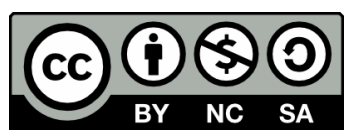
Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 2: *Typeclasses, Functions, Laziness*
Stand: 22. April 2016



Wiederholung: Typen & Typklassen

Werte und Typen

In typisierten Sprachen wie `Haskell` hat jeder *Wert* (mindestens) einen *Typen*. Wir unterscheiden zwischen der Ebene der Typen und der Werte. Beide sind es wert, genauer über sie nachzudenken.

Werte und Typen

In typisierten Sprachen wie `Haskell` hat jeder *Wert* (mindestens) einen *Typen*. Wir unterscheiden zwischen der Ebene der Typen und der Werte. Beide sind es wert, genauer über sie nachzudenken.

Die meisten Typen haben mehr als einen Wert (Bewohner), mit Ausnahme von `()` und $\bot$ (dem leeren Typen). Ein Wert kann aber auch mehr als einen Typen haben, er wird dann „polymorph“ genannt.

```haskell
--   This typechecks fine!   --
foo :: Either Bool Int
foo = Right 1
```

```haskell
bar :: Either () Int
bar = Right 1
```

Parametrisierte Typen

Polymorphismus funktioniert für Funktionen als auch für Typen. In `Haskell` können z.B. Typen auch andere Typen „mittragen“.

Die Worte Parameter und Konstruktor werden hier auf Typebene sehr ähnlich zur Werteebene verwendet.

```haskell
-- option type
data Maybe a = Nothing
             | Just a
```

Parametrisierte Typen

Polymorphismus funktioniert für Funktionen als auch für Typen. In `Haskell` können z.B. Typen auch andere Typen „mittragen“.

Die Worte Parameter und Konstruktor werden hier auf Typebene sehr ähnlich zur Werteebene verwendet.

```haskell
-- option type
data Maybe a = Nothing
             | Just a
```

Wichtig zu verstehen ist hier, dass `Maybe` alleine noch kein vollständiger Typ ist, `Maybe Int` oder `Maybe String` hingegen schon.

Typklassen

Wenn wir polymorphe Typen so einschränken wollen, dass der Typ polymorph bleiben kann (solange gewisse Funktionen definiert sind) können wir *Typklassen* benutzen.

Einige der wichtigsten Typklassen haben wir schon kennen gelernt:

- `Eq: (==)`, Gleichheit
- `Ord: (<=)`, Lineare Anordnung
- `Show: (show)`, Anzeigbarkeit
- `Num: (+), (*), ...`, Zahlen

Ord for Bools

Wir erinnern uns an die Typklassen `Eq` (Typen mit Gleichheit) und `Ord` (Typen, die linear anzuordnen sind).

```
class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  (<=) :: a -> a -> Bool
```

Ord for Bools

Wir erinnern uns an die Typklassen `Eq` (Typen mit Gleichheit) und `Ord` (Typen, die linear anzuordnen sind).

```
class Eq a where
  (==) :: a -> a -> Bool

class Eq a => Ord a where
  (<=) :: a -> a -> Bool
```

Hier nochmal die Instanzen für den Typen `Bool`:

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False

instance Ord Bool where
  False <= True  = True
  True  <= True  = True
  False <= False = True
  True  <= False = False
```

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Was ist ein Typparameter?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Was ist ein Typparameter?
- Was der Unterschied zwischen einem Typkonstruktor und einem Wertekonstruktor?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Was ist ein Typparameter?
- Was der Unterschied zwischen einem Typkonstruktor und einem Wertekonstruktor?
- Was ist eine Typklasse?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Was ist ein Typparameter?
- Was der Unterschied zwischen einem Typkonstruktor und einem Wertekonstruktor?
- Was ist eine Typklasse?
- Was sind die zwei wichtigen Schritte bei Typklassen?

Verständnisfragen

Ein paar Fragen, die ihr beantworten können solltet:

- Was ist ein Typparameter?
- Was der Unterschied zwischen einem Typkonstruktor und einem Wertekonstruktor?
- Was ist eine Typklasse?
- Was sind die zwei wichtigen Schritte bei Typklassen?
- Welches Problem versuchen Typklassen zu lösen?

Mehr Typklassen

Monoide in der Mathematik

In der Mathematik ist ein Monoid M eine Menge, auf der eine (abgeschlossene, binäre, assoziative) Funktion $\cdot$ definiert ist.

$$\cdot : M \times M \to M$$

$$\forall\, a, b, c \in M\,.\,(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Monoide in der Mathematik

In der Mathematik ist ein Monoid M eine Menge, auf der eine (abgeschlossene, binäre, assoziative) Funktion $\cdot$ definiert ist.

$$\cdot : M \times M \to M$$

$$\forall\, a, b, c \in M \,.\, (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Es muss außerdem ein neutrales Element $e \in M$ definiert sein:

$$\forall\, x \in M \,.\, e \cdot x = x = x \cdot e$$

.

Monoide in der Mathematik

In der Mathematik ist ein Monoid M eine Menge, auf der eine (abgeschlossene, binäre, assoziative) Funktion $\cdot$ definiert ist.

$$\cdot : M \times M \to M$$

$$\forall\, a, b, c \in M\,.\,(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Es muss außerdem ein neutrales Element $e \in M$ definiert sein:

$$\forall\, x \in M\,.\,e \cdot x = x = x \cdot e$$

.

Bekannte Monoide: $(\mathbb{N}, +, 0)$

Monoide in der Mathematik

In der Mathematik ist ein Monoid M eine Menge, auf der eine (abgeschlossene, binäre, assoziative) Funktion $\cdot$ definiert ist.

$$\cdot : M \times M \to M$$

$$\forall\, a, b, c \in M\,.\,(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Es muss außerdem ein neutrales Element $e \in M$ definiert sein:

$$\forall\, x \in M\,.\,e \cdot x = x = x \cdot e$$

.

Bekannte Monoide: $(\mathbb{N}, +, 0)$, $(\mathbb{R}, \cdot, 1)$

Monoide in der Mathematik

In der Mathematik ist ein Monoid M eine Menge, auf der eine (abgeschlossene, binäre, assoziative) Funktion $\cdot$ definiert ist.

$$\cdot : M \times M \to M$$

$$\forall\, a, b, c \in M \,.\, (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

Es muss außerdem ein neutrales Element $e \in M$ definiert sein:

$$\forall\, x \in M \,.\, e \cdot x = x = x \cdot e$$

.

Bekannte Monoide: $(\mathbb{N}, +, 0)$, $(\mathbb{R}, \cdot, 1)$, (Strings, concat, ε) ...

Monoide in Haskell

Diese Struktur können wir auch in `Haskell` abbilden:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a  -- (<>) for infix usage
```

