Fortgeschrittene Funktionale Programmierung

Teil 2: Typklassen, Funktoren, Kinds und Module

Steffen Jost

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

23. Oktober 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 2: Typklassen, Funktoren, Kinds und Module

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:

  `type` Typabkürzungen für Lesbarkeit
  `data` Deklaration wirklich neuer Typen
  `newtype` wie `data`, falls 1 Konstruktor mit 1 Arg.

- Datentypen können andere Typen als Parameter haben, **Konstruktoren** können als Funktionen betrachtetet 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 Verwechseln

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
```

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.

$\Rightarrow$ *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.

$\Rightarrow$ *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 Kompiler prüft mit **statischem Typsystem** 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 **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

```
fst :: (a,b) -> a          snd :: (a,b) -> b
fst    (x,_) =  x          snd    (_,y) =  y
```

Das Typsystem unterscheidet verschiedene Typparameter:

`a -> b -> (a,b)`	`b -> a -> (b,a)`	Identischer Typ
`a -> b -> (a,b)`	`a -> b -> (b,a)`	Unterschiedlich
`a -> b -> (a,b)`	`a -> a -> (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 -> a -> (a,a)` dürfen auch dort eingesetzt werden wo `a -> b -> (a,b)` erwartet wird, aber nicht umgekehrt!

  Beispiel

  Die Funktion `fst` kann auch auf ein Paar `(True,True) :: (Bool,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 ⇒ **Ad-hoc-Polymorphismus.**

Probleme:

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

Polymorphe Gleicheit

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 Gleicheit

Welchen Typ hat `(==)`?

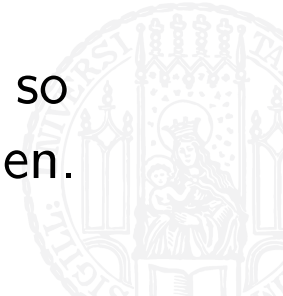
```
> :type (==)
(==) :: Eq a => a -> a -> Bool
```

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.



Beispiel

Verwendung der polymorphen Vergleichsfunktion in einer Funktionsdefinition, welche den ersten abweichenden Listenwert zweier Listen berechnet:

```
data Maybe a = Nothing | Just a -- zur Erinnerung
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing

> firstDiff [1..10] [1,2,3,99,5,6]
Just 99
```

Beispiel

Verwendung der polymorphen Vergleichsfunktion in einer Funktionsdefinition, welche den ersten abweichenden Listenwert zweier Listen berechnet:

```
data Maybe a = Nothing | Just a -- zur Erinnerung
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing

> firstDiff [1..10] [1,2,3,99,5,6]
Just 99
```

Die Typvariablen in der Typsignatur der Funktion `firstDiff` sind **eingeschränkt quantifiziert**:

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

Beispiel

```
firstDiff :: Eq a => [a] -> [a] -> Maybe a
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing
```

Beim Kompilieren fügt der Kompiler der Funktion ein implizites Argument pro class constraint hinzu. Dieses Argument ist ein Paket mit alle Funktionen der Typklasse `Eq` für den entsprechenden Typ, welches zur Laufzeit übergeben wird.
In diesem Fall also der passende Code für `(==)` und `(/=)`,
Wir brauchen uns darum nicht zu kümmern!

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` instanziieren.

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 für uns automatisch ableiten, aber wir können diese auch explizit hinschreiben und erzwingen.

Typklassen

Eine **Typklasse** ist eine Menge von Typen, plus eine Menge von Funktionen, welche auf alle Typen dieser Klasse anwendbar sind.

Beispiel einer Typklassen-Definition:

```
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: 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.
Standardbibliothek definiert viel Instanzen: `Eq Bool`, `Eq Double`,...

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

Eq Typklasse

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

- Klassennamen schreiben wir wie Typen immer groß
- In Klassendefinition ist `Eq a =>` implizit, d.h.
  `:type (==)` liefert `(==) :: Eq a => a -> a -> Bool`
- Klassendefinition können Default-Implementierung anbieten, welche übernommen oder ignoriert werden können
- `(==)` nur auf zwei Werte des gleichen Typs anwendbar.
  Der Ausdruck `"Ugh" == 42` ist nicht Typ-korrekt, obwohl `String` und `Int` beide in der Klasse `Eq` sind.
  Signatur `foo :: (Kl a, Kl b) => a -> b -> Bool` würde einen Ausdruck `"Ugh" `foo` 42` erlauben,
  falls `String` und `Int` Instanzen der Klasse `Kl` wären.

