Programmierung und Modellierung mit Haskell

Polymorphie, Typklassen & Module

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

7. Mai 2015

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where   rev_aux :: [Char] -> [Char] -> [Char]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das “vorläufige” Ergebnis:

```
reverse “live“
↪ rev_aux 'l':'i':'v':'e':[]          []
↪ rev_aux     'i':'v':'e':[]        'l':[]
↪ rev_aux         'v':'e':[]    'i':'l':[]
↪ rev_aux             'e':[]'v':'i':'l':[]
↪ rev_aux                 []'e':'v':'i':'l':[]
↪
```

Wo ist wichtig, dass die Listenelemente Typ `Char` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where   rev_aux :: [Char] -> [Char] -> [Char]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das "vorläufige" Ergebnis:

```
reverse "live"
↝ rev_aux 'l':'i':'v':'e':[]                    []
↝ rev_aux     'i':'v':'e':[]                'l':[]
↝ rev_aux         'v':'e':[]            'i':'l':[]
↝ rev_aux             'e':[]        'v':'i':'l':[]
↝ rev_aux                 []    'e':'v':'i':'l':[]
↝
```

Wo ist wichtig, dass die Listenelemente Typ `Char` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where   rev_aux :: [Char] -> [Char] -> [Char]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das “vorläufige” Ergebnis:

```
reverse “live“
↝ rev_aux 'l':'i':'v':'e':[]                  []
↝ rev_aux     'i':'v':'e':[]                'l':[]
↝ rev_aux         'v':'e':[]            'i':'l':[]
↝ rev_aux             'e':[]        'v':'i':'l':[]
↝ rev_aux                 []    'e':'v':'i':'l':[]
↝
```

Wo ist wichtig, dass die Listenelemente Typ `Char` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where   rev_aux :: [Char] -> [Char] -> [Char]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das "vorläufige" Ergebnis:

```
reverse "live"
↝ rev_aux 'l':'i':'v':'e':[]                  []
↝ rev_aux     'i':'v':'e':[]                'l':[]
↝ rev_aux         'v':'e':[]            'i':'l':[]
↝ rev_aux             'e':[]        'v':'i':'l':[]
↝ rev_aux                 []    'e':'v':'i':'l':[]
↝
```

Wo ist wichtig, dass die Listenelemente Typ `Char` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where   rev_aux :: [Char] -> [Char] -> [Char]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das "vorläufige" Ergebnis:

```
reverse "live"
↝ rev_aux 'l':'i':'v':'e':[]                  []
↝ rev_aux     'i':'v':'e':[]                'l':[]
↝ rev_aux         'v':'e':[]            'i':'l':[]
↝ rev_aux             'e':[]        'v':'i':'l':[]
↝ rev_aux                 []    'e':'v':'i':'l':[]
↝
```

Wo ist wichtig, dass die Listenelemente Typ `Char` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where   rev_aux :: [Char] -> [Char] -> [Char]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das "vorläufige" Ergebnis:

```
reverse "live"
↝ rev_aux 'l':'i':'v':'e':[]                  []
↝ rev_aux     'i':'v':'e':[]                'l':[]
↝ rev_aux         'v':'e':[]            'i':'l':[]
↝ rev_aux             'e':[]        'v':'i':'l':[]
↝ rev_aux                 []    'e':'v':'i':'l':[]
↝
```

Wo ist wichtig, dass die Listenelemente Typ `Char` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Char] -> [Char]
reverse xs = rev_aux xs []
  where   rev_aux :: [Char] -> [Char] -> [Char]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das “vorläufige” Ergebnis:

```
reverse “live“
↝ rev_aux 'l':'i':'v':'e':[]                 []
↝ rev_aux     'i':'v':'e':[]               'l':[]
↝ rev_aux         'v':'e':[]           'i':'l':[]
↝ rev_aux             'e':[]       'v':'i':'l':[]
↝ rev_aux                 []   'e':'v':'i':'l':[]
↝ “evil“
```

Wo ist wichtig, dass die Listenelemente Typ `Char` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [Int] -> [Int]
reverse xs = rev_aux xs []
  where   rev_aux :: [Int] -> [Int] -> [Int]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das "vorläufige" Ergebnis:

```
reverse [1,2,3,4]
⇝ rev_aux  1 : 2 : 3 : 4 :[]                []
⇝ rev_aux      2 : 3 : 4 :[]              1 :[]
⇝ rev_aux          3 : 4 :[]          2 : 1 :[]
⇝ rev_aux              4 :[]      3 : 2 : 1 :[]
⇝ rev_aux                  []  4 : 3 : 2 : 1 :[]
⇝ [4,3,2,1]
```

Wo ist wichtig, dass die Listenelemente Typ `Int` haben?