Monoide in Haskell

Diese Struktur können wir auch in `Haskell` abbilden:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a  -- (<>) for infix usage
```

Das erlaubt uns, mathematisches Wissen für unseren Code zu verwenden; was wir über Monoide in der Mathematik wissen, gilt auch für Monoide in `Haskell`. Das könnte zum Beispiel zu Optimierungen auf der Compilerebene führen.

Monoidengesetze

Die Gesetze für Monoide aus der Mathematik können direkt in `Data.Monoid` übertragen werden:

```
-- Instances should satisfy the following laws:
--  * mempty <> x   == x
--  * x <> mempty   == x
--  * x <> (y <> z) == (x <> y) <> z
```

Monoidengesetze

Die Gesetze für Monoide aus der Mathematik können direkt in `Data.Monoid` übertragen werden:

```
-- Instances should satisfy the following laws:
--  * mempty <> x   == x
--  * x <> mempty   == x
--  * x <> (y <> z) == (x <> y) <> z
```

Theoretisch könnte natürlich eine Instanz sich dazu entscheiden, diese Gesetze zu ignorieren. Es wird nicht vom Typsystem überprüft, dass sie eingehalten werden.

Instanzen

Instanzen für `Monoid` in Haskell sind zum Beispiel die folgenden:

```haskell
-- General lists, also works for String
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Instanzen

Instanzen für `Monoid` in Haskell sind zum Beispiel die folgenden:

```
-- General lists, also works for String
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Wir können auch allgemeinere Instanzen schreiben:

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing <> m = m
  m <> Nothing = m
  Just m1 <> Just m2 = Just (m1 <> m2)
```

Functor

Viele der uns bekannten parametrisierte Typen (`[]`, `Maybe`, `...`) haben außerdem Instanzen für die Typklasse `Functor`.
Oft hilft es, sich Funktoren als Container vorzustellen.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor

Viele der uns bekannten parametrisierte Typen (`[]`, `Maybe`, `...`) haben außerdem Instanzen für die Typklasse `Functor`.
Oft hilft es, sich Funktoren als Container vorzustellen.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Die Funktion `fmap` nimmt eine Funktion (`(a -> b)`) als Parameter und „hebt sie in den Funktor“ (`(f a -> f b)`).

Functor

Viele der uns bekannten parametrisierte Typen (`[]`, `Maybe`, `...`) haben außerdem Instanzen für die Typklasse `Functor`.
Oft hilft es, sich Funktoren als Container vorzustellen.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Die Funktion `fmap` nimmt eine Funktion (`(a -> b)`) als Parameter und „hebt sie in den Funktor“ (`(f a -> f b)`).

Wichtig: Wir übergeben dem `f` hier an zwei Stellen einen Typen als Parameter (z.B. das `a` in `f a`). Es ist also ein *Typkonstruktor*, kein fertiger Typ.

Funktorgesetze

Natürlich gibt es auch für `Functor` „ungeschriebene“ Gesetze, die alle Instanzen berücksichtigen sollten.

```
-- Instances should satisfy the following laws:
--   *  fmap id        ==  id
--   *  fmap (f . g)   ==  fmap f . fmap g
```

Funktorgesetze

Natürlich gibt es auch für `Functor` „ungeschriebene“ Gesetze, die alle Instanzen berücksichtigen sollten.

```
-- Instances should satisfy the following laws:
--   *  fmap id        ==  id
--   *  fmap (f . g)   ==  fmap f . fmap g
```

Streng genommen folgt das zweite Gesetz aus dem ersten, wird aber für Klarheit und Verständnis meist extra erwähnt.

