

FORTGESCHRITTENE FUNKTIONALE PROGRAMMIERUNG

TEIL 2: TYPKLASSEN, FUNKTOREN, KINDS UND MODULE

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

2. Mai 2017



1 POLYMORPHISMUS

- Motivation
- Parametrisch
- Ad-hoc
- Typklassen
- Unterklassen

2 FUNKTOREN

- Beispiele
- Kinds
- Newtype
- Zusammenfassung Funktoren
- Zusammenfassung Polymorphismus

3 MODULE

- Export
- Import
- Zusammenfassung Module
- Beispiel: Endliche Abbildungen



DATENTYPDEKLARATION

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – muss mit Großbuchstaben beginnen
- optionale Typparameter
- optionale Alternativen lies | als “oder”
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Zahl bekannte Typen oder Typparameter
- optionale `deriving`-Klausel mit Liste von Typklassen



ZUSAMMENFASSUNG: DATENTYPEN

- Ein Typ (oder **Datentyp**) ist eine Menge von Werten
- Typdeklarationen in Haskell:
 - `data` Deklaration wirklich neuer Typen
 - `type` Typabkürzungen für Lesbarkeit
- Datentypen können andere Typen als Parameter haben, **Konstruktoren** können als Funktionen betrachtet werden
- Datentypen können (wechselseitig) rekursiv definiert werden
- Unter einer **Datenstruktur** versteht man einen Datentyp plus alle darauf verfügbaren Operationen
- **Polymorphe Funktionen** können mit gleichem Code verschiedene Typen verarbeiten
- Datentypen verhindern versehentliches Verwecheln



BEISPIEL: VERWECHSLUNG VERHINDERN

Drei Datentypen, welche alle lediglich ein Tripel von `Double` sind, jedoch unterschiedlich zu nutzen sind:

```
type Point3D = (Double, Double, Double)      -- bietet keine Sicherheit
data Circle  = Circle Double Double Double --          | gegen Verwechslung
data Monster = Monster Double Double Double
```

```
hydralisk :: Monster
hydralisk = Monster 17.3 17.3 80.0
```

```
myCircle :: Circle
myCircle = Circle 17.3 17.3 80.0
```

```
area :: Circle -> Double
area (Circle _ _ r) = pi * r^2
```

```
> area myCircle
20106.192982974677
```

```
> area hydralisk
<interactive>:13:6:
  Couldn't match expected type `Circle' with actual type `Monster'
  In the first argument of `area', namely `hydralisk'
  In the expression: area hydralisk
  In an equation for `it': it = area hydralisk
```



PROBLEM

Jeder Datentyp muss extra behandelt werden!

Allerdings möchte man trotzdem ähnliche Operationen auf verschiedenen Datentypen durchführen: z.B. Anzahl Elemente einer Datenstruktur, egal ob Liste, Baum, Menge, etc.

⇒ *Funktionen mit gleicher Bedeutung*

Andererseits möchten wir Code-Duplikation vermeiden, z.B. es macht keinen Unterschied ob wir die Anzahl der Elemente in einen Baumes über `Int` oder `Circle` zählen.

⇒ *Funktionen mit nahezu gleichem Code*

Lösung: **Polymorphismus!** Generalisierung dank Polymorphismus vermeidet Redundanz und erhöht Lesbarkeit und Wartbarkeit.



BEISPIEL: LIST-REVERSAL

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where rev_aux :: [Char] -> [Char] -> [Char]
        rev_aux [] acc = acc
        rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das "vorläufige" Ergebnis:

```
reverse "live"
~> rev_aux 'l':'i':'v':'e':[]
~> rev_aux      'i':'v':'e':[] 'l':[]
~> rev_aux          'v':'e':[] 'i':'l':[]
~> rev_aux              'e':[] 'v':'i':'l':[]
~> rev_aux                  [] 'e':'v':'i':'l':[]
~> "evil"
```

Wo ist wichtig, dass die Listenelemente Typ Char haben?



BEISPIEL: LIST-REVERSAL

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Int] -> [Int]
reverse xs = rev_aux xs []
  where rev_aux :: [Int] -> [Int] -> [Int]
        rev_aux [] acc = acc
        rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das “vorläufige” Ergebnis:

```
reverse [1,2,3,4]
~> rev_aux  1 : 2 : 3 : 4 : []
~> rev_aux      2 : 3 : 4 : []      1 : []
~> rev_aux          3 : 4 : []      2 : 1 : []
~> rev_aux              4 : []      3 : 2 : 1 : []
~> rev_aux                  []      4 : 3 : 2 : 1 : []
~> [4,3,2,1]
```

Wo ist wichtig, dass die Listenelemente Typ Int haben?



BEISPIEL: LIST-REVERSAL

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [ a ] -> [ a ]
reverse xs = rev_aux xs []
  where  rev_aux :: [ a ] -> [ a ] -> [ a ]
         rev_aux [] acc = acc
         rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das “vorläufige” Ergebnis:

```
reverse [1,2,3,4]
~> rev_aux  1 : 2 : 3 : 4 : []
~> rev_aux      2 : 3 : 4 : []      1 : []
~> rev_aux          3 : 4 : []      2 : 1 : []
~> rev_aux              4 : []      3 : 2 : 1 : []
~> rev_aux                  []      4 : 3 : 2 : 1 : []
~> [4,3,2,1]
```

Wo ist wichtig, dass die Listenelemente Typ Int haben?



