Programmieren für Fortgeschrittene - eine Einführung in Haskell

Teil drei - noch etwas mehr

Stephan Mielke, 15.12.2014

Technische Universität Braunschweig, IPS

Überblick

- Typkonstruktoren
- Typüberprüfung und -berechnung
- Typkonversion (Cast-Anweisungen)
- Polymorphismus

Überblick

- **Typkonstruktoren**
- Typüberprüfung und -berechnung
- Typkonversion (Cast-Anweisungen)
- Polymorphismus

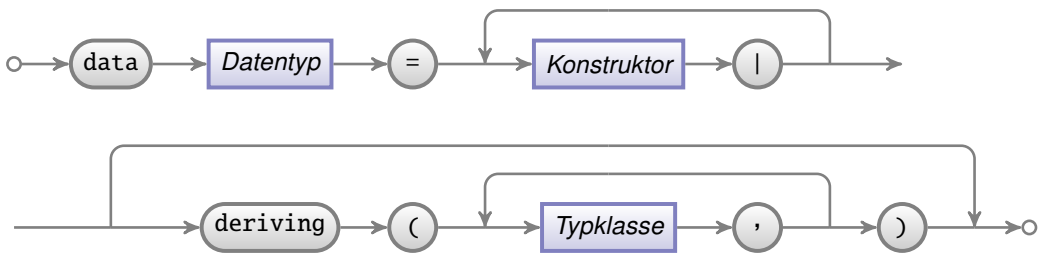
Typkonstruktoren- Eigene Datentypen

- **Typkonstruktoren**
  - Eigene Datentypen
  - Typ-Synonyme
  - Rekursive Datenstrukturen
  - Listen

Typkonstruktoren - Aufzählungstyp

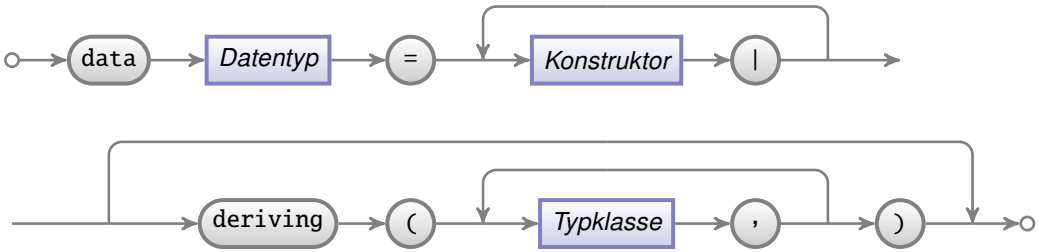
Aufzählungstypen sind mit Enums aus C bzw. C++ zu vergleichen und fassen inhaltlich Elemente zusammen

data

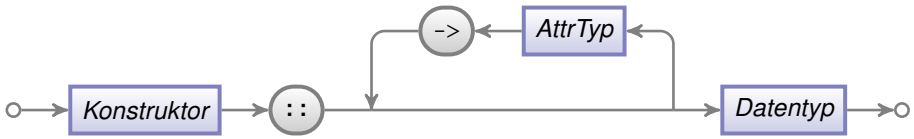


Typkonstruktoren - Aufzählungstyp

`data`



Konstruktor



In diesem Fall erhält der Konstruktor keine Parameter.

Typkonstruktoren - Aufzählungstyp

```
data Color = Blue | Cyan | Yellow | Orange | Green
```

Typkonstruktoren - Aufzählungstyp

```
data Color = Blue | Cyan | Yellow | Orange | Green
```

Wie werden die Konstuktoren aussehen?

Typkonstruktoren - Aufzählungstyp

```
data Color = Blue | Cyan | Yellow | Orange | Green
```

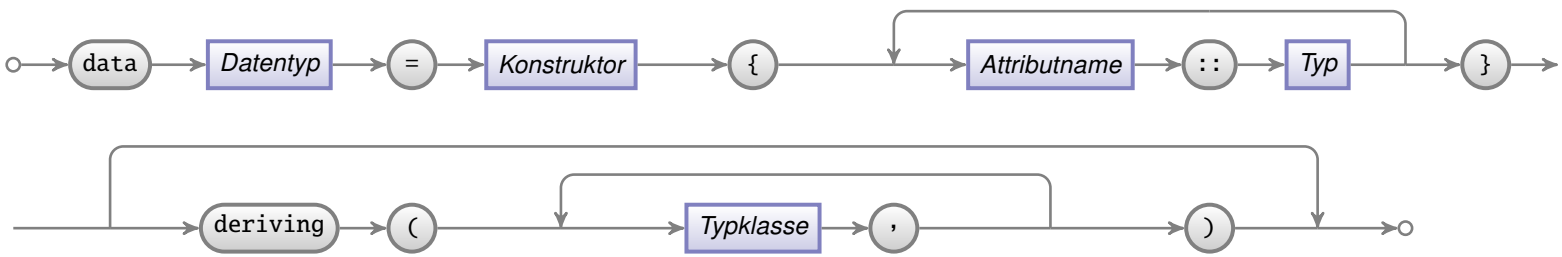
Konstruktoren:

```
Blue   :: Color
Cyan   :: Color
Yellow :: Color
Orange :: Color
Green  :: Color
```

