Kapitel 2

Haskell und Objektorientierung

2.1 Typklassen und Überladung in Haskell

Typklassen wurden entwickelt, um Überladung von Funktionen im polymorphen Typsystem systematisch zu benutzen und auch benutzerdefinierbar zu machen. (Überladung: Es gibt mehrere Funktionen mit gleichem Namen). Dazu muss spätestens zum Zeitpunkt der Ausführung bekannt sein, welche Funktion denn tatsächlich ausgeführt werden soll. Diese Problem ist praxisrelevant, denn es ist nicht einzusehen, daß man z.B. 2+5 schreiben darf, aber bei komplexen Zahlen statt $(1+i)+(3+2i)$ in etwa schreiben müßte: $(1+i) complexadd (3+2i)$. Als weiteren Nachteil des Überladungsverbots vermehren sich die Namen der allgemeinen Funktionen, z.B. Summe einer Liste müßte für Zahlen und komplexe Zahlen extra definiert werden.

Ein spezifisches Problem der Typklassen in Milner-getypten Sprachen wie Haskell ist die korrekte Kombination der normalen Typisierung mit Tyklassen.

- $+, *, -, /$ will man für verschiedene Arten von Zahlen verwenden: integer, Fließkomma, rationale, ...
- Es gibt Funktionalitäten, die nicht auf allen Typen definierbar sind, sondern nur auf einer Menge von vorgegebenen Typen: z.B. Gleichheit `==` oder die Funktion `show`, die Objekte in eine druckbare Form überführt.

2.1.1 Typklasse

Eine Typklasse kann man sich vorstellen als eine Menge von Typen zusammen mit zugehörigen Funktionen bzw. Methoden. Die Typklassen werden als eigene syntaktische Einheiten (haben einen Namen) eingeführt.

Typklassendefinition

Eine neue Typklasse in Haskell kann definiert werden durch:

class ⟨Vorbedingung⟩ => NEWCLASS(a)

⟨Vorbedingung⟩ : Klassenbedingungen an die Variable a
NEWCLASS: neuer Klassenname. Im allgemeinen gibt es nur eine Variable a. Die Bedingung in Haskell ist nur eine Unterklassenbeziehung von Typklassen:

```
class (Eq a, Show a, Eval a) => Num a
```

Bedeutet: Nur für Typen a, die bereits in den Klassen `Eq`, `Show`, `Eval` sind, wird a auch in `Num` definiert: d.h. `Num` ist Unterklasse von `EQ`, `Show`, `Eval`. Leider ist dies von der logischen Schreibweise her falsch geraten.

Notwendig ist eine extra Syntax zur Deklaration von Typklassen und zur Deklaration von Typen von Funktionen.

Beispiel 1. Der Typ von `==` ist

```
EQ(a) =>  a -> a -> Bool
```

Das bedeutet, daß für alle `a` die in `Eq` sind, die Funktion den Typ `a -> a -> Bool` hat.

Bei der aktuellen Verwendung von `==` in einem Ausdruck wird der Typechecker dann prüfen, ob die getesteten Objekte tatsächlich in der Klasse Eq sind.

In `Eq` sind alle Typen, bei denen ein Gleichheitstest implementiert werden kann.

Nicht sinnvoll ist der Vergleich von Funktionen mit `==`, obwohl man das auch definieren kann.

Beispiel 2. Definition von `Eq`:

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

Vergleicht man das mit der üblichen objektorientierten Programmierung (z.B. Java), so entspricht `Eq` einer Klasse, die Methoden sind `==` und `/=`. Für die Methode `==` wird nur die Schnittstelle zur Verfügung gestellt, während `/=` standardmäßig bereits aufbauend auf `==` definiert wird.

Umgekehrt ist es nicht sinnvoll, alle Klassen einer objektorientierten Sprache auch als Typklassen in Haskell zu implementieren. Oft ist die richtige Übersetzung ein Typ: z.B. Listen.