Beispiel: List-Reversal

Umdrehen einer Liste mithilfe eines Akkumulatoren:

```
reverse :: [ a ] -> [ a ]
reverse xs = rev_aux xs []
  where   rev_aux :: [ a ] -> [ a ] -> [ a ]
          rev_aux    [] acc = acc
          rev_aux (h:t) acc = rev_aux t (h:acc)
```

Der Akkumulator merkt sich das "vorläufige" Ergebnis:

```
reverse [1,2,3,4]
⇝ rev_aux  1 : 2 : 3 : 4 :[]                []
⇝ rev_aux      2 : 3 : 4 :[]              1 :[]
⇝ rev_aux          3 : 4 :[]          2 : 1 :[]
⇝ rev_aux              4 :[]      3 : 2 : 1 :[]
⇝ rev_aux                 []  4 : 3 : 2 : 1 :[]
⇝ [4,3,2,1]
```

Wo ist wichtig, dass die Listenelemente Typ `Int` haben?

Parametrischer Polymorphismus

Das **statische Typsystem** von Haskell prüft während dem Kompilieren alle Typen:

- Keine Typfehler und keine Typprüfung zur Laufzeit
- Kompiler kann besser optimieren

Dennoch können wir generischen Code schreiben:

```
rev_aux :: [a] -> [a] -> [a]
rev_aux   []  acc = acc
rev_aux (h:t) acc = rev_aux t (h:acc)
```

Wird von Haskell's Typsystem geprüft und für sicher befunden, da der **universell quantifizierte** Typ `a` keine Rolle spielt:

- Werte von Typ `a` werden herumgereicht, aber nie inspiziert
- Code kann unabhängig von Typ `a` ausgeführt werden

`reverse` hat einen **Typparameter** und ist damit **polymorph**

Polymorphe Funktionen

Parametrisch polymorphe Funktionen aus der Standardbibliothek:

```
id :: a -> a
id x = x

fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y

replicate :: Int -> a -> [a]

drop :: Int -> [a]-> [a]

zip :: [a] -> [b] -> [(a,b)]
```

Instantiierung

Wir dürfen für **jede** Typvariable **einen** beliebigen Typ einsetzen,
d.h. wir können `reverse :: [a] -> [a]` instantiieren mit

```
[Int]  -> [Int]
[Bool] -> [Bool]
String -> String
[[([(Int,Bool)],String)]] -> [[([(Int,Bool)],String)]]
```

Solche Typen bezeichnet man auch als **Monotypen**

Gegenbeispiel

Instantiierung beispielsweise zu `[Int] -> [Bool]` ist nicht erlaubt,
denn dazu wäre der polymorphe Typ `[c] -> [d]` notwendig!

Hinweis

Die äußere Liste hat hier keine Sonderrolle:
der Typ `Maybe a -> a` instantiiert genauso zu `Maybe Int -> Int`
oder `Maybe (Int,[Bool]) -> (Int,[Bool])`

Mehrere Typparameter

```
fst :: (a,b) -> a          snd :: (a,b) -> b
fst    (x,_) =  x          snd    (_,y) =  y
```

Das Typsystem unterscheidet verschiedene Typparameter:

`a -> b -> (a,b)`	`b -> a -> (b,a)`	Identischer Typ
`a -> b -> (a,b)`	`a -> b -> (b,a)`	Unterschiedlich
`a -> b -> (a,b)`	`a -> a -> (a,a)`	Unterschiedlich

- Namen der Typparameter sind unerheblich; wichtig ist aber, ob zwei Typparameter den gleichen Namen tragen!
- Verschiedene Typparameter dürfen gleich instantiiert werden; Werte des Typs `a -> a -> (a,a)` dürfen auch dort eingesetzt werden wo `a -> b -> (a,b)` erwartet wird, aber nicht umgekehrt!

  Beispiel

  Die Funktion `fst` kann auch auf ein Paar `(True,True) :: (Bool,Bool)` angewendet werden.

Eingeschränkte Polymorphie

Manchmal müssen wir die Polymorphie jedoch einschränken:
Z.B. sind Sortierverfahren generisch, aber wie vergleichen wir zwei Elemente, wenn wir deren Typ nicht kennen?
Vergleich zweier Werte des Typs `Char` funktioniert z.B. ganz anders als etwa bei zwei Werten des Typs `[Maybe Double]`

Sortierende Funktion mit Typ `[a] -> [a]` ist also nicht möglich.

Lösungen

1. **Typen höherer Ordnung**
   `sortBy :: (a -> a -> Bool) -> [a] -> [a]`
   Vergleichsoperation als Argument übergeben
   $\Rightarrow$ spätere Vorlesung "Typen höherer Ordnung"
2. **Typklassen** `sort :: Ord a => [a] -> [a]`
   Gleiche Idee, jedoch versteckt: Funktion bekommt zur Laufzeit ein **Wörterbuch** mit (mehreren) Funktion implizit übergeben.

Eingeschränkte Polymorphie

Manchmal müssen wir die Polymorphie jedoch einschränken:
Z.B. sind Sortierverfahren generisch, aber wie vergleichen wir zwei Elemente, wenn wir deren Typ nicht kennen?
Vergleich zweier Werte des Typs `Char` funktioniert z.B. ganz anders als etwa bei zwei Werten des Typs `[Maybe Double]`

Sortierende Funktion mit Typ `[a] -> [a]` ist also nicht möglich.