Für den Beweis siehe:
`www.schoolofhaskell.com/user/edwardk/snippets/fmap`

Instanzen (I)

Die Instanz für Listen wird euch sicherlich bekannt vorkommen:

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = (f x) : (fmap f xs)
```

Instanzen (I)

Die Instanz für Listen wird euch sicherlich bekannt vorkommen:

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = (f x) : (fmap f xs)
```

Es ist genau die Implementation von `map`, wie wir sie aus **A&D** kennen. Wenn wir das schon in scope haben, können wir auch einfach sagen:

```
instance Functor [] where
  fmap = map
```

Instanzen (II)

Der Optiontype `Maybe` hat auch eine Funktorinstanz. Welche?

Zur Erinnerung:

```
data Maybe a = Nothing
             | Just a
```

Instanzen (II)

Der Optiontype `Maybe` hat auch eine Funktorinstanz. Welche?

Zur Erinnerung:

```
data Maybe a = Nothing
             | Just a
```

Lösung:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

Instanzen (II)

Die Instanz für `Identity` ist relativ simpel, aber eventuell nicht sofort offensichtlich. Wie würde sie aussehen?

Zur Erinnerung:

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

Instanzen (II)

Die Instanz für `Identity` ist relativ simpel, aber eventuell nicht sofort offensichtlich. Wie würde sie aussehen?

Zur Erinnerung:

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

Lösung:

```
instance Functor Identity where
  fmap f i = Identity (f (runIdentity i))
```

Beispiel: Functor

Ein beliebtes Beispiel für einen `Functor`, der nicht in der Prelude liegt, ist `Tree`. Wie würde die Funktorinstanz aussehen?

```
-- simple binary trees
data Tree a = Nil
            | Leaf a
            | Branch (Tree a) (Tree a)
```

Beispiel: Functor

Ein beliebtes Beispiel für einen `Functor`, der nicht in der Prelude liegt, ist `Tree`. Wie würde die Funktorinstanz aussehen?

```
-- simple binary trees
data Tree a = Nil
            | Leaf a
            | Branch (Tree a) (Tree a)
```

Hier ist die rekursive Implementation:

```
instance Functor Tree where
  fmap _ Nil           = Nil
  fmap f (Leaf a)      = Leaf (f a)
  fmap f (Branch l r) = Branch (fmap f l) (fmap f r)
```

further functor reading

Die `Functor`-Typklasse wird uns noch häufiger begegnen und bildet in den nächsten Vorlesungen das Fundament für eine extrem wichtige Hierarchie von Typklassen. Es lohnt sich hier besonders, das Konzept gut zu verstehen.

Das Kapitel in LYAH:
`http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass`

Functors etc. in pictures:
`http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

Funktionen in `Haskell`

higher-order functions

Funktionen sind so genannte "first-class citizens" in `Haskell`. Das bedeutet, dass sie auch wie andere Werte Typen haben und umhergereicht werden können. In vielen klassischen Sprachen ist das nur beschränkt und erst seit Kurzem möglich.

higher-order functions

Funktionen sind so genannte "first-class citizens" in `Haskell`. Das bedeutet, dass sie auch wie andere Werte Typen haben und umhergereicht werden können. In vielen klassischen Sprachen ist das nur beschränkt und erst seit Kurzem möglich.

Das ermöglicht die Weiterverwendung von Funktionen und die Existenz von so genannten „Funktionen höherer Ordnung“.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter f (x:xs)
  | f x         = x : filter f xs
  | otherwise   =     filter f xs
```

function composition (I)

Oft will man in der Praxis einen Wert nicht nur an eine Funktion geben, sondern das Ergebnis dann auch direkt gleich an die nächste. Hier wird *Funktionsverkettung* (Operator `(.)`) nützlich.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

function composition (I)

Oft will man in der Praxis einen Wert nicht nur an eine Funktion geben, sondern das Ergebnis dann auch direkt gleich an die nächste. Hier wird *Funktionsverkettung* (Operator `(.)`) nützlich.

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Verknüpft sind Funktionen nicht nur übersichtlicher, sondern oft auch effizienter, weil sich Zwischenergebnisse nicht gemerkt werden müssen.

function composition (II)

Hier sind zwei Programme, die das gleiche tun. Aber `rev'` ist etwas übersichtlicher.

```
rev :: String -> String
rev str = unwords (reverse (words str))

rev' :: String -> String
rev' str = (unwords . reverse . words) str
```

function composition (II)

Hier sind zwei Programme, die das gleiche tun. Aber `rev'` ist etwas übersichtlicher.