Danach kann man noch selbst-definierte Typen zu Instanzen der Typklasse `Eq` erklären. Oder auch rekursive Regeln zur Erzeugung von Instanztypen angeben. Dies bedeutet: == (ist gleich) muß bei jeder solchen Instanzdefinition mit definiert werden, `/=` (ungleich) ist dann automatisch ebenfalls definiert. Allerdings gibt es auch einen Default, der die Strukturgleichheit implementiert, wobei rekursiv wieder == verwendet wird.

Beispiel 3.
```
instance Eq char      where
x == y    =   ord c == ord d
```

Dies erklärt `char` zu einem Typ der Typklasse `Eq`, auf den == angewendet werden kann. Gleichheit auf Zahlen wird im Haskell-Prelude mittels eingebauter Funktionen implementiert.

Beispiel 4. `instance Eq  Int     where (==)    = primEqInt`
```
instance Eq  Integer where (==)    = primEqInteger
instance Ord Int     where compare = primCmpInt
instance Ord Integer where compare = primCmpInteger
```

Bei (selbstdefinierten) algebraischen Datentypen, die auch noch rekursiv definiert sind (hier Beispiel Listen) ist das etwas komplizierter:

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

- Dies ist eine rekursive Definition einer unendlichen Menge von `Eq`-Typen
- Die Gleichheit ist explizit für alle Konstruktor-möglichkeiten definiert
- In Haskell ist dies auch möglich, allerdings wird dort eine default Definition benutzt, bei der direkt an der Typdeklaration steht: `deriving (Eq,Ord)`. Diese Möglichkeit gibt es auch in Hugs, aber nicht in Gofer.

2.1.2 Die Klasse Ord

Das ist die Klasse der Typen mit Ordnungsrelation $<$.

Beispiel 5. `data Ordering = LT | EQ | GT`
```
deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

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      = x
```

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

Die primitive Operation ist offensichtlich $<=$, die anderen sind darauf aufgebaut.

2.1.3 Typklassen Read und Show

Die Klasse `Show` enthält alle Typen, deren Objekte eine Repräsentation als Text haben. Die Klasse `Read` enthält die Typen, deren Objekte aus einer Textrepräsentation zu rekonstruieren sind. Wenn ein Typ sowohl zu Show als auch Read gehört, dann sollte die Textrepräsentation wieder eindeutig zu parsen sein und das Objekt ergeben, d.h. $Read(Show(x)) = x$.

Funktion (Methode), die auf solchen Typen (α) immer vorhanden ist:

`showS`: $\alpha \rightarrow$ string
`readS`: string $\rightarrow \alpha$

Diese ist für einige eingebaute Typen bereits vordefiniert und muß für andere entsprechend definiert werden.

2.1.4 Mehrfache Vererbung

Es gibt auch etwas analoges zur multiplen Vererbung:

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

    x - y            = x + negate y
    fromInt          = fromIntegral

instance Num Int where
    (+)            = primPlusInt
    (-)            = primMinusInt
    negate         = primNegInt
    (*)            = primMulInt
    abs            = absReal
    signum         = signumReal
    fromInteger    = primIntegerToInt
    fromInt x      = x
```

2.1.5 Auflösung der Überladung

Wird an einer Stelle in einer Funktion eine überladene Funktion (Methode) verwendet, dann muß das Typsystem während der Compilezeit dafür sorgen, daß die Überladung aufgelöst wird, bzw. zur Laufzeit aufgelöst werden kann. Es wird durch den Typechecker sichergestellt, daß eine Anwendung von Methoden auf primitive Objekte mit der richtigen primitiven Methode erfolgt.

!!! Im Haskell /Gofer -Laufzeitsystem gibt es keine Typinformationen mehr. D.h. man kann zur Laufzeit Listen-objekte von Bäumen nicht unterscheiden (mittels tags). Insbesondere ist eine Java-Methode wie `instanceof` in Haskell nicht möglich.

Zum Zeitpunkt des Typchecks hat das manchmal merkwürdige Konsequenzen: z.B. läßt sich [] = [] nicht typen, da ein [] von Typ $list(a)$, das andere von Typ $list(b)$ ist, und für a und b der (Typklassen-)Kontext nicht bestimmt werden kann.

Im Vergleich dazu existieren in Java Laufzeitsystem noch die Typinformationen und können auch in Abfragen verwendet werden.

Die interne Überladungsauflösung funktioniert so, daß überladene Funktionen intern einen zusätzlichen Parameter, den sogenannten `Dictionary`-Parameter bekommen. Zum Aufrufzeitpunkt einer Funktion muss dieser Parameter dann so bekannt sein, daß die zugehörige Implementierung im dictionary gefunden werden kann. Außerdem wird intern ein"Dictionary" aufgebaut, das zur Laufzeit als normale Haskell-Datenstruktur vorhanden ist, und die Funktionen mit den richtigen"Methoden"versorgt. Wenn der Dictionary-Parameter einem primitiven Typ wie `Int` entspricht, dann wird die zugehörige Gleichheit benutzt. D.h. die rekursive Aufrufhierarchie bzgl. der Typen als Aufrufhierarchie des dictionary von == abgebildet.

Eine Konsequenz dieses Verfahrens ist, daß in Haskell zur Compilezeit genügend viel Typinformation (auch für Unterausdrücke) zur Verfügung stehen muß, um überladene Funktionen mit den richtigen Dictionary-Parametern zu versehen.

Da z.B. die Funktion == überladen ist, und nur für Argumente der Typklasse `Eq` zur Verfügung steht, muß für ein Gleichung $s == t$ für die Ausdrücke s und t entweder ein Grundtyp berechnet werden, oder, wenn Typvariablen vorkommen, muß aus dem übergeordneten Ausdruck hervorgehen, welches Dictionary zu benutzen ist. Dieser Dictionary-Parameter ist i.a. ein (implizites) Argument einer übergeordneten Funktion.

Ein Dictionary kann auch rekursiv aufgebaut sein, wobei jedoch die Bedingung der Endlichkeit eingehalten werden muß.

2.2 Verwendung der Definition der Gleichheit

Beispiel 6. **Mengen**

```
data Menge a = Set [a]

Instance Eq a => Eq (Menge a)  where
```

```
Set xs == Set ys  = subset xs ys && subset ys xs

subset: Eq a => a -> a -> a
subset xs ys = all  ('elem' ys)  xs

instance (Ord a) => Ord (Menge a)  where
Set xs <= Set ys = subset xs ys
```

Damit kann man erreichen, daß verschiedene algebraische Datentypen auf ihre eigene und korrekte Weise auf Gleichheit verglichen werden, auch wenn diese verschachtelt sind, z.B. Mengen von Mengen, Mengen von Listen, oder Listen von Mengen.

2.3 Aufzählungsklasse `Enum`

Damit kann man die Datentypen erfassen, deren Elemente sich (vorwärts und rückwärts) aufzählen lassen, wie nichtnegative ganze Zahlen; die Zeichen eines Alphabets, oder Farben, falls man diese in eine Ordnung bringt. Für Elemente dieser Typen kann man das Aufzählungsverfahren haben mit der Syntax analog zu Zahlen verwenden: $[2..], [a..z], \ldots$.

Als Beispiel aus dem Hugs-Prelude die Definition der Klasse.

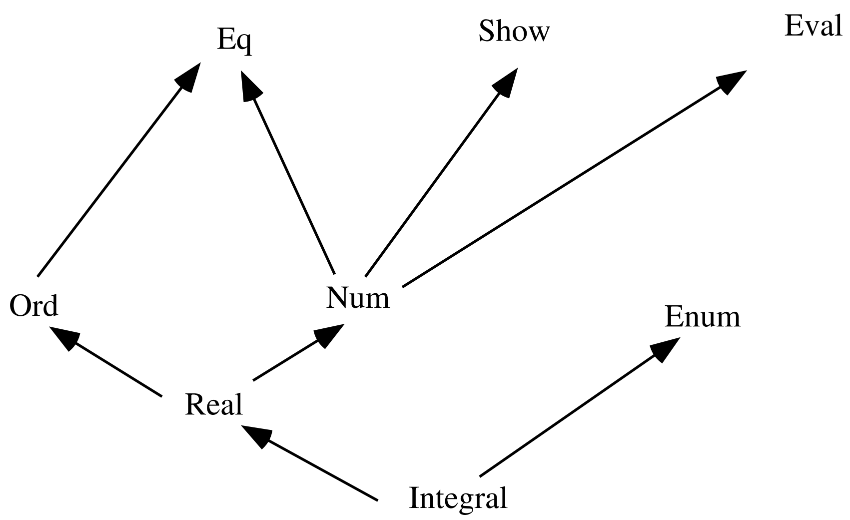
```
class Enum a where
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
    enumFrom            :: a -> [a]             -- [n..]
    enumFromThen        :: a -> a -> [a]        -- [n,m..]
    enumFromTo          :: a -> a -> [a]        -- [n..m]
    enumFromThenTo      :: a ->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