  $\Rightarrow$ Klassendefinition grob vergleichbar mit Interfaces in Java

Beispiel

```
foo :: Kl1 a, Kl1 c, Kl2 c => a -> b -> (a,c)
```

- `a` ist irgendein Typ der Typklasse `Kl1`
- `b` kann beliebiger Typ sein
- `c` ist irgendein Typ der Typklassen `Kl1` und `Kl2`

Ein Typ kann mehreren Typklassen auf einmal angehören, aber es ist nur *eine* Instanz pro Klasse definierbar. Abhilfe: 2.28: `newtype`

Instanzen definieren

Man kann leicht neue Instanzen für eigene Datentypen definieren:

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

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

Damit können wir Äpfel mit 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 Frucht = Apfel (Int,Int) | Birne Int Int

instance Eq Frucht 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 Frucht = Apfel (Int,Int) | Birne Int Int
  deriving (Eq, Ord, Show, Read)
```

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

Typklassen Instanzen

- Instanzdeklarationen müssen alle Funktionen der Klasse implementieren, welche keine Default-Implementierung haben
- Ein Typ kann mehreren Typklassen auf einmal angehören
- Aber nur *eine* Instanz pro Klasse definierbar

Beispiel: Wenn wir `Frucht` *manchmal* nur nach Preis vergleichen wollen, dann brauchen wir einen neuen Typ dafür:

```
data PreisFrucht = PF Frucht                -- besser: newtype
instance Eq PreisFrucht where
  PF(Apfel (_,p1)) == PF(Apfel (_,p2)) = p1 == p2
  PF(Apfel (_,p1)) == PF(Birne p2 _  ) = p1 == p2
  PF(Birne p1 _  ) == PF(Apfel (_,p2)) = p1 == p2
  PF(Birne p1 _  ) == PF(Birne p2 _  ) = p1 == p2
```

...oder wir schreiben ohne Typklassen nur eine Funktion

```
preisvergleich :: Frucht -> Frucht -> Bool
```

Grundlegende Typklassen

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

Mit `deriving` ableitbar sind die oberen 7 Typklassen und weitere.

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`:
Nach Größe ordnen setzt voraus, dass wir auch vergleichen können!

```
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 dieser 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.

Newtype Deklaration

`newtype`-Deklaration ist ein optimierter Speziallfall 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` $\neq$ `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 ausgeführt 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!"
```

Typklasse Semigroup

Modul `Data.Semigroup` definiert eine simple Klasse für Halbgruppen (Menge mit assoziativer binärer Operation):

```
class Semigroup a where
  (<>) :: a -> a -> a                  -- should be associative

  stimes  :: Integral n => n -> a -> a   -- fails for n<1
  sconcat :: NonEmpty a -> a
```

Gesetz `x <> (y <> z) == (x <> y) <> z`

Sonstiges

- Instanzdeklarationen müssen nur `(<>)` definieren; die anderen Funktionen besitzen Standardimplementierungen
- `NonEmpty a` ist ein Typ für nicht-leere Listen;
  Konstruktor: `(:|) :: a -> [a] -> NonEmpty a`
  Beispiel: `1 <| 2 <| 3 :| [4,5] == 1 :| [2,3,4,5]`

Beispiel: Semigroup

Ebenfalls definiert in `Data.Semigroup` (und analog für `Max`):

```
newtype Min a = Min a
getMin :: Min a -> a
getMin (Min x) = x

instance Ord a => Semigroup (Min a) where
  Min a <> Min b = Min (min a b)
```

Anwendung:

```
> getMin $ Min 7 <> Min 5
5

Prelude Data.Semigroup Data.List.NonEmpty
> getMin $ sconcat $ map Min $ 5:|[0,-5,3,11,-7,9]
-7
```

Typklasse Monoid

Modul `Data.Monoid` der Standardbibliothek definiert:

```
class Semigroup a => Monoid a where
  mempty  :: a                  -- eine Konstante
                                -- neutrales Element für (<>)
  mappend :: a -> a -> a
  mappend = (<>)                -- immer noch assoziativ

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Gesetze Konstante `mempty` ist neutral zu `mappend`:

```
mempty <> x   ==   x   ==   x <> mempty
```

Mathematiker bezeichnen mit **Monoid** eine Halbgruppe mit Eins, d.h. eine Halbgruppe, welche ein neutrales Element besitzt.