```
rev :: String -> String
rev str = unwords (reverse (words str))

rev' :: String -> String
rev' str = (unwords . reverse . words) str
```

Tatsächlich können wir noch einen Schritt weiter gehen, zum so genannten "pointfree style":

```
rev'' :: String -> String
rev'' = unwords . reverse . words
```

Curryfizierung (I)

Funktionen in Haskell sind *gecurried* (nach Haskell Curry). Das bedeutet, das nicht alle Parameter vorliegen müssen, sondern auch *partial application* (Rückgabe von Funktion statt Wert) möglich ist.

```
foo_curried   :: a -> b -> c -> d
foo_uncurried :: (a, b, c) -> d
```

Curryfizierung (I)

Funktionen in Haskell sind *gecurried* (nach Haskell Curry). Das bedeutet, das nicht alle Parameter vorliegen müssen, sondern auch *partial application* (Rückgabe von Funktion statt Wert) möglich ist.

```
foo_curried   :: a -> b -> c -> d
foo_uncurried :: (a, b, c) -> d
```

In Signaturen werden oft Klammern weg gelassen. Folgendes sind alles valide Lesearten vom Typ von `foo`:

```
foo :: a -> b -> c -> d
foo :: a -> b -> (c -> d)
foo :: a -> (b -> (c -> d))
```

Curryfizierung (II)

Was wir über Curryfizierung gelernt haben, bringt uns zum Fakt:

Alle Funktionen in Haskell nehmen *genau* einen Parameter!

Funktionen, die „mehr als einen“ Parameter zu nehmen scheinen, nehmen tatsächlich nur einen und geben eine Funktion zurück, die einen Parameter weniger erwartet.

Curryfizierung (II)

Was wir über Curryfizierung gelernt haben, bringt uns zum Fakt:

> Alle Funktionen in Haskell nehmen *genau* einen Parameter!

Funktionen, die „mehr als einen“ Parameter zu nehmen scheinen, nehmen tatsächlich nur einen und geben eine Funktion zurück, die einen Parameter weniger erwartet.

Deal: Wir können weiterhin wie gewohnt sprechen („Die Funktion `filter` nimmt zwei Parameter. . . “), aber nur solange das nicht dem Verständnis im Weg steht.

Lazy Evaluation

"Garbage collection means that the programmer does not have to worry about the end of a value's lifetime.

Lazy evaluation means that she doesn't have to worry about the beginning of that lifetime, either."

– Doaitse Swierstra

laziness, the idea(I)

Lazy Evaluation ist eine Auswertungsstrategie, die mit dem Auswerten eines Ausdrucks wartet, bis sein Wert abgefragt wird und wiederholte Auswertungen zu vermeiden versucht.

laziness, the idea(I)

Lazy Evaluation ist eine Auswertungsstrategie, die mit dem Auswerten eines Ausdrucks wartet, bis sein Wert abgefragt wird und wiederholte Auswertungen zu vermeiden versucht.

Die Idee hinter laziness ist, unnötige Berechnungen zu vermeiden und somit Laufzeit einzusparen.

In der Praxis bedeutet das, dass `Haskell` Berechnungen in *thunks* speichert, die später weiter ausgewertet werden können. Diese werden in einem Graphen vorgehalten und analysiert.

```haskell
let (x,y) = (length "Hello World!", 3 + 4)
-- (x,y) is now a pair of unevaluated thunks
```

lazy control flow

Die meisten Programmiersprachen haben short-circuit-Operatoren wie `(&&)` oder `(||)`, die kleinere Optimierungen erlauben.

In `Haskell` wird dieser Gedanke zu seiner logischen Schlussfolgerung weiter gedacht: Laziness by default!

lazy control flow

Die meisten Programmiersprachen haben short-circuit-Operatoren wie `(&&)` oder `(||)`, die kleinere Optimierungen erlauben.

In `Haskell` wird dieser Gedanke zu seiner logischen Schlussfolgerung weiter gedacht: Laziness by default!

```haskell
-- Lazy by default

foo :: Int -> Int -> Int
foo x _ = x * 2

ghci: foo 128 undefined
256
```

lazy control flow

Die meisten Programmiersprachen haben short-circuit-Operatoren wie `(&&)` oder `(||)`, die kleinere Optimierungen erlauben.

In `Haskell` wird dieser Gedanke zu seiner logischen Schlussfolgerung weiter gedacht: Laziness by default!

```haskell
-- Lazy by default

foo :: Int -> Int -> Int
foo x _ = x * 2

ghci: foo 128 undefined
256
```

```haskell
{-# LANGUAGE BangPatterns #-}

foo' :: Int -> Int -> Int
foo' x !y = x * 2

ghci: foo' 128 undefined
*** Exception:
      Prelude.undefined
```

infinite data structures

Laziness erlaubt uns, „unendliche“ Datenstrukturen nach einer allgemeinen Regel aufzubauen. Diese können wir dann exakt so lange auswerten, wie wir es benötigen.

Ein beliebtes Beispiel sind die Fibonacci-Zahlen:

