müssen.

Pfeile in Hask sind in `Haskell`-Funktionen, die zwei Objekte (Typen) verbinden (markiert durch einen „Pfeil“ (`->`)).

In `Haskell` haben wir nur die Kategorie der `Haskell`-Typen (auch „Hask“ genannt), sodass wir hier nur noch Pfeile auf Pfeile mappen müssen.

Pfeile in Hask sind in `Haskell`-Funktionen, die zwei Objekte (Typen) verbinden (markiert durch einen „Pfeil“ (`->`)).

Für einen Funktor ergibt sich somit:

In `Haskell` haben wir nur die Kategorie der `Haskell`-Typen (auch „Hask“ genannt), sodass wir hier nur noch Pfeile auf Pfeile mappen müssen.

Pfeile in Hask sind in `Haskell`-Funktionen, die zwei Objekte (Typen) verbinden (markiert durch einen „Pfeil“ (->)).

Für einen Funktor ergibt sich somit:

```
fmap :: (a -> b) ->  f a -> f b
```

In `Haskell` haben wir nur die Kategorie der `Haskell`-Typen (auch „Hask“ genannt), sodass wir hier nur noch Pfeile auf Pfeile mappen müssen.

Pfeile in Hask sind in `Haskell`-Funktionen, die zwei Objekte (Typen) verbinden (markiert durch einen „Pfeil“ (->)).

Für einen Funktor ergibt sich somit:

```
fmap :: (a -> b) -> (f a -> f b)
```

In `Haskell` haben wir nur die Kategorie der `Haskell`-Typen (auch „Hask“ genannt), sodass wir hier nur noch Pfeile auf Pfeile mappen müssen.

Pfeile in Hask sind in `Haskell`-Funktionen, die zwei Objekte (Typen) verbinden (markiert durch einen „Pfeil“ (->)).

Für einen Funktor ergibt sich somit:

```
fmap :: (b -> a) -> (f b -> f a)
```

In `Haskell` haben wir nur die Kategorie der `Haskell`-Typen (auch „Hask“ genannt), sodass wir hier nur noch Pfeile auf Pfeile mappen müssen.

Pfeile in Hask sind in `Haskell`-Funktionen, die zwei Objekte (Typen) verbinden (markiert durch einen „Pfeil“ (->)).

Für einen Funktor ergibt sich somit:

```
fmap :: (b -> a) -> (f b -> f a)
```

Wir sehen also: Es gibt keinen Co-Funktor, weil dies wieder ein Funktor ist.

Co-Monaden

Wie sieht das nun für Monaden aus?

Wie sieht das nun für Monaden aus?

```
class Functor m => Monad m where
    return  :: a   -> m a
    bind    :: (a -> m b) -> (m a -> m b)
    join    :: m (m a) -> m a
```

Co-Monaden

Wie sieht das nun für Monaden aus?

```
class Functor w => Monad w where
  coreturn  :: a   -> w a
  cobind    :: (a -> w b) -> (w a -> w b)
  cojoin    :: w (w a) -> w a
```

Wenn die Monade `m` heisst, so heisst die Co-Monade normalerweise `w`, weil dies ein umgedrehtes `m` ist.[a]

[a] Ja.. Mathematiker sind so witzig! :)

Wie sieht das nun für Monaden aus?

```
class Functor w => Monad w where
  coreturn  :: w a -> a
  cobind    :: (w b -> a) -> (w b -> w a)
  cojoin    :: w a     -> w (w a)
```

Wir drehen die Pfeile

Wie sieht das nun für Monaden aus?

```
class Functor w => Monad w where
  extract   :: w a -> a
  (<<=)     :: (w b -> a) -> (w b -> w a)
  duplicate :: w a      -> w (w a)
```

Und benennen die Sachen um

Co-Monaden

Gut. Nun kennen wir Co-Monaden. Aber was kann man damit machen?

Co-Monaden

Gut. Nun kennen wir Co-Monaden. Aber was kann man damit machen?

Bei Monaden haben wir immer davon gesprochen, dass diese einen „versteckten“ Effekt haben. Co-Monaden beschreiben somit Co-Effekte.

Co-Monaden

Gut. Nun kennen wir Co-Monaden. Aber was kann man damit machen?

Bei Monaden haben wir immer davon gesprochen, dass diese einen „versteckten“ Effekt haben. Co-Monaden beschreiben somit Co-Effekte.

