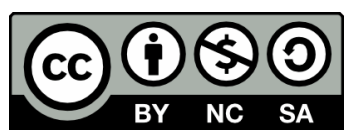
Fortgeschrittene Funktionale Programmierung in Haskell

Jonas Betzendahl
Stefan Dresselhaus

Vorlesung 1: *Thinking in Types!*
Stand: 17. April 2016



Organisatorisches

Veranstaltungen

Vorlesung:

Freitags, 14 - 16 Uhr, V2-205

Übungen:

Montags, 16 - 18 Uhr, V2-222 (GZI)
Montags, 18 - 20 Uhr, V2-222 (GZI)

Wahl der Tutorien ist frei, je nach eurem Stundenplan. Der erste Termin ist der 18. April (nächste Woche).

Übungszettel werden in den Übungen vorgestellt, teilweise dort auch bearbeitet und eine Woche später besprochen.

Leistungspunkte

Das Modul **FFPiHaskell** bringt fünf (echte) Leistungspunkte.

Wegen bürokratischer Hürden können diese LP *ausschließlich* in der Individuellen Ergänzung (BA/MA) untergebracht werden.

Voraussetzung für die Vergabe der LP sind Programmierabgaben (größere Projekte und/oder kleinere Aufgaben). Näheres später.

Personen

Jonas Betzendahl (`jbetzend@techfak...`)
Dozent (nur Vorlesungen), aktuell im ISY Master

Stefan Dresselhaus (`sdressel@techfak...`)
Dozent (Vorlesungen und Tutorien), aktuell im NWI Master

Hatem Shugaa Addin (`hshugaaaddin@techfak...`)
Tutor, FFPI-Haskell-Absolvent vom letzten Jahr

Alexander Sczyrba (`asczyrba@techfak...`)
Arbeitsgruppe Computational Metagenomics, verantwortlich für das Modul und für Beschwerden über uns.

Materialien

PDFs:

Alle Foliensätze, Übungsblätter und sonstige Unterlagen findet ihr entweder im *ekVV*-Lernraum oder auf GitHub zum Selberklonen:

`https://github.com/FFPiHaskell`

Videos:

Videomitschnitte der Vorlesungen, Screencasts etc. findet ihr auf YouTube auf dem Kanal der Vorlesung:

`https://youtube.com/channel/UC5yZfQZrZnug0sgvveTTfnw`

Die Materialien aus dem letzten Jahr haben wir gekennzeichnet online gelassen, weil sie noch für einige nützlich sein könnten.

Technik

Wir verwenden in dieser Vorlesung als Standard den `Glasgow Haskell Compiler (GHC)`, Versionen ≥ 7.10.

Das rundum-sorglos-Paket für zu Hause ist die *Haskell Platform* (`https://www.haskell.org/platform/`), erhältlich für GNU/Linux, Mac OS X und Windows.

Einen aktuellen GHC gibt es im GZI mit dem rcinfo-Paket `ghc`.

Technik

Wir verwenden in dieser Vorlesung als Standard den `Glasgow Haskell Compiler (GHC)`, Versionen ≥ 7.10.

Das rundum-sorglos-Paket für zu Hause ist die *Haskell Platform* (`https://www.haskell.org/platform/`), erhältlich für GNU/Linux, Mac OS X und Windows.

Einen aktuellen GHC gibt es im GZI mit dem rcinfo-Paket `ghc`.

Wichtig:
Der Haskell-Interpreter `Hugs` wird von uns nicht unterstützt!

Voraussetzungen

Wir setzen (informell) ein Minimum an Vorwissen voraus.

Voraussetzungen

Wir setzen (informell) ein Minimum an Vorwissen voraus.

- Grundlegende Datenstrukturen wie Listen, Bäume
  (siehe: Algorithmen und Datenstrukturen)

Voraussetzungen

Wir setzen (informell) ein Minimum an Vorwissen voraus.

- Grundlegende Datenstrukturen wie Listen, Bäume
  (siehe: Algorithmen und Datenstrukturen)
- Grundlegende Erfahrung mit Haskell / FunProg
  (siehe: Programmieren in Haskell)

Voraussetzungen

Wir setzen (informell) ein Minimum an Vorwissen voraus.

- Grundlegende Datenstrukturen wie Listen, Bäume
  (siehe: Algorithmen und Datenstrukturen)
- Grundlegende Erfahrung mit Haskell / FunProg
  (siehe: Programmieren in Haskell)
- Bereitschaft, sich auf Theorie einzulassen

Voraussetzungen

Wir setzen (informell) ein Minimum an Vorwissen voraus.

- Grundlegende Datenstrukturen wie Listen, Bäume
  (siehe: Algorithmen und Datenstrukturen)
- Grundlegende Erfahrung mit Haskell / FunProg
  (siehe: Programmieren in Haskell)
- Bereitschaft, sich auf Theorie einzulassen
- Bereitschaft, sich auf Praxis einzulassen

Wenn es damit (oder sonst auch) Probeleme gibt, helfen unsere ...