PARAMETRISCHER POLYMORPHISMUS

Haskell's **statisches Typsystem** prüft beim Kompilieren alle Typen:

- Keine Typfehler und keine Typprüfung zur Laufzeit
- Kompiler kann besser optimieren

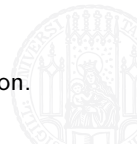
Dennoch können wir generischen Code schreiben:

```
rev_aux :: [a] -> [a] -> [a]
rev_aux [] acc = acc
rev_aux (h:t) acc = rev_aux t (h:acc)
```

Wird von Haskell's Typsystem geprüft und für sicher befunden, da der **universell quantifizierte** Typ `a` keine Rolle spielt:

- Werte von Typ `a` werden herumgereicht, aber nie inspiziert
- Code kann *unabhängig* von Typ `a` ausgeführt werden

`rev_aux` hat einen **Typparameter** und ist eine **polymorphe** Funktion.



POLYMORPHE FUNKTIONEN

Parametrisch polymorphe Funktionen aus der Standardbibliothek:

```
id :: a -> a
```

```
id x = x
```

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

```
snd :: (a,b) -> b
```

```
snd (_,y) = y
```

```
replicate :: Int -> a -> [a]
```

```
drop :: Int -> [a]-> [a]
```

```
zip :: [a] -> [b] -> [(a,b)]
```



POLYMORPHE FUNKTIONEN

Parametrisch polymorphe Funktionen aus der Standardbibliothek:

```
id :: a -> a
```

```
id x = x
```

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

```
snd :: (a,b) -> b
```

```
snd (_,y) = y
```

```
replicate :: Int -> a -> [a]
```

```
drop :: Int -> [a]-> [a]
```

```
zip :: [a] -> [b] -> [(a,b)]
```

Polymorphie macht Typen auch zu einer starken Spezifikation: es gibt nur eine *semantische* Möglichkeit, `id` oder `snd total` zu implementieren!

(Bei `replicate` gilt das nicht.)



INSTANTIIERUNG

Wir dürfen für *jede* Typvariable *einen* beliebigen Typ einsetzen, d.h. wir können `reverse :: [a] -> [a]` instantiieren mit

```
[Int] -> [Int]
```

```
[Bool] -> [Bool]
```

```
String -> String
```

```
[[[(Int,Bool)],String]] -> [[[(Int,Bool)],String]]
```

Solche Typen bezeichnet man auch als **Monotypen**

GEGENBEISPIEL

Instantiierung beispielsweise zu `[Int] -> [Bool]` ist nicht erlaubt, denn dazu wäre der polymorphe Typ `[c] -> [d]` notwendig!

HINWEIS

Die äußere Liste hat hier keine Sonderrolle:

der Typ `Maybe a -> a` instantiiert genauso zu `Maybe Int -> Int` oder `Maybe (Int, [Bool]) -> (Int, [Bool])`



MEHRERE TYPPARAMETER

$$\text{fst} :: (a,b) \rightarrow a$$

$$\text{fst} \quad (x,_) = x$$

$$\text{snd} :: (a,b) \rightarrow b$$

$$\text{snd} \quad (_,y) = y$$

Das Typsystem unterscheidet verschiedene Typparameter:

$a \rightarrow b \rightarrow (a,b)$	$b \rightarrow a \rightarrow (b,a)$	Identischer Typ
$a \rightarrow b \rightarrow (a,b)$	$a \rightarrow b \rightarrow (b,a)$	Unterschiedlich
$a \rightarrow b \rightarrow (a,b)$	$a \rightarrow a \rightarrow (a,a)$	Unterschiedlich

- Namen der Typparameter sind unerheblich; wichtig ist aber, ob zwei Typparameter den gleichen Namen tragen!
- Verschiedene Typparameter dürfen gleich instantiiert werden; Werte des Typs $a \rightarrow a \rightarrow (a,a)$ dürfen auch dort eingesetzt werden wo $a \rightarrow b \rightarrow (a,b)$ erwartet wird, aber nicht umgekehrt!

BEISPIEL

Die Funktion `fst` kann auch auf ein Paar

$(\text{True}, \text{True}) :: (\text{Bool}, \text{Bool})$ angewendet werden.



EINGESCHRÄNKTE POLYMORPHIE

Manchmal müssen wir die Polymorphie jedoch einschränken:

Z.B. sind Sortierverfahren generisch, aber wie vergleichen wir zwei Elemente, wenn wir deren Typ nicht kennen?

Vergleich zweier Werte des Typs `Char` funktioniert z.B. ganz anders als etwa bei zwei Werten des Typs `[Maybe Double]`

Sortierende Funktion mit Typ `[a] -> [a]` ist also nicht möglich.

LÖSUNGEN

① Typen höherer Ordnung

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

Vergleichsoperation als Argument übergeben

② Typklassen `sort :: Ord a => [a] -> [a]`

Gleiche Idee, jedoch versteckt: Funktion bekommt zur Laufzeit ein **Wörterbuch** mit (mehreren) Funktion implizit übergeben.



