Haskell's Typklassensystem

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 22. Juli 2020 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

Ziele

- Klassen, Methoden, Instanzen, Vererbung
- Verwendung eigener Typklassen
- Standardklassen und Methoden in Haskell:
  `Eq`, `Ord`, `Read`, `Show`, `Num`
- Kinds und Konstruktorklassen
- Neuere Klassen: Monoid, Semigroup, Foldable
- Auflösung der Überladung:
  Übersetzung in typklassenfreies Haskell

Klassen & Instanzen

Motivation, Klassendefinition, Instanzen, Veerbung

Polymorphismus (1)

Parametrischer Polymorphismus:

- Funktion f ist für eine Menge von verschiedenen Typen definiert
- Verhalten ist für alle Typen gleich
- Implementierung ist unabhängig von den konkreten Typen
- Beispiele:
  - `(++) :: [a] -> [a] -> [a]`
    kann für beliebige Listen verwendet werden
  - `(\x -> x) :: a -> a`
    kann auf beliebiges Argument angewendet werden
  - `map :: (a -> b) -> [a] -> [b]`
    kann für passende Funktion und Listen verwendet werden

Polymorphismus (2)

Ad hoc Polymorphismus:

- Eine Funktion (bzw. Funktionsname) f wird mehrfach für verschiedene Datentypen definiert.
- I.a. ist die Stelligkeit unabhängig vom Typ.
- Implementierungen von f für verschiedene Typen sind unterschiedlich.
- Ad hoc-Polymorphismus nennt man auch Überladung.
- Beispiele
  - `+` für `Integer`- und für `Double`-Werte.
  - `==` für `Bool` und `Integer`
  - `map` für Listen und Bäume

Haskells Typklassen implementieren Ad hoc Polymorphismus

Polymorphismus (3)

Manchmal kann man parametrischen Polymorphismus verwenden,
indem man zusätzliche Parameter übergibt:

Beispiel: Sortieren von Listen

- `sort :: [a] -> [a]`
  geht nicht, da Sortierung abhängig vom Elementtyp
- `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`
  geht, da Vergleichsfunktion als Argument mitgeliefert wird.
- `sort :: Ord a => [a] -> [a]`
  geht, da der Constraint `Ord a` besagt, dass man nur solche Typen `a` verwenden kann, für die man die Vergleichsfunktion kennt.

Bei der Auflösung der Überladung (Haskell mit Typklassen → Haskell ohne Typklassen) wird (mehr oder weniger) aus `sort` das `sortBy` gemacht (sehen wir später)

Typklassen und Instanzen

Typklasse:

- Namen
- Klassenfunktionen

Instanzen:

- Typen, die zur Typklasse gehören
- mit der jeweils spezifischen Implementierung der Klassenfunktionen

Beispiel

- Typklasse: GenericTree
- Klassenfunktionen, z.B. gmap, fold
- (Typ-)Instanzen: Listen, BBaum,...
- Klassenfunktion-Instanzen: map, bmap, foldr, foldb

Typklassen

In der Klassendefinition wird festgelegt:

- Typ der Klassenfunktionen
- Optional: Default-Implementierungen der Klassenfunktionen

Typklassen

In der Klassendefinition wird festgelegt:

- Typ der Klassenfunktionen
- Optional: Default-Implementierungen der Klassenfunktionen

Pseudo-Syntax für den Kopf:

```
class [OBERKLASSE =>] Klassenname a where
  ... Typdeklarationen und Default-Implementierungen ...
```

- definiert die Klasse `Klassenname`
- `a` ist der Parameter für den Typ. Es ist nur eine solche Variable erlaubt (Es gibt Erweiterungen...)
- `OBERKLASSE` ist eine Klassenbedingung (optional)
- Einrückung beachten!

Die Klasse Eq

Definition der Typklasse Eq:

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

        x /= y  = not (x == y)
        x == y  = not (x /= y)
```

- Keine Klassenbedingung
- Klassenmethoden sind == und /=
- Es gibt Default-Implementierungen für beide Klassenmethoden
- Instanzen müssen mindestens == oder /= definieren

Typklasseninstanz (1)

- Instanzen definieren die Klassenmethoden für einen konkreten Typ
- Instanzen können Default-Implementierungen überschreiben

Syntax für Instanzen:

```
instance [KLASSENBEDINGUNGEN => ] KLASSENINSTANZ where
  ...Implementierung der Klassenmethoden ...
```

- `KLASSENINSTANZ` besteht aus Klasse und der Instanz, z.B. `Eq [a]` oder `Eq Int`
- `KLASSENBEDINGUNGEN` optional

Typklasseninstanz (2)

Beispiele:

```
instance Eq Int where
 (==) = primEQInt
```

Typklasseninstanz (2)

Beispiele:

```
instance Eq Int where
 (==) = primEQInt

instance Eq Bool where
   x == y    =  (x && y) || (not x && not y)
   x /= y    =  not (x == y)
```

Typklasseninstanz (2)

Beispiele:

```
instance Eq Int where
 (==) = primEQInt

instance Eq Bool where
   x == y    =  (x && y) || (not x && not y)
   x /= y    =  not (x == y)

instance Eq Wochentag where
 Montag     == Montag     = True
 Dienstag   == Dienstag   = True
 Mittwoch   == Mittwoch   = True
 Donnerstag == Donnerstag = True
 Freitag    == Freitag    = True
 Samstag    == Samstag    = True
 Sonntag    == Sonntag    = True
 _          == _          = False
```

Typklasseninstanz (3)

Beispiel mit Klassenbedingung:

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

- Für die Abfrage `x == y` muss der Gleichheitstest auf Listenelementen vorhanden sein
- `Eq a => Eq [a]` drückt diese Bedingung aus.
- „Typ `[a]` ist nur dann eine Instanz von `Eq`,
  wenn Typ `a` bereits Instanz von `Eq` ist“

Fehlende Definitionen

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

        x /= y  = not (x == y)
        x == y  = not (x /= y)
```

Beispiel:

```
data RGB = Rot | Gruen | Blau
 deriving(Show)

instance Eq RGB where
```

- Instanz syntaktisch korrekt
- Keine Fehlermeldung oder Warnung
- `Rot == Gruen` terminiert nicht!