Typkonstruktoren - Produkttyp

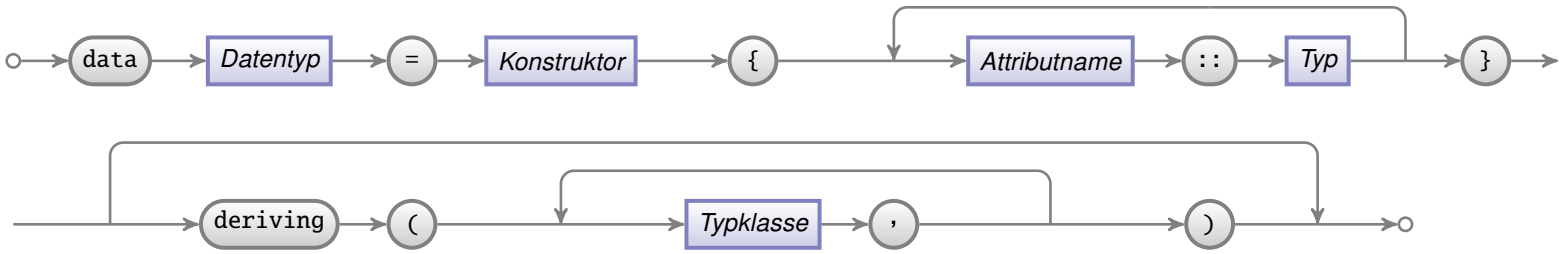
- Produkttyp ist ein Tupel der einzelnen Attribute
- Tupel fassen Gruppen von Daten zusammen, die logisch zusammengehören und gemeinsam etwas Neues und Eigenständiges bilden

`data` mit allen Angaben

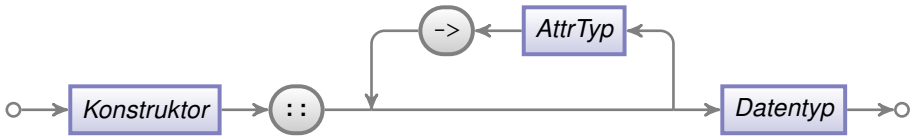


Typkonstruktoren - Produkttyp

`data` mit allen Angaben

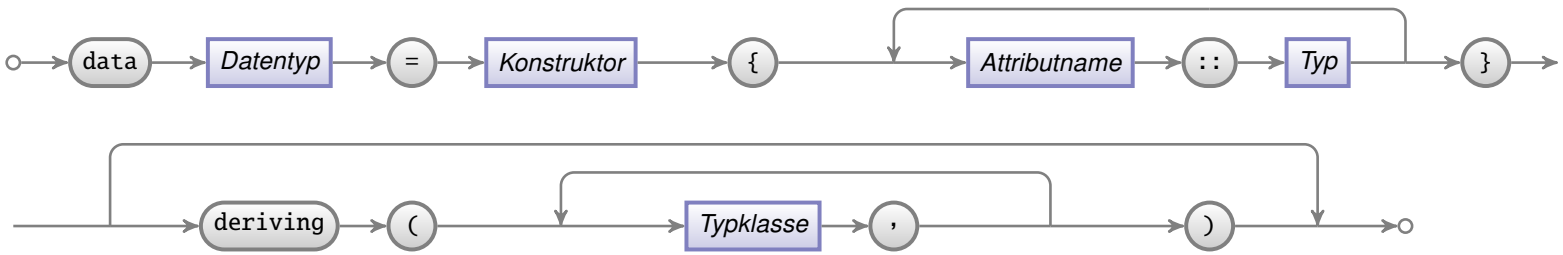


Konstruktor



Typkonstruktoren - Produkttyp

`data` mit allen Angaben



Selektor (bei allen Angaben automatisch erstellt)

Typkonstruktoren - Produkttyp

```haskell
data Point = Point{x :: Double, y :: Double}

data Circle = Circle{center :: Point, radius :: Double}
```

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
```

Kurz

```
data Point = Point Double Double
```

```
data Circle = Circle{center :: Point, radius :: Double}
```

Kurz

```
data Circle = Circle Point Double
```

Kurz Schreibweise

Bei der Kurz-Schreibweise werden keine Selektoren erstellt

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

Konstruktorfunktionen: ?

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

Konstruktorfunktionen:

```
Point :: Double -> Double -> Point
Circle :: Point -> Double -> Circle
```

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

Selektorfunktionen: ?

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

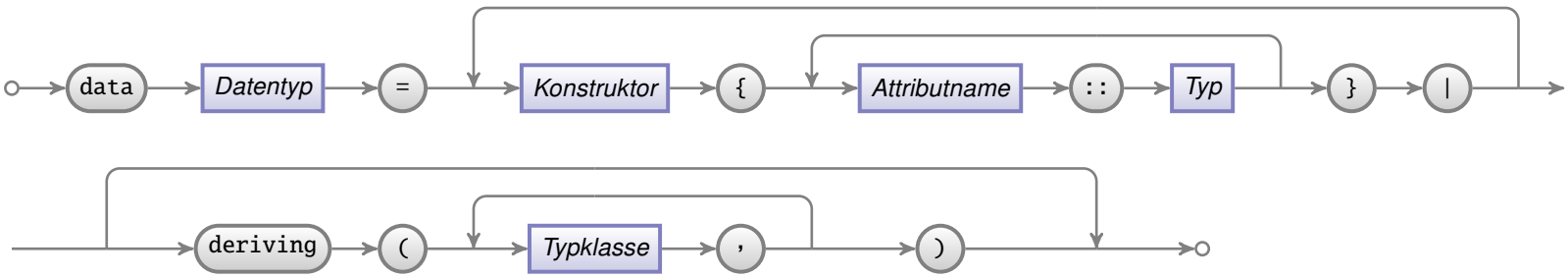
Selektorfunktionen:

```
x :: Point -> Double
y :: Point -> Double
center :: Circle -> Point
radius :: Circle -> Double
```

Typkonstruktoren - Summentyp

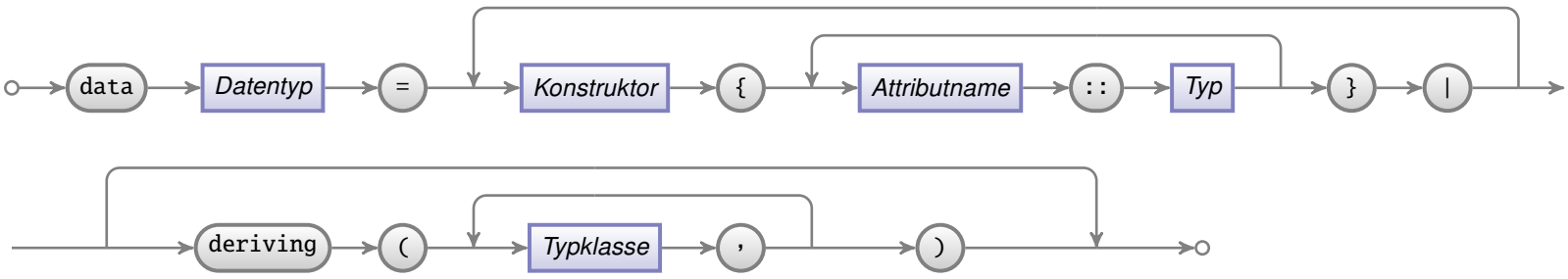
- Summentypen fassen inhaltlich verwandte (aber struktuell verschiedene) Elemente zusammen
- Sind eine Fusion von Aufzählungs- und Produkttyp

`data` mit allen Angaben

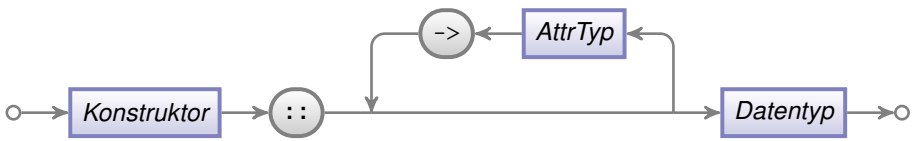


Typkonstruktoren - Summentyp

`data` mit allen Angaben

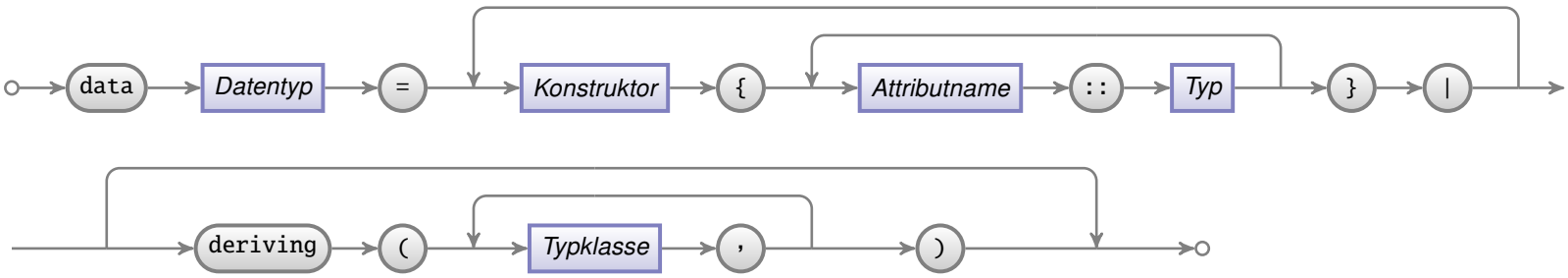


Konstruktor



Typkonstruktoren - Summentyp

`data` mit allen Angaben



Selektor (bei allen Angaben automatisch erstellt)

Typkonstruktoren - Summentyp