Lösungen

1. **Typen höherer Ordnung**
   `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`
   Vergleichsoperation als Argument übergeben
   $\Rightarrow$ spätere Vorlesung "Typen höherer Ordnung"
2. **Typklassen** `sort :: Ord a => [a] -> [a]`
   Gleiche Idee, jedoch versteckt: Funktion bekommt zur Laufzeit ein **Wörterbuch** mit (mehreren) Funktion implizit übergeben.

Eingeschränkte Polymorphie

Manchmal müssen wir die Polymorphie jedoch einschränken:
Z.B. sind Sortierverfahren generisch, aber wie vergleichen wir zwei Elemente, wenn wir deren Typ nicht kennen?
Vergleich zweier Werte des Typs `Char` funktioniert z.B. ganz anders als etwa bei zwei Werten des Typs `[Maybe Double]`

Sortierende Funktion mit Typ `[a] -> [a]` ist also nicht möglich.

Lösungen

1. **Typen höherer Ordnung**
   `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`
   Vergleichsoperation als Argument übergeben
   $\Rightarrow$ spätere Vorlesung "Typen höherer Ordnung"
2. **Typklassen** `sort :: Ord a => [a] -> [a]`
   Gleiche Idee, jedoch versteckt: Funktion bekommt zur Laufzeit ein **Wörterbuch** mit (mehreren) Funktion implizit übergeben.

Ad-hoc-Polymorphismus

Wie kann man eine Vergleichsfunktion für alle Typen implementieren?

```
(==) :: a -> a -> Bool
x == y  =  ???
```

↯ Geht nicht, da wir je nach Typ `a` anderen Code brauchen!
Z.B. Vergleich von zwei Strings komplizierter als von zwei Zeichen.

In vielen Sprachen sind **überladene** Operationen eingebaut:
Eine Typprüfung zur Laufzeit entscheidet über den zu verwendenden Code ⇒ **Ad-hoc-Polymorphismus.**

Probleme:

- Typprüfung benötigt Typinformation zur Laufzeit
- eingebaute Operation können dann oft nur schwerlich mit benutzerdefinierten Datentypen umgehen

Polymorphe Gleicheit

Welchen Typ hat `(==)` in Haskell?

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

Lies "für alle Typen `a` aus der Typklasse `Eq`", also "für alle Typen `a`, welche wir vergleichen können"

Polymorphe Gleicheit

Welchen Typ hat `(==)` in Haskell?

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

Lies "für alle Typen `a` aus der Typklasse `Eq`", also
"für alle Typen `a`, welche wir vergleichen können"

`Eq a =>` ist **Typklasseneinschränkung** (engl. **class constraint**)

- Polymorphismus der Vergleichsfunktion `(==)` ist eingeschränkt
- Zur Laufzeit wird ein Wörterbuch (Dictionary) mit dem Code der passenden Funktionen *implizit* als weiteres Argument übergeben; keine Typprüfung zur Laufzeit

Verwenden wir `(==)` in einer Funktionsdefinition, so muss auch deren Typsignatur entsprechend eingeschränkt werden.

Beispiel

Verwendung der polymorphen Vergleichsfunktion in einer Funktionsdefinition, welche den ersten abweichenden Listenwert zweier Listen berechnet:

```
data Maybe a = Nothing | Just a -- zur Erinnerung
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing

> firstDiff [1..10] [1,2,3,99,5,6]
Just 99
```

Beispiel

Verwendung der polymorphen Vergleichsfunktion in einer Funktionsdefinition, welche den ersten abweichenden Listenwert zweier Listen berechnet:

```
data Maybe a = Nothing | Just a -- zur Erinnerung
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing

> firstDiff [1..10] [1,2,3,99,5,6]
Just 99
```

Die Typvariablen in der Typsignatur der Funktion `firstDiff` sind eingeschränkt quantifiziert:

```
> :type firstDiff
firstDiff :: Eq a => [a] -> [a] -> Maybe a
```

Beispiel

```
firstDiff :: Eq a => [a] -> [a] -> Maybe a
firstDiff (h1:t1) (h2:t2)
  | h1 == h2  = firstDiff t1 t2
  | otherwise = Just h2
firstDiff _ _ = Nothing
```

Beim Kompilieren fügt der Kompiler der Funktion ein implizites Argument pro class constraint hinzu. Dieses Argument ist ein Paket mit alle Funktionen der Typklasse `Eq` für den entsprechenden Typ, welches zur Laufzeit übergeben wird.
In diesem Fall also der passende Code für `(==)` und `(/=)`,

Wir brauchen uns darum nicht zu kümmern!