Fehlende Definitionen (2)

Eigene „Eq"-Klasse

```
class MyEq a where
  (===), (=/=) :: a -> a -> Bool
  (===)  a b = not (a =/= b)
```

Nur (===) hat eine Default-Implementierung

```
instance MyEq RGB where
```

- Instanz syntaktisch korrekt
- Warnung vom Compiler (kein Fehler):

  ```
  Warning: No explicit method nor default method for `=/='
  In the instance declaration for `MyEq RGB'
  ```

- `Rot === Gruen` gibt Laufzeitfehler:

  ```
  *Main> Rot === Gruen
  *** Exception: TypklassenBeispiele.hs:37:9-16:
  No instance nor default method for class operation Main.=/=
  ```

Instanzen: data, type, newtype

- Mit `data` definierte Typen können zu Instanzen gemacht werden.
- Mit `type` definierte Typsynonyme können **nicht** zu Instanzen gemacht werden!
  Grund: Compiler kann die Synonyme nicht unterscheiden, welche Instanz soll er nehmen?
- Mit `newtype` definierte Typsynonyme **können** zu Instanzen gemacht werden.

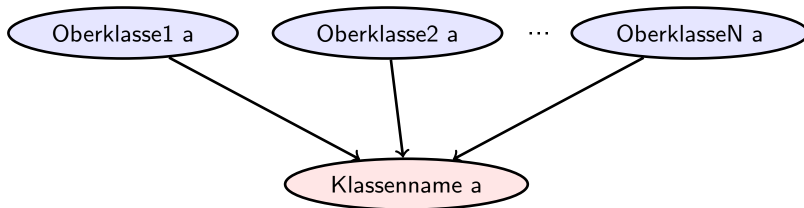
Beispiele: Zwei Eq-Instanzen für Bool

```
newtype Wahrheitswert = W Bool
instance Eq Wahrheitswert where
   (W True) == (W True) = True
   (W False) == (W False) = True
   _ == _ = False

newtype VerkehrteWelt = VW Bool
instance Eq VerkehrteWelt where
   (VW True) == (VW False) = True
   (VW False) == (VW True) = True
   _ == _ = False
```

Typklassen: Vererbung

```
class (Oberklasse1 a, ..., OberklasseN a) => Klassenname a where
  ...
```



- `Klassenname` ist Unterklasse von `Oberklasse1`,...,`OberklasseN`
- In der Klassendefinition können die überladenen Operatoren der Oberklassen verwendet werden (da sie geerbt sind)
- Beachte: `=>` ist nicht als logische Implikation zu interpretieren
- Da mehrere Oberklassen erlaubt sind, ist Mehrfachvererbung möglich

Klasse mit Vererbung: Ord

Ord ist Unterklasse von Eq:

```
class  (Eq a) => Ord a  where
  compare        :: a -> a -> Ordering
  (<),(<=),(>=),(>) :: a -> a -> Bool
  max, min          :: a -> a -> a
  compare x y | x == y    = EQ
              | x <= y    = LT
              | otherwise = GT
  x <= y  = compare x y /= GT
  x <  y  = compare x y == LT
  x >= y  = compare x y /= LT
  x >  y  = compare x y == GT
  max x y | x <= y    =  y
          | otherwise =  x
  min x y | x <= y    =  x
          | otherwise =  y
```

Ord: lineare Ordnungen!

Ordering ist definiert als:

```
data Ordering = LT | EQ | GT
```

Instanzen müssen entweder
  <= oder compare
definieren.

Beispiel: Instanz für Wochentag

```
instance Ord Wochentag where
 a <= b =
   (a,b) `elem`  [(a,b) | i <- [0..6],
                          let a = ys!!i,
                          b <- drop i ys]
  where ys = [Montag, Dienstag, Mittwoch,
              Donnerstag, Freitag, Samstag, Sonntag]
```

Wenn Wochentag Instanz von `Enum`,
dann kann man die Definition ersetzen:

```
a <= b =
  (a,b)`elem`[(a,b) | a <- [Montag..Sonntag],b <- [a..Sonntag]]
```

Vererbung im Typsystem

Beispiel:

```
f x y = (x == y) && (x <= y)
```

Da == und <= verwendet werden würde man erwarten:

Typ von `f`?

Vererbung im Typsystem

Beispiel:

```
f x y = (x == y) && (x <= y)
```

Da == und <= verwendet werden würde man erwarten:

```
f :: (Eq a, Ord a) => a -> a -> Bool
```

Vererbung im Typsystem

Beispiel:

```
f x y = (x == y) && (x <= y)
```

Da == und <= verwendet werden würde man erwarten:

```
f :: (Eq a, Ord a) => a -> a -> Bool
```

Da `Ord` jedoch Unterklasse von `Eq`:

```
f :: Ord a => a -> a -> Bool
```

Klassenbeschränkungen bei Instanzen

```
instance (Klasse1 a1, ..., KlasseN aN) => Klasse Typ where
 ...
```

- Es sind mehrere Typvariablen erlaubt.
- Die Typvariablen `a1`,...,`aN` müssen alle im Typ `Typ` vorkommen.