Überlebenstipps

If you are learning programming (or anything else) what you do when you get "stuck" will be the sole determinant of your success!

Sarah Mount (`@snim2` auf Twitter)

`https://twitter.com/snim2/status/404560982577389568`

Bücher

Es gibt einige Bücher frei im Netz, die insbesondere für Neulinge gut geeignet sind:

- **Learn You A Haskell For Great Good**
  Bewährt und beliebt, das Standardwerk. Hat schon so manchen durch A& D gebracht.
  `http://learnyouahaskell.com/`

Bücher

Es gibt einige Bücher frei im Netz, die insbesondere für Neulinge gut geeignet sind:

- **Learn You A Haskell For Great Good**
  Bewährt und beliebt, das Standardwerk. Hat schon so manchen durch A& D gebracht.
  `http://learnyouahaskell.com/`

- **Happy Learn Haskell Tutorial**
  Ein neueres Buch, nocht nicht komplett offen verfügbar. Dafür allerdings aktueller.
  `http://www.happylearnhaskelltutorial.com`

Sonstiges

Aktive Communities, in denen eins leicht Hilfe, Material und Unterstützung finden kann:

- **Reddit**
  `www.reddit.com/r/haskell` diskutiert auf vielen Ebenen Neuigkeiten um Haskell, konkrete Fragen sind in `www.reddit.com/r/haskellquestions` besser aufgehoben.

Sonstiges

Aktive Communities, in denen eins leicht Hilfe, Material und Unterstützung finden kann:

- **Reddit**
  `www.reddit.com/r/haskell` diskutiert auf vielen Ebenen Neuigkeiten um Haskell, konkrete Fragen sind in `www.reddit.com/r/haskellquestions` besser aufgehoben.
- **IRC**
  Auf `freenode` findet ihr dutzende Channel (z.B. `#haskell`) mit hilfsbereiten, aktiven Leuten, die euch helfen können, egal ob ihr `foldl'` oder `CoYoneda` verstehen wollt.

Sonstiges

Aktive Communities, in denen eins leicht Hilfe, Material und Unterstützung finden kann:

- **Reddit**
  `www.reddit.com/r/haskell` diskutiert auf vielen Ebenen Neuigkeiten um Haskell, konkrete Fragen sind in `www.reddit.com/r/haskellquestions` besser aufgehoben.
- **IRC**
  Auf `freenode` findet ihr dutzende Channel (z.B. `#haskell`) mit hilfsbereiten, aktiven Leuten, die euch helfen können, egal ob ihr `foldl'` oder `CoYoneda` verstehen wollt.
- **StackOverflow**
  There is always StackOverflow ...
  (`stackoverflow.com/questions/tagged/haskell`)

Andere Menschen

. . . und wenn all diese Menschen im Internet euch nicht weiter helfen können, gibt es natürlich auch immer noch

- die Tutorien,
- die Fachschaft,
- andere Studierende mit Erfahrung und sogar
- E-Mails an Tutoren / Dozenten.

Manchmal helfen die auch weiter. :D

Thinking in Types

Thinking in Types

. . . is the most valuable lesson I can teach you!

Was ist ein Typ?

In typisierten Programmiersprachen (`Haskell`, Idris, Java ...) hat jeder *Term* einen *Typen* zugeordnet. Anders herum sagen wir auch, dass jeder *Typ* eine Menge von *Bewohnern* hat.

Was ist ein Typ?

In typisierten Programmiersprachen (`Haskell`, Idris, Java ...) hat jeder *Term* einen *Typen* zugeordnet. Anders herum sagen wir auch, dass jeder *Typ* eine Menge von *Bewohnern* hat.

```
gerade :: Integer -> Bool      -- Signatur
gerade 0 = True
gerade n = not $ gerade (n-1)
```

Was ist ein Typ?

In typisierten Programmiersprachen (`Haskell`, Idris, Java ...) hat jeder *Term* einen *Typen* zugeordnet. Anders herum sagen wir auch, dass jeder *Typ* eine Menge von *Bewohnern* hat.

```
gerade :: Integer -> Bool      -- Signatur
gerade 0 = True
gerade n = not $ gerade (n-1)
```

In `Haskell` sind Terme (Werte) und Typen unterschiedliche Dinge. Wir unterhalten uns also über zwei Ebenen:

- die Typebene (type level) und
- die Werteebene (value level)

Einzigartigkeit

Es ist darauf hinzuweisen, dass jedem Term *genau* ein Typ zugeordnet wird. Auch wenn es für das Auge so aussieht: Der gleiche Term kann *niemals* zwei unterschiedliche Typen haben!

```
foo1 :: Integer
foo1 = 5

foo2 :: Int
foo2 = 5

equal :: Bool
equal = foo1 == foo2 -- type error
```

Pragmatik

Ein hinreichend mächtiges Typsystem hat verschiedene Vorteile:

- Programmstruktur klarer ersichtlich
- Erfolgreiches type checking impliziert Korrektheit^{TM}
  "The compiler is your friend!"
- Hilfreiche Dokumentation