```
data Point = Point{x :: Double, y :: Double}

data Shape = Circle{center :: Point,
    radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Frage

Was sind hier Selektoren und Konstruktoren?

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Konstruktoren: ?

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Konstruktoren:

```
Circle :: Point -> Double -> Shape
Rectangle :: Point -> Double -> Double -> Shape
Triangle :: Point -> Point -> Point -> Shape
```

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Selektoren: ?

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Selektoren:

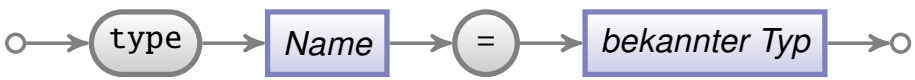
```
center :: Shape -> Point
radius :: Shape -> Double
point :: Shape -> Point
width :: Shape -> Double
...
```

Typkonstruktoren- Typ-Synonyme

- **Typkonstruktoren**
  - Eigene Datentypen
  - Typ-Synonyme
  - Rekursive Datenstrukturen
  - Listen

Typ-Synonyme

- Typ-Synonyme sind keine eigenen Typen sondern führen nur neue Namen für bekannte Typen ein
- Vorteile:
  - Verbesserte Lesbarkeit
  - Intern wird bei `type Euro = Int` wieder `Int`



Typ-Synonyme

```haskell
type Euro = Int
type Cent = Int
type Preis = (Euro, Cent)

type Tupel = (Int, Int)
```

Achtung

`Preis` und `Tupel` sind für uns und intern `(Int, Int)`

Typ-Synonyme

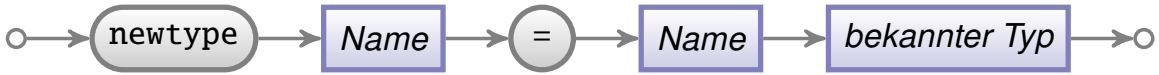
```
type Euro = Int
type Cent = Int

add :: Euro -> Euro -> Euro
add a b = a + b

add' :: Euro -> Cent -> Int
add' a b = a + b
```

Achtung

`add' 5 (add 5 8)` funktioniert

Typ-Synonyme mit Typsicherheit

- `newtype` wird statt `type` verwendet, wenn Typsicherheit benötigt wird
- `newtype` verhält sich somit geauso wie `data`
- Jedes `newtype` kann durch `data` ersetzt werden
- Jedoch `data` kann nur in Ausnahmefällen durch `newtype` ersetzt werden



Typ-Synonyme mit Typsicherheit

- `newtype` wird statt `type` verwendet, wenn Typsicherheit benötigt wird
- `newtype` verhält sich somit geauso wie `data`
- Jedes `newtype` kann durch `data` ersetzt werden
- Jedoch `data` kann nur in Ausnahmefällen durch `newtype` ersetzt werden

```
newtype Euro = Euro Int
newtype Cent = Cent Int
```

Achtung

`Euro` und `Cent` sind nicht kompatibel

Typkonstruktoren- Rekursive Datenstrukturen

- **Typkonstruktoren**
  - Eigene Datentypen
  - Typ-Synonyme
  - Rekursive Datenstrukturen
  - Listen

Rekursive Datenstrukturen

- Datenstrukturen können auf sich selbst verweisen
- Es sind „unendliche“ Rekursionen erlaubt, solange der Arbeitsspeicher mitspielt

Rekursive Datenstrukturen

```
data List = []
          | Cons {head :: Int, tail :: List}

data Tree = Nil
          | Tree {left :: Tree, value :: Int,
            right :: Tree}

data Nat = Zero
         | Succ {pred :: Nat}
```

Typkonstruktoren- Listen

- **Typkonstruktoren**
  - Eigene Datentypen
  - Typ-Synonyme
  - Rekursive Datenstrukturen
  - Listen

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

- Listen sind Folgen von Elementen gleichen Types
- `a` ist hier der Platzhalter für einen Typ
  somit kann das `a` für `Int`, `Integer` usw. stehen

Konstruktoren: ?

