



Einführung in die Funktionale Programmierung: Haskell (2)

Prof. Dr. Manfred Schmidt-Schauß

WS 2025/26

Stand der Folien: 16. Dezember 2025

1 Typklassen

- Klassen und Instanzen
- Konstruktorklassen
- Auflösung der Überladung
- Erweiterung von Typklassen

2 Haskells hierarchisches Modulsystem (Nicht Prüfungs-Relevant)

Haskells Typklassensystem

Ziele des Kapitels

- Klassen, Methoden, Ober-/Unterklassen, Vererbung in Haskell.
- Definition und Verwendung selbstdefinierter Typklassen in Haskell.
- Standardklassen und Methoden in Haskell:
Eq, Ord, Read, Show, Num
die weitverbreitet in Haskell-Modulen verwendet werden
- Auflösung der Überladung (in Haskell):
Übersetzung in typklassenfreies Haskell

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:
 - $(++) :: [a] \rightarrow [a] \rightarrow [a]$
kann für beliebige Listen verwendet werden
 - $(\lambda x \rightarrow x) :: a \rightarrow a$
kann auf beliebiges Argument angewendet werden
 - $map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$
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 fix d.h. 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 und ...
 - $==$ für Bool und Integer und...
 - map für Listen und Bäume

Haskells **Typklassen** implementieren **Ad hoc Polymorphismus**

Typklassen und Instanzen in Haskell

- Typklasse: besteht aus
 - Name
 - Klassenfunktionen
- **Instanzen:** sind die Typen die zur Typklasse gehören:
zusammen mit der jeweils spezifischen Implementierung der Klassenfunktionen

Beispiel

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

Typklassen

In der **Klassendefinition** wird festgelegt:

- **Name** und **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
- **OBERKLASSE** ist eine **Klassenbedingung** (optional)

Die Klasse Eq

Definition der Typklasse Eq (Teil von Haskell)

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

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

- Keine Klassenbedingung (d.h. es gibt keine Oberklasse)
- Klassenmethoden sind == und /=
- Es gibt Default-Implementierungen für beide Klassenmethoden: /= und == ; scheinbar zyklisch.
- Sinnvolle Instanzen sollten mindestens == oder /= definieren

Beispiel für Oberklassen - Syntax

Auszug aus der Definition der Klasse Ord: (später)

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

  ...
```

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
- Erlaubt sind nur:** Typkonstruktor, oder Typkonstruktor angewendet auf verschiedene Typ-Variablen.
- KLASSENBEDINGUNGEN optional

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)
```

Vorsicht: Vermeidung von zyklischen Aufrufen!

```
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: (und Rekursion auf der Klassenebene)

Definiert wird was [a] bedeuten soll, D.h. Liste von a.

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

- Für die Abfrage $x == y$ muss der Gleichheitstest zum Typ der Listenelemente schon definiert sein
- $\text{Eq } a \Rightarrow \text{Eq } [\text{a}]$ drückt diese Bedingung gerade aus.
- „Typ $[\text{a}]$ 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 (in meiner Version)
- Rot == Gruen terminiert nicht!
(wg. Schleife in der Defaultimplementierung durch fehlende Definitionen)

Fehlende Definitionen (2)

Eigene „Eq“-Klasse (Nur als Beispiel...)

```
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):

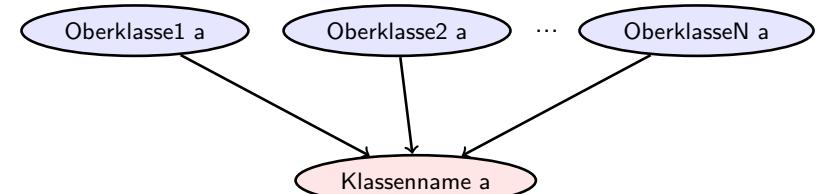
Warning: No explicit method nor default method for ‘=/’
In the instance declaration for ‘MyEq RGB’

- Rot === Gruen gibt Laufzeitfehler:

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

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?

Typ von f?

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

Da Ord jedoch Unterklasse von Eq:

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

(Ist aber inhaltlich das Gleiche.)

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.
- Der Typ Typ kann nur sein:
ein Typkonstruktor angewendet auf verschiedene Typvariablen.