    enumFromTo x y       = map toEnum [ fromEnum x .. fromEnum y ]
    enumFromThenTo x y z  = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]

succ, pred :: Enum a => a -> a
succ        = toEnum . (1+)       . fromEnum
pred        = toEnum . subtract 1 . fromEnum

instance Enum Char where
    toEnum           = primIntToChar
    fromEnum         = primCharToInt
    enumFrom c       = map toEnum [fromEnum c .. fromEnum (maxBound::Char)]
    enumFromThen c d =
              map toEnum [fromEnum c, fromEnum d .. fromEnum (lastChar::Char)]
       where lastChar = if d < c then minBound else maxBound
```

Typklassenhierarchie (Ausschnitt aus den vordefinierten Typklassen)



Beispiel 7. Definition einer überladenen Konstanten.

```
class Finite a  where members :: [a]

instance Finite Bool where members = [True, False]

instance (Finite a, Finite b) => Finite (a, b)  where
members = [(x,y) | x <- members, y <- members]
```

Dies definiert eine Konstante in einer Typklasse, die abhängig von Ihrem Typkontext ihren Wert ändert. Weitere Typen können mittels `Bool` und Paarbildung zusammengesetzt werden.

Damit erhält man z.B.

```
members::Bool  -> [ True, False]
length (members :: [((Bool, Bool), (Bool, Bool)]])  --->  16
```

Beispiel 8. Allgemeine Summe über alle Listen von Typen, die zur Typklasse `Num` gehören.

Der folgende Versuch scheitert:

```
summe [] =  0
summe (x:xs) = x+summe xs
```

Der Grund ist, daß die Zahl 0 den festen Typ Int hat.

Die allgemeine Summe über eine Liste erhält man folgendermaßen (noch effizienter mit Akkumulator):

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

Dies löst das "Problem" der überladenen Konstanten.

Ein Problem mit Verallgemeinerungen ist: Läßt man mehr Variablen in der Bedingung zu, dann ist unklar, ob es einen entscheidbaren Typcheckalgorithmus für Typen und Typklassen gibt.

Möglicherweise ist ein Typsystem einfacher zu handhaben, das abhängige Typen (dependent types) verwendet. Dies bedeutet: ein sehr ausdrucksstarkes allerdings unentscheidbares Typsystem.

2.3.1 Klasse Functor

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

data Maybe a = Just a | Nothing
       deriving (Eq, Ord, Read, Show)

instance Functor Maybe where
    map f Nothing  = Nothing
    map f (Just x) = Just (f x)


instance Functor [] where
    map f []     = []
    map f (x:xs) = f x : map f xs
```

Sinnvoll ist es, auch die Datenstrukturen, für die man ein `map` definieren kann, zu Instanzen von "Functor" zu machen.

```
data Baum a = Blatt a | Knoten (Baum a) (Baum a)
      deriving (Eq, Show)

instance Functor Baum where
    map = baummap


baummap f (Blatt a) = Blatt (f a)
baummap f (Knoten x y) = Knoten (baummap f x) (baummap f y)

--   map (\x-> x) (Blatt 1)
```