Klassenbeschränkungen bei Instanzen

```
instance (Klasse1 a1, ..., KlasseN aN) => Klasse Typ where
 ...
```

- Keine Vererbung!
- Bedeutet: Es gibt nur dann eine Instanz, wenn es Instanzen für die Typvariablen a1,...,aN der entsprechenden Klassen gibt.

Klassenbeschränkungen bei Instanzen

```
instance (Klasse1 a1, ..., KlasseN aN) => Klasse Typ where
 ...
```

- Keine Vererbung!
- Bedeutet: Es gibt nur dann eine Instanz, wenn es Instanzen für die Typvariablen a1,...,aN der entsprechenden Klassen gibt.

Beispiel:

```
instance Eq a => Eq (BBaum a) where
 Blatt a == Blatt b              = a == b
 Knoten l1 r1 == Knoten l2 r2 = l1 == l2 && r1 == r2
 _   ==   _                      = False
```

Nur wenn man Blattmarkierungen vergleichen kann, dann auch Bäume

Klassenbeschränkungen bei Instanzen (2)

```
*Main List> Blatt 1 == Blatt 2
False
Main List> Blatt 1 == Blatt 1
True
*Main List> Blatt (\x ->x) == Blatt (\y -> y)
<interactive>:1:0:
    No instance for (Eq (t -> t))
      arising from a use of `==' at <interactive>:1:0-32
    Possible fix: add an instance declaration for (Eq (t -> t))
    In the expression: Blatt (\ x -> x) == Blatt (\ y -> y)
    In the definition of `it':
        it = Blatt (\ x -> x) == Blatt (\ y -> y)
```

Klassenbeschränkungen bei Instanzen (3)

Beispiel mit mehreren Klassenbeschränkungen:

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

Either-Instanz für Eq:

```
instance (Eq a, Eq b) => Eq (Either a b) where
 Left x  == Left y  = x == y -- benutzt Eq-Instanz f\"ur a
 Right x == Right y = x == y -- benutzt Eq-Instanz f\"ur b
 _       == _       = False
```

Die Klassen Read und Show

Die Klassen Read und Show dienen zum Einlesen (Parsen) und Anzeigen von Datentypen.

```
show :: Show a => a -> String
read :: Read a => String -> a
```

Allerdings ist read keine Klassenfunktion!
Komplizierter:

```
type  ReadS a = String -> [(a,String)]
type  ShowS   = String -> String
```

- reads ist wie ein Parser mit Erfolgslisten
- shows kann noch einen String als Argument nehmen
  (oft schneller als geschachteltes ++)

```
reads :: Read a => ReadS a
shows :: Show a => a -> ShowS
```

Die Klassen Read und Show (2)

```
class  Read a  where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]
    -- ... default decl for readList given in Prelude

class  Show a  where
    showsPrec :: Int -> a -> ShowS
    show      :: a -> String
    showList  :: [a] -> ShowS

    showsPrec _ x s   = show x ++ s
    show x            = showsPrec 0 x ""
    -- ... default decl for showList given in Prelude
```

Show für BBaum

```
showBBaum :: (Show t) => BBaum t -> String
showBBaum (Blatt a)     = show a
showBBaum (Knoten l r) =
   "<" ++ showBBaum l ++ "|" ++ showBBaum r ++ ">"
```

Schlecht, da quadratische Laufzeit, insbesondere bei tiefen baumartigen Strukturen

Besser:

```
showBBaum' :: (Show t) => BBaum t -> String
showBBaum' b = showsBBaum b []

showsBBaum :: (Show t) => BBaum t -> ShowS
showsBBaum (Blatt a)     = shows a
showsBBaum (Knoten l r) =
  showChar '<' . showsBBaum l . showChar '|'  . showsBBaum r . showChar '>'
```

Show für BBaum (2)

Tests:

```
*Main> last $ showBBaum t
'>'
(73.38 secs, 23937939420 bytes)
*Main> last $ show t
'>'
(0.16 secs, 10514996 bytes)
```

Hierbei ist `t` ein Baum mit ca. 15000 Knoten.

Show-Instanz für `BBaum a`:

```
instance Show a => Show (BBaum a) where
 showsPrec _ = showsBBaum
```

Rand eines Baumes: analog

```
bRand (Blatt a) = [a]
bRand (Knoten links rechts) = (bRand links) ++ (bRand rechts)


--  Variante 1
bRand' tr =  bRand_it [] tr
bRand_it acc (Blatt a)  =  (a:acc)
bRand_it acc (Knoten links rechts)  =
         bRand_it (bRand_it acc rechts) links

--  Variante 2
bRand'' tr = reverse (bRand_it' tr [])
bRand_it' (Blatt a)  =  \x-> (a:x)
bRand_it'  (Knoten links rechts)  =
       (bRand_it' rechts)  . (bRand_it' links)
```

Rand eines Baumes: analog

```
*Main> let t = genBaum [1..10000]
*Main> :s +s
*Main> length (bRand t)
...
*Main> let t' = genBaum' [1..100000]
*Main> length (bRandschnell t)
```

Read für BBaum

Elegant mit List Comprehensions:

```
instance Read a => Read (BBaum a) where
 readsPrec _ = readsBBaum

readsBBaum :: (Read a) => ReadS (BBaum a)
readsBBaum ('<':xs) =
   [(Knoten l r, rest) | (l, '|':ys)   <- readsBBaum xs,
                         (r, '>':rest) <- readsBBaum ys]
readsBBaum s      =
    [(Blatt x, rest)    | (x,rest)      <- reads s]
```

Auflösung erfordert manchmal Typ

```
Prelude> read "10"
<interactive>:1:0:
 Ambiguous type variable `a' in the constraint
 `Read a' arising from a use of `read' at <interactive>:1:0-8
 Probable fix: add a type signature that
               fixes these type variable(s)
Prelude> (read "10")::Int
10
```

Ähnliches Problem bei überladenen Konstanten, z.B. `0`

Im GHC Defaulting: Für Zahlen ist dies der Typ `Integer`.

