



Funktionale Programmierung

Woche 6 – Typklassen

René Thiemann Philipp Dablander Joshua Ocker Michael Schaper Lilly Schönherr Adam Pescoller

Institut für Informatik

Typklassen – Definition

Letzte Vorlesung

- Layout-Regel: definiere Blöcke durch Einrückung oder mit { ...; ...; ...}
- case-Ausdrücke: Pattern-Matching in rechten Seiten


```
case expr of { pat -> expr; ... ; pat -> expr }
```
- lokale Definitionen mit `let` und `where`

```
let { pat = expr; fName pat ... pat = expr } in expr

fName pat ... pat = expr
where pat          = expr
      fName pat .. pat = expr
```
- bedingte Gleichungen


```
fName pat ... pat
| cond = expr
| ... = ... -- + optional where-block
```
- Rekursion mit Zahlen

RT et al. (IFI @ UIBK)

Woche 6

2/23

Typklassen – Aktueller Stand

- kurze Einführung, dass es Typklassen gibt, z.B., `Num a`, `Eq a`, ...
- Typklassen bieten **einheitlichen Zugriff** auf Funktionen, die für unterschiedliche Typen implementiert werden können
- Beispiel
 - `(<) :: Ord a => a -> a -> Bool` ist Name einer Funktion, um zwei Elemente zu vergleichen
 - jeder der folgenden Typen hat eine andere Implementierung von `(<)`
 - `(<) :: Int -> Int -> Bool`
 - `(<) :: Char -> Char -> Bool`
 - `(<) :: Bool -> Bool -> Bool`
 - `(<) :: Ord a => [a] -> [a] -> Bool`
 - `(<) :: (Ord a, Ord b) => (a, b) -> (a, b) -> Bool`
- diese Vorlesung: **Definition** von Typklassen
 - Vorstellung der Definition von existierenden Typklassen
 - Spezifikation von neuen Typklassen
- diese Vorlesung: **Instantiierung** von Typklassen
 - Definition einer Implementierung für die Kombination aus Typ und Typklasse

Typklassen – Definition

- Typklassen werden mit dem Schlüsselwort `class` definiert:

```
class Tcname a where
  fName :: ty      -- type ty + description of fName
  ...
  lhs = rhs        -- optional default implementation
  ...
```

 - `Tcname` ein Name der Typklasse ist, beginnend mit einem Großbuchstaben
 - `a` ist eine einzelne Typvariable
 - es gibt Typ-Definitionen von (mehreren) Funktionen – ohne definierende Gleichungen
 - für jede Funktion `fName` sollte es eine *informelle Beschreibung* geben
 - es kann Standard-Implementierungen für jede Funktion `fName` geben
- bei Verwendung einer Typklassen-Einschränkung (**TK-Einschränkung**) `Tcname a => ...` werden **alle** Funktionen `fName` verfügbar
- Definition einer Typklassen-Instanz benötigt Implementierung aller Funktionen
- Ausnahme: Funktionen, für die es eine Standard-Implementierung gibt, brauchen nicht separat implementiert werden

Typklassen – Beispiel Equality

- ```
class Equality a where
 equal :: a -> a -> Bool -- equality
 different :: a -> a -> Bool -- inequality
 -- equal x x should evaluate to True
 -- equal and different should be symmetric
 -- exactly one of equal x y and different x y should be True
 equal x y = not (different x y) -- default implementation
 different x y = not (equal x y) -- default implementation
```
- wenn TK-Einschränkung `Equality b` in Funktionstyp von `f` vorkommt, dann dürfen `equal :: b -> b -> Bool` und `different :: b -> b -> Bool` in den definierenden Gleichungen von `f` benutzt werden
  - wenn ein konkreter Typ `Ty` Instanz von `Equality` ist, dann können `equal :: Ty -> Ty -> Bool` und `different :: Ty -> Ty -> Bool` ohne Angabe einer TK-Einschränkung genutzt werden
  - um einen Typ zu einer Instanz von `Equality` zu machen, muss mindest eine der Funktionen `equal` oder `different` für diesen Typ implementiert werden