Typklasse Monoid

Modul `Data.Monoid` der Standardbibliothek definiert:

```
class Semigroup a => Monoid a where
  mempty  :: a                  -- eine Konstante
                                -- neutrales Element für (<>)
  mappend :: a -> a -> a
  mappend = (<>)                -- immer noch assoziativ

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Hinweis: `Monoid` ist erst ab GHC 8.4.x eine Unterklasse von `Semigroup`. Vorher waren beide Typklassen unabhängig voneinander und die Typklasse `Monoid` forderte zusätzlich, dass `mappend` assoziativ ist.
Wenn man bei Monoid-Instanzen `import Data.Semigroup` und explizit `mappend = (<>)` hinschreibt, dann geht es mit beiden Versionen.

d.h. eine Halbgruppe, welche ein neutrales Element besitzt.

Beispiel: Listen bilden Monoide

Ein wichtiges Monoid sind Listen:

```
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)   = (++)
instance Monoid [a] where
  -- mempty :: [a]
  mempty  = []
```

Das die geforderten Gesetze gelten ist offensichtlich:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiel: Listen bilden Monoide

Ein wichtiges Monoid sind Listen:

```
import Data.Semigroup                    - für GHC <8.4.x
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)    = (++)
instance Monoid [a] where
  -- mempty :: [a]
  mempty  = []
  mappend = (<>)                         - für GHC <8.4.x
```

Das die geforderten Gesetze gelten ist offensichtlich:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiel: Listen bilden Monoide

Ein wichtiges Monoid sind Listen:

```
import Data.Semigroup                    - für GHC <8.4.x
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)   = (++)
instance Monoid [a] where
  -- mempty :: [a]
  mempty  = []
  mappend = (<>)                         - für GHC <8.4.x
```

Das die geforderten Gesetze gelten ist offensichtlich:

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiele sind kein Beweis!

Beispiel: Listen bilden Monoide

Ein wichtiges Monoid sind Listen:

```
import Data.Semigroup                    - für GHC <8.4.x
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)    = (++)
```

Dies ist ein Beispiel für eine *nicht-kommutative* Halbgruppe, denn

```
[1,2]++[3,4,5] =[1,2,3,4,5]
               ≠[3,4,5,1,2] = [3,4,5]++[1,2]
```

Das

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiele sind kein Beweis!

Beispiel: Listen bilden Monoide

Ein wichtiges Monoid sind Listen:

```
import Data.Semigroup                    - für GHC <8.4.x
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>)    = (++)
```

Dies ist ein Beispiel für eine *nicht-kommutative* Halbgruppe, denn

```
[1,2]++[3,4,5] =[1,2,3,4,5]
               ≠[3,4,5,1,2] = [3,4,5]++[1,2]
```

Das

```
([1,2,3] ++ [4,5,6]) ++ [7,8,9] == [1,2,3,4,5,6,7,8]
[1,2,3] ++ ([4,5,6] ++ [7,8,9]) == [1,2,3,4,5,6,7,8]

[] ++ [1,2,3] == [1,2,3]
[1,2,3] ++ [] == [1,2,3]
```

Beispiele sind kein Beweis!

FRAGE: Welchen Sinn hat es, `[]` und `(++)` neue Namen zu geben?

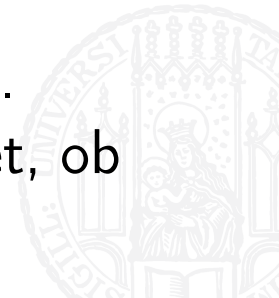
ANTWORT: Verallgemeinerung! `(++)` funktioniert nur auf Listen, aber `(<>)` funktioniert auf allen Halbgruppen/Monoiden. *Beispiele:*

```
> [1,2,3] <> mempty <> [4,5,6]
[1,2,3,4,5,6]

> Just [1,2,3]  <>  Just [4,5,6]
Just [1,2,3,4,5,6]
> Nothing  <>  Just [4,5,6]
Just [4,5,6]

> ([1,2,3],"abc") <> ([4,5,6],"def")
([1,2,3,4,5,6],"abcdef")
```

Nur das erste Beispiel hier würde auch mit `[]` und `(++)` gehen. Für eine Bibliothek ist das sehr nützlich: Der Nutzer entscheidet, ob er `[a]`, `Maybe [a]` oder `([a],[b])`, usw. haben möchte!



Beispiel: Maybe als Monoid

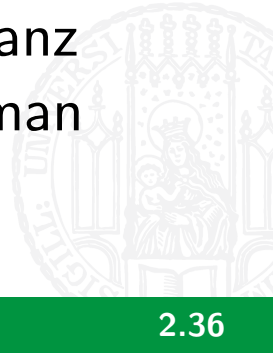
Wenn der Inhalt ein Monoid bildet, dann bildet auch die Verpackung `Maybe` ein Monoid:

```
instance Semigroup a => Semigroup (Maybe a) where
  Nothing <> b       = b
  a       <> Nothing = a
  Just a  <> Just b  = Just (a <> b)

instance Semigroup a => Monoid (Maybe a) where
  mempty = Nothing
```

Das die Gesetze gelten, kann man hier leicht nachrechnen!

Interessanterweise reicht als Voraussetzung für die Monoid-Instanz hier bereits die Halbgruppe aus. Dass dies hier gut geht muss man nachrechnen, oder man weiß es bereits aus der Mathematik.



Beispiel: Tupel als Monoide

Wenn die Inhalte ein Monoid bilden, dann bilden auch Paare davon wieder ein Monoid:

```
instance (Semigroup a, Semigroup b) =>
  Semigroup (a, b) where
    (a,b) <> (a',b') = (a<>a', b<>b')

instance (Monoid a, Monoid b) => Monoid (a,b) where
    mempty = (mempty, mempty)
```

Beachte: In der Definition von `(<>)` taucht `a<>a'` und `b<>b'` auf: Dies sind keine rekursiven Aufrufe! Stattdessen werden die Definition von `Semigroup a` und `Semigroup b` verwendet — was auch immer diese sind!

Beispiel: Zahlen bilden Monoide

Additives Monoid $(+, 0)$: Es gilt $(x + y) + z = x + (y + z)$

Multiplikatives Monoid $(\cdot, 1)$: Es gilt $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

In der Haskell Standardbibliothek wurde entschieden, dass man sich explizit entscheiden muss, welches Monoid man meint:

```
> getSum     $ Sum     3 <> mempty <> Sum     4
7
> getProduct $ Product 3 <> mempty <> Product 4
12
```

Realisiert wird das mit Hilfe von `newtypes`:

```
newtype Sum a     = Sum     { getSum :: a }
newtype Product a = Product { getProduct :: a }
instance Num a => Semigroup (Sum a)     where
  Sum     x <> Sum     y = Sum     (x+y)
instance Num a => Semigroup (Product a) where
  Product x <> Product y = Product (x*y)
instance Num a => Monoid (Sum a)     where mempty = Sum 0
instance Num a => Monoid (Product a) where mempty = Product 1
```

Beispiel: Zahlen bilden Monoide

Additives Monoid $(+, 0)$: Es gilt $(x + y) + z = x + (y + z)$

Multiplikatives Monoid $(\cdot, 1)$: Es gilt $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

In der Haskell Standardbibliothek wurde entschieden, dass man sich explizit entscheiden muss, welches Monoid man meint:

```
> getSum     $ mconcat $ map Sum     [3,4,5]
12
> getProduct $ mconcat $ map Product [3,4,5]
60
```

Realisiert wird das mit Hilfe von `newtypes`:

```
newtype Sum a     = Sum     { getSum :: a }
newtype Product a = Product { getProduct :: a }
instance Num a => Semigroup (Sum a)     where
  Sum     x <> Sum     y = Sum     (x+y)
instance Num a => Semigroup (Product a) where
  Product x <> Product y = Product (x*y)
instance Num a => Monoid (Sum a)     where mempty = Sum 0
instance Num a => Monoid (Product a) where mempty = Product 1
```

Beispiel: Komposition als Monoid

Funktionen mit Typ `a->a` bilden ein Monoid unter Komposition:

```
instance Semigroup (a->a) where
  f <> g = f . g

instance Monoid (a->a) where
  mempty  = id
```

Geht so nicht:
Instanzdeklaration dürfen in Standard-Haskell keine Gleichheit zwischen Typvariablen erzwingen.
`instance Monoid (a->b)` wäre aber erlaubt.
Abhilfe: `newtype`

Beispiel: Komposition als Monoid

Funktionen mit Typ `a->a` bilden ein Monoid unter Komposition:

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Semigroup (Endo a) where
  Endo f <> Endo g = Endo (f . g)

instance Monoid (Endo a) where
  mempty  = Endo id
```

Beispiel:

```
> let f= mconcat $ map Endo [(4+),(10*),succ,max 1,\n -> n*n+1]
> appEndo f 1
34
```

Bemerkung: Dies ist wieder ein nicht-kommutatives Monoid, denn

`(\x->x*x) . (\y->y+1)` $\neq$ `(\y->y+1) . (\x->x*x)`

Beispiel: Komposition als Monoid

Funktionen mit Typ `a->a` bilden ein Monoid unter Komposition:

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Semigroup (Endo a) where
  Endo f <> Endo g = Endo (f . g)

instance Monoid (Endo a) where
  mempty  = Endo id
```

„Endo“ ist griechisch für „innerhalb“; wir haben es hier ja mit 1-stelligen *inneren* Abbildung zu tun.

Beispiel:

```
> let f= mconcat $ map Endo [(4+),(10*),succ,max 1,\n -> n*n+1]
> appEndo f 1
34
```

Bemerkung: Dies ist wieder ein nicht-kommutatives Monoid, denn

`(\x->x*x) . (\y->y+1)` $\neq$ `(\y->y+1) . (\x->x*x)`

Zusammenfassung Halbgruppen und Monoide

- Halbgruppe: hat *assoziative* binäre Operation `(<>)::a->a->a`
- Monoid: hat *neutrales* Element `mempty` bezüglich `(<>)`
- Instanzen von `Monoid` sollten zu Instanzen von `Semigroup` passen (Pflicht ab GHC 8.4.x) und die Gesetze beachten.
- Wer GHC älter als 8.4.*x* verwendet, muss anstatt `(<>)` immer `` `mappend` `` schreiben; oder `Data.Semigroup` importieren.
- Viele Instanzen in Standardbibliothek vordefiniert
- Erlaubt sehr starke verallgemeinerte Programmierung
  $\Rightarrow$ erhöht Wiederverwendbarkeit, erleichtert Wartung

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:

$$\kappa ::= * \mid \kappa \to \kappa$$

`(*)` 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.

`* -> (* -> *)` = `* -> * -> *` ≠ `(* -> *) -> *`

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 (->)
(->) :: * -> * -> *
```

Constraints

```
kind Eq
Eq :: * -> Constraint

> :kind Ord
Ord :: * -> Constraint

> :kind Show
Show :: * -> Constraint
```

Befragt man GHC nach dem Kind einer Typklasse, so erhält man als Antwort den Kind der Parameter und `-> Constraint`.

Die oben gezeigten Beispiele lesen sich also wie folgt: „wenn man in `Eq` noch einen konkreten Typen hineinsteckt, so erhält man eine Typklasseneinschränkung“.
Anstatt `foo :: Eq => a -> a` müssen wir also
`foo :: Eq a => a -> a` schreiben!

Typklasse Functor

Modul `Data.Functor` definiert:

```
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)`

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

Parameter hat also Kind `f :: * -> *` und ist damit kein konkreter Typ wie z.B. `Tree Int`, sondern ein Typkonstruktor wie z.B. `Tree`. `Functor` ist die Typkklasse 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

Wegen `[] :: * -> *` können wir Listen zu Funktoren machen:

```
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

Wegen `[] :: * -> *` können wir Listen zu Funktoren machen:

`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 Kompiler.
- 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 ?

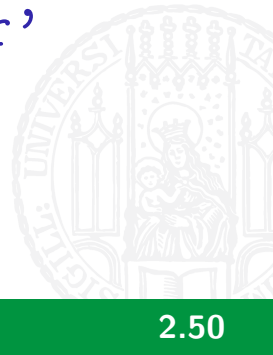
`Either` können wir aber so nicht zu einer Instanz von `Functor` machen:

```
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 Tykonstruktor mit zwei Argumenten.



Funktor-Instanz für Either

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

Auch wenn wir `Either` nicht zur Funktoreninstanz 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
```

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)
```

Funktoren

```
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 Funktoren

- Funktoren sind ein Programmierschema für die punktweise Anwendung einer Funktion auf eine Datenstruktur
- *Identität:* Funktoren 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.

$\Rightarrow$ Änderungen durch Funktoren 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).

Ausblick

Funktoren behandeln wir weiter in Abschnitt ??:

Klasse	Beschreibung
`Data.Functor`	Funktionen auf Inhalte anwendbar
`Control.Applicative`	Applikative Funktoren
`Control.Monad`	Monaden
`Data.Monoid`	eine assoziative binäre Operation
`Data.Foldable`	Faltbare Container, z.B. aufsummieren
`Data.Traversable`	In fester Reihenfolge begehbare Container

Hinweis: Viele Funktionen der Standardbibliothek, welche früher nur auf Listen definiert waren, sind inzwischen mit den obigen Typklassen verallgemeinert worden. Dadurch entstehen furchteinflößend kompliziert aussehende Fehlermeldungen.

Aufgaben von Modulsystemen

Größere Softwareprojekte benötigt Struktur $\Longrightarrow$ 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
  ...
```

- Modulenamen 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 `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.null` in `Data.Set` auch als `Set.null`
- 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

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 einfügt 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 kan 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 `{-# LANGAUGE TypeFamilies #-}` in die Klasse aufnehmen