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 ml == Blatt mr      = ml == mr
  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 (Drucken) 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
- Diese Methode der Komposition ist schneller als geschachteltes ++: $O(n * \log(n))$ vs. quadratisch

```
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

```

M. Schmidt-Schauß (06) Haskell (2)

25 / 76

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 '>'

```

M. Schmidt-Schauß (06) Haskell (2)

26 / 76

Show für BBaum (2)

Tests:

```

*Main> last $ showBBaum t
'>
(73.38 secs, 23937939420 bytes)
*Main> last $ show t
(das ist showBBaum')
'>
(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

```

M. Schmidt-Schauß (06) Haskell (2)

27 / 76

Rand eines Baumes: analog

```

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

-- Variante 1
bRandschnell tr = reverse (bRandschnell_r [] tr)
bRandschnell_r r (Blatt a) = (a:r)
bRandschnell_r r (Knoten links rechts) =
  bRandschnell_r (bRandschnell_r r rechts) links

-- Variante 2
bRandschnell'_tr = bRandschnell'_r tr []
bRandschnell'_r (Blatt a) = \x-> (a:x)
bRandschnell'_r (Knoten links rechts) =
  (bRandschnell'_r rechts) . (bRandschnell'_r links)

```

M. Schmidt-Schauß (06) Haskell (2)

28 / 76

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)
```

M. Schmidt-Schauß (06) Haskell (2)

29 / 76

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]
```

M. Schmidt-Schauß (06) Haskell (2)

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.

M. Schmidt-Schauß (06) Haskell (2)

31 / 76

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`

M. Schmidt-Schauß (06) Haskell (2)

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

$$\mathbf{T} ::= TV \mid TC \ \mathbf{T}_1 \dots \mathbf{T}_n \mid \mathbf{T}_1 \rightarrow \mathbf{T}_2$$

Erweiterte Typen \mathbf{T}_e mit Typklassen:

$$\mathbf{T}_e ::= \mathbf{T} \mid \mathbf{Kon} \Rightarrow \mathbf{T}$$

Kon ist ein **Typklassenkontext**:

Kon ::= Klassename $TV \mid (\text{Klassename}_1 \ TV, \dots, \text{Klassename}_n \ TV)$

Zusätzlich: Für **Kontext** \Rightarrow **Typ** muss gelten:

Alle Typvariablen von **Kontext** kommen auch in **Typ** vor.

Z.B. $\text{elem} : (\text{Eq } a) \Rightarrow a \rightarrow [\mathbf{a}] \rightarrow \text{Bool}$.

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.

- In Haskell ist es auch möglich über **Typkonstrukturen** zu abstrahieren
- Solche Klassen heißen **Konstruktorklassen**
- Zu Erinnerung: Z.B. ist **Baum a** ein Typ aber **Baum** ein Typkonstruktor.

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.

Functor-Instanz für Listen

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

Hier ist der Typkonstruktor `[]` gemeint! Wie in `[a]`

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)’`

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.

Kinds

(“Kind” ist englisch gemeint)

- Kinds sind quasi **Typen über Typen**
- Syntax: $K ::= * \mid K \rightarrow K$
- Hierbei steht $*$ für einen Typ
- Beispiele:
 - der Typkonstruktor `BBaum` hat den Kind `* -> *`,
 - der Typ `Int` hat den Kind `*`,
 - der Typkonstruktor `Either` hat den Kind `* -> * -> *`.

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
```

Neue Typklassen in Haskell ghc 7.10.2

Monoid

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

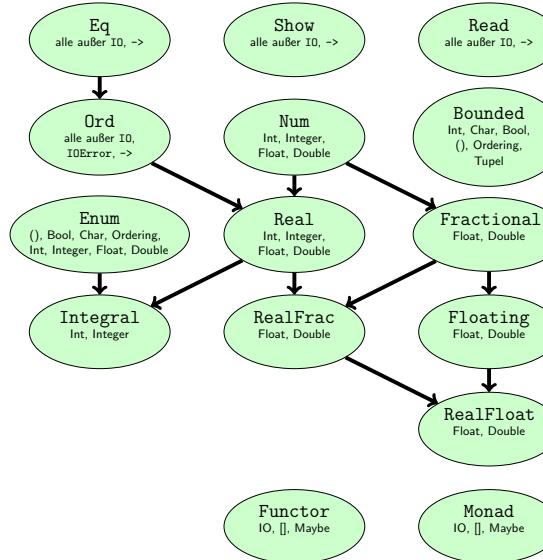
Foldable

```
class Foldable (t :: * -> *) where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  ...
```