## Operator Syntax, Typklasse Eq

- `Eq` ist eine vordefinierte Typklasse für Gleichheit
- Unterschied zu `Equality`: Nutzung von Operatoren anstelle von Funktionen
- in Haskell kann jeder **Operator in eine Funktion verwandelt werden und umgekehrt**
  - Klammern verwandeln einen Operator `&` in Funktion `(&)`
  - `a & b` ist das gleiche wie `(&) a b`
  - `(&) :: ty` ist die Syntax, um den Typ eines Operators zu spezifizieren
  - Back-Ticks wandeln binäre Funktion `fName` in einen Operator ``fName``
  - `fName a b` ist das gleiche wie `a `fName` b`, z.B. `12 `div` 5`
- Konsequenz: in der Typklassen Definition sind `(==)` und `(/=)` einfach Funktionsnamen

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

<http://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#t:Eq>

## Typklassen Hierarchien

- Typklassen können hierarchisch definiert werden
- Syntax: `class (Tclass1 a, ..., TclassN a) => TclassNew a where ...`
- Konsequenzen
  - TK-Einschränkung `TclassNew a` fügt implizit die TK-Einschränkungen `Tclass1 a, ..., TclassN a` hinzu
  - bei Nutzung von TK-Einschränkung `TclassNew a`, werden alle Funktionen in `TclassNew, Tclass1, ..., TclassN` verfügbar
  - eine Instanz von `TclassNew` für einen Typ ist nur möglich, wenn dieser Typ bereits Instanz von `Tclass1, ..., TclassN` ist
  - Standard-Implementierungen in `TclassNew` können alle Funktionen von `Tclass1, ..., TclassN` nutzen

## Example: Type Class Ord

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

- minimale vollständige Definition: `compare` oder `(<=)`
- Anmerkung: Standard-Implementierung nutzt `(<=)` aus Typklasse `Eq`
- <http://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#t:Ord>

## Typklassen Instanzen

- viele Typen sind Instanzen von `Eq` und `Ord`
- Beispiel
  - `Eq Int` Bedeutung: `Int` ist eine Instanz von `Eq`
  - `Eq Char, Eq Integer, Eq Bool, ...`
  - `Eq a => Eq [a]`
    - Bedeutung: `[a]` ist eine Instanz von `Eq`, wenn `a` eine Instanz von `Eq` ist
    - Folgerung: `Eq [Int], Eq String, Eq [[Integer]], ...`
  - `Eq a => Eq (Maybe a), (Eq a, Eq b) => Eq (Either a b), ...`
  - `Eq (), (Eq a, Eq b) => Eq (a,b), ...` für Tupel mit maximal 15 Einträgen
  - `Ord Bool, Ord Char, Ord Integer, Ord Double, ...`
  - `Ord a => Ord [a], (Ord a, Ord b) => Ord (Either a b), ...`
  - `Ord (), (Ord a, Ord b) => Ord (a,b), ...` für Tupel mit maximal 15 Einträgen
  - `Ord a => Ord [(String, Either (a,Int) [Double])]`
  - Funktionen sind nicht Instanzen von `Eq` und `Ord`: `Eq (Int -> Int)` gilt nicht

## Typklassen Hierarchien für Zahlen

## Typklasse Num

- `Num a` bietet grundlegende arithmetische Operationen
- Spezifikation

```
(+) :: a -> a -> a
(*) :: a -> a -> a
(-) :: a -> a -> a
abs :: a -> a
signum :: a -> a
fromInteger :: Integer -> a
negate :: a -> a
```
- minimale vollständige Definition: man darf nur eines von `negate` oder `(-)` weglassen
- **ganzzahlige Literale** sind verfügbar für jeden `Num a` Typen: `4715 :: Num a => a`
- Instanzen: `Int, Integer, Float, Double`

## Typklasse Fractional – Division

- Definition: `class Num a => Fractional a where ...`
- Auswahl von Funktionen: `(/) :: a -> a -> a`
- Kommazahlen sind verfügbar für jeden `Fractional a` Typen:  
`5.72 :: Fractional a => a`
- Instanzen: `Float, Double`