Pragmatik

Ein hinreichend mächtiges Typsystem hat verschiedene Vorteile:

- Programmstruktur klarer ersichtlich
- Erfolgreiches type checking impliziert Korrektheit^{TM}
  "The compiler is your friend!"
- Hilfreiche Dokumentation

Aber ein *zu* ausdrucksstarkes Typsystem (Dependent Types) kann auch Nachteile haben:

- Mehr Arbeit (point of diminishing returns)
- Unentscheidbares Typechecking

Pragmatik

Ein hinreichend mächtiges Typsystem hat verschiedene Vorteile:

- Programmstruktur klarer ersichtlich
- Erfolgreiches type checking impliziert Korrektheit^{TM}
  "The compiler is your friend!"
- Hilfreiche Dokumentation

Aber ein *zu* ausdrucksstarkes Typsystem (Dependent Types) kann auch Nachteile haben:

- Mehr Arbeit (point of diminishing returns)
- Unentscheidbares Typechecking

Es geht also wieder mal um das Richtige Werkzeug für den richtigen Job.

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer?`

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer`? A: `0,1,2,3,9001,-314...`

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer`? A: `0,1,2,3,9001,-314...`
- `"Hello World!"`?

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer`? A: `0,1,2,3,9001,-314...`
- `"Hello World!"`? A: `String` (oder `[Char]`)

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer`? A: `0,1,2,3,9001,-314...`
- `"Hello World!"`? A: `String` (oder `[Char]`)
- `Integer -> Bool`?

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer`? A: `0,1,2,3,9001,-314...`
- `"Hello World!"`? A: `String` (oder `[Char]`)
- `Integer -> Bool`? A: `even, odd, ...`

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer?` A: `0,1,2,3,9001,-314...`
- `"Hello World!"?` A: `String` (oder `[Char]`)
- `Integer -> Bool?` A: `even, odd, ...`
- `...`

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer?` A: `0,1,2,3,9001,-314...`
- `"Hello World!"?` A: `String` (oder `[Char]`)
- `Integer -> Bool?` A: `even, odd, ...`
- `...`

Wir merken: Auch *Funktionen* sind Werte eines Typen (sie sind so genannte "first class citizens" in `Haskell`). Dazu später mehr.

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer`? A: `0,1,2,3,9001,-314...`
- `"Hello World!"`? A: `String` (oder `[Char]`)
- `Integer -> Bool`? A: `even, odd, ...`
- `...`

Wir merken: Auch *Funktionen* sind Werte eines Typen (sie sind so genannte "first class citizens" in `Haskell`). Dazu später mehr.

Was wäre der simpelste Typ, den ihr euch vorstellen könnt?

(sehr) simple Typen

Um etwas Intuition zu üben: Bitte paart Bewohner und Typen!

- `Integer`? A: `0,1,2,3,9001,-314...`
- `"Hello World!"`? A: `String` (oder `[Char]`)
- `Integer -> Bool`? A: `even, odd, ...`
- `...`

Wir merken: Auch *Funktionen* sind Werte eines Typen (sie sind so genannte "first class citizens" in `Haskell`). Dazu später mehr.

Was wäre der simpelste Typ, den ihr euch vorstellen könnt?
Mögliche Antworten: `Bool, Unit, ...`

There and back again (I)

Die interessante Fragen bei einem neuen Typen, sind in der Regel immer die gleichen:

- Wie komme ich an Bewohner dieses Typen?
- Was kann ich mit einem Bewohner dieses Typens machen?

Schauen wir uns an, wie das in `Haskell` funktioniert.

There and back again (II)

In `Haskell` werden neue Datentypen mit dem Keywort `data` erstellt. Eine Möglichkeit ist, alle Werte per Konstruktoren anzugeben.

```haskell
-- Fresh from the prelude
data ()   = ()
data Bool = False | True
```

There and back again (II)

In `Haskell` werden neue Datentypen mit dem Keywort `data` erstellt. Eine Möglichkeit ist, alle Werte per Konstruktoren anzugeben.

```haskell
-- Fresh from the prelude
data ()   = ()
data Bool = False | True
```

Es ist auch möglich, auf bestehende Typen zurückzugreifen.

```haskell
data Temperature = Fahrenheit Int
                 | Celsius    Int
                 | Kelvin     Int
```

There and back again (II)

In `Haskell` werden neue Datentypen mit dem Keywort `data` erstellt. Eine Möglichkeit ist, alle Werte per Konstruktoren anzugeben.

```haskell
-- Fresh from the prelude
data ()   = ()
data Bool = False | True
```

Es ist auch möglich, auf bestehende Typen zurückzugreifen.

```haskell
data Temperature = Fahrenheit Int
                 | Celsius    Int
                 | Kelvin     Int
```

Auch auf mehrere auf einmal:

```haskell
data Artikel = Artikel String Int Int
```

There and back again (II)