Co-Effekte nennt man auch Ursachen. Somit können wir einen Algorithmus schreiben, der konkrete Ursachen berücksichtigt.

Gut. Nun kennen wir Co-Monaden. Aber was kann man damit machen?

Bei Monaden haben wir immer davon gesprochen, dass diese einen „versteckten“ Effekt haben. Co-Monaden beschreiben somit Co-Effekte.

Co-Effekte nennt man auch Ursachen. Somit können wir einen Algorithmus schreiben, der konkrete Ursachen berücksichtigt.

Klassische Beispiele sind z.B. Algorithmen, die sich ihrer Umgebung bewusst sind:

- Bildfilter, die den Wert eines Pixels gegeben der Nachbarschaft errechnen
- Graphen, die den nächsten Wert eines Knotens abhängig von Nachbarknoten machen
- Streams mit Focus

Beispiel:

```
data Zipper a = Zipper [a] a [a]

z = Zipper [1,2] 3 [4]
z            -- |1 2 >3< 4|
extract z    -- 3
duplicate z -- ||>1< 2 3 4| |1 >2< 3 4| >|1 2 >3< 4|< |1 2 3 >4<||

f :: Zipper Int -> Int
f (Zipper l a r) = l' + a + r'
  where
    l' = if l == [] then 0 else last l
    r' = if r == [] then 0 else head r

z =>> f      -- |3 6 >9< 7|
```

Co-Monaden

Weitere gängige Co-Monaden:

- Store (Co-State)
- Env (Co-Reader)
- Traced (Co-Writer)
- CoFree

Free und CoFree

Free und CoFree

Abstrakt gesprochen ist ein Free-Irgendwas in der Mathematik etwas, was genau die Minimalansprüche erfüllt, aber nichts darüber hinaus.

Free und CoFree

Abstrakt gesprochen ist ein Free-Irgendwas in der Mathematik etwas, was genau die Minimalansprüche erfüllt, aber nichts darüber hinaus.

Free ist die freie Monade und CoFree ist die freie Co-Monade - also benutzen beide nur die Funktor-Funktionalität um das Minimum erweitert.

Free und CoFree

Abstrakt gesprochen ist ein Free-Irgendwas in der Mathematik etwas, was genau die Minimalansprüche erfüllt, aber nichts darüber hinaus.

Free ist die freie Monade und CoFree ist die freie Co-Monade - also benutzen beide nur die Funktor-Funktionalität um das Minimum erweitert.

```
data Free f a = Pure a
              | Free (f (Free f a))
```

Free und CoFree

Abstrakt gesprochen ist ein Free-Irgendwas in der Mathematik etwas, was genau die Minimalansprüche erfüllt, aber nichts darüber hinaus.

Free ist die freie Monade und CoFree ist die freie Co-Monade - also benutzen beide nur die Funktor-Funktionalität um das Minimum erweitert.

```
data Free f a = Pure a
              | Free (f (Free f a))

instance Functor f => Monad (Free f) where
  return x       = Pure x
  (Pure x) >>= f = f x
  (Free x) >>= f = Free (fmap (>>= f) x)
```

Abstrakt gesprochen ist ein Free-Irgendwas in der Mathematik etwas, was genau die Minimalansprüche erfüllt, aber nichts darüber hinaus.

Free ist die freie Monade und CoFree ist die freie Co-Monade - also benutzen beide nur die Funktor-Funktionalität um das Minimum erweitert.

```
data Free f a = Pure a
              | Free (f (Free f a))

instance Functor f => Monad (Free f) where
  return x       = Pure x
  (Pure x) >>= f = f x
  (Free x) >>= f = Free (fmap (>>= f) x)
```

Bind ist hier nichts anderes als „wende `fmap` an, wenn du im Funktor bist, wende die Funktion an im `Pure`-Fall“.

Free und CoFree

Abstrakt gesprochen ist ein Free-Irgendwas in der Mathematik etwas, was genau die Minimalansprüche erfüllt, aber nichts darüber hinaus.

Free ist die freie Monade und CoFree ist die freie Co-Monade - also benutzen beide nur die Funktor-Funktionalität um das Minimum erweitert.

```
data Free f a = Pure a
              | Free (f (Free f a))

instance Functor f => Monad (Free f) where
  return x       = Pure x
  (Pure x) >>= f = f x
  (Free x) >>= f = Free (fmap (>>= f) x)
```

Bind ist hier nichts anderes als „wende `fmap` an, wenn du im Funktor bist, wende die Funktion an im `Pure`-Fall“.