```
-- "infinite" list of fibonnaci numbers
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

infinite data structures

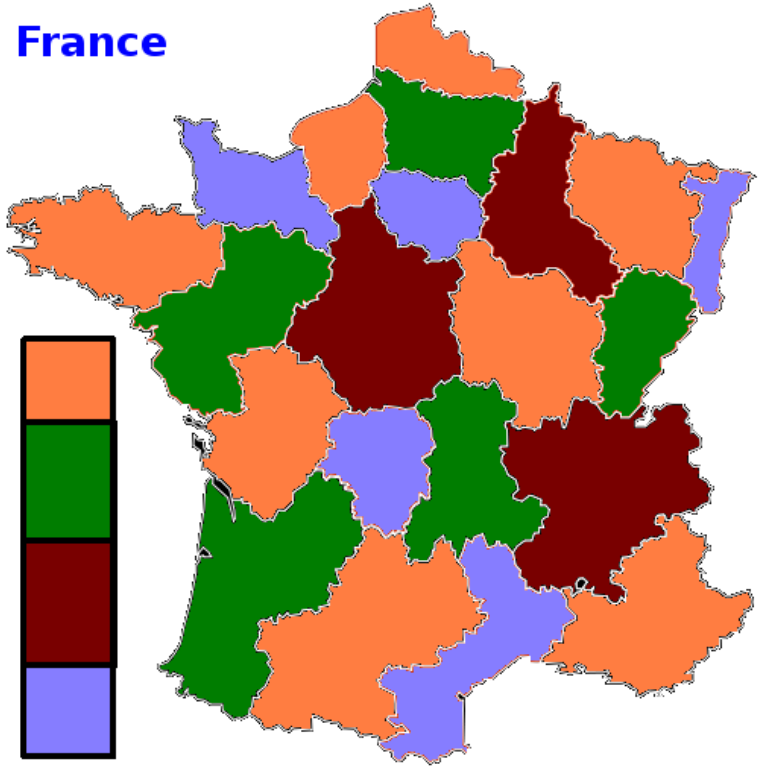
Laziness erlaubt uns, „unendliche“ Datenstrukturen nach einer allgemeinen Regel aufzubauen. Diese können wir dann exakt so lange auswerten, wie wir es benötigen.

Ein beliebtes Beispiel sind die Fibonacci-Zahlen:

```
-- "infinite" list of fibonnaci numbers
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

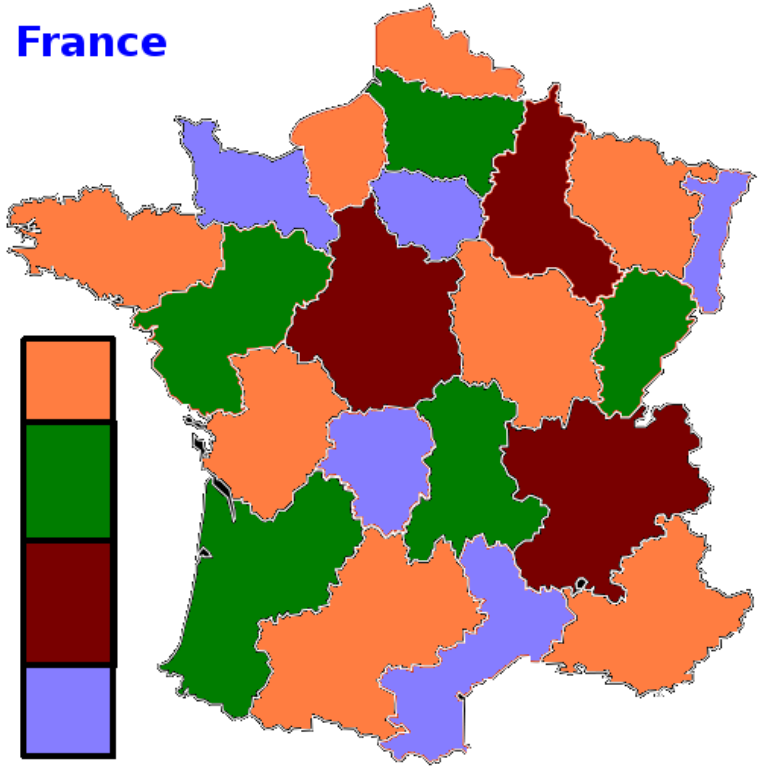
Selbstverständlich sind diese Strukturen nicht wirklich unendlich groß, sondern durch Speicher, Rechenpower etc. begrenzt.

infinite data structures by example (I)



Nach dem Vierfarbensatz kann jede (!) Landkarte mit max. vier Farben eingefärbt werden, sodass keine zwei Regionen, die sich eine Grenze teilen, die gleichen Farben haben.

infinite data structures by example (I)



Nach dem Vierfarbensatz kann jede (!) Landkarte mit max. vier Farben eingefärbt werden, sodass keine zwei Regionen, die sich eine Grenze teilen, die gleichen Farben haben.

Fun Fact: Der Vierfarbensatz war einer der ersten Sätze, die computergestützt bewiesen wurden.

infinite data structures by example (II)

Angenommen, wir haben bereits eine Funktion, die uns für eine gegebene Karte alle validen Einfärbungen berechnet.