Klassen mit Mehrfachvererbung: Num

`Num` überlädt Operatoren für Zahlen:

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
```

Mit `fromInteger` werden Zahlenkonstanten überladen, z.B.

```
length [] = 0
length (x:xs) = 1+(length xs)
```

Der Compiler kann den Typ `length :: (Num a) => [b] -> a` herleiten, da `0` eigentlich für `fromInteger (0::Integer)` steht.
In Haskell: `genericLength` aus Modul `Data.List`

Die Klasse Enum

`Enum` ist für Typen geeignet deren Werte aufzählbar sind:

```
class  Enum a  where
    succ, pred     :: a -> a
    toEnum         :: Int -> a
    fromEnum       :: a -> Int
    enumFrom       :: a -> [a]             -- [n..]
    enumFromThen   :: a -> a -> [a]        -- [n,n'..]
    enumFromTo     :: a -> a -> [a]        -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]   -- [n,n'..m]
```

Die Klasse Enum (2)

Enum-Instanz für Wochentag

```
instance Enum Wochentag where
 toEnum i = tage!!(i `mod` 7)
 fromEnum t = case elemIndex t tage of
               Just i -> i

tage = [Montag,Dienstag,Mittwoch,Donnerstag,
        Freitag,Samstag,Sonntag]
```

Ein Aufruf:

```
*Main> enumFromTo  Montag Sonntag
[Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag,Sonntag]
*Main> [(Montag)..(Samstag)]
[Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag]
```

Typisierung unter Typklassen

Syntax von polymorphen Typen ohne Typklassen

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

Erweiterte Typen $\mathbf{T_e}$ mit Typklassen:

$$\mathbf{T_e} ::= \mathbf{T} \mid \mathbf{Kon}\ \texttt{=>}\ \mathbf{T}$$

$\mathbf{Kon}$ ist ein Typklassenkontext:

$$\mathbf{Kon} ::= \text{Klassenname}\ TV \mid (\text{Klassenname}_1\ TV, \ldots, \text{Klassenname}_n\ TV)$$

Zusätzlich: Für *Kontext* `=>` *Typ* muss gelten:
Alle Typvariablen von *Kontext* kommen auch in *Typ* vor.

Z.B. `elem ::(Eq a) => a -> [a] -> Bool`.

Kinds & Konstruktorklassen

Was sind Kinds? / Klassen für Typkonstruktoren. / Die Klasse Functor

Kinds

- Typen sind quasi „Typen über Termen“
- **Kinds** (engl. für Sorte) sind „Typen über Typen“
- Einfache Syntax für Kinds: $\kappa := * \mid \kappa \to \kappa$ Basiskind $*$ und Funktionskind.
- Jeder Haskell-Typ hat Kind $*$.
- Im GHCi anzeigen:

```
> :kind Bool
Bool :: *
> :kind (Bool -> [Int] -> Char)
(Bool -> [Int] -> Char) :: *
```

- Typkonstruktoren TC mit Stelligkeit n haben den Kind $\underbrace{\texttt{* -> } \cdots \texttt{ -> *}}_{n \text{ Sterne}} \texttt{ -> *}$

Der Typkonstruktor TC erwartet n Argumente, die selbst Typen sind. Wendet man TC auf solche Argumente an, so erhält man einen Typ.

- Beispiel: `Maybe` ist einstellig und hat Kind `* -> *`

Kinds: Beispiele

```
Prelude> :kind Maybe
Maybe :: * -> *
Prelude> :kind Either
Either :: * -> * -> *
Prelude> :kind Maybe Bool
Maybe Bool :: *
Prelude> :kind Either Bool
Either Bool :: * -> *
Prelude> :kind Either Bool Char
Either Bool Char :: *
```

Kinds: Listen, Funktionen

Listen:

- `[]` ist der Typkonstruktor.
- Statt `[] a` schreibt man meistens `[a]`

```
Prelude> :kind []
[] :: * -> *
Prelude> :kind [Bool]
[Bool] :: *
Prelude> :kind [] Bool
[] Bool :: *
```

Funktionspfeil:

- `->` verhält sich wie ein Typkonstruktor
- Verlangt zwei Typen als Argumente

```
Prelude> :kind (->)
(->) :: * -> * -> *
```

Typklassenbeschränkungen: Constraint als Kind

Neuer Basiskind für Typklassenbeschränkungen: `Constraint`

Beispiele:

```
Prelude> :kind (Show Bool)    -- Show Bool ist ein Constraint
(Show Bool) :: Constraint
Prelude> :kind Show           -- Show erwartet einen Typ, um dann
Show :: * -> Constraint       -- ein Constraint zu werden
```

Konstruktorklassen

- Die bisherigen Klassen abstrahieren über einen Typ
  Z.B.

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

  - für die Variable a kann ein Typ eingesetzt werden.
  - a ist vom Kind $*$

Konstruktorklassen

- Die bisherigen Klassen abstrahieren über einen Typ
  Z.B.

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

  - für die Variable a kann ein Typ eingesetzt werden.
  - a ist vom Kind *
- In Haskell ist es auch möglich über Typkonstruktoren zu abstrahieren.
  D.h. Klassen, wobei die Variable a von Kind `* -> * -> ... -> *`
- Solche Klassen heißen **Konstruktorklassen**

Die Konstruktorklasse Functor

Definition der Klasse Functor:

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

f ist eine Variable für einen Typkonstruktor.

In Data.Functor wird ein Synonym definiert

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

Functor-Instanz für Listen

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

Die Konstruktorklasse Functor

Functor-Instanz für `BBaum`