Syntax Class Constraints

In Typsignaturen kann der Bereich eingeschränkt werden, über den eine Typvariable quantifiziert ist:

```
firstDiff :: Eq a => [a] -> [a] -> Maybe a
```

Die polymorphe Funktion `firstDiff` kann nur auf Werten von Typen angewendet werden, welche zur Typklasse `Eq` gehören.

Es ist auch möglich, mehrere Einschränkungen anzugeben:

```
foo :: (Eq a, Read a) => [String] -> [a]
```

. . . und/oder mehrere Typvariablen einzuschränken:

```
bar :: (Eq a, Eq b, Read b, Ord c) => [(a,b)] -> [c]
```

Solche Typsignaturen kann GHC für uns automatisch ableiten, aber wir können diese auch explizit hinschreiben und erzwingen.

Ausblick: Polymorphismus

Es gibt noch weitere Arten von Polymorphismus, welche wir in dieser Vorlesung nicht behandeln.

GHC kennt Erweiterungen für manche, z.B.:

- Rank-N Types
- Impredicative Types

aber nicht für alle, z.B.:

- Subtyping

Die Verwendung solcher Erweiterungen benötigt oft die explizite Angabe von Typsignaturen, da die Inferenz solcher Typen im allgemeinen schnell unentscheidbar wird.

Auch für **Polymorphe Rekursion**, d.h. der Typ des rekursiven Aufrufs ändert sich, benötigt GHC eine explizite Typsignatur.

Übersicht Polymorphismus

Parametrischer Polymorphismus

- Unabhängig vom tatsächlichen Typ wird immer der gleiche generische Code ausgeführt.
- Realisiert in Haskell über Typparameter: `a -> a`

Ad-hoc-Polymorphismus

- Funktionsnamen werden überladen, d.h. abhängig vom tatsächlichen Typ wird spezifischer Code ausgeführt
- Realisiert in Haskell durch Einschränkung der Typparameter auf Typklassen: `Klasse a => a -> a`

Subtyp Polymorphismus

- Mischform: Es kann der gleiche Code oder spezifischer Code ausgeführt werden.
- Irrelevant für uns, da GHC hat keinen direkten Subtyp-Mechanismus hat (wie z.B. Vererbung in Java).

Typklassen

Eine **Typklasse** ist eine Menge von Typen, plus eine Menge von Funktionen, welche auf alle Typen dieser Klasse anwendbar sind.

Beispiel einer Typklassen-Definition:

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

Auf jeden Typ `a`, der eine **Instanz** der Klasse `Eq` ist, können wir die beiden 2-stelligen Funktionen `(==)` und `(/=)` anwenden.

Die Standardbibliothek definiert diese Klasse `Eq` und zahlreiche Instanzen dazu:

```
Eq Bool      Eq Double
Eq Char      Eq Float   ...
Eq String    Eq Int
```

Eq Typklasse

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

- Klassennamen schreiben wir wie Typen immer groß
- Klassendefinition kann Default-Implementierung enthalten, diese können überschrieben werden
- `(==)` nur auf zwei Werte des gleichen Typs anwendbar.
  Der Ausdruck `"Ugh" == [1..5]` ist nicht Typ-korrekt, obwohl `String` und `[Int]` beide in der Klasse `Eq` sind.

  Signatur `foo :: (Kl a, Kl b) => a -> b -> Bool` würde einen Ausdruck `"Ugh" `foo` [1..5]` erlauben, falls `String` und `[Int]` Instanzen der Klasse `Kl` wären.

⇒ Klassendefinition grob vergleichbar mit Interfaces in Java

Instanzen definieren

Man kann leicht neue Instanzen für eigene Datentypen definieren:

```
data Frucht = Apfel (Int,Int) | Birne Int Int

instance Eq Frucht where
  Apfel x1    == Apfel x2    = x1 == x2
  Birne x1 y1 == Birne x2 y2 = x1 == x2 && y1 == y2
  _           == _           = False
```

Damit können wir nun Äpfel und Birnen vergleichen:

```
> Apfel (3,4) == Apfel (3,4)
True
> Apfel (3,4) == Apfel (3,5)
False
> Apfel (3,4) /= Birne 3 4
True
```

Definition für /= verwendet Klassen Default-Implementierung, da wir diese nicht neu definiert haben.

Instanzen ableiten

Man kann leicht neue Instanzen für eigene Datentypen definieren:

```
data Frucht = Apfel (Int,Int) | Birne Int Int

instance Eq Frucht where
  Apfel x1    == Apfel x2    = x1 == x2
  Birne x1 y1 == Birne x2 y2 = x1 == x2 && y1 == y2
  _           == _           = False
```

Diese Instanz Deklaration ist langweilig und ohne Überraschungen:

Bei gleichen Konstruktoren vergleichen wir einfach nur die Argumente der Konstruktoren.