EINGESCHRÄNKTE POLYMORPHIE

Manchmal müssen wir die Polymorphie jedoch einschränken:

Z.B. sind Sortierverfahren generisch, aber wie vergleichen wir zwei Elemente, wenn wir deren Typ nicht kennen?

Vergleich zweier Werte des Typs `Char` funktioniert z.B. ganz anders als etwa bei zwei Werten des Typs `[Maybe Double]`

Sortierende Funktion mit Typ `[a] -> [a]` ist also nicht möglich.

LÖSUNGEN

① Typen höherer Ordnung

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

Vergleichsoperation als Argument übergeben

② Typklassen `sort :: Ord a => [a] -> [a]`

Gleiche Idee, jedoch versteckt: Funktion bekommt zur Laufzeit ein **Wörterbuch** mit (mehreren) Funktion implizit übergeben.



AD-HOC-POLYMORPHISMUS

Wie kann man eine Vergleichsfunktion für alle Typen implementieren?

```
(==) :: a -> a -> Bool  
x == y = ???
```

⚡ Geht nicht, da wir je nach Typ `a` anderen Code brauchen!
Z.B. Vergleich von zwei Listen komplizierter als von zwei Zeichen.

In vielen Sprachen sind **überladene** Operationen eingebaut:
Meistens entscheidet eine Typprüfung *zur Laufzeit* über den zu verwendenden Code \Rightarrow **Ad-hoc-Polymorphismus**.

PROBLEME:

- Typprüfung benötigt Typinformation zur Laufzeit
- eingebaute Operation können dann oft nur schwerlich mit benutzerdefinierten Datentypen umgehen



POLYMORPHE GLEICHHEIT

Welchen Typ hat (==)?

```
> :type (==)
```

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

Lies “für alle Typen `a` aus der Typklasse `Eq`”, also
“für alle Typen `a`, welche wir vergleichen können”



POLYMORPHE GLEICHHEIT

Welchen Typ hat `(==)`?

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

Lies “für alle Typen `a` aus der Typklasse `Eq`”, also
“für alle Typen `a`, welche wir vergleichen können”

`Eq a =>` ist **Typklasseneinschränkung** (engl. class constraint):

- Polymorphismus der Vergleichsfunktion `(==)` ist eingeschränkt
- Zur Laufzeit wird ein Wörterbuch (Dictionary) mit dem Code der passenden Funktionen *implizit* als weiteres Argument übergeben;
keine Typprüfung zur Laufzeit in Haskell

Verwenden wir `(==)` in einer polymorphen Funktionsdefinition, so muss auch deren Typsignatur entsprechend eingeschränkt werden.



SYNTAX CLASS CONSTRAINTS

In Typsignaturen kann der Bereich eingeschränkt werden, über den eine Typvariable quantifiziert ist:

```
firstDiff :: Eq a => [a] -> [a] -> Maybe a
```

Die polymorphe Funktion `firstDiff` kann nur auf Werten von Typen angewendet werden, welche `Eq` instanziiieren.

Es ist auch möglich, mehrere Einschränkungen anzugeben:

```
foo :: (Eq a, Read a) => [String] -> [a]
```

... und/oder mehrere Typvariablen einzuschränken:

```
bar :: (Eq a, Eq b, Read b, Ord c) => [(a,b)] -> [c]
```

Solche Typsignaturen kann GHC nicht nur inferieren, sondern wir können diese auch explizit in unserem Code hinschreiben und erzwingen.



TYPKLASSEN

Typklasse ist Menge von Typen plus Menge von Funktions-signaturen, welche auf alle Typen der Klasse anwendbar sind.

BEISPIEL:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y     = not (x == y)
```

Auf jeden Typ `a`, der eine **Instanz** der Klasse `Eq` ist, können wir die beiden 2-stelligen Funktionen `(==)` und `(/=)` anwenden.

Mit Typklassen funktioniert **Überladen** in Haskell sicher und ohne eingebaute Sonderbehandlungen, auch für benutzerdefinierte Datentypen!



TYPKLASSEN

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y      = not (x == y)
```

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- Keywords `class` und `where`
- Klassennamen wie Typen immer groß schreiben
- Klassendefinition kann überschreibbare Defaults enthalten

TYPSIGNATUR SAGT HIER: `(==)` nur auf Werte gleichen Typs anwendbar
Obwohl `String` und `Int` beide Instanzen der Klasse `Eq` sind, ist
Ausdruck `"Ugh" == 5` ist nicht Typ-korrekt.

GHC-Fehlermeldungen enthalten oft viele Typklassen Angaben
⇒ Klassendefinition entsprechen grob Java-Interfaces

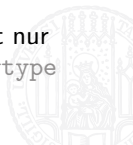
BEISPIEL

```
foo :: K11 a, K11 c, K12 c => a -> b -> (a,c)
```

- `a` ist irgendein Typ der Typklasse `K11`
- `b` kann beliebiger Typ sein
- `c` ist irgendein Typ der Typklassen `K11` und `K12`

Ein Typ kann mehreren Typklassen auf einmal angehören, aber es ist nur *eine* Instanz pro Klasse definierbar.