```
solutions :: Map -> [ColourMap]
solutions = ... -- cleverness goes here!
```

infinite data structures by example (II)

Angenommen, wir haben bereits eine Funktion, die uns für eine gegebene Karte alle validen Einfärbungen berechnet.

```
solutions :: Map -> [ColourMap]
solutions = ... -- cleverness goes here!
```

Dann können wir diese Datenstruktur anwenden, egal ob wir nur irgendein Ergebnis (`head solutions`) oder nur bestimmte Ergebnisse nach einem Muster (`filter foobar solutions`) haben wollen.
Die Datenstruktur wird nicht weiter ausgewertet, als benötigt.
Wenn wir nur eine Karte haben wollen wird auch nur die erste berechnet.

problems with laziness

In manchen Situationen kann Lazy Evaluation auch *gerade nicht* das sein, was ihr wollt. Einige davon sind:

problems with laziness

In manchen Situationen kann Lazy Evaluation auch *gerade nicht* das sein, was ihr wollt. Einige davon sind:

- File-`IO` (Dateien schließen, bevor Text gelsesen wird . . .)

problems with laziness

In manchen Situationen kann Lazy Evaluation auch *gerade nicht* das sein, was ihr wollt. Einige davon sind:

- File-`IO` (Dateien schließen, bevor Text gelsesen wird . . .)
- Performance (trotz Compileranalyse, auch Overhead)

problems with laziness

In manchen Situationen kann Lazy Evaluation auch *gerade nicht* das sein, was ihr wollt. Einige davon sind:

- File-`IO` (Dateien schließen, bevor Text gelsesen wird . . .)
- Performance (trotz Compileranalyse, auch Overhead)
- Exception Handling

problems with laziness

In manchen Situationen kann Lazy Evaluation auch *gerade nicht* das sein, was ihr wollt. Einige davon sind:

- File-`IO` (Dateien schließen, bevor Text gelsesen wird . . .)
- Performance (trotz Compileranalyse, auch Overhead)
- Exception Handling
- . . .

problems with laziness

In manchen Situationen kann Lazy Evaluation auch *gerade nicht* das sein, was ihr wollt. Einige davon sind:

- File-`IO` (Dateien schließen, bevor Text gelsesen wird . . .)
- Performance (trotz Compileranalyse, auch Overhead)
- Exception Handling
- . . .

Es gibt Möglichkeiten, laziness by default zu umgehen (z.B. *BangPatterns*). Wer hohe Anforderungen an die Performance seines Codes hat, kommt da quasi nicht drum herum.

Die Feinheiten benötigen einiges an Können. Es herrscht nach wie vor Uneinigkeit darüber, was der Standard sein sollte.

Purity

referential transparency

Definition: Wir sagen, ein Ausdruck ist *referentially transparent* oder kurz *pur*, wenn wir ihn bei jedem Vorkommen durch seinen Rückgabewert ersetzen können, ohne dass sich das Verhalten des Programms ändert.

referential transparency

Definition: Wir sagen, ein Ausdruck ist *referentially transparent* oder kurz *pur*, wenn wir ihn bei jedem Vorkommen durch seinen Rückgabewert ersetzen können, ohne dass sich das Verhalten des Programms ändert.

Wir können zum Beispiel in `Haskell` jederzeit `(even 42)` durch `True` ersetzen, ändern tut sich dadurch nichts. Also ist der Ausdruck pur.
Ein weiterer purer Ausdruck ist `(filter (>0) [-1337..1337])`.

Seiteneffekte (I)

In der Mathematik sind alle Funktionen pur. Sobald wir jedoch programmieren, können Funktionen allerdings *Seiteneffekte* haben und dadurch ihre referential transparency verlieren.

Seiteneffekte (I)

In der Mathematik sind alle Funktionen pur. Sobald wir jedoch programmieren, können Funktionen allerdings *Seiteneffekte* haben und dadurch ihre referential transparency verlieren.

Definition: Wir sagen, dass ein Ausdruck einen *Seiteneffekt* (side effect) hat, wenn er einen Zustand (state) (ob global oder lokal) ändert oder Auswirkungen auf die Außenwelt hat (Dateien löschen, PC herunterfahren, ...).

Seiteneffekte zu haben und referential transparency schließen sich gegenseitig aus!

Seiteneffekte (II)

Ein einfaches Beispiel aus der Programmiersprache Java:

```
public static int fuenf()
{
    System.out.println("Seiteneffekt!");
    return 5;
}
```

Seiteneffekte (II)

Ein einfaches Beispiel aus der Programmiersprache Java:

```
public static int fuenf()
{
    System.out.println("Seiteneffekt!");
    return 5;
}
```

Die folgenden Ausdrücke haben somit alle eine unterschiedliche Bedeutung, auch wenn jedes Mal 10 zurück gegeben wird.

```
return 5 + 5;
return 5 + fuenf();
return fuenf() + fuenf();
```

Seiteneffekte (III)

In `Haskell` funktioniert das hingegen so nicht. Deswegen wird `Haskell` auch als eine *pure* Programmiersprache bezeichnet.