In `Haskell` werden neue Datentypen mit dem Keywort `data` erstellt. Eine Möglichkeit ist, alle Werte per Konstruktoren anzugeben.

```haskell
-- Fresh from the prelude
data ()    = ()
data Bool = False | True
```

Es ist auch möglich, auf bestehende Typen zurückzugreifen.

```haskell
data Temperature = Fahrenheit Int      -- Summentyp
                 | Celsius    Int
                 | Kelvin     Int
```

Auch auf mehrere auf einmal:

```haskell
data Artikel = Artikel String Int Int -- Produkttyp
```

There and back again (III)

Natürlich sind Konstruktoren nicht der einzige Weg, an Werte eines Typs zu gelangen. Ein anderer Weg, sind *Funktionen*!

```
even :: Int -> Bool   -- Bools from Ints
not  :: Bool -> Bool  -- Bools from other Bools
```

Auf gewisse Art und Weise sind *Funktionen* und *Konstruktoren* gar nicht so unterschiedlich.

Bleibt die Frage: was können wir mit Werten eines Typs machen?

There and back again (III)

Natürlich sind Konstruktoren nicht der einzige Weg, an Werte eines Typs zu gelangen. Ein anderer Weg, sind *Funktionen*!

```
even :: Int -> Bool  -- Bools from Ints
not  :: Bool -> Bool -- Bools from other Bools
```

Auf gewisse Art und Weise sind *Funktionen* und *Konstruktoren* gar nicht so unterschiedlich.

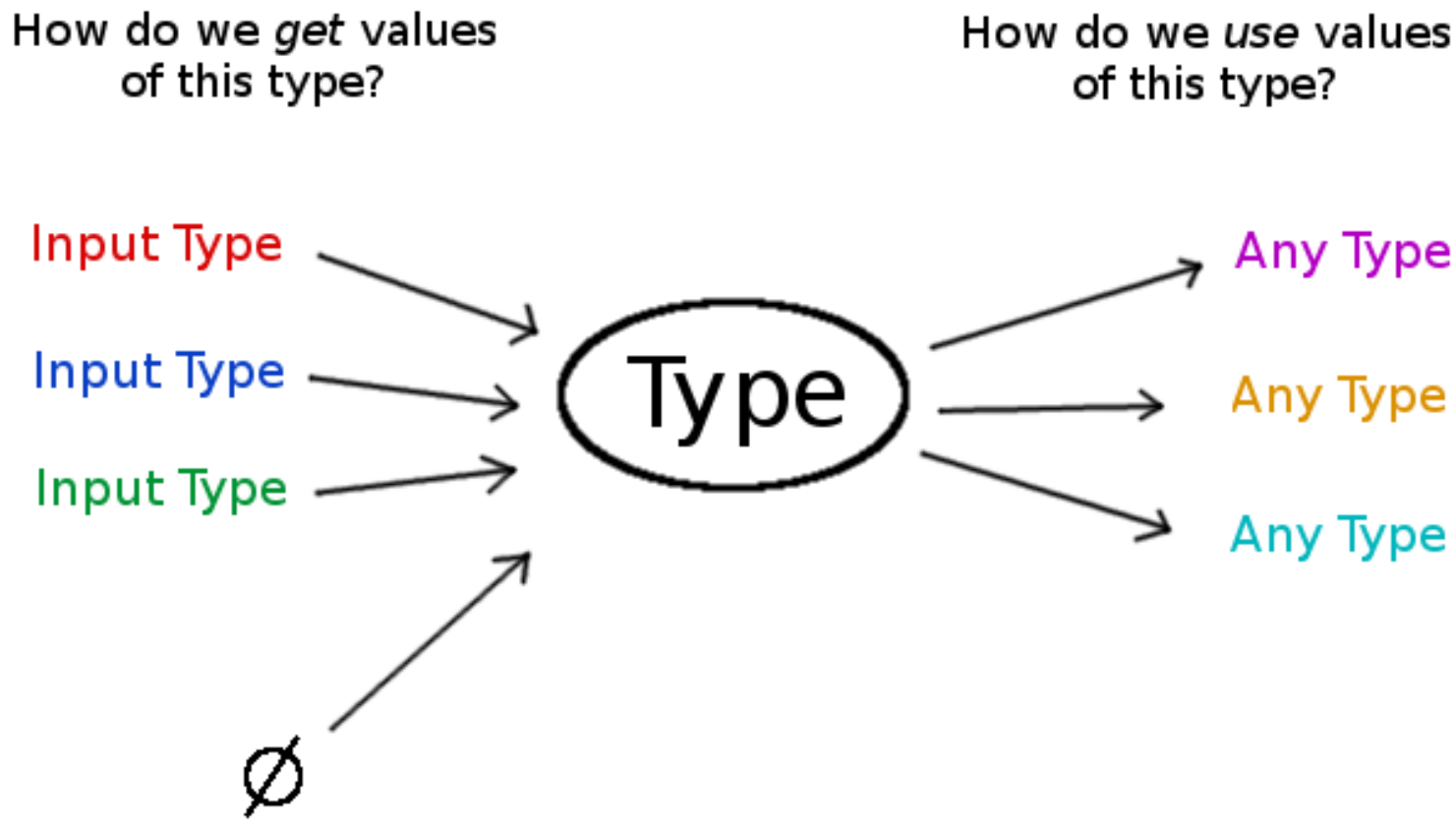
Bleibt die Frage: was können wir mit Werten eines Typs machen?