```
instance Functor BBaum where
 fmap = bMap
```

Bei Instanzbildung muss der Typkonstruktor `BBaum` und nicht der Typ `(BBaum a)` angegeben werden.

Die Konstruktorklasse Functor

Falsch mit `BBaum a` statt `BBaum`:

```
instance Functor (BBaum a) where
 fmap = bMap
```

ergibt den Fehler:

```
Kind mis-match
The first argument of `Functor' should have kind `* -> *',
but `BBaum a' has kind `*'
In the instance declaration for `Functor (BBaum a)'
```

Instanzen von Functor

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

Beachte, dass (`Either` a) wie ein 1-stelliger Typkonstruktor wirkt, da `Either` 2-stellig ist.
siehe `Data.Either`

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

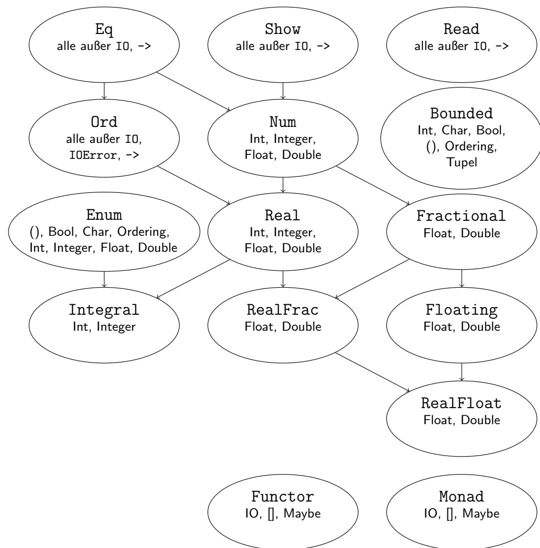
Instanzen von `Functor` sollten die folgenden beiden Gesetze erfüllen:

```
fmap id = id
fmap (f . g)  = fmap f . fmap g
```

Monoide und Halbgruppen

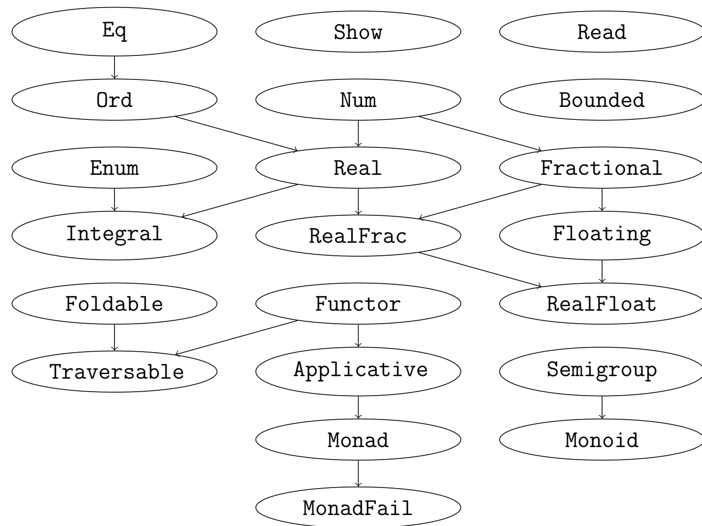
Übersicht über vordefinierte Klassen, die Klassen Monoid und Semigroup

Übersicht über die vordefinierten Typklassen (veraltet)



Diese Typklassen-Hierarchie entspricht dem Haskell 2010 Report

Aktuelle Übersicht



Diese Typklassen-Hierarchie entspricht der base-Library Version 4.13 (GHC 8.8.3)

Änderungen:

- Neu: Semigroup, Monoid, Foldable, Traversable, Applicative, MonadFail
- Klassenbeziehung Functor → Applicative → Monad → MonadFail
- Beziehung Num, Eq, Show geändert

Automatisches Ableiten von Instanzen

- Mit Schlüsselwort `deriving` nach der `data`-Deklaration
- Compiler generiert automatisch Typklasseninstanz
- Anzeigen mit `-ddump-deriv`

Beispiel:

```
Prelude> :set -ddump-deriv
Prelude> data WW = Wahr | Falsch deriving (Eq)

==================== Derived instances ====================
Derived class instances:
  instance GHC.Classes.Eq Ghci1.WW where
    (GHC.Classes.==) (Ghci1.Wahr) (Ghci1.Wahr) = GHC.Types.True
    (GHC.Classes.==) (Ghci1.Falsch) (Ghci1.Falsch) = GHC.Types.True
    (GHC.Classes.==) _ _ = GHC.Types.False
```

Semigroup

- Halbgruppe: Menge mit binärer assoziativer Verknüpfung
- In Haskell: Typklasse `Semigroup`

```
class Semigroup a where
  (<>) :: a -> a -> a              -- should be associative
  stimes  :: Integral n => n -> a -> a  -- fails for n<1
  sconcat :: NonEmpty a -> a
```

- Es genügt die Operation `<>` zu definieren.
- Es muss gelten: `x <> (y <> z) == (x <> y) <> z`
- `stimes` n a erzeugt

$$\underbrace{a \texttt{ <> } a \texttt{ <> } \ldots \texttt{ <> } a}_{n \text{ viele } a\text{s}}.$$

- `sconcat` konkateniert eine nicht-leere Liste von Werten, durch Verknüpfung mit `<>`
- `data NonEmpty a = a :| [a]`.

Semigroup: Beispiel Minimum-Operation