## Typklasse Integral – Division mit Rest

- Definition: `class (Num a, Ord a) => Integral a where ...`
- Auswahl von Funktionen  
`toInteger :: a -> Integer`  
`div :: a -> a -> a` -- quotient  
`mod :: a -> a -> a` -- remainder
- Instanzen: `Int, Integer`

## Unterschiedliches Verhalten bei Division durch 0

- testen Sie in ghci: `1 `div` 0, 1 / 0, 1 / (-0), 0 == -0`

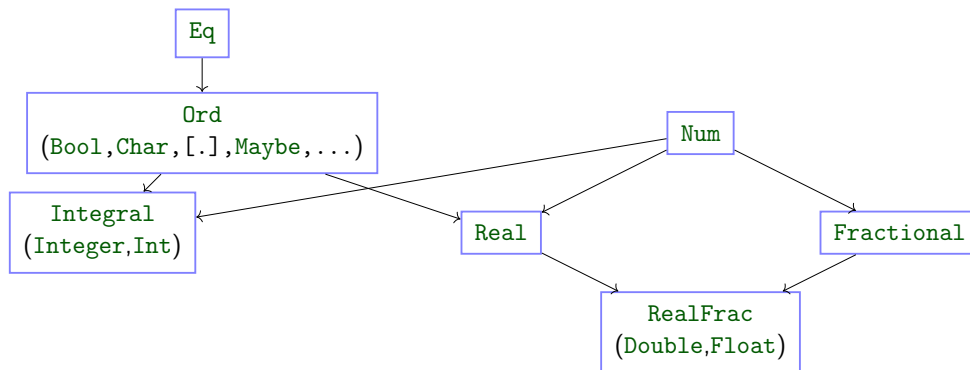
## Typklasse RealFrac – Runden

- Definition: `class (Real a, Fractional a) => RealFrac a where ...`
- Auswahl von Funktionen  
`floor :: Integral b => a -> b` -- round down to next integer  
`ceiling :: Integral b => a -> b` -- round upwards to next integer  
`round :: Integral b => a -> b` -- round to closest integer
- Instanzen: `Float, Double`

## Konvertierung von Zahlen

- von `Integral` in beliebigen Zahlentyp  
`fromIntegral :: (Integral a, Num b) => a -> b`  
`fromIntegral x = fromInteger (toInteger x)`
- Beispiel: Bestimmung der Nachkommastellen  
`fractionalPart :: RealFrac a => a -> a`  
`fractionalPart x = x - fromInteger (floor x)`

## Auszug aus der Typklassen Hierarchie



- Dokumentation:  
<http://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html>

Instantiierung von Typklassen

## Instanziierung einer Typklasse

- bislang: Definition von Typklassen, Auflistung existierenden Instanzen
- jetzt: Definition eigener Instanzen; Syntax:

```
instance (optional class assertion) => TClass (TConstr a1 .. aN) where
... -- implementation of functions
```

  - `a1 .. aN` sind unterschiedliche Typvariablen
  - diese dürfen in den TK-Einschränkungen genutzt werden
  - Implementierung muss definierende Gleichungen für jede Funktion `f :: ty` in der Definition von `TClass a` enthalten
    - wichtig: `f` muss für Typ `ty` implementiert werden, der aus `ty` entsteht, in dem man `a` durch `TConstr a1 .. aN` ersetzt
    - Funktionen `f`, die eine Standard-Implementierung besitzen, dürfen ausgelassen werden
    - Implementierung darf TK-Einschränkungen nutzen
- die Nutzung von `deriving TClass` in Datentyp Definitionen liefert Standard-Instanzen; diese sind für Typklassen `Eq`, `Ord`, `Show` verfügbar

## Beispiel: Komplexe Zahlen

```
data Complex = Complex Double Double -- real and imaginary part
```

```
-- remark: we do not write deriving (Eq, Show),
-- but implement these instances on our own
```

```
instance Eq Complex where
 Complex r1 i1 == Complex r2 i2 = r1 == r2 && i1 == i2
-- for (/=) use default implementation
```

```
instance Show Complex where
 show (Complex r i)
 | i == 0 = show r
 | r == 0 = show i ++ "i"
 | i < 0 = show r ++ show i ++ "i"
 | otherwise = show r ++ "+" ++ show i ++ "i"
```