Der Funktor-Fall ist sehr ähnlich zu dem bereits bekannten `Fix`, allerdings erlaubt uns der `Pure`-Fall jederzeit abzubrechen, wenn wir die Struktur abbauen wollen.

Verglichen mit

```
data Fix f = Fix (f (Fix f))

unfix :: Fix f     -> f (Fix f)
unfix (Fix f) = f

fix   :: f (Fix f) -> Fix f
fix f = Fix f
```

Verglichen mit

```
data Fix f = Fix (f (Fix f))

unfix :: Fix f      -> f (Fix f)
unfix (Fix f) = f

fix   :: f (Fix f) -> Fix f
fix f = Fix f
```

```
data Free f a = Pure a
              | Free (f (Free f a)

free   :: f (Free f a) -> Free f a
free f = Free f

unfree :: Free f a     -> f (Free f a)
unfree (Free f) = f
unfree (Pure a) = error "kaboom"
```

stossen wir bei der Definition von `unfree` auf Probleme. Somit können wir einen Fixpunkt nur noch auf-, aber nicht mehr Abbauen.

Genau das Gegenteil (Überraschung!) passiert bei der Definition der CoFree Co-Monade. Wo die Monade noch eine Summe war, ist die Co-Monade ein Produkt:

```
data CoFree f a = (:<) a (CoFree (f (CoFree f a)))
data CoFree f a = a :< CoFree (f (CoFree f a))
```

Genau das Gegenteil (Überraschung!) passiert bei der Definition der CoFree Co-Monade. Wo die Monade noch eine Summe war, ist die Co-Monade ein Produkt:

```
data CoFree f a = (:<) a (CoFree (f (CoFree f a)))
data CoFree f a = a :< CoFree (f (CoFree f a))

instance Functor f => Comonad (CoFree f) where
  extract (a :< _) = a
  duplicate c@(_ :< fs) = c :< (fmap duplicate fs)
```

Genau das Gegenteil (Überraschung!) passiert bei der Definition der CoFree Co-Monade. Wo die Monade noch eine Summe war, ist die Co-Monade ein Produkt:

```
data CoFree f a = (:<) a (CoFree (f (CoFree f a)))
data CoFree f a = a :< CoFree (f (CoFree f a))
```

```
instance Functor f => Comonad (CoFree f) where
  extract (a :< _) = a
  duplicate c@(_ :< fs) = c :< (fmap duplicate fs)
```

`extract` holt uns hier unseren Fokus heraus und `duplicate` setzt den Fokus auf das aktuelle und reicht das duplizieren mittels `fmap` durch.

Verglichen mit

```
data Fix f = Fix (f (Fix f))

unfix :: Fix f      -> f (Fix f)
unfix (Fix f) = f

fix   :: f (Fix f) -> Fix f
fix f = Fix f
```

Verglichen mit

```
data Fix f = Fix (f (Fix f))

unfix :: Fix f     -> f (Fix f)
unfix (Fix f) = f

fix   :: f (Fix f) -> Fix f
fix f = Fix f

data CoFree f a = a :< CoFree (f (CoFree f a))

uncofree :: CoFree f a     -> f (CoFree f a)
uncofree (_ :< c) = c

cofree   :: f (CoFree f a) -> CoFree f a
cofree c = error "kaboom" :< c
```

stossen wir bei der Definition von `cofree` auf Probleme. Somit können wir einen Fixpunkt nur noch ab-, aber nicht mehr Aufbauen.

Wozu brauchen wir das überhaupt?

[4] Vollständiger Post: http://dlaing.org/cofun/posts/free_and_cofree.html

[5] Vortrag dazu: https://yow.eventer.com/yow-lambda-jam-2015-1305/cofun-with-cofree-comonads-by-david-laing-1891

Wozu brauchen wir das überhaupt?

Aufgrund der Zeit reißen wir auch hier nur die Möglichkeiten an.[4,5]

[4] Vollständiger Post: http://dlaing.org/cofun/posts/free_and_cofree.html

[5] Vortrag dazu: https://yow.eventer.com/yow-lambda-jam-2015-1305/cofun-with-cofree-comonads-by-david-laing-1891

Wozu brauchen wir das überhaupt?