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

- Listen sind Folgen von Elementen gleichen Types
- `a` ist hier der Platzhalter für einen Typ
  somit kann das `a` für `Int`, `Integer` usw. stehen

Konstruktoren:

```
[] :: [a]
Cons :: a -> [a] -> [a]
```

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

Selektoren: ?

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

Selektoren:

```
head :: [a] -> a
tail :: [a] -> [a]
```

Listen in Funktionen

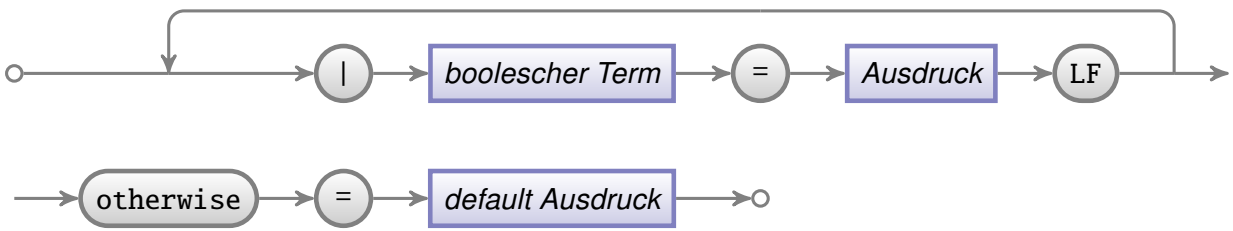
```haskell
length :: [Int] -> Int
length []     = 0
length (_:xs) = 1 + length xs

append :: [Int] -> [Int] -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys

sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Listen in Funktionen

```haskell
filter :: [Int] -> [Int]
filter []     = []
filter (x:xs) | ok x       = x : filter xs
              | otherwise = filter xs
              where
                ok x = (mod x 2) == 1
```

Was macht diese Funktion?

Was ist der Funktionskopf von `ok`?



Listen in Funktionen

```haskell
filter :: [Int] -> [Int]
filter []     = []
filter (x:xs) | ok x       = x : filter xs
              | otherwise = filter xs
              where
                ok x = (mod x 2) == 1
```

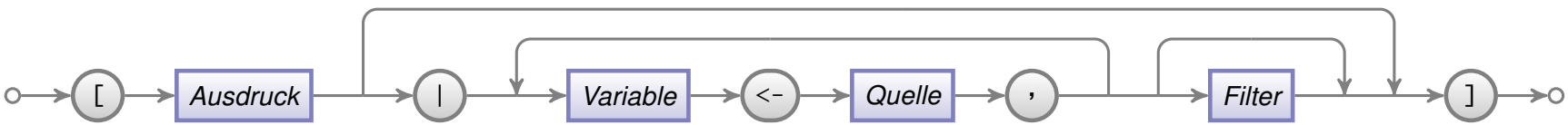
Was macht diese Funktion?

Was ist der Funktionskopf von `ok`?

```haskell
ok :: Int -> Bool
ok      x =  (mod x 2) == 1
```

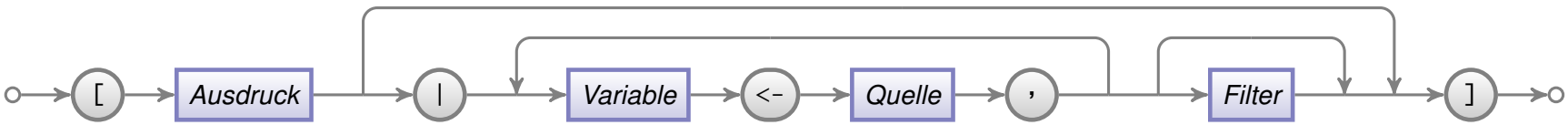
Listen Generatoren

- Für Listen existieren Generatoren
- Sonst müsste jedes Element von Hand aufgeschrieben werden
- `[0..]` generiert eine unendliche Liste
- Warum es unendliche Listen geben kann kommt später
- `[0..10]` generiert `[0,1,2,3,4,5,6,7,8,9,10]`



Listen Generatoren

```
[x * x| x <- [1..5]]
[(i,j)|i <- [1,2], j <- [1..4]]

[y | y <- [1..], even y]
[a * a | a <- [1..], odd a]
[square x | x <- [0..], square x < 10]
```



Listen Generatoren

Berechnung aller Primzahlen

```
primes = sieves [2..]
  where
    sieves (p:xs) = p:sieves [x|x<- xs, mod x p > 0]
```

Wichtige Liste

- Erinnert ihr euch noch an `Char`?
- Erinnert ihr euch noch an `type`?
- Wie wird wohl `String` definiert sein?

Wichtige Liste

- Erinnert ihr euch noch an `Char`?
- Erinnert ihr euch noch an `type`?
- Wie wird wohl `String` definiert sein?