```
fuenf :: Int
fuenf = putStrLn "Seiteneffekt!" >> 5
```

Stattdessen kriegen wir einen Typfehler:

```
Couldn't match expected type 'Int'
            with actual type 'IO b0'
In the expression: putStrLn "Seiteneffekt!" >> 5
```

xkcd by Randall Munroe, CC-BY-NC
https://xkcd.com/1312/

The good...

Die Vorteile von purity sind enorm! Dadurch, dass wir Seiteneffekte kontrolliert halten, erhalten wir Typsicherheit für unsere Programme und erlauben dem Compiler eine Reihe von möglichen Optimisierungen (Memoisation, Parallelisation, . . .).

The good...

Die Vorteile von purity sind enorm! Dadurch, dass wir Seiteneffekte kontrolliert halten, erhalten wir Typsicherheit für unsere Programme und erlauben dem Compiler eine Reihe von möglichen Optimisierungen (Memoisation, Parallelisation, . . .).

Für die Entwicklung von Software ist es ebenfalls oft hilfreich, sich auf unabhängige Teile des Codes zu konzentrieren, die nicht an einem globalen State hängen (Stichwort: Modularität).

The good...

Die Vorteile von purity sind enorm! Dadurch, dass wir Seiteneffekte kontrolliert halten, erhalten wir Typsicherheit für unsere Programme und erlauben dem Compiler eine Reihe von möglichen Optimisierungen (Memoisation, Parallelisation, . . .).

Für die Entwicklung von Software ist es ebenfalls oft hilfreich, sich auf unabhängige Teile des Codes zu konzentrieren, die nicht an einem globalen State hängen (Stichwort: Modularität).

Zuletzt können einige mehr oder minder große Katastrophen verhindert werden, weil nicht einfach irgendwo Funktionen wie diese aufgerufen werden können.

```
launchMissiles :: IO ()
launchMissiles = ... -- uh oh!
```

...and the bad

Allerdings ist eine Programmiersprache *komplett* ohne Seiteneffekte meist (mit ein paar wohldefinierten Ausnahmen) keine sehr nützliche Sprache.

Wenn wir keine Daten einlesen, keine Ergebnisse ausgeben und auch sonst nicht mit der Welt interagieren können, sind wir nicht mehr nützlich. Wir können zwar für uns interessante Werte berechnen (solange die Berechnung hardgecodet wurde), aber der Nutzer merkt davon nur, dass die CPU wärmer wird.

IO in Haskell

Wenn wir in Haskell programmieren, müssen wir IO, einen speziellen Typen für diese Zwecke, verwenden. Zum Beispiel mit diesen Funktionen:

```
getLine  :: IO String        -- String einlesen
putStrLn :: String -> IO ()  -- String ausgeben
```

IO in Haskell

Wenn wir in Haskell programmieren, müssen wir IO, einen speziellen Typen für diese Zwecke, verwenden. Zum Beispiel mit diesen Funktionen:

```
getLine  :: IO String        -- String einlesen
putStrLn :: String -> IO ()  -- String ausgeben
```

Dabei ist zu beachten, dass es keine* Funktion mit dem Typen `(IO a -> a)` gibt. Wir sind also dazu gezwungen, Interaktionen mit der Außenwelt klar anzusagen und bestimmten Regeln zu folgen (dazu später mehr).

IO by example

Ein sehr simples Programm mit IO könnte so aussehen:

```haskell
-- All good! =)
main :: IO ()
main = do
  putStrLn "Wie ist Ihr Name?"
  name <- getLine
  -- name :: String
  -- Trotzdem noch im IO-Typen
  putStrLn $ "Hallo, " ++ name ++ "!"
```

IO by example

Ein sehr simples Programm mit IO könnte so aussehen:

```haskell
-- All good! =)
main :: IO ()
main = do
  putStrLn "Wie ist Ihr Name?"
  name <- getLine
  -- name :: String
  -- Trotzdem noch im IO-Typen
  putStrLn $ "Hallo, " ++ name ++ "!"
```

Für einen einfachen Start gibt es in Haskell die do-Notation, die einen quasi-imperativen Programmierstil für den IO-Typen erlaubt. Mehr dazu in späteren Vorlesungen.

`IO` by bad example

Es gibt tatsächlich doch eine Funktion mit Typ `(IO a -> a)`, sie heißt `unsafePerformIO`. Der Name ist hier Programm.

Mit dieser Funktion *kann* man den `IO`-Typen umschiffen. Wir stellen aber fest, dass das oft nicht das ist, was wir *wollen*, weil wir so auch alle Vorteile von gekapseltem `IO` wieder verlieren.

`IO` by bad example

Es gibt tatsächlich doch eine Funktion mit Typ `(IO a -> a)`, sie heißt `unsafePerformIO`. Der Name ist hier Programm.

Mit dieser Funktion *kann* man den `IO`-Typen umschiffen. Wir stellen aber fest, dass das oft nicht das ist, was wir *wollen*, weil wir so auch alle Vorteile von gekapseltem `IO` wieder verlieren.