Aufgrund der Zeit reißen wir auch hier nur die Möglichkeiten an.[4,5]
Gegeben eine „Programmiersprache“

```
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  | Total (Int -> k)
```

[4] Vollständiger Post: http://dlaing.org/cofun/posts/free_and_cofree.html

[5] Vortrag dazu: https://yow.eventer.com/yow-lambda-jam-2015-1305/cofun-with-cofree-comonads-by-david-laing-1891

Wozu brauchen wir das überhaupt?

Aufgrund der Zeit reißen wir auch hier nur die Möglichkeiten an.[4,5]
Gegeben eine „Programmiersprache“

```
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  | Total (Int -> k)
```

können wir mittels

```
type Adder a = Free AdderF a

add :: Int -> Adder Bool
add x = liftF (Add x id)

clear :: Adder ()
clear = liftF (Clear ())

total :: Adder Int
total = liftF (Total id)
```

[4] Vollständiger Post: http://dlaing.org/cofun/posts/free_and_cofree.html
[5] Vortrag dazu: https://yow.eventer.com/yow-lambda-jam-2015-1305/cofun-with-cofree-comonads-by-david-laing-1891

Wozu brauchen wir das überhaupt?

Aufgrund der Zeit reißen wir auch hier nur die Möglichkeiten an.[4,5]
Gegeben eine „Programmiersprache“

```
data AdderF k =
    Add Int (Bool -> k)
  | Clear k
  | Total (Int -> k)
```

können wir mittels

```
type Adder a = Free AdderF a

add :: Int -> Adder Bool
add x = liftF (Add x id)

clear :: Adder ()
clear = liftF (Clear ())

total :: Adder Int
total = liftF (Total id)
```

eine Monade definieren und direkt benutzen:

[4] Vollständiger Post: http://dlaing.org/cofun/posts/free_and_cofree.html
[5] Vortrag dazu: https://yow.eventer.com/yow-lambda-jam-2015-1305/cofun-with-cofree-comonads-by-david-laing-1891

```haskell
findLimit :: Adder Int
findLimit = do
  t <- total       -- save counter
  clear            -- set to 0
  r <- execStateT findLimit' 0 --find overflow
  clear            -- set to 0
  _ <- add t       -- restore counter
  return r         -- return result

findLimit' :: StateT Int Adder ()
findLimit' = do
  r <- lift $ add 1 -- add 1
  when r $ do
    -- if no overflow, add to our state counter ...
    modify (+ 1)
    -- and continue
    findLimit'
```

Allerdings können wir nicht nur die Freie Monade, sondern auch die Freie Co-Monade über dem entsprechenden Funktor dieser Sprache definieren:

```
data CoAdderF k = CoAdderF
                { addF   :: Int -> (Bool, k)
                , clearF :: k
                , totalF :: (Int, k)

type CoAdder a = CoFree CoAdderF a
```

Allerdings können wir nicht nur die Freie Monade, sondern auch die Freie Co-Monade über dem entsprechenden Funktor dieser Sprache definieren:

```
data CoAdderF k = CoAdderF
                { addF   :: Int -> (Bool, k)
                , clearF :: k
                , totalF :: (Int, k)

type CoAdder a = CoFree CoAdderF a
```

allerdings benötigen wir noch Startwerte um anzufangen:

```
type Limit = Int
type Count = Int

runCoAdder :: Limit -> Count -> CoAdder (Limit, Count)
runCoAdder l c = coiter next (l,c)
  where
    next w = CoAdderF (coAdd w) (coClear w) (coTotal w)
```

```haskell
coClear :: (Limit, Count) -> (Limit, Count)
coClear (limit, count) = (limit, 0)

coTotal :: (Limit, Count) -> (Int, (Limit, Count))
coTotal (limit, count) = (count, (limit, count))

coAdd :: (Limit, Count) -> Int -> (Bool, (Limit, Count))
coAdd (limit, count) x = (test, (limit, next))
  where
    count' = count + x                        -- calculate new value
    test = count' <= limit                    -- test if over limit
    next = if test then count' else count -- return correct value
```

Da Free und CoFree dual sind, können wir sie miteinander paaren:

```
class (Functor f, Functor g) => Pairing f g where
  pair (a -> b -> r) -> f a -> g b -> r

instance Pairing f g => Pairing (Cofree f) (Free g) where
  pair p (a :< _ ) (Pure x)  = p a x
  pair p (_ :< fs) (Free gs) = pair (pair p) fs gs

instance Pairing CoAdderF AdderF where
  pair f (CoAdderF a _ _) (Add x k) = pair f (a x) k
  pair f (CoAdderF _ c _) (Clear k) = f c k
  pair f (CoAdderF _ _ t) (Total k) = pair f t k
```