```
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)
```

Einige Beispielaufrufe:

```
*> getMin (Min 7 <> Min 8 <> Min 10 <> Min 3)
3
> getMin $ sconcat $ (Min 4):| (map Min [1,2,3,4,10,-1,2,3])
-1
```

Monoide

- Monoid $=$ Menge mit binärer assoziativer Verknüpfung $\otimes$ und neutralem Element 1 $(1 \otimes m = m = m \otimes 1)$
- Monoid $=$ Halbgruppe mit neutralem Element
- In Haskell: Klassen `Monoid`

```
class Semigroup a => Monoid a where
  mempty  :: a              -- neutrales Element
  mappend :: a -> a -> a   -- Verknuepfung
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Beachte:

- `Monoid` ist erst ab GHC 8.4.x eine Unterklasse von `Semigroup`.
- Vorher: unabhängig und `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.

Monoid: Listen

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

Gesetze gelten: Nachrechnen, wobei man für endliche Listen Induktion verwenden kann, für unendliche Listen andere Methoden braucht (z.B. Co-Induktion).
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]
```

Monoid: Maybe

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> my      = my
    mx      <> Nothing = mx
    Just x  <> Just y  = Just (x <> y)

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

Wiederum kann man nachrechnen, dass die Gesetze gelten. Interessanterweise genügt es, für die Monoid-Instanz zu fordern, dass der verpackte Typ eine Halbgruppe ist – neutrale Elemente werden auf diesem Typ nicht benötigt.

Monoid: Paare

```
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)
```

Monoid: Zahlen

Auch Zahlen bilden Monoide:

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

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

D.h. man kann zwei Instanzen angeben. In der Haskell-Standardbibliothek:

```
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
```

Monoid: Funktionen

Auch Funktionen bilden mit der Operation der Funktionskomposition ein Monoid.

```
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: Endomorphismus eine 1-stellige-Abbildung in sich selbst (also eine Funktion mit Haskell-Typ `a -> a`).

Foldable

Klasse Foldable

Typen, die sich falten lassen (Verallgemeinerung von Listen)

```
class Foldable t where          {-# MINIMAL foldMap | foldr #-}
    fold     :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap' :: Monoid m => (a -> m) -> t a -> m
    foldr    :: (a -> b -> b) -> b -> t a -> b
    foldr' :: (a -> b -> b) -> b -> t a -> b
    foldl    :: (b -> a -> b) -> b -> t a -> b
    foldl' :: (b -> a -> b) -> b -> t a -> b
    toList   :: t a -> [a]
    null     :: t a -> Bool
    length   :: t a -> Int
    elem     :: Eq a => a -> t a -> Bool
    sum, product :: Num a => t a -> a
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
```

Klasse Foldable (2)

Default-Implementierungen für `fold` und `foldMap`:

```
--  Combine the elements of a structure using a monoid.
fold :: Monoid m => t m -> m
fold = foldMap id

-- Map each element of the structure to a monoid, and combine the results.
foldMap :: Monoid m => (a -> m) -> t a -> m
foldMap f = foldr (mappend . f) mempty
```

Beispiele zur Verwendung:

```
*> import Data.Foldable
*> import Data.Monoid
*> fold [Product 1, Product 2,Product 3]
Product {getProduct = 6}
*> foldMap Product [2,3,4]
Product {getProduct = 24}
```

```
*> foldMap Sum [2,3,4]
Sum {getSum = 9}
*> fold [[1,2,3],[3,4,5],[5,6,7]]
[1,2,3,3,4,5,5,6,7]
*> foldMap (++ " ") ["H","A","L","L","O"]
"H A L L O "
```

Klasse Foldable (3)

Gesetze:

- Identität: `fold == foldMap id`
- Funktoren-Komposition: `foldMap f . fmap g == foldMap (f . g)`
- Gesetze für `foldMap` und `foldr`/`foldl` foldr f z t == foldMap

Auflösung der Überladung

Wie kodiert man Typklassen wieder weg?

Auflösung der Überladung

- Irgendwann muss die richtige Implementierung für eine überladene Funktion benutzt werden
- Early Binding: Auflösung zur Compilezeit
- Late Binding: Auflösung zur Laufzeit

Zur Erinnerung:

- Parametrisch polymorph: Implementierung ist unabhängig von den konkreten Typen
- Ad hoc polymorph (Typklassen): Implementierung ist abhängig von den konkreten Typen

Auflösung der Überladung in Haskell

- Es gibt keine Typinformation zur Laufzeit

  $\Longrightarrow$ Auflösung zur Compilezeit

- Auflösung benötigt Typinformation,
  daher Auflösung zusammen mit dem Typcheck.

- Nicht immer (einfach) möglich, zB. `quadrat`

- Typen-Ersatz für Late Binding:
  Datenstruktur die „Typinformation“ enthält

- Typcheck ist notwendig für korrekte Auflösung der Überladung
  Insbesondere lokale Typinformation
  Auch für Optimierungen durch early Binding.