Wir können sie wieder in *Funktionen* benutzen, bis wir den Wert haben, den wir gesucht haben.

```
reverse   :: [a] -> [a]
pronounce :: Integer -> String
```

There and back again (IV)

Zusammengefasst. Für welche Konzepte stehen die Pfeile?



Polymorphismus

Eine von Haskells Stärken ist, dass Polymorphismus einfach und unkompliziert auszudrücken ist.

```
-- Don't need to know about the type
repeat' :: a -> [a]
repeat' x = x:(repeat' x)
```

Diese Funktion kann für *jeden* Typen a aufgerufen werden und liefert immer das entsprechende Ergebnis. Haskell unterstützt aktiv einen allgemeinen Codingstil (code reuse!).

Vergleich: Polymorphismus

Haskell:

```
selectionSort :: (Ord a) => [a] -> [a]
selectionSort xs = ...
```

Vergleich: Polymorphismus

Haskell:

```
selectionSort :: (Ord a) => [a] -> [a]
selectionSort xs = ...
```

Scala:

```
def selectionSort[T <% Ordered[T]]
          (list: List[T]): List[T] = {...}
```

Vergleich: Polymorphismus

Haskell:

```haskell
selectionSort :: (Ord a) => [a] -> [a]
selectionSort xs = ...
```

Scala:

```scala
def selectionSort[T <% Ordered[T]]
         (list: List[T]): List[T] = {...}
```

Java:

```java
public static <E extends Comparable<E>>
       void selectionSort(E[] list)
{ ... }
```

Polymorphe Typen (I)

Dieses Konzept von Polymorphismus eignet sich hervorragend, um polymorphe Funktionen zu schreiben. Es kann aber noch darüber hinaus eingesetzt werden.

Polymorphe Typen sind ebenfalls möglich. Als Beispiel dient oft ein simpler Listentyp:

```
-- Isomorphic to regular lists
data List a = Empty | Cons a (List a)
```

Polymorphe Typen (II)

```haskell
-- Isomorphic to regular lists
data List a = Empty | Cons a (List a)
```

Das a spielt in dieser Definition verschiedene Rollen:

Polymorphe Typen (II)

```haskell
-- Isomorphic to regular lists
data List a = Empty | Cons a (List a)
```

Das a spielt in dieser Definition verschiedene Rollen:

- Das erste a gibt an, dass der Typ `List` einen *Parameter* übergeben bekommt. `List` an sich ist also kein fertiger Typ; `List Int` z.B. hingegen schon.

Polymorphe Typen (II)

```haskell
-- Isomorphic to regular lists
data List a = Empty | Cons a (List a)
```

Das `a` spielt in dieser Definition verschiedene Rollen:

- Das erste `a` gibt an, dass der Typ `List` einen *Parameter* übergeben bekommt. `List` an sich ist also kein fertiger Typ; `List Int` z.B. hingegen schon.
- Das zweite `a` bezieht sich auf einen *Wert* des Typen.

Polymorphe Typen (II)

```haskell
-- Isomorphic to regular lists
data List a = Empty | Cons a (List a)
```

Das `a` spielt in dieser Definition verschiedene Rollen:

- Das erste `a` gibt an, dass der Typ `List` einen *Parameter* übergeben bekommt. `List` an sich ist also kein fertiger Typ; `List Int` z.B. hingegen schon.
- Das zweite `a` bezieht sich auf einen *Wert* des Typen.
- Das dritte `a` besagt, dass im `Cons`-Fall wieder eine Liste mit dem gleichen Parametertypen erwartet wird.

Polymorphe Typen (III)

Diese Parametrisierung erlaubt uns, sehr nützliche polymorphe Typen zu bauen. Zum Beispiel für Berechnungen, die fehlschlagen können:

```
data Maybe a = Nothing  -- computation failed
             | Just a   --        "       successful
```

Polymorphe Typen (III)

Diese Parametrisierung erlaubt uns, sehr nützliche polymorphe Typen zu bauen. Zum Beispiel für Berechnungen, die fehlschlagen können:

```
data Maybe a = Nothing  -- computation failed
             | Just a   --      "       successful
```

Wenn wir dann auch noch zum Beispiel eine Fehlermeldung erhalten wollen, können wir auch mehrere Parameter mitgeben.

```
data Either a b = Left  a  -- comp. failed     (error)
                | Right b  -- comp. successful (result)
```

Parametricity

Parametricity

Typen allgemein statt konkret zu halten gibt uns einige Vorteile. Anhand einer Typsignatur können wir oft schon viel über den Code erfahren!