Dies erlaubt uns z.B. das eben geschriebene Programm auf allen Interpretern auszuführen und deren Limit herauszufinden:

```
runLimit :: CoAdder a -> Int
runLimit w = pair (\_ b -> b) w findLimit
```

Dies erlaubt uns z.B. das eben geschriebene Programm auf allen Interpretern auszuführen und deren Limit herauszufinden:

```
runLimit :: CoAdder a -> Int
runLimit w = pair (\_ b -> b) w findLimit
```

Ebenso können wir auch eine Eigenschaft definieren und so z.B. unseren Interpreter mit Quickcheck testen:

```
testLimit :: Int -> Bool
testLimit x = runLimit (mkCoAdder x 0) == x
```

Im Gegensatz zu normalen Monaden und Co-Monaden erlauben uns Freie Monaden und Freie Co-Monaden eine Verknüpfung. Man kann definieren:

```
(:+:) :: Free   f a -> Free   f a -> Free   f a
(:*:) :: CoFree f a -> CoFree f a -> CoFree f a
```

und somit die Sprache und den Interpreter Befehl für Befehl zusammenstecken.

Weitere Technologien

Weitere Technologien

Dinge, die leider nicht mehr in die Vorlesung gepasst haben, wir aber kurz noch erwähnen möchten:

Weitere Technologien

Dinge, die leider nicht mehr in die Vorlesung gepasst haben, wir aber kurz noch erwähnen möchten:

Recursion-Schemes
: Wir konnten die Techniken hier nur anreissen. Früher gab es die Vorlesung „Algebraic Dynamic Programming“, in der ein ganzes Semester `hylo` und `dyna` gewidmet waren.

Weitere Technologien

Dinge, die leider nicht mehr in die Vorlesung gepasst haben, wir aber kurz noch erwähnen möchten:

Recursion-Schemes
: Wir konnten die Techniken hier nur anreissen. Früher gab es die Vorlesung „Algebraic Dynamic Programming“, in der ein ganzes Semester `hylo` und `dyna` gewidmet waren.

Cache-Oblivious Datastructures
: Wir hatten hin und wieder mal Caches erwähnt und dass es wichtig ist diese für eine gute Laufzeit auszunutzen. CODS machen dies „von alleine“, indem sie sich automatisch auf den passenden Cache optimieren.

Weitere Technologien

Dinge, die leider nicht mehr in die Vorlesung gepasst haben, wir aber kurz noch erwähnen möchten:

Recursion-Schemes
: Wir konnten die Techniken hier nur anreissen. Früher gab es die Vorlesung „Algebraic Dynamic Programming“, in der ein ganzes Semester `hylo` und `dyna` gewidmet waren.

Cache-Oblivious Datastructures
: Wir hatten hin und wieder mal Caches erwähnt und dass es wichtig ist diese für eine gute Laufzeit auszunutzen. CODS machen dies „von alleine“, indem sie sich automatisch auf den passenden Cache optimieren.

Succinct Datastructures
: Succinct DS sind Datenstrukturen, die nur Speicher in der Größenordnung der (Shannon-)Entropie benötigen.

Weitere Technologien

Dinge, die leider nicht mehr in die Vorlesung gepasst haben, wir aber kurz noch erwähnen möchten:

Recursion-Schemes
: Wir konnten die Techniken hier nur anreissen. Früher gab es die Vorlesung „Algebraic Dynamic Programming“, in der ein ganzes Semester `hylo` und `dyna` gewidmet waren.

Cache-Oblivious Datastructures
: Wir hatten hin und wieder mal Caches erwähnt und dass es wichtig ist diese für eine gute Laufzeit auszunutzen. CODS machen dies „von alleine“, indem sie sich automatisch auf den passenden Cache optimieren.

Succinct Datastructures
: Succinct DS sind Datenstrukturen, die nur Speicher in der Größenordnung der (Shannon-)Entropie benötigen.

2-3-Finger-Trees
: Einmal kurz angeklungen sind 2-3-Finger-Trees als Ersatz für Listen (Definiert in Data.Sequence). Bessere Laufzeit und in alle Punkten gleich schnell oder schneller.