Auflösung der Überladung

Unser Vorgehen zur Transformation in Kernsprache:

- Wir nehmen an, Typinformation, auch für Unterausdrücke, ist vorhanden
- Wir trennen jedoch Auflösung
  und Typcheck (Typberechnung), d.h.
  wir behandeln erstmal nur die Auflösung durch Übersetzung

Auflösung der Überladung (2)

Wir machen das anhand von Beispielen:
Auflösung von `Eq`

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

Auflösung der Überladung (2)

Wir machen das anhand von Beispielen:
Auflösung von `Eq`

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

- Anstelle der Typklasse tritt eine Datentypdefinition:
- Dieser Dictionary-Datentyp ist ein Produkttyp (record), der für jede Klassenmethode eine Komponente erhält.

Auflösung der Überladung (2)

Wir machen das anhand von Beispielen:
Auflösung von Eq

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

- Anstelle der Typklasse tritt eine Datentypdefinition:
- Dieser Dictionary-Datentyp ist ein Produkttyp (record), der für jede Klassenmethode eine Komponente erhält.

```
data EqDict a = EqDict {
                  eqEq  :: a -> a -> Bool, -- f"ur ==
                  eqNeq :: a -> a -> Bool  -- f"ur /=
                }
```

Auflösung der Überladung (3)

Methoden bzw. Operatoren - Implementierungen:

- Sie werden als „normale“ Funktionen implementiert
- und erhalten als zusätzliches Argument ein Dictionary (also ein Tupel der Klassenfunktion-Implementierungen)

Anstelle der überladenen Operatoren:

Implementierung von $==$

```
overloadedeq :: EqDict a -> a -> a -> Bool
overloadedeq dict a b = eqEq dict a b
```

Implementierung von $/=$

```
overloadedneq :: EqDict a -> a -> a -> Bool
overloadedneq dict a b = eqNeq dict a b
```

Auflösung der Überladung (4)

Beachte bei der Übersetzung: aus

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

wird:

```
overloadedeq :: EqDict a -> a -> a-> Bool
```

Überladene Funktionen können nun durch zusätzliche Dictionary-Parameter angepasst werden:
Beispiel:

```
elem :: (Eq a) => a -> [a] -> Bool
elem e []     = False
elem e (x:xs)
 | e == x    = True
 | otherwise = elem e xs
```

$\Longrightarrow$

```
elemEq :: EqDict a -> a -> [a] -> Bool
elemEq dict e []     = False
elemEq dict e (x:xs)
 | (eqEq dict) e  x    = True
 | otherwise           = elemEq dict e xs
```

Auflösung der Überladung (5)

Bei konkreten Typen müssen jedoch die konkreten Dictionaries eingesetzt werden

Z.B. aus

```
... True == False ...
```

wird

```
... overloadedeq eqDictBool True False
```

Auflösung der Überladung

Beispiel:

Aus

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

wird

```
eqDictBool = EqDict {eqEq = eqBool, eqNeq =  default_eqNeq}
```

und

```
eqBool True  True  = True
eqBool False False = True
eqBool _     _     = False
```

Auflösung der Überladung: Grundtypen

Übersetzung: `True == True` Typ von $==$ ist hier `Bool -> Bool -> Bool`

wird zu

```
overloadedeq eqDictBool True True
```

Das kann man durch Beta-Reduktion (zur Compilezeit) weiter vereinfachen:

```
eqEq eqDictBool True True
```

und zu

```
eqBool True True
```

Ergebnis: early binding bei bekanntem Grundtyp

Auflösung der Überladung: Eq auf Listen

```
instance Eq a => Eq [a] where
 ....
 x:xs == y:ys = x == y && xs == ys
```

wird als Funktion übersetzt: dict für Elemente $\mapsto$ dict für Liste

```
 eqDictList::  EqDict a -> EqDict [a]
 eqDictList dict  =  EqDict {eqEq = eqList dict,
           eqNeq = default_eqNeq (eqDictList dict)}   where

  eqList .... =
  eqList dict (x:xs)  (y:ys)
      = overloadedeq dict x y && overloadedeq (eqDictList dict) xs ys
```

Auflösung der Überladung: Defaults

Default-Implementierungen:

falls Klassenmethoden nicht bei `instance` angegeben:
Hier für die `Eq`-Typklasse

```
-- Default-Implementierung f"ur ==:
default_eqEq eqDict  x y = not (eqNeq eqDict x y)

-- Default-Implementierung f"ur /=:
default_eqNeq eqDict x y = not (eqEq eqDict x y)
```

Auflösung der Überladung (6)

Weitere Beispiele für konkrete Dictionaries

Auflösung der Überladung (6)

Weitere Beispiele für konkrete Dictionaries

Aus der Instanzdefinition:

```
instance Eq Wochentag where
 Montag     == Montag     = True
 Dienstag   == Dienstag   = True
 Mittwoch   == Mittwoch   = True
 Donnerstag == Donnerstag = True
 Freitag    == Freitag    = True
 Samstag    == Samstag    = True
 Sonntag    == Sonntag    = True
 _          == _          = False
```

⇒

```
eqDictWochentag :: EqDict Wochentag
eqDictWochentag =
 EqDict {
   eqEq = eqW,
   eqNeq = default_eqNeq eqDictWochentag
 }
  where  eqW Montag     Montag     = True
         eqW Dienstag   Dienstag   = True
         eqW Mittwoch   Mittwoch   = True
         eqW Donnerstag Donnerstag = True
         eqW Freitag    Freitag    = True
         eqW Samstag    Samstag    = True
         eqW Sonntag    Sonntag    = True
         eqW _          _          = False
```

Beachte die Verwendung der Default-Implementierung für eqNeq

Auflösung der Überladung (7)

Eq-Dictionary für `Ordering`

```
instance Eq Ordering where
    LT == LT = True
    EQ == EQ = True
    GT == GT = True
    _  == _  = False
```