GHC kann solche Instanzen mit `deriving` automatisch generieren:

```
data Frucht = Apfel (Int,Int) | Birne Int Int
  deriving (Eq, Ord, Show, Read)
```

Typklassen Instanzen

- Instanzdeklarationen müssen mind. alle Funktionen der Klasse implementieren, die keine Default-Implementierung haben
- Ein Typ kann mehreren Typklassen auf einmal angehören
- Aber nur *eine* Instanz pro Klasse definierbar

Beispiel: Wenn wir `Frucht` *manchmal* nur nach Preis vergleichen wollen, dann brauchen wir einen neuen Typ dafür:

```
data PreisFrucht = PF Frucht
instance Eq PreisFrucht where
  PF(Apfel (_,p1)) == PF(Apfel (_,p2)) = p1 == p2
  PF(Apfel (_,p1)) == PF(Birne p2 _  ) = p1 == p2
  PF(Birne p1 _  ) == PF(Apfel (_,p2)) = p1 == p2
  PF(Birne p1 _  ) == PF(Birne p2 _  ) = p1 == p2
```

. . . oder wir schreiben ohne Typklassen nur eine Funktion

```
preisvergleich :: Frucht -> Frucht -> Bool
```

Show Typklasse

Die Typklasse `Show` erlaubt die Umwandlung eines Wertes in ein String, um den Wert zum Beispiel am Bildschirm anzuzeigen:

```
class  Show a  where
  show       :: a    -> String
  showList   :: [a] -> ShowS
  ...
```

Beispiel:

```
> show (Apfel (3,4))
"Apfel (3,4)"
> show (Birne 5 6)
"Birne 5 6"
```

Die Standardbibliothek definiert viele Instanzen dazu:

```
Show Bool       Show Double       Show Int
Show Char       Show Float        ...
```

Show Typklasse

Wenn uns die automatisch abgeleitete `show` Funktion nicht gefällt, dann entfernen wir `Show` aus der `deriving` Klausel der Datentypdeklaration für `Frucht`, und definieren die Instanz selbst:

```
instance Show Frucht where
  show (Apfel (z,p)) = "Apfel("++(show z)++")"
  show (Birne p z)   = "Birne("++(show z)++")"
```

Damit erhalten wir dann entsprechend:

```
> show (Apfel (3,4))
"Apfel(3)"
> show (Birne 5 6)
"Birne(6)"
```

Read Typklasse

Das Gegenstück zu `Show` ist die Typklasse `Read`:

```
class Read a where
  read :: String -> a
  ...
```

Da der Typ `a` nur im Ergebnis vorkommt, müssen wir GHCi manchmal sagen, welchen Typ wir brauchen:

```
> read "7" :: Int
7
> read "7" :: Double
7.0
```

Innerhalb eines Programmes kann GHC oft den Typ inferieren:

```
> [ read "True", 6==9, 'a'/='z' ]
[True,False,True] :: [Bool]
```

Read Typklasse

Das Gegenstück zu `Show` ist die Typklasse `Read`:

```
class Read a where
  readsPrec :: Int -> ReadS a
  ...

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

- Kann ebenfalls automatisch abgeleitet werden
- Viele Instanzen in der Standardbibliothek definiert
- Wenn man die `Show`-Instanz selbst definiert,
  dann sollte man auch die `Read`-Instanz selbt definieren!
- Funktion `read` kann eine Ausnahme werfen

```
> read "7" :: Bool
*** Exception: Prelude.read: no parse
```

Überladene Instanzen

Interssanterweise können wir auch gleich automatisch Paare und Listen von Früchten vergleichen:

```
> (Apfel (3,4), Birne 5 6) == (Apfel (3,4), Birne 5 6)
True

> [Apfel (3,4), Birne 5 6] /= [Apfel (3,4), Birne 5 6]
False
```

Dies liegt daran, dass die Deklaration von Typklassen und Instanzen auch selbst wieder überladen werden dürfen!

Überladene Instanzen

Wenn wir je zwei Werte von Typ `a` und zwei von Typ `b` vergleichen können, dann können wir auch Tupel des Typs `(a,b)` auf Gleichheit testen, dank dieser Deklaration der Standardbibliothek:

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = (x1==x2) && (y1==y2)
```

Wenn wir jeweils zwei Werte von Typ `a` vergleichen können, dann können wir auch gleich eine Liste des Typs `[a]` vergleichen:

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

Überladene Instanzen

Wenn wir je zwei Werte von Typ `a` und zwei von Typ `b` vergleichen können, dann können wir auch Tupel des Typs `(a,b)` auf Gleichheit testen, dank dieser Deklaration der Standardbibliothek:

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = (x1==x2) && (y1==y2)
```

Wenn wir jeweils zwei Werte von Typ `a` vergleichen können, dann können wir auch gleich eine Liste des Typs `[a]` vergleichen:

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

Erster Vergleich: `Eq a`

Überladene Instanzen

Wenn wir je zwei Werte von Typ `a` und zwei von Typ `b` vergleichen können, dann können wir auch Tupel des Typs `(a,b)` auf Gleichheit testen, dank dieser Deklaration der Standardbibliothek:

```
instance (Eq a,Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = (x1==x2) && (y1==y2)
```

Wenn wir jeweils zwei Werte von Typ `a` vergleichen können, dann können wir auch gleich eine Liste des Typs `[a]` vergleichen:

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

Erster Vergleich: `Eq a` Zweiter Vergleich: `Eq [a]` rekursiv!

Ord Typklasse – Unterklasse von Eq

Werte eines Typs aus Typklasse `Ord` können wir nach Größe ordnen:

```
data Ordering = LT | EQ | GT

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

- Alle Typen der Typklasse `Ord` müssen auch in `Eq` sein!
  `Ord` ist also Unterklasse von `Eq`.