```
type String = [Char]
```

String

- Strings sind Listen von `Char`s
- Alle Funktionen die generisch für Listen definiert sind, sind auch für Strings definiert
- Strings werden intern nicht anders als normale Listen behandelt
- Also kein Stringspool, keine Unabänderbarkeit von Strings usw. . . .

String

```
wochenTag :: Int -> String
wochenTag 1 = "Montag"
wochenTag 2 = "Dienstag"
wochenTag 3 = "Mittwoch"
wochenTag 4 = "Donnerstag"
wochenTag 5 = "Freitag"
wochenTag 6 = "Samstag"
wochenTag 7 = "Sonntag"
wochenTag _ = undefined
```

„Bottom Element“

jeder Wert außer 1, 2, 3, 4, 5, 6, 7 führt zum „Bottom Element“ und führt zu einer Exception vom Typ `Prelude.undefined`

Überblick

- Typkonstruktoren
- **Typüberprüfung und -berechnung**
- Typkonversion (Cast-Anweisungen)
- Polymorphismus

Typüberprüfung

- Es sind nur typengleich, welche auch gleich sind!
- `Int` ist kompatibel zu `Int`
- Aber `Int` ist nicht kompatibel zu `Integer`

Typüberprüfung

- Typ-Synonyme sind gleich, wenn sie auf den gleichen Typ abbilden
- `type Euro = Int`
- `type Laenge = Int`
- Zwischen beiden Typen gibt es keinen Unterschied

Typberechnung

- Wenn keine Typen angegeben wurden, wird der passende Typ berechnet
- Bei Eingaben im GHCi wird der richtige Typ „erraten“

Typberechnung

- Wenn keine Typen angegeben wurden, wird der passende Typ berechnet
- Bei Eingaben im GHCi wird der richtige Typ „erraten“
- Ok statt „zu raten“ wird der Typcheck-Algorithmus von Robin Milner verwendet

Überblick

- Typkonstruktoren
- Typüberprüfung und -berechnung
- **Typkonversion (Cast-Anweisungen)**
- Polymorphismus

Typkonversion

- In Haskell existieren keine Cast-Anweisungen wie in Java
- Jeder Cast wird über eine Funktion realisiert
- `toInteger :: a -> Integer`
- `fromInteger :: Integer -> a`
- `...`

Überblick

- Typkonstruktoren
- Typüberprüfung und -berechnung
- Typkonversion (Cast-Anweisungen)
- **Polymorphismus**

Polymorphismus- Typparameter

- **Polymorphismus**
  - Typparameter
  - Typklassen

Polymorphismus

- Bisher hatten wir nur Funktionen für genau einen Typ
- Nun lernen wir Typklassen und Typparameter kennen

Typparameter

Erinnert ihr euch noch an Listen?

```
data List = []
          | Cons {head :: Int, tail :: List}
```

Damit wir nicht für jeden Datentyp eine neue Liste definieren müssen

`Cons` ist nun `(:)`

```
data List a = []
            | (:) {head :: a, tail :: List a}
```

Typparameter

Erinnert ihr euch noch an Listen?

```
data List = []
          | Cons {head :: Int, tail :: List}
```

kurz:

```
data [a] = []
         | (:) {head :: a, tail :: [a]}
```

Typparameter

Genauso in Funktionen

```
append :: [Int] -> [Int] -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Typparameter

Genauso in Funktionen

```
append :: [Int] -> [Int] -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Damit wir nicht für jeden Datentyp eine neue Funktion definieren müssen

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Polymorphismus- Typklassen

- **Polymorphismus**
  - Typparameter
  - Typklassen

Typklassen

- Typklassen fassen Typen zusammen, die ähnliche Operationen unterstützen
- Alle Ausprägungen einer Funktion einer Typklassen tragen dann den gleichen Namen
- $\rightarrow$ Overloading, d.h. der gleiche Funktionsname steht für unterschiedliche Implementierungen

Typklassen

Zu allgemein

```
(+) :: a -> a -> a
```

Typklassen

Zu allgemein

```
(+) :: a -> a -> a
```

Zu speziell

```
(+) :: Int -> Int -> Int
```

Typklassen

Zu allgemein

```
(+) :: a -> a -> a
```

Zu speziell

```
(+) :: Int -> Int -> Int
```

Genau richtig

```
(+) :: Num a =>a -> a -> a
```

Typklassen

Nutzung von Typklassen

```
summe :: Num a => [a] -> a
summe []     = 0
summe (x:xs) = x + summe xs
```

Typklassen

Eigene Datentypen mit Typklassen