$\Rightarrow$

```
eqDictOrdering :: EqDict Ordering
eqDictOrdering =
 EqDict {
  eqEq = eqOrdering,
  eqNeq = default_eqNeq eqDictOrdering
 }
  where
   eqOrdering LT LT = True
   eqOrdering EQ EQ = True
   eqOrdering GT GT = True
   eqOrdering _  _  = False
```

Auflösung der Überladung (8)

Instanzen mit Klassenbeschränkungen:

```
instance Eq a => Eq (BBaum a) where
 Blatt a == Blatt b             = a == b
 Knoten l1 r1 == Knoten l2 r2 = l1 == l2 && r1 == r2
 _  ==  _                       = False
```

Auflösung: das Dictionary für `BBaum a` ist eine Funktion, die das Dictionary für a erhält:

```
eqDictBBaum :: EqDict a -> EqDict (BBaum a)
eqDictBBaum dict = EqDict {
                    eqEq  = eqBBaum dict,
                    eqNeq = default_eqNeq (eqDictBBaum dict)
                 }
 where
  eqBBaum dict (Blatt a) (Blatt b) =
    overloadedeq dict a b -- hier Eq-Dictionary f"ur dict
  eqBBaum dict (Knoten l1 r1) (Knoten l2 r2) =
    eqBBaum dict l1 l2 && eqBBaum dict r1 r2
  eqBBaum dict x y = False
```

Auflösung der Überladung (9)

Auflösung bei **Unterklassen**: Der Datentyp enthält die Dictionaries der Oberklassen

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

 compare x y | x == y    = EQ
             | x <= y    = LT
             | otherwise = GT

 x <= y  = compare x y /= GT
 x <  y  = compare x y == LT
 x >= y  = compare x y /= LT
 x >  y  = compare x y == GT

 max x y | x <= y    =  y
         | otherwise =  x
 min x y | x <= y    =  x
         | otherwise =  y
```

$\Longrightarrow$

```
data OrdDict a =
 OrdDict {
  eqDict :: EqDict a,   -- Dictionary
                        -- der Oberklasse
  ordCompare :: a -> a -> Ordering,
  ordL  :: a -> a -> Bool,
  ordLT :: a -> a -> Bool,
  ordGT :: a -> a -> Bool,
  ordG  :: a -> a -> Bool,
  ordMax :: a -> a -> a,
  ordMin :: a -> a -> a
  }
```

Auflösung der Überladung (10)

Übersetzung der Default-Implementierungen:

```
compare x y | x == y    = EQ
            | x <= y    = LT
            | otherwise = GT
```

⇒

```
default_ordCompare dictOrd x y
  | (eqEq (eqDict dictOrd)) x y = EQ
  | (ordLT dictOrd) x y         = LT
  | otherwise                   = GT
```

```
x <= y  = compare x y /= GT
```

⇒

```
default_ordLT dictOrd x y =
  let compare = (ordCompare dictOrd)
      nequal  = eqNeq (eqDictOrdering)
  in (compare x y) `nequal` GT
```

```
x <  y  = compare x y == LT
```

⇒

```
default_ordL dictOrd x y =
  let compare = (ordCompare dictOrd)
      equal    = eqEq eqDictOrdering
  in (compare x y) `equal` LT
```

usw.

Auflösung der Überladung (11)

Ord-Dictionary für `Wochentag`:

```
instance Ord Wochentag where
 a <= b = (a,b) `elem`  [(a,b) | i <- [0..6], let a = ys!!i, b <- drop i ys]
  where ys = [Montag,Dienstag,Mittwoch,Donnerstag,Freitag,Samstag,Sonntag]
```

⇒

```
ordDictWochentag = OrdDict {
  eqDict = eqDictWochentag,
  ordCompare = default_ordCompare ordDictWochentag,
  ordL   = default_ordL ordDictWochentag,
  ordLT = wt_lt,
  ordGT = default_ordGT ordDictWochentag,
  ordG   = default_ordG ordDictWochentag,
  ordMax = default_ordMax ordDictWochentag,
  ordMin = default_ordMin ordDictWochentag
  }
  where
   wt_lt  a b = (a,b) `elem` [(a,b) | i <- [0..6], let a = ys!!i, b <- drop i ys]
   ys = [Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag]
```

Auflösung der Überladung (11)

Konstruktorklassen:

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

Typvariablen `a` und `b` müssen dem Dictionary hinzugefügt werden:

```
data FunctorDict a b f = FunctorDict {
  functorFmap :: (a -> b) -> f a -> f b}
```

Die überladene `fmap`-Funktion nach der Übersetzung:
(kein gültiger polymorpher Typ, aber Haskell-Typ)

```
overloaded_fmap :: (FunctorDict a b f) -> (a -> b) -> f a -> f b
overloaded_fmap dict = functorFmap dict
```

Instanz für BBaum:

```
functorDictBBaum = FunctorDict {functorFmap = bMap}
```

Erweiterung von Typklassen

Multiparameter Klassen

- Haskell erlaubt nur eine „Kind“-Variable in der Klassendeklaration
- Multiparameter-Klassen erlauben auch mehrere „Kind“-Variablen, z.B.

```
class Indexed c a i where
sub :: c -> i -> a
idx :: c -> a -> Maybe i
```

- Überlappende bzw. flexible Instanzen sind in Haskell verboten:

```
instance Eq (Bool,Bool) where
(a,b) == (c,d) = ...
```

- Funktionale Abhängigkeiten

```
class MyClass a b | a -> b where
```

  bedeutet in etwa: Der Typ `b` wird durch den Typ `a` bestimmt.

Problem fast aller Erweiterungen: Typsystem wird unentscheidbar!