  Achtung: `compare` und `(==)` sollten übereinstimmen;
  wird aber **nicht** von GHC geprüft
- Instanzen müssen mindestens `compare` oder `(<=)` angeben
- Kann oft automatisch abgeleitet werden
- Viele praktische Funktion in der Standardbibliothek:

```
minimum, maximum :: Ord a => [a] -> a
sort             :: Ord a => [a] -> [a]
```

Beispiel: Überladene Instanz

Wenn wir die Ordnung umdrehen möchten, können wir das leicht mit einer überladenen Instanz leicht tun.
Die Standardbibliothek definiert in Modul `Data.Ord` für uns:

```
data Down a = Down a deriving (Eq, Show, Read)

instance Ord a => Ord (Down a) where
  compare (Down x) (Down y) = y `compare` x
```

Dies funktioniert analog zur Idee von Folie 05-19 `PreisFrucht`:

```
> sort [2,4,1,3]
[1,2,3,4]

> sort [Down 2,Down 4,Down 1,Down 3]
[Down 4,Down 3,Down 2,Down 1]

> [ y | Down y <- sort [ Down x | x <- [2,4,1,3]]]
[4,3,2,1]
```

Enum Typklasse

Die Typklasse `Enum` ist für alle aufzählbaren Typen:

```
class  Enum a   where
  succ, pred     :: a -> a     -- Nachfolger/Vorgänger
  fromEnum       :: a -> Int   -- Ordnungszahl
  toEnum         :: Int -> a
  ...
  enumFromTo     :: a -> a -> [a]       -- [a..e]
  enumFromThenTo :: a -> a -> a -> [a] -- [a,s..e]
```

- Wenn alle Konstruktoren eines Datentyps keine Argumente haben, kann `Enum` automatisch abgeleitet werden
  Beispiel `data Day = Mon|Tue|Wed|Thu|Fri|Sat|Sun`
- Viele Instanzen in der Standardbibliothek vordefiniert
- `Enum` impliziert nicht automatisch `Ord`!

Bounded Typklasse

Die Klasse aller beschränkten Typen:

```
class  Bounded a  where
  minBound, maxBound :: a

> maxBound :: Bool
True
> minBound :: Int
-9223372036854775808
```

- Klasse `Ord` ist nicht in `Bounded` enthalten, da nicht jeder geordnete Typ ein größtes oder kleinstes Element haben muss
- Kann für alle Typen der Klasse `Enum` abgeleitetet werden
- Kann für Datentypen mit einem Konstruktor abgeleitet werden, wenn alle Argumente Typen aus `Bounded` haben

Num Typklasse

Auch die arithmetischen Funktion sind mit diesem Mechanismus überladen, es wird keine Sonderbehandlungen benötigt:

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

- `(+)` leicht erweiterbar, um z.B. mit `Frucht` zu rechnen
- Kann nicht automatisch abgeleitet werden
  mit Einschränkung möglich für newtype Deklarationen
- Instanzdeklarationen dürfen höchstens entweder `negate` oder `(-)` weglassen

Integral und Real Typklassen

Erweitert die Klasse `Num` um ganzzahlige Divison:

```
type Rational = Ratio Integral -- Ganzzahl Brüche

class (Num a, Ord a) => Real a where
  toRational ::  a -> Rational

class  (Real a, Enum a) => Integral a where
  div :: a -> a -> a
  rem :: a -> a -> a
  mod :: a -> a -> a
  ...
  toInteger :: a -> Integer
```

- `Real` enthält nur Umwandlungsfunktion zu Typ `Rational`
- Die Klasse `Integral` impliziert also `Real`, `Enum`, `Num`, `Ord`, `Eq`

Typklasse Fractional

Typklasse für alle Zahlen, welche Fließkomma-Divison unterstützen:

```
class  (Num a) => Fractional a  where
  (/)          :: a -> a -> a
  recip        :: a -> a          -- Kehrwert
  fromRational :: Rational -> a
```

Typklasse für alle Fließkommazahlen:

```
class  (Fractional a) => Floating a  where
  pi             :: a
  exp, log, sqrt :: a -> a
  (**), logBase  :: a -> a -> a
  sin, cos, tan  :: a -> a
  ...
```