Codensity

Die Codensity-Monade erlaubt für jeden Monoiden das Linksassoziative `mappend` in ein Rechtsassoziatives umzuwandeln, indem man lediglich die Typen ändert. Insbesondere bei Laufzeitunterschieden durch falsche Klammerung kann dies die Rettung sein.

Codensity

: Die Codensity-Monade erlaubt für jeden Monoiden das Linksassoziative `mappend` in ein Rechtsassoziatives umzuwandeln, indem man lediglich die Typen ändert. Insbesondere bei Laufzeitunterschieden durch falsche Klammerung kann dies die Rettung sein.

Optics

: Wir haben zwar Lens angesprochen, allerdings nichts zu Prisms und nur wenig zu Traversals gesagt. Auch dort liegen noch viele Vorlesungen vergraben.

Codensity

Die Codensity-Monade erlaubt für jeden Monoiden das Linksassoziative `mappend` in ein Rechtsassoziatives umzuwandeln, indem man lediglich die Typen ändert. Insbesondere bei Laufzeitunterschieden durch falsche Klammerung kann dies die Rettung sein.

Optics

Wir haben zwar Lens angesprochen, allerdings nichts zu Prisms und nur wenig zu Traversals gesagt. Auch dort liegen noch viele Vorlesungen vergraben.

GPU-Computing

Man kann mittels `accelerate` aus einer speziellen Monade direkt CUDA-Code generieren und somit auf der Grafikkarte rechnen.

Codensity
: Die Codensity-Monade erlaubt für jeden Monoiden das Linksassoziative `mappend` in ein Rechtsassoziatives umzuwandeln, indem man lediglich die Typen ändert. Insbesondere bei Laufzeitunterschieden durch falsche Klammerung kann dies die Rettung sein.

Optics
: Wir haben zwar Lens angesprochen, allerdings nichts zu Prisms und nur wenig zu Traversals gesagt. Auch dort liegen noch viele Vorlesungen vergraben.

GPU-Computing
: Man kann mittels `accelerate` aus einer speziellen Monade direkt CUDA-Code generieren und somit auf der Grafikkarte rechnen.

FPGA-Computing
: Ein Subset von Haskell findet sich in der Programmiersprache $C\lambda ASH$. Dieses generiert Code für FPGAs, welche ihr z.B. in DEP programmiert.

Codensity
: Die Codensity-Monade erlaubt für jeden Monoiden das Linksassoziative `mappend` in ein Rechtsassoziatives umzuwandeln, indem man lediglich die Typen ändert. Insbesondere bei Laufzeitunterschieden durch falsche Klammerung kann dies die Rettung sein.

Optics
: Wir haben zwar Lens angesprochen, allerdings nichts zu Prisms und nur wenig zu Traversals gesagt. Auch dort liegen noch viele Vorlesungen vergraben.

GPU-Computing
: Man kann mittels `accelerate` aus einer speziellen Monade direkt CUDA-Code generieren und somit auf der Grafikkarte rechnen.

FPGA-Computing
: Ein Subset von Haskell findet sich in der Programmiersprache $C\lambda ASH$. Dieses generiert Code für FPGAs, welche ihr z.B. in DEP programmiert.

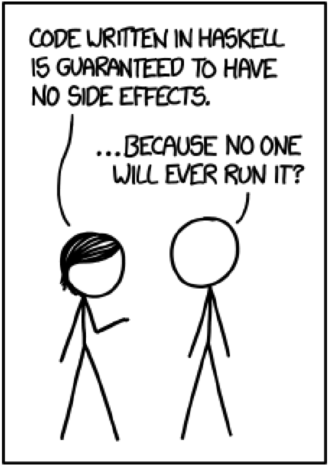
PureScript
: PureScript ist eine fast 1:1 Haskell-Implementation, die zu lesbarem und effizientem JavaScript kompiliert und so Webprogrammierung sehr einfach macht.

und vieles mehr

Viele Themen haben wir selbst in dieser Auflistung sicher vergessen...

Vielen Dank für die Teilnahme an der Vorlesung, eine gute vorlesungsfreie Zeit, viel Erfolg bei den Klausuren und Spass beim Projekt.

Jonas Betzendahl & Stefan Dresselhaus



Hovertext: „The problem with Haskell is that it's a language built on lazy evaluation and nobody's actually called for it.“

https://xkcd.com/1312/