Abhilfe: 2.38: `newtype`



INSTANZEN DEFINIEREN

Man kann leicht neue Instanzen für eigene Datentypen definieren:

```
data Früchte = Apfel (Int,Int) | Birne Int Int

instance Eq Früchte where
  Apfel x1      == Apfel x2      = x1 == x2
  Birne x1 y1 == Birne x2 y2 = x1 == x2 && y1 == y2
  _             == _             = False
```

Damit können wir Äpfel und Birnen vergleichen, wenn wir möchten:

```
> Apfel (3,4) == Apfel (3,4)
True
> Apfel (3,4) == Apfel (3,5)
False
> Apfel (3,4) /= Birne 3 4
True
```

Definition für `/=` verwendet Klassen Default-Implementierung, da wir diese nicht neu definiert haben.



INSTANZEN ABLEITEN

Man kann leicht neue Instanzen für eigene Datentypen definieren:

```
data Früchte = Apfel (Int,Int) | Birne Int Int

instance Eq Früchte where
  Apfel x1      == Apfel x2      = x1 == x2
  Birne x1 y1 == Birne x2 y2 = x1 == x2 && y1 == y2
  _             == _             = False
```

Diese Instanz Deklaration ist langweilig und ohne Überraschungen:
Bei gleichen Konstruktoren vergleichen wir einfach
nur die Argumente der Konstruktoren.

GHC kann solche Instanzen mit `deriving` automatisch generieren:

```
data Früchte = Apfel (Int,Int) | Birne Int Int
  deriving (Eq, Ord, Show, Read)
```

Anzeige des erzeugten Codes mit: `-ddump-deriv`



GRUNDLEGENDE TYPKLASSEN

Klasse	Funktion
<code>Eq</code>	Gleichheit testen
<code>Ord</code>	Größenvergleiche
<code>Ix</code>	Als Index mit Teilbereichen verwendbar
<code>Enum</code>	Aufzählbar (erlaubt <code>[x..y]</code>)
<code>Bounded</code>	hat Größtes/Kleinste Element
<code>Show</code>	Konvertierung nach <code>String</code>
<code>Read</code>	Konvertierung von <code>String</code>
<code>Num</code>	Ganzzahlige Dinge
<code>Integral</code>	Ganzzahlige mit Ganzzahl-Division <code>div</code>
<code>Rational</code>	Ganzzahlige Brüche
<code>Real</code>	Zu <code>Rational</code> konvertierbar
<code>Fractional</code>	Zahlen mit Fließkomma-Division
<code>Floating</code>	Fließkommazahlen

Mit `deriving` ableitbar sind genau die oberen 7 Typklassen.



ZAHLEN

- Es gibt noch weitere spezielle Zahlenklassen (z.B. `RealFrac`)
- Wichtige Funktionen zur Umwandlung von Zahlen sind:

```

fromIntegral :: (Integral a,      Num b) => a -> b
realToFrac   :: (Real a, Fractional b) => a -> b
round        :: (Integral b, RealFrac a) => a -> b
ceiling      :: (Integral b, RealFrac a) => a -> b
floor        :: (Integral b, RealFrac a) => a -> b

```

- Viele numerische Funktion außerhalb von Typklassen definiert, welche so bequeme Verwendung ohne explizite Umwandlung ermöglichen:

```

(^)  :: (Num a, Integral b) =>      a -> b -> a
     -- verlangt positive Exponenten
(^^) :: (Fractional a, Integral b) => a -> b -> a
     -- erlaubt negative Exponenten

```



UNTERKLASSEN

Beispiel: Klasse `Ord` für geordnete Typen ist **Unterklasse** von `Eq`:
 Alles was wir nach Größe ordnen können, können auch vergleichen!

```
data Ordering = LT | EQ | GT

class (Eq a) => Ord a where
  compare          :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min         :: a -> a -> a
```

Alle Typen der Typklasse `Ord` müssen auch in `Eq` sein.

Ob die Semantik von `compare` und `(==)` übereinstimmt, wird von GHC jedoch *nicht* geprüft! Invarianten beachten!

- Eine Klasse kann mehrere Oberklassen haben
- Einschränkung auf Unterklasse reicht, d.h. `Ord a => a -> a` ist äquivalent zu `(Eq a, Ord a) => a -> a`.
- Unabhängig davon können auch die Funktionen einer Klasse eigene Klassenbeschränkungen besitzen.



ÜBERLADENE INSTANZEN

Auch Instanzdeklarationen dürfen selbst überladen werden:

BEISPIEL

Wenn wir jeweils zwei Werte von Typ `a` und zwei von Typ `b` vergleichen können, dann können wir auch automatisch Tupel des Typs `(a,b)` auf Gleichheit testen, dank der Definition:

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = (x1==x2) && (y1==y2)
```

BEISPIEL

Wenn wir jeweils zwei Werte von Typ `a` vergleichen können, dann können wir auch automatisch eine Liste des Typs `[a]` vergleichen:

```
instance (Eq a) => Eq [a] where
  (x:xs) == (y:ys) = x == y && xs == ys
  []       == []   = True
  _xs     == _ys   = False
```

Erster Vergleich: `Eq a`

Zweiter Vergleich: `Eq [a]` rekursiv!



ZUSAMMENFASSUNG TYPKLASSEN

- Typklassen definieren ein **abstraktes Interface** bzw. Typklassen definieren polymorphe Funktionen für eine eingeschränkte Menge von Typen
- Überladen wird in Haskell durch Typklassen realisiert, welches benutzerdefiniertes Überladen erlaubt
- Ein Typ kann mehreren Typklassen angehören, je nach Eigenschaften. Keine Hierarchie nötig! anders als z.B. in Java
- Jeder Typ kann nur eine Instanz pro Klasse deklarieren
- Typklassen können mit Unterklassen erweitert werden
- Beliebige Funktionsdefinition können mit Typklassen eingeschränkt werden

Siehe auch *“ML-Modules and Haskell Type Classes: A Constructive Comparison”* von Stefan Wehr and Manuel M. T. Chakravarty über die Mächtigkeit von Typklassen.



TYPKLASSE FUNCTOR

Im Module `Data.Functor` findet sich folgende Definition:

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

  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap      -- Infix-Synonym
```

Gesetze:

IDENTITÄT: `fmap id == id`

KOMPOSITION: `fmap f . fmap g == fmap (f . g)`

Der Parameter `f` der Typklasse `Functor` steht also nicht für einen konkreten Typ wie z.B. `Tree Int`, sondern für einen Typkonstruktor wie z.B. `Tree`.

Die Typklasse `Functor` ist also die Klasse aller Typen-in-Kontext, welche es erlauben Ihre Inhalte auf andere im gleichen Kontext abzubilden.

BEISPIEL: FUNCTOR-INSTANZ FÜR LISTEN

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

BEISPIEL: Instanz für Listen

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

Typ instantiiert zu `fmap :: (a -> b) -> [a] -> [b]`

```
> even <$> [1..5]
[False,True,False,True,False]
```


BEISPIEL: FUNCTOR-INSTANZ FÜR LISTEN

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

BEISPIEL: Instanz für Listen

`fmap` für Listen kennen wir bereits unter dem Namen `map`:

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

Der Typ `map :: (a -> b) -> [a] -> [b]` passt genau!

```
> even `map` [1..5]
[False,True,False,True,False]
```

BEISPIEL: FUNCTOR-INSTANZ FÜR BÄUME

“Container”-Datentypen können wir leicht zu Instanzen von `Functor` machen:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

```
instance Functor Tree where
```

```
  fmap f (Leaf a)      = Leaf (f a)
```

```
  fmap f (Node l a r) = Node (fmap f l) (f a) (fmap f r)
```

```
> fmap even (Node (Leaf 1) 2 (Leaf 6))  
Node (Leaf False) True (Leaf True)
```

```
> fmap (*2) (Node (Leaf 1) 2 (Leaf 3))  
Node (Leaf 2) 4 (Leaf 6)
```

BEISPIEL: FUNCTOR-INSTANZ FÜR MAYBE

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

Eine Funktion auf ein Maybe anwenden:

```
data Maybe a = Nothing | Just a
instance Functor Maybe where
  fmap g (Just x) = Just (g x)
  fmap _ Nothing = Nothing
```

`Nothing` bleibt `Nothing`; aber auf Inhalte von `Just` wird die gegebene Funktion angewendet und das Ergebnis wieder verpackt:

```
> fmap even (Just 42)
Just True
> fmap even Nothing
Nothing
```

GENERISCHE FUNKTOR INSTANZEN

Für GHC beinhaltet eine Erweiterung, welche `Functor`-Instanzen für Container-Datentypen automatisch generieren kann:

```
{-# LANGUAGE DeriveFunctor #-}
```

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)  
  deriving (Functor )
```

- **Pragma** `{-# ... #-}` am Anfang der Datei beeinflusst den Compiler.
- Mit dem Pragma `LANGUAGE` werden **Spracherweiterung** gegenüber dem Haskell Standard aktiviert Mehrere Spracherweiterung werden dabei durch Kommas getrennt.
- Weitere Spracherweiterungen erlauben automatische Ableitung von Instanzen für einige weitere Klassen:
`DeriveFoldable`, `DeriveTraversable`, `DeriveLift`,...

FUNCTOR EITHER ?

`Either` können wir aber so nicht zu einer Instanz von `Functor` machen:

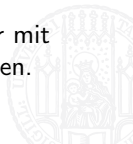
```
data Either a b = Left a | Right b
```

```
instance Functor Either where
  fmap f (Left a) = Left $ f a
  fmap f (Right b) = Right $ f b
```

Error:

- Expecting one more argument to ‘Either’
Expected kind ‘* -> *’, but ‘Either’ has kind ‘* -> * -> *’
- In the first argument of ‘Functor’, namely ‘Either’
In the instance declaration for ‘Functor Either’

Es liegt ein **Kind**-Fehler vor: `Functor` erwartet einen Typkonstruktor mit einem Argument, aber `Either` ist Typkonstruktor mit zwei Argumenten.



KIND

Der **Kind** (*engl. für Sorte*) eines Typen beschreibt die Art des Typen, also die Anzahl und Art der Argumente eines Typkonstruktors.

Ein Kind ist entweder `*` oder aus zwei Kinds per Pfeil zusammengesetzt:

$$\kappa ::= * \mid \kappa \rightarrow \kappa$$

- (`*`) steht für alle konkreten Datentypen, z.B. `Int`, `Bool`, `Double` und auch `[Int]`, `Maybe Bool`, `Either String Double`
- (`* -> *`) steht für alle Typkonstruktoren mit genau einem Argument, z.B. `[]`, `Maybe` und auch `Either String`.
- (`* -> (* -> *)`) steht für alle Typkonstruktoren mit genau zwei Argumenten, z.B. `Either`.

Wie bei Funktionstypen ist die Rechtsklammerung implizit, d.h.

$$* \rightarrow (* \rightarrow *) = * \rightarrow * \rightarrow * \neq (* \rightarrow *) \rightarrow *$$



KIND

GHCi kann den Kind eines Typen mit `:kind` anzeigen:

```
> :kind Integer
Integer :: *
> :ki Maybe
Maybe :: * -> *
> :k Either
Either :: * -> * -> *
```

`Maybe` und `Either` haben also unterschiedliche Kinds, da diese Typkonstruktoren unterschiedliche viele Parameter verlangen.

Nicht vergessen, Funktionstypen sind auch Typen:

```
> :k (Int -> Int)
(Int -> Int) :: *
> :k ((->) Int)
((->) Int) :: * -> *
> :k (->)
(->) :: * -> * -> *
```



KIND OF FUNCTOR

Angewendet auf die Typklasse `Functor` gibt GHCi folgendes heraus:

```
> :k Functor
Functor :: (* -> *) -> Constraint
```

Funktoreninstanzen benötigen ein Typargument mit Kind `* -> *`

```
> :k Functor Maybe
Functor Maybe :: Constraint
```

```
> :k Functor f => (a -> b) -> f a -> f b
Functor f => (a -> b) -> f a -> f b :: *
```

Die GHC-speziielle Antwort `Constraint` bedeutet dabei, dass es sich nicht um einen Typen, sondern um eine Typklasse(-nbeschränkung) handelt, wie wir sie vor dem Doppelpfeil `=>` schreiben.



FUNKTOR-INSTANZ FÜR EITHER

```
> :k Either Int
Either Int :: * -> *
```

Auch wenn wir `Either` nicht zur Funktoereninstanz machen können, so können wir immerhin eine für `Either a` definieren:

```
data Either a b = Left a | Right b
  deriving (Eq, Ord, Read, Show, Typeable)
```

```
instance Functor (Either a) where
  fmap _ (Left x) = Left x
  fmap f (Right y) = Right (f y)
```

So ist es im Modul `Data.Either` auch definiert.

BEISPIEL:

```
> fmap (*3) $ Left "Error"
Left "Error"
> fmap (*3) $ Right 23
Right 69
```



FUNCTOR-INSTANZ FÜR EITHER II

Ein `flip`-Äquivalent gibt es nicht direkt, aber wie können mit `newtype` einen äquivalenten Datentyp mit eigener Instanz einführen:

```
newtype FlipEither b a = FE (Either a b)
  deriving Show
```

```
instance Functor (FlipEither b) where
  fmap f (FE (Left y)) = FE $ Left $ f y
  fmap _ (FE (Right x)) = FE $ Right x
```

Dies ist auch zwingend notwendig, da ja anhand des Typen entschieden wird, welcher Code ausgeführt wird!

Immer nur eine Instanz pro Typ und Klasse!

```
> fmap (*2) $ Left 21
Left 21
> fmap (*2) $ FE $ Left 21
FE (Left 42)
```



NEWTYPEDeKLARATION

newtype-Deklaration ist ein optimierter Spezialfall zum "Kopieren" eines existierenden Typs. Das Typsystem unterscheidet beide Typen!

```
newtype Typname = Konstruktor <typ> deriving <Typklassen>
```

- Frischer Typname und frischer Konstruktor mit 1 Argument
- Typ und Konstruktor werden oft gleich benannt, da Typen und Werte in getrennten Namensräumen leben Bool ≠ True
- Übernahme beliebiger vorhandener Instanzen erlaubt mit Erweiterung GeneralisedNewtypeDeriving

UNTERSCHIED ZUR HERKÖMMLICHEN DEKLARATION

```
data Typname = Konstruktor <typ> deriving <Typklassen>
```

- `newtype` effizienter, da Konstruktor zur Laufzeit nicht existiert
- `newtype` ist fauler, da kein Pattern-Match ausgeführt wird

Beide Varianten erlauben eigene Typklassen-Instanzen!



BEISPIEL: DOWN

Modul `Data.Ord` definiert:

```
newtype Down a = Down a deriving (Eq, Show, Read)
```

```
instance Ord a => Ord (Down a) where  
  compare (Down x) (Down y) = y `compare` x
```

Damit können wir für jeden Typ `a` aus der Klasse `Ord` die Ordnung leicht umdrehen:

```
> sort [5,3,1,4,2]  
[1,2,3,4,5]  
> sort $ Down <$> [5,3,1,4,2]  
[Down 5,Down 4,Down 3,Down 2,Down 1]
```

Mit `newtype` können wir also leicht alternative Instanzdeklarationen definieren, also die Beschränkung, pro Typ und Klasse immer nur eine Instanz zu haben, umgehen.



LAZINESS VON NEWTYPE

`newtype` ist *fauler*, da keine Pattern-Matches ausgefuehrt werden – der Konstruktor existiert ja zur Laufzeit gar nicht:

```
data    Foo a = Foo a deriving Show
newtype Baz a = Baz a deriving Show
```

```
foo :: Foo a -> String
foo (Foo _) = "OK!"
```

```
baz :: Baz a -> String
baz (Baz _) = "OK!"
```

```
> foo undefined
*** Exception: Prelude.undefined
> baz undefined
"OK!"
> foo (Foo undefined)
"OK!"
```



FUNKTOEREN

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

`fmap f` wendet Funktion `f` punktweise auf eine Datenstruktur an.

Dabei sollten folgende Gesetze erfüllt werden:

- 1 Identität: `fmap id == id`
- 2 Komposition: `fmap f . fmap g == fmap (f . g)`

GHC prüft dies nicht, Programmierer muss dies alleine sicherstellen!

Der Begriff "Funktor" kommt aus der Mathematik: ein Funktor ist strukturerhaltende Abbildung zwischen zwei Kategorien



ZUSAMMENFASSUNG FUNKTOEREN

- Funktoeren sind ein Programmierschema für die punktweise Anwendung einer Funktion auf eine Datenstruktur
- *Identität*: Funktoeren verändern nie die Form einer Datenstruktur, sondern nur deren Inhalt
- *Komposition*: Es ist egal, ob wir mehrfach über die Datenstruktur gehen oder nur einmal und dabei gleich mehrere Funktionen hintereinander punktweise anwenden.

⇒ Änderungen durch Funktoeren innerhalb einer Datenstruktur sind immer lokal und voneinander unabhängig!



AUSBLICK: POLYMORPHISMUS

Es gibt noch weitere Arten von Polymorphismus. GHC kennt Erweiterungen für manche, z.B.:

- Rank-N Types
- Impredicative Types

aber nicht für alle, z.B.:

- Subtyping

Die Verwendung solcher Erweiterungen benötigt oft die explizite Angabe von Typsignaturen, da die Inferenz solcher Typen im allgemeinen schnell unentscheidbar wird.

Auch für **Polymorphe Rekursion**, d.h. der Typ des rekursiven Aufrufs ändert sich, benötigt GHC eine explizite Typsignatur.



ÜBERSICHT POLYMORPHISMUS

PARAMETRISCHER POLYMORPHISMUS

- Unabhängig vom tatsächlichen Typ wird immer der gleiche generische Code ausgeführt.
- Realisiert in Haskell über Typparameter: `a -> a`

AD-HOC-POLYMORPHISMUS

- Funktionsnamen werden überladen, d.h. abhängig vom tatsächlichen Typ wird spezifischer Code ausgeführt
- Realisiert in Haskell durch Einschränkung der Typparameter auf Typklassen: `Klasse a => a -> a`

SUBTYP POLYMORPHISMUS

- Mischform: Es kann der gleiche Code oder spezifischer Code ausgeführt werden.
- Irrelevant für uns, da GHC hat keinen direkten Subtyp-Mechanismus hat (wie z.B. Vererbung in Java).



WEITERE WICHTIGE TYPKLASSEN:

Klasse	Beschreibung
<code>Data.Functor</code>	Funktionen auf Inhalte anwendbar
<code>Control.Applicative</code>	Applikative Funktoeren später: ??
<code>Control.Monad</code>	Monadern später: ??
<code>Data.Foldable</code>	Faltbare Container, z.B. aufsummieren
<code>Data.Traversable</code>	In fester Reihenfolge begehbare Container
<code>Data.Monoid</code>	eine assoziative binäre Operation

HINWEIS: Viele Funktionen der Standardbibliothek, welche früher nur auf Listen definiert waren, sind inzwischen mit den obigen Typklassen verallgemeinert worden.

Durch diese Verallgemeinerung können manchmal furchteinflößend kompliziert aussehende Fehlermeldungen entstehen (welche aber letztendlich simpel sind).



AUFGABEN VON MODULSYSTEMEN

Größere Softwareprojekte benötigt Struktur \implies Module

- Module erlauben Untergliederung in Teilprojekte, welche *separat kompiliert* werden können
- Module organisieren und separieren **Namensraum**
- Module ermöglichen es, Implementierungsdetails zu verstecken. Damit kann eine leichte Austauschbarkeit eines Moduls sichergestellt werden.

Teile und Herrsche:

Gute Modularisierung bedeutet gute Wartbarkeit



MODULE IN HASKELL

Ein Haskell Programm besteht aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulnamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen GHC
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
`ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs` GHC
- Jedes Modul hat seinen eigenen Namensraum:
`Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
Es gibt viele weitere nützliche Standard-Module:
`Data.List`, `Data.Set`, `Data.Map`, `System.IO`, ...
- Haskell Module sind recht simpel, z.B. nicht parametrisiert



EXPORT

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)           = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von Datentyp `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

Module erlauben so Erstellen von echten **abstrakte Datentypen**, deren Implementierung nicht zugänglich ist, indem keine Konstruktoren exportiert werden, z.B. falls oben `Oak()` darstünde.



BEISPIEL: ABSTRAKTER DATENTYP

```
module SomeData (SD(), create, toInt) where

  data SD = External Int | Internal String
    deriving Show

  create :: Int -> SD
  create i = External i

  toInt :: SD -> Int
  toInt (External i) = i
  toInt (Internal s) = length s
```

Benutzer des Moduls können weder `External` noch `Internal` benutzen, d.h. nur mit `create` können Werte des Typs `SD` erzeugt werden; und nur mit `toInt` verarbeitet werden; und ansonsten nur weitergereicht werden.

IMPORT