```
data Point = Point{x :: Double, y :: Double}
  deriving (Eq, Show)

data Circle = Circle{center :: Point,
  radius :: Double}
  deriving (Eq, Show)
```

mit `deriving` wird geraten, wie die Implementierung von `Eq`, `Ord`, `Show` usw. sein sollen

Typklassen

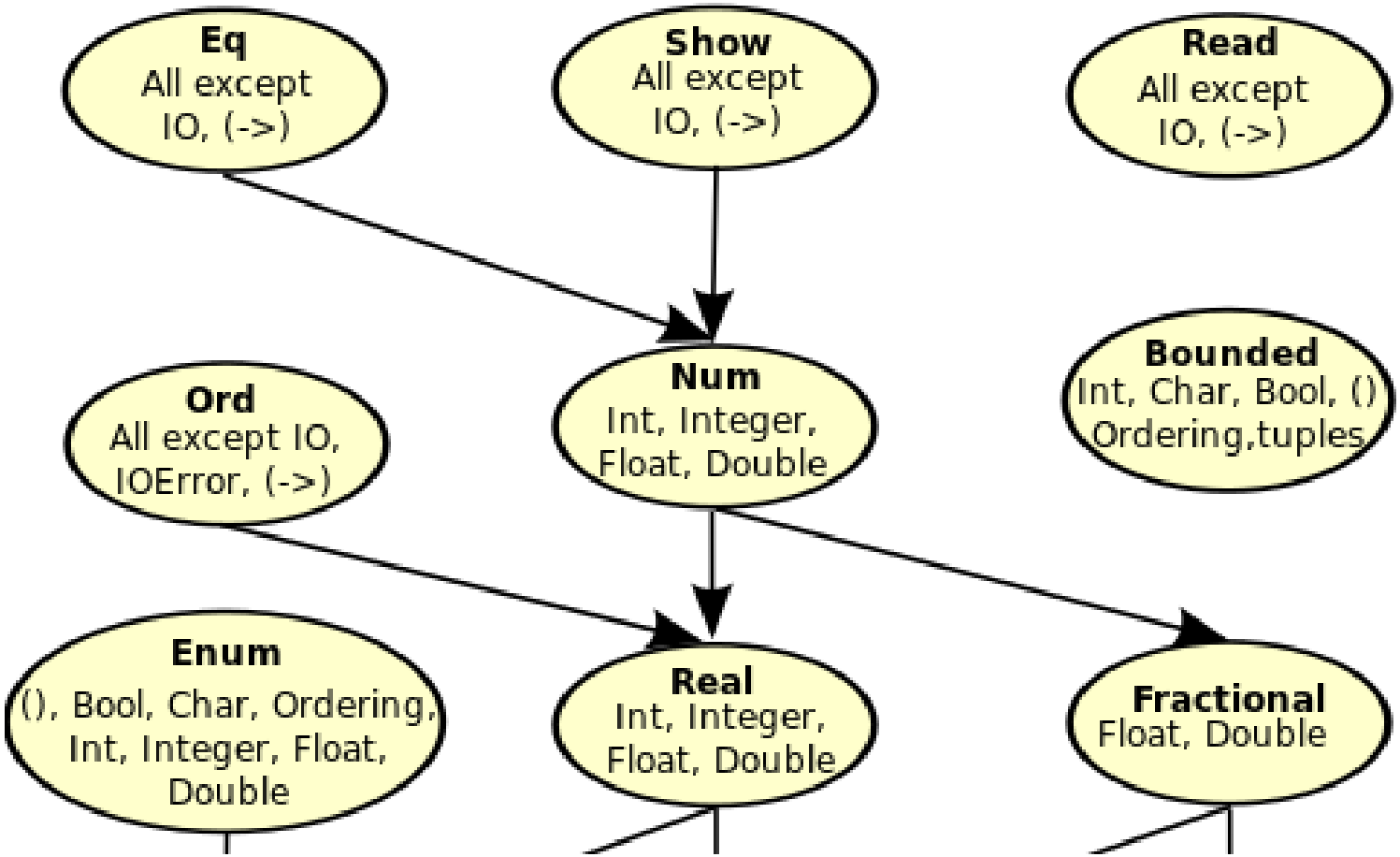


Abbildung 1: Typklassen Teil 1 ©wikibooks.org

Typklassen

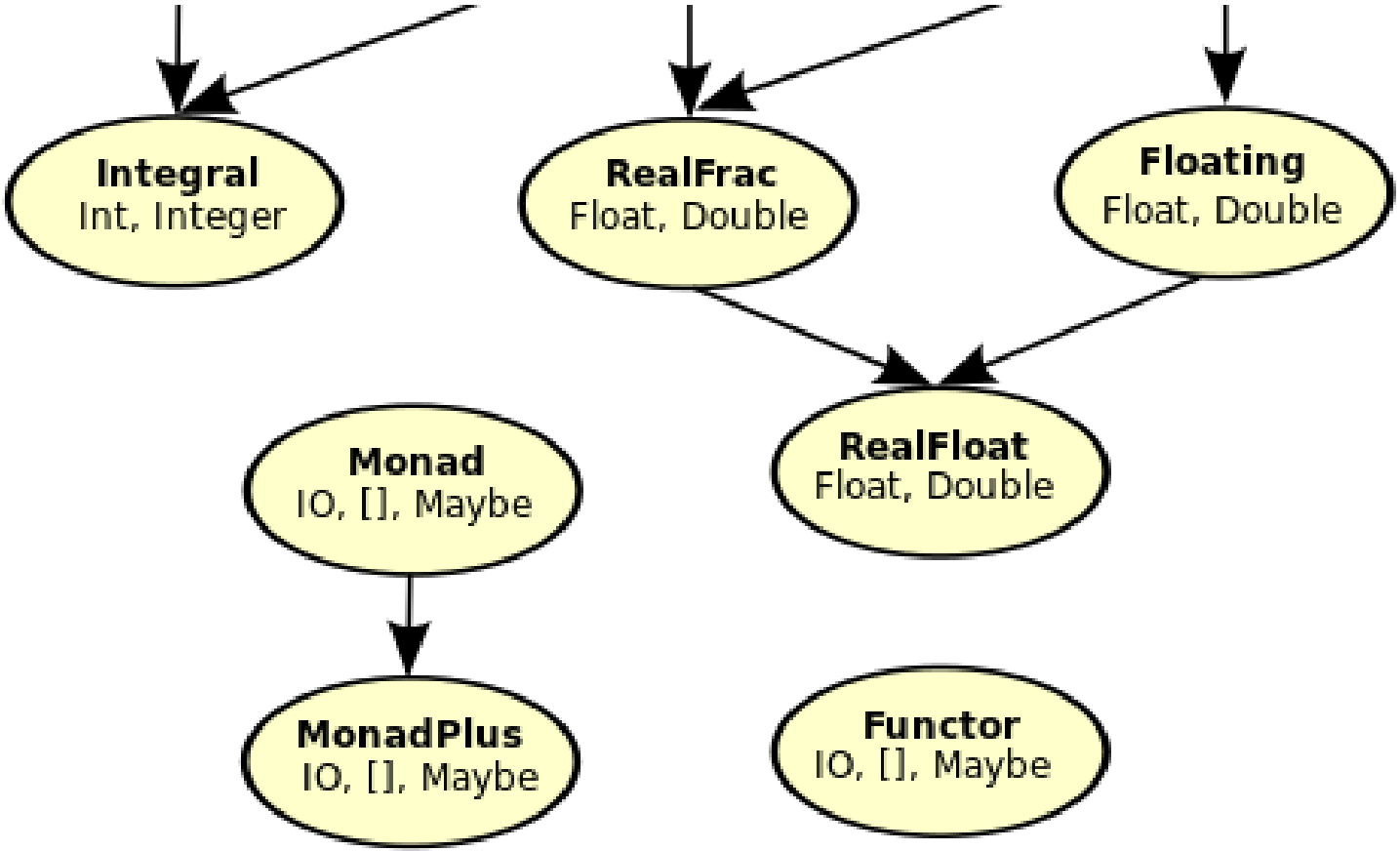


Abbildung 2: Typklassen Teil 2 ©wikibooks.org

Typklassen

Die `Eq` Typklasse

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

Achtung

Die Definition ist zirkulär!

Typklassen

Die `Ord` Typklasse

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

Achtung

Die Definition ist zirkulär!

Typklassen

`instance` von `Eq` und `Ord` für `Buch`