Beispiel

```
fold :: (Foldable t, Monoid m) => t m -> m
```

Übersicht über einige der vordefinierten Typklassen



Auflösung der Überladung

Empfehlung

zur eigenen Recherche:
Nutzen Sie :info <...> im ghc.

z.B.:

```
Main> :info Functor
```

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$>) :: a -> f b -> f a
  -- Defined in 'GHC.Base'
instance Functor (Either a) -- Defined in 'Data.Either'
instance Functor [] -- Defined in 'GHC.Base'
instance Functor Maybe -- Defined in 'GHC.Base'
instance Functor IO -- Defined in 'GHC.Base'
instance Functor ((->) r) -- Defined in 'GHC.Base'
instance Functor ((,) a) -- Defined in 'GHC.Base'
```

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 durch Typklassen

Haskellprogramm mit Typklassen

↓ Vor-Übersetzung

Haskellprogramm ohne Typklassen (erweiterte Datentypen)

↓ (normale Compilation)

Programm für abstrakte Maschine (Graphdatenstrukturen)

Auflösung der Überladung in Haskell

- Es gibt **keine Typinformation zur Laufzeit**
deshalb Auflösung zur Compilezeit.
- Diese Auflösung der Überladung ist eine **Übersetzung in typklassenfreies Haskell**.
UND: Es werden Hilfsdatenstrukturen erzeugt
UND: Funktionen bekommen tlw zusätzliche Argumente.
- Auflösung benötigt Typinformation,
daher Auflösung zusammen mit dem Typcheck.
- 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, dass Typinformation, auch für Unterausdrücke, vorhanden ist.
- Wir trennen jedoch Auflösung und Typcheck (Typberechnung), d.h. wir behandeln **zunächst** nur die Auflösung durch Übersetzung (late binding)

Auflösung der Überladung (2)

Erklärung 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 !=
}
```

M. Schmidt-Schauß (06) Haskell (2)

49 / 76

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

M. Schmidt-Schauß (06) Haskell (2)

51 / 76

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
```

M. Schmidt-Schauß (06) Haskell (2)

50 / 76

M. Schmidt-Schauß (06) Haskell (2)

50 / 76

Auflösung der Überladung (5)

`... x == y...` wird zu `... overloadedeq dict x y`

Bei konkreten Typen müssen jedoch die konkreten Dictionaries eingesetzt werden

Z.B. aus

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

wird

`... overloadedeq eqDictBool True False`

M. Schmidt-Schauß (06) Haskell (2)

52 / 76

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: 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: 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: 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

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

Eq-Dictionary für Ordering

```
instance Eq Ordering where
  LT == LT = True
  EQ == EQ = True
  GT == GT = True
  - == - = False ⇒
    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 11 r1 == Knoten 12 r2 = 11 == 12 && r1 == r2
  - == - = False
```

Auflösung: das Dictionary für BBaum a ist eine Funktion,
Diese erhält das Dictionary für a als Argument:

```
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 11 r1) (Knoten 12 r2) =
    eqBBaum dict 11 12 && eqBBaum dict r1 r2
  eqBBaum dict x y = False
```

Auflösung der Überladung (7)

Eq-Dictionary für Ordering

```
instance Eq Ordering where
  LT == LT = True
  EQ == EQ = True
  GT == GT = True
  - == - = False ⇒
    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 (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
```

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)

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:

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

Instanz für BBaum:

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

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]

```

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!**

Haskells hierarchisches Modulsystem

(kein Prüfungsstoff)

Aufgaben von Modulen

- Strukturierung / Hierarchisierung
- Kapselung
- Wiederverwendbarkeit

Moduldefinition in Haskell

```
module Modulname(Exportliste) where
  Modulimporte,
  Datentypdefinitionen,
  Funktionsdefinitionen, ...
  } Modulrumpf
```

- Name des Moduls muss mit Großbuchstaben beginnen
- Exportliste enthält die nach außen sichtbaren Funktionen und Datentypen
- Dateiname = Modulname.hs (bei hierarchischen Modulen Pfad+Dateiname = Modulname.hs)
- Ausgezeichnetes Modul Main muss Funktion main :: IO () exportieren