```
-- You're doing it wrong! >.<
pureInput :: String
pureInput = unsafePerformIO . getLine
```

In Abgaben für diese Veranstaltung ist `unsafePerformIO` (sofern nicht explizit geregelt) an keiner Stelle im Code erlaubt.

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

- Typklassen
  - Monoide und `Monoid`
  - `Functor` als Container
    oder Kontext

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

- Typklassen
  - Monoide und `Monoid`
  - `Functor` als Container oder Kontext
- Funktionen in `Haskell`
  - Funktionsverkettung
  - Partielle Anwendung
  - Curryfizierung (Deal)

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

- Typklassen
  - Monoide und `Monoid`
  - `Functor` als Container oder Kontext
- Funktionen in `Haskell`
  - Funktionsverkettung
  - Partielle Anwendung
  - Curryfizierung (Deal)
- Lazy Evaluation
  - Definition
  - Unendliche Strukturen
  - Fauler Kontrollfluss
  - Problemstellung

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

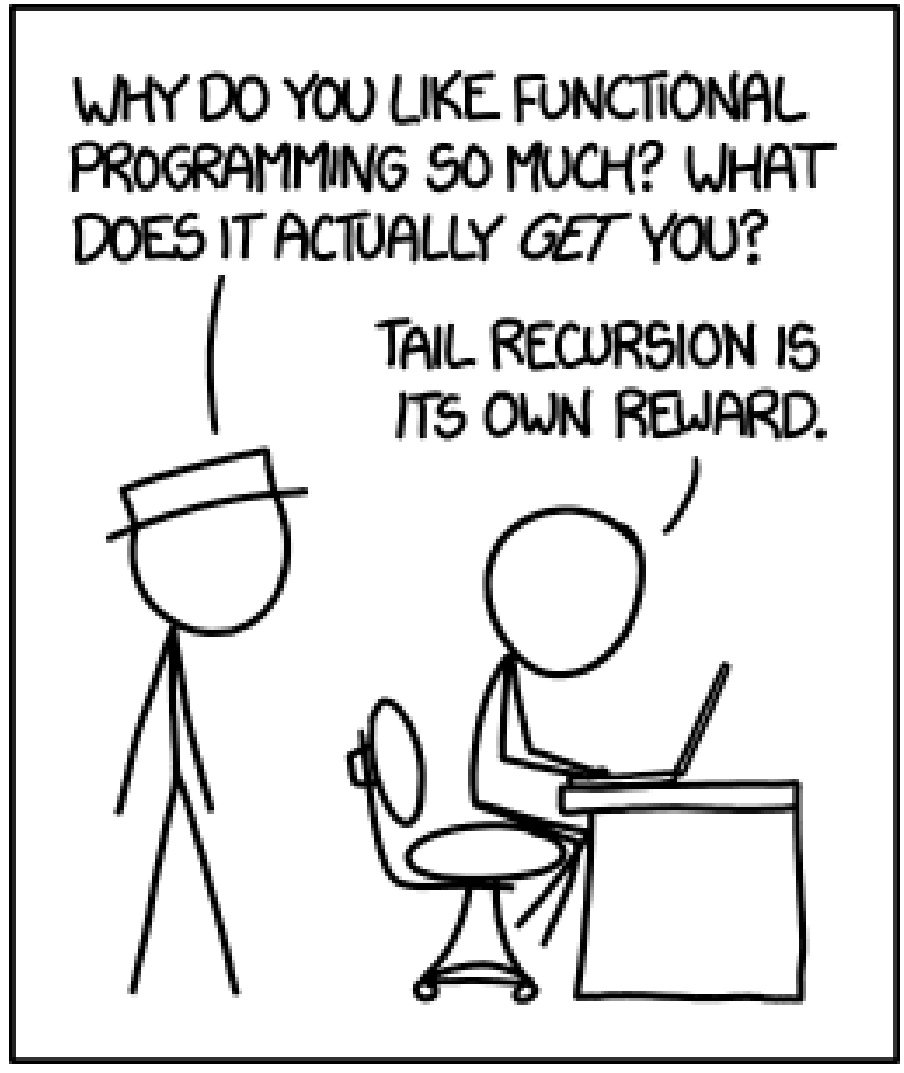
- Typklassen
  - Monoide und `Monoid`
  - `Functor` als Container oder Kontext
- Funktionen in `Haskell`
  - Funktionsverkettung
  - Partielle Anwendung
  - Curryfizierung (Deal)
- Lazy Evaluation
  - Definition
  - Unendliche Strukturen
  - Fauler Kontrollfluss
  - Problemstellung
- Purity
  - Definitionen von Purity und Seiteneffekt
  - `IO`-Typ und Motivation
  - ~~unsafePerformIO~~

Ausblick

Vorschau: Was machen wir nächste Woche?

- Wiederholung: Vorlesung 2
- `Functor`, `Applicative`, `Monad`

Fragen?



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