## Komplexe Zahlen – fortgeführt

```
instance Num Complex where
 Complex r1 i1 + Complex r2 i2 = Complex (r1 + r2) (i1 + i2)
 Complex r1 i1 * Complex r2 i2 =
 Complex (r1 * r2 - i1 * i2) (r1 * i2 + r2 * i1)
 fromInteger x = Complex (fromInteger x) 0
 negate (Complex r i) = Complex (negate r) (negate i)
 abs c = Complex (absComplex c) 0
 signum c@(Complex r i)
 | c == 0 = 0
 | otherwise = Complex (r / a) (i / a)
 where a = absComplex c
```

```
-- auxiliary functions must be defined outside
```

```
-- the class instantiation
```

```
absComplex (Complex r i) = sqrt (r2 + i2)
```

## Beispiel: Polymorphe komplexe Zahlen

```
data Complex a = Complex a a -- polymorphic: type a instead of Double
```

```
instance Eq a => Eq (Complex a) where
 Complex r1 i1 == Complex r2 i2 = r1 == r2 && i1 == i2
-- comparing r1 and r2 (i1 and i2) requires equality on type a
```

```
-- for Show not only Show a is required, but also Ord a and Num a
```

```
instance (Show a, Ord a, Num a) => Show (Complex a) where
```

```
 show (Complex r i)
 | i == 0 = show r
 | r == 0 = show i ++ "i"
 | i < 0 = show r ++ show i ++ "i"
 | otherwise = show r ++ "+" ++ show i ++ "i"
```

```
instance (Floating a, Eq a) => Num (Complex a) where ...
```

```
-- sqrt :: Floating a => a -> a, Floating a implies Num a
```

## Beispiel: Polymorphe komplexe Zahlen

```
> (Complex 0 1)^2
-1.0

> 2 + 5 :: Complex Float
7.0

> abs (Complex 1 3) :: Complex Float
3.1622777
> abs (Complex 1 3) :: Complex Double
3.1622776601683795

> 2 * Complex 7 2.5
14.0+5.0i

> 2.4 * Complex 7 2.5
error: No instance for (Fractional (Complex Double))
```

## Einschränkungen bei der Instantiierung von Typklassen

- Form der Instantiierung: `instance ... => TClass (TConstr a1 .. aN)`
- die Typvariablen können **nicht** durch konkrete Typen ersetzt werden
  - Beispiel: folgende Instantiierung ist nicht erlaubt

```
-- show Boolean lists as bit-strings: "011" vs. "[False,True,True]"
instance Show [Bool] where
 show (b : bs) = (if b then '1' else '0') : show bs
 show [] = ""
```
  - Lösung mittels separater Funktion: `showBits :: [Bool] -> String`
- jede Kombination aus Typklassen und Typ kann **maximal 1 Instanz** haben
  - Beispiel: folgende Instantiierung ist nicht erlaubt

```
-- case-insensitive comparison of characters (for sorting)
import Data.Char
instance Ord Char where -- clashes with existing Ord Char instance
 c <= d = ord (toUpper c) <= ord (toUpper d)
```
  - Lösung: parametrisiere Sortier-Algorithmus, ... mit Ordnung, anstelle von (`<=`) zu nutzen

## Zusammenfassung

- viele Typklassen sind in Prelude vordefiniert
- Hierarchie von Typklassen für Zahlen
- es ist möglich, eigene Typklassen zu erstellen
  - Liste von Funktionsnamen mit Typen
  - Beschreibung, was diese Funktionen tun sollen
  - optional: Standard-Implementierung für manche der Funktionen
- Typklassen können von Typen instantiiert werden, entweder manuell oder mittels `deriving (Show,Eq,Ord)`
- jede Kombination aus Typklassen und Typ darf **maximal eine Instanz** haben
- Konvertierung zwischen Operatoren und Funktionen: `(+)` `(25 `div` 3)` `2`