Beispiel: *Ich weiß genau, was die Funktion `foo` tut. Ihr auch?*

```
foo :: a -> b -> c -> c
```

Parametricity

Typen allgemein statt konkret zu halten gibt uns einige Vorteile.
Anhand einer Typsignatur können wir oft schon viel über den Code erfahren!

Beispiel: *Ich weiß genau, was die Funktion `foo` tut. Ihr auch?*

```
foo :: a -> b -> c -> c
```

Antwort: Es gibt nur eine korrekte Möglichkeit!

```
foo :: a -> b -> c -> c
foo _ _ x = x
```

`foo` *muss* seine ersten beiden Argumente wegschmeißen und das dritte zurück geben.
Es gibt keinen anderen Weg vorbei am Typchecker.

Mehr Parametricity

Ich weiß genau, was die Funktion `bar` tut!

```
bar :: a -> (a, a, a, a)
```

Mehr Parametricity

Ich weiß genau, was die Funktion `bar` tut!

```
bar :: a -> (a, a, a, a)
```

Ich weiß fast, was die Funktion `baz` tut!

```
baz :: (a, a, a, a) -> a
```

Mehr Parametricity

Ich weiß genau, was die Funktion `bar` tut!

```
bar :: a -> (a, a, a, a)
```

Ich weiß fast, was die Funktion `baz` tut!

```
baz :: (a, a, a, a) -> a
```

Wie viele mögliche Funktionen gibt es für diese Signatur?

```
snafu :: [a] -> [b]
```

Mehr Parametricity

Ich weiß genau, was die Funktion `bar` tut!

```
bar :: a -> (a, a, a, a)
```

Ich weiß fast, was die Funktion `baz` tut!

```
baz :: (a, a, a, a) -> a
```

Wie viele mögliche Funktionen gibt es für diese Signatur?

```
snafu :: [a] -> [b]
```

... und welche Rolle spielt hier der Input?

```
quux :: a -> Integer
```

Typklassen

Motivation

Ein (vorerst) letztes Beispiel zu Parametricity. Was kann diese Funktion tun?

```
foobar :: a -> a -> Bool
```

Motivation

Ein (vorerst) letztes Beispiel zu Parametricity. Was kann diese Funktion tun?

```
foobar :: a -> a -> Bool
```

Es stellt sich heraus, dass es nur zwei valide Implementationen gibt:

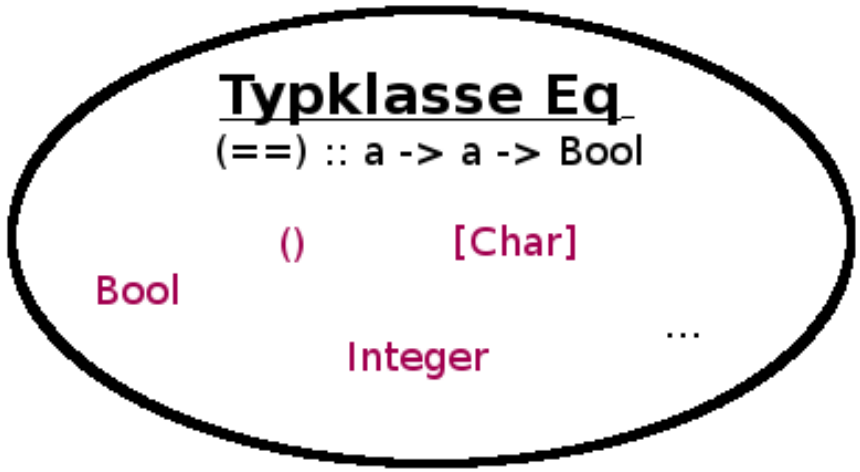
```
foobar'  _ _ = True
foobar'' _ _ = False
```

Aber wäre es nicht gut, wenn wir sagen könnten, ob wir zweimal das gleiche a übergeben bekommen haben?

Type classes to the rescue!

Typklassen

Typklassen sind Mengen von Typen und kommen alle mit (mindestens) einer Funktion. Es sind immer genau die Typen in einer Typklasse, auf denen diese Funktion definiert ist.



Im Falle von `Eq` ist es die Funktion `(==)`, die auf einem Typen definiert sein muss, bevor er in die Typklasse aufgenommen werden kann.

Eq from scratch (I)

Möchte ich mir meine eigene Typklasse bauen und sie sinnvoll benutzen, muss ich sie sowohl *definieren* als auch *instanziieren*.