```
module Import where
  import Prelude
  import MyModul - definiert foo und bar

  myfun x = foo x + MyModul.bar x
  ...
```

- Wenn ein Modul importiert wird, werden alle exportierten Deklaration in dem importierenden Modul sichtbar.
- Man kann diese direkt verwenden, oder mit `ModulName.bezeichner` ansprechen
- Mehrfachimport unproblematisch, so lange alle Pfade zum selben Modul führen
- Re-export ist auch möglich, z.B. mit

```
module Import (module MyModule) where
```



IMPORT

Beim Importieren kann weiter eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree     hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus MyModul werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Tree`



IMPORT

Beim Importieren kann weiter eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden wenn wir beide brauchen



IMPORT

Beim Importieren kann weiter eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden wenn wir beide brauchen



IMPORT

Beim Importieren kann weiter eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree     hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen



IMPORT

Beim Importieren kann weiter eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

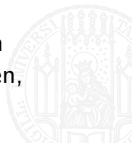


IMPORT

Beim Importieren kann weiter eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen



ZUSAMMENFASSUNG MODULE

Module erlauben also:

- Unterteilung des Namensraums
- Aufteilung von Code auf mehrere Dateien
- Unterstützen Modularisierung und Abstraktion durch das Verstecken von Implementierungsdetails
- Gegenseitige Abhängigkeiten zwischen Modulen nicht erlaubt
- *Achtung*: alle Instanzdeklarationen für Typklassen werden *immer* exportiert und importiert

In GHCi importieren wir Module mit

```
> :module + Data.List Data.Set
```

und schließen Module mit

```
> :module - Data.List Data.Set
```



BEISPIEL: ENDLICHE ABBILDUNGEN

```
module FinMap (FinMap, emptyM, insertM, insertL, lookupM) where

import Data.List

class FinMap m where
  emptyM  :: (Ord k) => m k v
  insertM :: (Ord k) => k -> v -> m k v -> m k v
  lookupM :: (Ord k) => k -> m k v -> Maybe v
  insertL :: (Ord k) => [(k,v)] -> m k v -> m k v
  insertL xs mp = foldl' (\m (k,v)-> insertM k v m) mp xs
```

Klasse für endliche Abbildungen mit minimalen Interface:

- nur `emptyM` liefert uns eine Abbildung
- `insertM` können Schlüssel/Wert-Paare eingefügt werden
- `lookupM` erlaubt abfrage eines Schlüssels



BEISPIEL: ENDLICHE ABBILDUNGEN

```
module FinMap.AL (module FinMap, Map()) where

import FinMap

-- data Map a b = Map [(a,b)]
newtype Map a b = Map [(a,b)]
    deriving Show

instance FinMap Map where
    emptyM    = Map []
    insertM x y (Map m) = Map $ (x,y):m
    lookupM x (Map []) = Nothing
    lookupM x (Map ((y,t):ys))
        | x == y    = Just t
        | otherwise = lookupM x (Map ys)
```

Implementierung durch Assoziationslisten;
vermutlich nicht die beste Wahl.



BEISPIEL: ENDLICHE ABBILDUNGEN

```
module FinMap.Map (module FinMap, Map()) where

import FinMap
import qualified Data.Map

newtype Map k v = Map (Data.Map.Map k v)

instance FinMap.Map where
  emptyM  = Map Data.Map.empty
  insertM k v (Map m) = Map $ Data.Map.insert k v m
  lookupM k (Map m) = Data.Map.lookup k m
```

Verwendung des Moduls `Data.Map` aus der Standardbibliothek; realisiert durch *size balanced binary trees*.



BEISPIEL: ENDLICHE ABBILDUNGEN

```
module Main where

-- import FinMap.AL
import FinMap.Map    -- einfach austauschbar

n  = 1999999
n2 = n `div` 2

garbage :: [(Integer,Integer)]
garbage = take (fromIntegral n) $ (n2,0) : zipWith (,) [1..] [-1,-2..]

m1 :: Map Integer Integer
m1 = emptyM
m11 = insertL garbage m1

main = do
  print $ lookupM 2 m11
  print $ lookupM 2 m11
  print $ lookupM n2 m11
```



KOMMENTAR ZUM BEISPIEL

- In der Standardbibliothek werden endliche Abbildungen durch das Modul `Data.Map` bereitgestellt.

Es wird dabei aber keine Klasse verwendet, sondern ausschließlich das Modulsystem: Man kann wählen zwischen `Data.Map.Lazy` und `Data.Map.Strict`. Das beide die gleichen Funktionen bieten, prüft der Compiler aber nicht.

- Für das Beispiel gibt es Alternativen:

- Durch die GHC-Erweiterungen

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}
```

kann man die Typen von Schlüssel und Werten explizit in die Klasse aufnehmen – ein Vor- und Nachteil!

```
class Ord k => FinMap m k v where
```

- Die Typen für Schlüssel und Werte lassen sich besser auch mit der Erweiterung `{-# LANGUAGE TypeFamilies #-}` in die Klasse aufnehmen