Umwandlung zwischen Zahlen

- Es gibt noch weitere spezielle Zahlenklassen (z.B. `RealFrac`)
- Wichtige Funktionen zur Umwandlung von Zahlen sind:

```
fromIntegral :: (Integral a, Num b)    => a -> b
realToFrac   :: (Real a, Fractional b) => a -> b
round        :: (Integral b, RealFrac a) => a -> b
ceiling      :: (Integral b, RealFrac a) => a -> b
floor        :: (Integral b, RealFrac a) => a -> b
```

- Viele numerische Funktion außerhalb von Typklassen definiert, ermöglichen bequeme Verwendung ohne explizite Umwandlung:

```
(^)  :: (Num a, Integral b) =>          a -> b -> a
     -- verlangt positive Exponenten
(^^) :: (Fractional a, Integral b) => a -> b -> a
     -- erlaubt negative Exponenten
```

Zusammenfassung Typklassen

- Typklassen definieren ein **abstraktes Interface** bzw. Typklassen definieren polymorphe Funktionen für eine eingeschränkte Menge von Typen
- Ermöglichen benutzerdefiniertes "Überladen" ohne fest eingebaute Sonderbehandlungen
- Jeder Typ kann nur eine Instanz pro Klasse deklarieren
- Typ kann vielen Typklassen angehören, je nach Eigenschaften

  keine Hierarchie nötig (wie z.B. in Java)!
- Typklassen können mit Unterklassen erweitert werden
- GHC versucht immer den **prinzipalen** (allgemeinsten) Typ einer Deklaration zu inferieren

  deshalb tauchen Klassen in Fehlermeldungen auf

Module

Ein Haskell Programm besteht oft aus mehreren Modulen:

```
module Main where
  import Prelude
  ...
```

- Modulenamen werden immer groß geschrieben
- Pro Modul eine Datei mit gleich lautendem Namen
- Punkte in Modulnamen entsprechen Verzeichnisunterteilung:
  `ModA.SubA.MyMod` ist in Datei `ModA\SubA\MyMod.hs`
- Jedes Modul hat seinen eigenen Namensraum:
  `Mod1.foo` kann eine völlig andere Funktion wie `Mod2.foo` sein
- Standardbibliothek ist das Modul `Prelude`.
  Es gibt viele weitere nützliche Standard-Module:
  `Data.List, Data.Set, Data.Map, System.IO, ...`
- Haskell Module sind nicht parametrisiert

Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

Export

Export und Import von Modulen kann eingeschränkt werden:

```
module Tree ( Tree(..), Oak(OakLeaf), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)
data Oak a  = OakLeaf | OakBranch (Tree a) (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right

hidden :: Tree a -> a
...
```

- `Tree` und alle seine Konstruktoren werden exportiert
- Von `Oak` wird nur Konstruktor `OakLeaf` exportiert
- Funktion `fringe` wird exportiert, aber `hidden` nicht

Import

```
module Import where
  import Prelude
  import MyModul - definiert foo und bar

  myfun x = foo x + MyModul.bar x
  ...
```

- Wenn ein Modul importiert wird, werden alle exportierten Deklaration in dem importierenden Modul sichtbar.
- Man kann diese direkt verwenden, oder mit `ModulName.bezeichner` ansprechen
- Mehrfachimport unproblematisch, so lange alle Pfade zum selben Modul führen

Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head`, `init`, `last`, `tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
- Zum Beispiel gibt es `Data.Map.map` und `Prelude.map`, wenn eine Funktion `map` da, dann muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen

  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

Import

Beim Importieren kann auch eingeschränkt und unbenannt werden:

```
module Import where
  import MyModul (foo, bar)
  import Prelude hiding (head, init, last, tail)
  import Tree    hiding (Oak(..))
  import Data.Map as Map
  import qualified Data.Set as Set
```

- Aus `MyModul` werden nur `foo` und `bar` importiert
- Wir importieren `Prelude`, außer `head, init, last, tail`
- Wir importieren `Tree` ohne alle Konstruktoren des Typs `Oak`
- Deklaration aus `Data.Map` sind auch mit `Map.` ansprechbar
- Deklaration aus `Data.Set` müssen wir mit `Set.` ansprechen
  Zum Beispiel gibt es `Data.Map` und in `Data.Set` eine Funktion `size`, daher muss eines von beiden qualifiziert importiert werden, wenn wir beide brauchen

Module

Module erlauben also:

- Unterteilung des Namensraums
- Aufteilung von Code auf mehrere Dateien
- Unterstützen Modularisierung und Abstraktion durch das Verstecken von Implementierungsdetails
- Instanzdeklarationen für Typklassen werden immer exportiert und importiert

In GHCI importieren wir Module mit

```
> :module + Data.List Data.Set
```

und schließen Module mit

```
> :module - Data.List Data.Set
```

Module

Module erlauben also:

- Unterteilung des Namensraums
- Aufteilung von Code auf mehrere Dateien
- Unterstützen Modularisierung und Abstraktion durch das Verstecken von Implementierungsdetails
- Instanzdeklarationen für Typklassen werden immer exportiert und importiert

In GHCI importieren wir Module mit

```
> :module + Data.List Data.Set
```

und schließen Module mit

```
> :module - Data.List Data.Set
```

Beispiel: Endliche Abbildungen

Eine **endliche Abbildung** über zwei Typen `Key` und `Value` ordnet endlich vielen **Schlüsseln** (Werten aus `Key`) jeweils *einen* Wert aus `Value` zu.