Eine Definition funktioniert über das Keyword `class` und die Angabe einer Funktionssignatur.

```
data Bool = False | True


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

Die Signatur gibt den Namen und den Typen für die Funktion vor, die alle Typen in der Klasse implementieren müssen.

Eq from scratch (II)

Ist eine Typklasse erfolgreich definiert, so ist sie erstmal leer. Das bedeutet, noch enthält sie keine der Typen, die Haskell kennt. Diese müssen mannuell hinzugefügt werden.

Eq from scratch (II)

Ist eine Typklasse erfolgreich definiert, so ist sie erstmal leer. Das bedeutet, noch enthält sie keine der Typen, die Haskell kennt. Diese müssen mannuell hinzugefügt werden.

Zu diesem Prozess wird in der Regel gesagt, dass eine *Instanz* der Typklasse für den speziellen Typen geschrieben wird.

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

Eq from scratch (II)

Ist eine Typklasse erfolgreich definiert, so ist sie erstmal leer. Das bedeutet, noch enthält sie keine der Typen, die Haskell kennt. Diese müssen mannuell hinzugefügt werden.

Zu diesem Prozess wird in der Regel gesagt, dass eine *Instanz* der Typklasse für den speziellen Typen geschrieben wird.

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

In besonders einfachen Fällen kann Haskell sich die Instanz auch automatisch herleiten:

```
data () = () deriving Eq
```

Anwendung

Jetzt, wo wir unsere Typklasse definiert und instanziiert haben, können wir polymorphe Typen auf bestimmte Klassen einschränken.

```
foobar :: (Eq a) => a -> a -> Bool
```

Anwendung

Jetzt, wo wir unsere Typklasse definiert und instanziiert haben, können wir polymorphe Typen auf bestimmte Klassen einschränken.

```
foobar :: (Eq a) => a -> a -> Bool
```

Unsere Funktion von vorhin kann mit dieser Einschränkung sehen, dass (==) zur Verfügung steht und es benutzen. Folgende Implementationen sind jetzt auch möglich:

```
foobar'     _ _ = True
foobar''    _ _ = False
foobar'''   x y = x == y
foobar'''' x y = not (x == y)
```

Andere (einfache) Typklassen

Andere Typklassen in `Haskell`, die oft hilfreich sind:

Andere (einfache) Typklassen

Andere Typklassen in `Haskell`, die oft hilfreich sind:

- `Ord` (Funktion: `(<=)`)
  Beispiele: `Integer`, `Double`, `String`, `Bool`, ...
  Alle Typen, die sich linear anordnen lassen.

Andere (einfache) Typklassen

Andere Typklassen in `Haskell`, die oft hilfreich sind:

- `Ord` (Funktion: `(<=)`)
  Beispiele: `Integer`, `Double`, `String`, `Bool`, ...
  Alle Typen, die sich linear anordnen lassen.
- `Show` (Funktion: `show`)
  Beispiele: `Bool`, `String`, `Integer`, ...
  Alle Typen, die über einen String anzeigbar sind.

Andere (einfache) Typklassen

Andere Typklassen in `Haskell`, die oft hilfreich sind:

- `Ord` (Funktion: `(<=)`)
  Beispiele: `Integer`, `Double`, `String`, `Bool`, ...
  Alle Typen, die sich linear anordnen lassen.
- `Show` (Funktion: `show`)
  Beispiele: `Bool`, `String`, `Integer`, ...
  Alle Typen, die über einen String anzeigbar sind.
- `Num` (Funktionen: `(+)`, `(-)` `(*)`, ...)
  Beispiele: `Int`, `Integer`, `Double`, ...
  Alle Zahlentypen. Genauer: Alle Typen, auf denen die üblichen Rechenoperationen definiert sind.

Von einer Typklasse zur nächsten (I)

Wir haben gerade `Ord` kennen gelernt, die Klasse der `Haskell`-Typen, auf denen irgendeine Form von $\leq$-Realtion definiert ist.
Diese Typklasse wird zum Beispiel gebraucht, wenn wir polymorphe Sortieralgorithmen implementieren wollen.

```
-- unsorted lists to sorted lists
sort :: (Ord a) => [a] -> [a]
```

Von einer Typklasse zur nächsten (I)

Wir haben gerade `Ord` kennen gelernt, die Klasse der `Haskell`-Typen, auf denen irgendeine Form von $\leq$-Realtion definiert ist.
Diese Typklasse wird zum Beispiel gebraucht, wenn wir polymorphe Sortieralgorithmen implementieren wollen.