```
data Buch = Buch Int [String] String deriving (Show)

instance Eq Buch where
  (Buch isbn1 _ _) == (Buch isbn2 _ _)
    = isbn1 == isbn2

instance Ord Buch where
   (Buch isbn1 _ _) `compare` (Buch isbn2 _ _)
     = isbn1 `compare` isbn2
```

Typklassen

instance von Eq

```
instance Eq Buch where
  (Buch isbn1 _ _) == (Buch isbn2 _ _)
    = isbn1 == isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo"== Buch 123 ["du"] "keiner"
```

Typklassen

instance von Eq

```
instance Eq Buch where
  (Buch isbn1 _ _) == (Buch isbn2 _ _)
    = isbn1 == isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo"== Buch 123 ["du"] "keiner"
```

Ausgabe

```
True
```

Typklassen

instance von Ord

```
instance Ord Buch where
   (Buch isbn1 _ _) `compare` (Buch isbn2 _ _)
     = isbn1 `compare` isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo"< Buch 123 ["du"] "keiner"
```

Typklassen

instance von Ord

```
instance Ord Buch where
   (Buch isbn1 _ _) `compare` (Buch isbn2 _ _)
     = isbn1 `compare` isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo"< Buch 123 ["du"] "keiner"
```

Ausgabe

```
False
```

Danke

Vielen Dank für die Aufmerksamkeit und das Interesse!