Import / Export

Beispiel:

```
module Spiel where
  data Ergebnis = Sieg | Niederlage | Unentschieden
  berechneErgebnis a b = if a > b then Sieg
                        else if a < b then Niederlage
                        else Unentschieden
  istSieg Sieg = True
  istSieg _     = False
  istNiederlage Niederlage = True
  istNiederlage _       = False
```

- Da keine Exportliste: Es werden alle Funktionen und Datentypen exportiert
- Außer importierte

Exporte

Exportliste kann enthalten

- Funktionsname (in Präfix-Schreibweise) Z.B.: ausschließlich berechneErgebnis wird exportiert
`module Spiel(berechneErgebnis) where`
- Datentypen mittels data oder newtype, drei Möglichkeiten
 - Nur Ergebnis in die Exportliste:
`module Spiel(Ergebnis) where`
 Typ Ergebnis wird exportiert, die Datenkonstruktoren, d.h. Sieg, Niederlage, Unentschieden jedoch nicht
 - `module Spiel(Ergebnis(Sieg, Niederlage))`
 Typ Ergebnis und die Konstruktoren Sieg und Niederlage werden exportiert, nicht jedoch Unentschieden.
 - Durch den Eintrag `Ergebnis(..)`, wird der Typ mit sämtlichen Konstruktoren exportiert.

Exporte (2)

Exportliste kann enthalten

- Typsynonyme, die mit type definiert wurden

```
module Spiel(Result) where
  ... wie vorher ...
  type Result = Ergebnis
```

- Importierte Module:

```
module Game(module Spiel, Result) where
  import Spiel
  type Result = Ergebnis
```

Game exportiert alle Funktionen, Datentypen und Konstruktoren, die auch Spiel exportiert sowie zusätzlich noch den Typ Result.

Importe

Einfachste Form: Importiert alle Einträge der Exportliste

```
import Modulname
```

Andere Möglichkeiten:

- Explizites Auflisten der zu importierenden Einträge:
`module Game where
 import Spiel(berechneErgebnis, Ergebnis(..))
 ...`
- Explizites Ausschließen einzelner Einträge:
`module Game where
 import Spiel hiding(istSieg, istNiederlage)
 ...`

Importe (2)

So importierte Funktionen und Datentypen sind mit ihrem unqualifizierten und ihrem qualifizierten Namen ansprechbar.
 Qualifizierter Name: *Modulname.unqualifizierter Name*

```
module A(f) where
  f a b = a + b
module B(f) where
  f a b = a * b
module C where
  import A
  import B
  g = f 1 2 + f 3 4 -- funktioniert nicht!
```

```
Prelude> :l C.hs
```

```
ERROR C.hs:4 - Ambiguous variable occurrence "f"
*** Could refer to: B.f A.f
```

Importe (3)

Mit qualifizierten Namen funktioniert es:

```
module C where
  import A
  import B
  g = A.f 1 2 + B.f 3 4
```

Mit Schlüsselwort qualified kann man nur die qualifizierten Namen importieren:

```
module C where
  import qualified A
  g = f 1 2  -- f ist nicht sichtbar
```

Prelude> :l C.hs

ERROR C.hs:3 - Undefined variable "f"

Importe (4)

Lokale Aliase: Schlüsselwort as:

```
import LangerModulName as C
```

Importe (5)

Übersicht:

Import-Deklaration	definierte Namen
import M	f, g, M.f, M.g
import M()	keine
import M(f)	f, M.f
import qualified M	M.f, M.g
import qualified M()	keine
import qualified M(f)	M.f
import M hiding ()	f, g, M.f, M.g
import M hiding (f)	g, M.g
import qualified M hiding ()	M.f, M.g
import qualified M hiding (f)	M.g
import M as N	f, g, N.f, N.g
import M as N(f)	f, N.f
import qualified M as N	N.f, N.g

Hierarchische Modulstruktur

- Modulnamen mit Punkten versehen geben Hierarchie an: z.B.
`module A.B.C`
- Allerdings ist das rein syntaktisch (es gibt keine Beziehung zwischen Modul A.B und A.B.C)
- Beim Import:
`import A.B.C`
sucht der Compiler nach dem File namens C.hs im Pfad A/B/