Funktionalität:

- Abfragen ob ein Schlüssel vorhanden ist
- Abfragen des Wertes eines eingetragenen Schlüssels
- Eintragen einer Schlüssel/Wert Zuordnung
- Entfernen einer Schlüssel/Wert Zuordnung

Die Reihenfolge der Schlüssel-Wert-Paare ist jedoch unerheblich!

Beispiele:

- Telefonbuch ordnet jedem Namen eine Telefonnumer zu; also eine endliche Abbildung zwischen Namen und Nummern.
- Zweispaltige Tabelle, bei der in der ersten Spalte keine Doppelten vorkommen. Zweite Spalte darf doppelte enthalten.

Beispiel: Data.Map

Modul `Data.Map` stellt endliche Abbildungen als *abstrakte* Datenstruktur für einen Typ `Map k a` bereit:

```
member :: Ord k => k -> Map k a -> Bool
```
Prüft, ob ein Schlüssel vorhanden ist

```
lookup :: Ord k => k -> Map k a -> Maybe a
```
Wert eines Schlüssels nachschlagen, wenn möglich

```
insert :: Ord k => k -> a -> Map k a -> Map k a
```
Schlüssel-Wert-Paar eintragen; überschreibt ggf.

```
delete :: Ord k => k -> Map k a -> Map k a
```
Löscht einen möglicherweise vorhandenen Schlüssel

. . . und noch viele mehr!

Beispiel:

```
> Data.Map.lookup "Steffen" telefonbuch
Just 9139
```

Beispiel: Data.Map

In Haskell bleibt natürlich alles rein funktional, d.h. Einfügen oder Löschen liefert lediglich eine veränderte Kopie der endl. Abbildung:

```
> let telefonbuch2 = Data.Map.delete "Martin" telefonbuch

> Data.Map.lookup "Martin" telefonbuch2
Nothing

> Data.Map.lookup "Martin" telefonbuch
Just 9341

> Data.Map.insert "Sigrid" 9337 telefonbuch2
fromList [("Sigrid",9337),("Steffen",9139)]

> Data.Map.lookup "Sigrid" telefonbuch
Nothing
```

Beispiel: Data.Map

Frage: Warum ist die Datenstruktur hier abstrakt?

`Data.Map` exportiert keinen Konstruktor des Datentyps `Map v k`
Stattdessen werden nur Funktionen zur Konstruktion bereitgestellt:

```
empty :: Map k a
```
Leere Abbildung — eine Konstante

```
singleton :: k -> a -> Map k a
```
Abbildung mit einem Schlüssel-Wert-Paar

```
fromList :: Ord k => [(k, a)] -> Map k a
```
Einlesen aus Liste von Paaren

Vorteil: Implementierung ist problemlos austauschbar.
Aus Effizienzgründen werden z.Zt. "size balanced binary tree" verwendet. Für uns als Anwender ist dies jedoch egal.
Das Modul definiert eine klare Schnittstelle;
Implementierungsdetails brauchen uns nicht zu kümmern.

Zusammenfassung Data.Map

Das Modul `Data.Map` der Standardbibliothek...

- ist ein Beispiel für eine abstrakte Datenstruktur bei dem die Implementierungsdetails durch das Modul-System versteckt werden
- stellt endliche Abbildungen bereit, welche oft ein sehr nützliches Hilfsmittel sind
- stellt *effiziente* Implementierung von endl. Abbildungen bereit
- sollte immer qualifiziert importiert werden:

  ```
  import qualified Data.Map.Strict as Map
  ```

  Viele Bezeichner überschneiden sich mit anderen Modulen, z.B. `Prelude`, `Data.List`, `Data.Set`, ...
  `empty` kann leere Liste, leere Menge oder leere Abbildung sein, aber `Map.empty` macht klar, was gemeint ist.

Zusammenfassung Data.Map

Das Modul `Data.Map` der Standardbibliothek...

- ist ein Beispiel für eine abstrakte Datenstruktur bei dem die Implementierungsdetails durch das Modul-System versteckt

Tipp: Aufgrund des qualifzierten Import erhält man Typsignaturen
`foo :: Map.Map k a -> Map.Map k a`
Mit der Deklaration `type Map k v = Data.Map k v`
kann man das vermeiden.

- sollte immer qualifiziert importiert werden:
  ```
  import qualified Data.Map.Strict as Map
  ```

  Viele Bezeichner überschneiden sich mit anderen Modulen, z.B. `Prelude`, `Data.List`, `Data.Set`, ...
  `empty` kann leere Liste, leere Menge oder leere Abbildung sein, aber `Map.empty` macht klar, was gemeint ist.