```
-- unsorted lists to sorted lists
sort :: (Ord a) => [a] -> [a]
```

Damit eine Relation wie (<=) aber überhaupt Sinn machen kann, muss auch eine Relation wie (==) vorhanden sein. In einer zu sortierenden Liste könnten ja auch zwei gleiche Elemente stehen.

Wie können wir das einrichten?

Von einer Typklasse zur nächsten (II)

Ein erster Versuch mit dem Wissen von eben würde vielleicht so aussehen:

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

Von einer Typklasse zur nächsten (II)

Ein erster Versuch mit dem Wissen von eben würde vielleicht so aussehen:

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

Aber das würde jede Menge Duplizierung von Code bedeuten. In den meisten Fällen benutzt `Ord` kein anderes `(==)` als `Eq`. Der gesamte Code der `Eq`-Instanz müsste hier nochmal geschrieben werden.

Von einer Typklasse zur nächsten (III)

Stattdessen eine Einschränkung auf die Typen legen, für die wir überhaupt eine `Ord`-Instanz schreiben können:

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

Auf diese Weise muss eine `Eq`-Instanz schon in scope liegen, die wir einfach wieder verwenden können.

Von einer Typklasse zur nächsten (III)

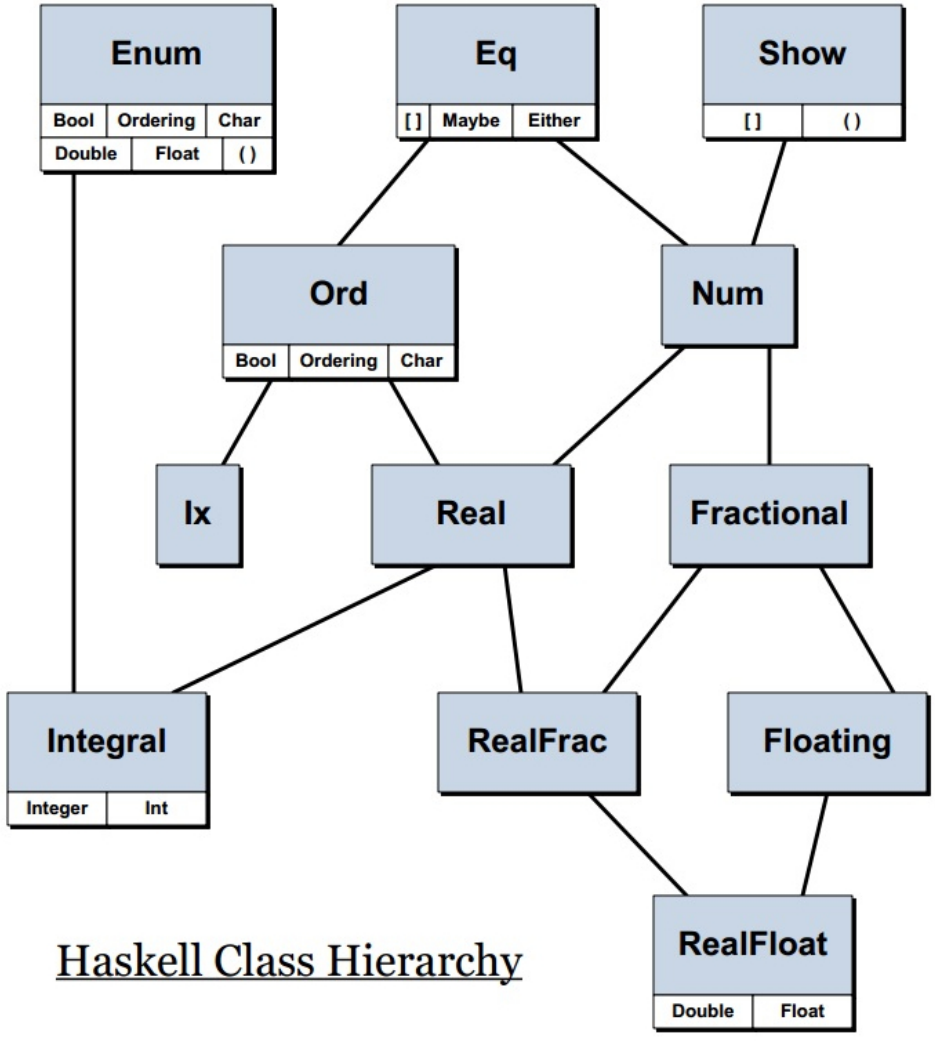
Stattdessen eine Einschränkung auf die Typen legen, für die wir überhaupt eine `Ord`-Instanz schreiben können:

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

Auf diese Weise muss eine `Eq`-Instanz schon in scope liegen, die wir einfach wieder verwenden können.

Auf diese Weise ergibt sich eine Form von Hierarchie der bekanntesten Typklassen:

Typklassenhierarchie



Haskell Class Hierarchy

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

- Typen
  - Genau ein Typ zu genau einem Term
  - Vorteile von Typsystemen
  - Konstruktoren und Induktion
  - Parametrisierte Typen
  - Parametricity

Rückblick

Ein Blick zurück: Was haben wir heute gelernt?

- Typen
  - Genau ein Typ zu genau einem Term
  - Vorteile von Typsystemen
  - Konstruktoren und Induktion
  - Parametrisierte Typen
  - Parametricity
- Typklassen
  - Motivation, Bedeutung
  - Definition & Instanziierung
  - Hierarchie von Typklassen
  - `Eq`, `Ord`, `Num`, `Show`

Ausblick

Vorschau: Was machen wir nächste Woche?

Ausblick

Vorschau: Was machen wir nächste Woche?

- Typklassen

Ausblick

Vorschau: Was machen wir nächste Woche?

- Typklassen
- Typklassen

Ausblick

Vorschau: Was machen wir nächste Woche?

- Typklassen
- Typklassen
- Vielleicht auch was anderes?

Ausblick

Vorschau: Was machen wir nächste Woche?

- Wiederholung: Vorlesung 1
- `Functor`! `Applicative`! `Monad`?
- Lazy Evaluation oder Purity

Fragen?